N° d'ordre 2826

THÈSE

présentée

DEVANT L'UNIVERSITÉ DE RENNES 1

pour obtenir

le grade de : *DOCTEUR L'UNIVERSITÉ DE RENNES 1*

Mention : Informatique

par

Simon Pickin

Équipe d'accueil : IRISA/TRISKELL
École doctorale : Matisse
Composant universitaire : IFSIC

Titre de la thèse :

Test des composants logiciels pour les télécommunications

Soutenue le 2 juillet 2003 devant la commission d'Examen

M. :	Claude	JARD	Président
MM. :	Roland	GROZ	Rapporteurs
	Juan	PAVON MESTRAS	
	Ina	SCHIEFERDECKER	
MM. :	Jean-Marc	JÉZÉQUEL	Examinateurs
	Thomas	JENSEN	

Table of contents

Acknowledgments v

Résumé en Français vii

Chapter I : Introduction 1

1 Context and motivation 3
- 1.1 Software testing 3
- 1.2 Component testing 4
- 1.3 Telecom testing 5
- 1.4 Testing languages 6

2 Objectives 7

3 Contributions 9

4 Guide to the rest of this document 10

Chapter II : Conceptual Background 11

1 Basic Concepts 13
- 1.1 The testing addressed in this document 13
- 1.2 The different actors in a test case 14
- 1.3 Structuring concepts 16
- 1.4 Behavioural concepts 19

2 Precursory work 21
- 2.1 An approach to component testing 21
- 2.2 Formal approaches to conformance testing 23
- 2.3 Scenario languages in testing 25

Chapter III : UML Sequence Diagrams as a basis for a test description language 29

1 Using UML for a Test Description Language 31
- 1.1 Motivation for approach taken 31
- 1.2 Semantics in the UML standard 32
- 1.3 UML 1.4 sequence-diagram semantics 33
- 1.4 UML 1.4 sequence diagrams for test descriptions 37

2 Other issues for our scenario-based test language 47
- 2.1 Level of abstraction of TeLa 47
- 2.2 The internal structure of the tester and the SUT in TeLa 48
- 2.3 Guards in TeLa 51
- 2.4 Verdicts in TeLa 52
- 2.5 Message order as a special case of event order in TeLa 55
- 2.6 Data in TeLa 56
- 2.7 Time in TeLa 57

3 Extending UML 1.4 sequence diagrams 58
- 3.1 Sequential composition 58
- 3.2 Internal actions 58
- 3.3 Synchronization messages (or virtual messages) 59
- 3.4 Explicit concurrency 59
- 3.5 Local ordering 60
- 3.6 Loops 60
- 3.7 Branching 61
- 3.8 Focus bars/suspension bars 62
- 3.9 TeLa textual language 63

4 UML 2.0 sequence diagrams …… 65
4.1 Semantics via an abstraction/conformance relation? …… 65
4.2 Abstraction/conformance relation constructs …… 68
4.3 Other problems with UML 2.0 sequence diagrams …… 69
4.4 Concluding remarks on UML 2.0 sequence diagrams …… 70

Chapter IV : A scenario-based test description language for component testing (TeLa) …… 71

1 Overview of TeLa language …… 73
1.1 One-tier or two-tier scenario structures …… 73
1.2 Limited parallelism …… 74
1.3 Other significant omissions from the language …… 74
1.4 The underlying component model …… 75
1.5 Verdicts …… 76
1.6 TeLa textual language / data language …… 77
1.7 TeLa test descriptions …… 77

2 TeLa sequence diagrams and 1-tier scenario structures …… 78
2.1 Basic constituents of TeLa sequence diagrams …… 78
2.2 Message arrow labels …… 82
2.3 TeLa sequence diagram references …… 88
2.4 Internal actions …… 90
2.5 Coregions …… 95
2.6 Local orderings …… 97
2.7 Focus bars …… 97
2.8 Lifeline composition …… 103
2.9 Sequence diagram loops …… 104
2.10 Sequence diagram choices …… 107

3 TeLa activity diagrams and 2-tier scenario structures …… 131
3.1 Overview of syntactic elements …… 131
3.2 TeLa activity-diagram references …… 133
3.3 Sequential composition (weak and strong) …… 134
3.4 TeLa activity-diagram choice …… 135
3.5 Implicit and explicit verdicts …… 141
3.6 TeLa activity-diagram loops …… 143

Chapter V : Elements of a formal semantics for a scenario-based test description language for component testing …… 145

1 Elements of structural semantics …… 147
1.1 Component structures and component models …… 147
1.2 Test description component models …… 151
1.3 Cuts and interaction frameworks …… 153
1.4 Generalised cuts and generalised interaction frameworks …… 156
1.5 Structural semantics of TeLa sequence diagrams …… 159
1.6 Structural consistency of TeLa sequence diagrams …… 162

2 Elements of behavioural semantics …… 168
2.1 Outline of behavioural semantics for TeLa test descriptions …… 168
2.2 Determinism and controllability in TeLa …… 177
2.3 Semantics of implicit verdicts …… 192

3 Incorporating a more restrictive behavioural semantics …… 198
3.1 Partially-ordered messages from partially-ordered events …… 198
3.2 Partially-ordered messages and explicit concurrency …… 200
3.3 Implementation of partially-ordered message semantics …… 200
3.4 Trace formulation of partially-ordered message semantics …… 202
3.5 Partially-ordered message semantics for a component …… 202

Chapter VI : Test Synthesis from UML Models of Distributed Software …… 205

1 Introduction …… 207

1.1 Background....207
1.2 Definition of behavioural concepts....208
1.3 Overview of the approach....215

2 Derivation of LTS from UML model.... 219
2.1 Prerequisites for the derivation of an LTS.... 219
2.2 Transforming the UML specification....228
2.3 Generating the simulation API....230
2.4 From LTS to IOLTS....236
2.5 "Umlaut-simulatable" model for the ATC application.... 241
2.6 Formal specification derivation: possible enhancements....242

3 Derivation of LTS from scenario-based test objective.... 244
3.1 The O-TeLa language....244
3.2 From O-TeLa expression to LTS.... 245
3.3 A test objective for the ATC application....249
3.4 Formal objective derivation: possible enhancements....251

4 Test synthesis from model LTS guided by objective LTS.... 252
4.1 The test synthesis engine....252
4.2 Test case synthesis for the ATC application.... 253
4.3 Test synthesis: possible enhancements....255

5 Derivation of scenario-based test case from IOLTS.... 257
5.1 The TeLa language....257
5.2 From IOLTS to TeLa expression.... 258
5.3 Synthesized test case for the ATC application....261
5.4 UML test case derivation: possible enhancements....267

Chapter VII : Conclusions....269

1 Original contributions.... 271
1.1 Structural semantics of scenario-based test descriptions.... 271
1.2 Behavioural semantics of scenario-based test descriptions....272
1.3 Definition of a scenario-based test description language....276
1.4 Application of TeLa: test synthesis using Umlaut/TGV....277
1.5 Conceptual framework.... 277

2 Future work.... 278
2.1 Semantics of TeLa....278
2.2 Syntax of TeLa.... 278
2.3 Tool support.... 278
2.4 Extensions to TeLa....279

Bibliography.... 281

Appendix A : Grammar of .aut files used by TGV/Umlaut.... 293

A1 Grammar of .aut input in system test synthesis.... 295
A1.1 Constraints....295
A1.2 Observations....295

A2 Grammar of .aut output in system test synthesis....296
A2.1 Constraints....296
A2.2 Observations....296

1 Acknowledgements

Starting at the beginning, rather than at the end, I'd first like to thank my parents for valuing education and in particular, my mother, for putting relatively scarce resources into books, lots of books! Next, I'd like to thank my brothers and sisters for teaching me about argument and debate, even if it did sometimes get a bit heated. Sincere thanks to the fast-disappearing British welfare state for paying for my education at Sussex University, Cambridge University, King's College London and Imperial College London, all of which came free and with a grant that included living expenses, something that has become a luxury in the post-Thatcher dark ages. Thanks also to the lecturers from this period of my life, notably Roger Fenn of Sussex University maths department.

Thanks to my friends and colleagues at France Télécom R&D (or CNET as it was then called) for introducing me to the world of telecom research. Thanks also to friends and colleagues at FT R&D, those at the different Spanish universities where I have worked (DIT/UPM, IT/UC3M and LSI/EUI/UPM), and to those of the different EU projects I have worked on, for many interesting discussions, technical and otherwise.

More directly concerning the doctoral studies, I would like to express my gratitude to Gonzalo Leon Serrano, not only for bringing me to Spain and taking me on as a PhD student when I endeavoured to complete a doctorate in Madrid, but also for his clear-sighted vision as to what constitute promising research directions. My gratitude also to my office colleagues at DIT, Carlos Sanchez Tarnawiecki, for helping me get on my feet with CORBA and Alvaro Rendón Gallón, for discussions about the use of MSCs in test description.

Most importantly, I would like to express my deep gratitude to Claude Jard and Jean-Marc Jézéquel for giving me the chance to finish my doctorate at the University of Rennes / IRISA and the chance to benefit from their clarity of thought and their wide professional experience while, at the same time, working in a quality research group at an excellent research centre in a country that, by and large, still values research. Particular thanks to Jean-Marc for undertaking the responsibility of supervising the PhD of a sometimes unruly, old hand like myself. In fact, I am indebted to all my colleagues at IRISA for demonstrating to me the efficacy of coherent and collaborative research groups as opposed to the every researcher for himself/herself philosophy (which in-vogue productivity measures seem to actively encourage). Jean-Marc, in particular, expends quite some effort in the thankless task of trying to maintain such coherence and collaboration.

Other colleagues at IRISA that deserve special thanks are Thierry Jéron and Loïc Hélouet for helpful technical discussions and clarification of some important concepts, Yves Le Traon and, especially, Clémentine Nébut for some productive technical collaboration, Jacques Malenfant, Heinz Schmidt and Clémentine Nébut for some varied discussions in the few months during which I shared an office with each of them, Didier Vojtisek, for his helpfulness, availability and ingenuity in solving some technical support problems, Romain Sagnimorte for some technical support advice, Marie-Noëlle Georgeault, the *atelier* staff and the cafeteria staff for their efficiency and good nature, and, finally, Franck Fleurey, Clémentine Nébut and, especially, Benoît Baudry for their invaluable help with PhD administrative tasks, particularly when I was no longer on French soil, making it impossible for me to carry them out on my own.

My gratitude to the reviewers (*rapporteurs* and *examinateurs*) of this thesis for the time and effort they dedicated to ploughing their way through what turned out to be a rather oversize document.

I would also like to thank all my colleagues from the COTE project – in the context of which much of the work reported on here was carried out – for their collaboration, particular those that were local and with whom the collaboration was closest, notably Eric Riou and Nicolas Bulteau of Softeam, and Thierry Heuillard of France Télécom R&D.

On a more personal note, I would like to thank Thierry Jéron, Sophie Pinchinat, Hervé Marchand, Bertrand Jeannet, Loïc Hélouet, Claude Jard, Philippe Darondeau, Benoit Caillaud and others for making my PhD phase more enjoyable, Vlad Rusu and Marek Bednarczyk for livening up this phase with some interesting, and occasionally heated, political discussions, Pat Morfitt, Marco Villagra, Paul Cavel and others for helping to keep me (vaguely) in shape during this time, and, finally, Angeles for her love and support (and taking on more than her fair share of the housework!), without which I would have had great difficulty getting through the ordeal of writing-up, not to mention for her proof-reading of certain sections and for some very pertinent technical remarks on parts of the technical content.

2 Resumé en Français

La plupart du travail présenté ici a été réalisé par l´auteur dans le contexte du projet COTE [JarPic01] du programme de recherche national RNTL (Réseau National des Technologies Logicielles). Les membres de ce projet étaient France Télécom R&D, Gemplus, IRISA, LSR-IMAG et Softeam. Ces travaux seront détaillés par la suite.

Le travail presenté dans la Section 2.1 du Chapitre 2 a été realisé pendant le séjour de l'auteur dans l'équipe dirigée par Gonzalo León Serrano au *Departamento de Ingeniería de Sistemas Telemáticos* (Département de l'Ingénierie de Systèmes Telematiques) de l'*Universidad Politécnica de Madrid* (Université Polytechnique de Madrid), Espagne, avec le support du programme national de recherche intitulé: *Estancias Temporales de Científicos y Tecnólogos Extranjeros en España* (Séjours Temporels de Scientifiques et Technologues Étrangers en Espagne). Une partie des résultats de ce travail a été publiée dans [PicSanYel96], [PicSanSan96].

2.1 Chapitre 1

Dans le premier chapitre, nous introduisons le document au moyen d'une brève description du contexte, de la motivation, des objectifs et des contributions de la thèse, ainsi qu'un guide pour le reste du document.

2.2 Chapitre 2

Dans le deuxième chapitre, nous présentons le cadre conceptuel du travail qui est décrit dans le reste du document.

La première section contient un glossaire qui définit la terminologie qui sera utilisée par la suite. Dans le domaine du test de composant, il n'y a toujours pas de base conceptuelle ni terminologie couramment acceptées. Bien que les définitions fournies ici soient inspirées par la terminologie du test de télécommunications et celle du test de logiciel standard, elles sont originales. Une version antérieure de ce glossaire a été fournie comme document de travail au consortium qui répond au RFP du OMG sur le profil de test fondé sur UML [OMG03].

La deuxième section résume brièvement les travaux précédents qui sont les plus pertinents pour cette thèse.

2.3 Chapitre 3

Dans le troisième chapitre, nous traitons le problématique de la conception d'un langage de description de test basé sur les diagrammes de séquence UML.

Dans la Section 1, nous étudions la possibilité de bâtir notre langage de description de test fonctionel, appelé TeLa, sur UML 1.4 [OMG01] et nous établissons la base sémantique de TeLa, en nous inspirant des différents travaux menés sur les langages MSC 2000 [ITU-T99] et TTCN-3 [ETSI03a], en particulier les travaux sur la sémantique des MSC. Les diagrammes de notre langage de description de test incluent des lignes de vie qui représentent des composants du testeur, mais aussi des lignes de vie qui représentent des élements du système

sous test (SUT). Cependant, le rôle des lignes de vie du SUT est limité puisque la sémantique est donnée par une projection sur les lignes de vie du testeur, conformément à la supposition de test « boîte noire ». A notre avis, un langage qui représente explicitement des lignes de vie du SUT est plus convivial qu'un langage qui utilise les *ports* de TTCN-3 ou les *gates* des MSC pour éviter de représenter explicitement ces lignes de vie.

La Section 2 présente d´autres concepts généraux importants pour la définition d'un langage de description de tests fondé sur les scénarios. La Section 3 présente les principales constructions nécessaires et qui sont absentes des diagrammes de séquence d'UML 1.4 [OMG01], ainsi qu'une justification de ces besoins. Finalement, dans la Section 4, nous faisons une évaluation des diagrammes de séquence d'UML 2.0 [U2P03], qui ont été définis alors que le travail présenté ici avait déjà commencé.

2.4 Chapitre 4

Dans le quatrième chapitre, nous présentons en détail le langage TeLa, avec une explication informelle de la sémantique non-entrelacée du langage. Dans [PicJarHeu01] nous avons présenté une version antérieure de TeLa.

La première section donne une vue d´ensemble des aspects clés de TeLa. La Section 2 introduit les différentes constructions des diagrammes de séquence de TeLa, ainsi que les structures de scénario à une couche qu'on peut construire avec eux. Une structure de scénario à une couche est composée de plusieurs diagrammes de séquence de TeLa connectés entre eux par des « références de diagrammes de séquence ». La Section 3 introduit les différentes constructions des diagrammes d'activité de TeLa, ainsi que les structures de scénario à deux couches qu'on peut construire avec ces constructions et celles des diagrammes de séquence de TeLa présentées dans la Section 2. Une structure de scénario à deux couches de TeLa est composée d'un diagramme d'activité de Tela dont les noeuds contiennent des diagrammes de séquence élémentaires de TeLa.

L'ensemble des tests qui peuvent être décrits par une structure de scénario à une couche est un sous-ensemble de celui des tests qui peuvent être décrits par une structure de scénarios à deux couches. Le sous-langage à une couche a été conçu pour faire en sorte que toutes les descriptions de test exprimées avec lui définissent des cas de test. Il a été conçu aussi pour être plus simple à utiliser que le langage complet, grâce aux opérateurs de choix et de boucle localment définies.

La présentation de TeLa ne vise pas à être la conception complète d'un langage, mais plutôt une analyse détaillée des questions soulevées par la définition d'un langage de test fondé sur les scénarios. La plupart des constructions, et plus particulièrement les constructions des structures de scénario à une couche, ont été proposées en tant que réponses à des besoins concrets qui se sont exprimés dans le projet COTE. Le choix de la syntaxe concrète, lui aussi, a été fortement influencé par les objectifs et les contraintes du projet COTE. Une des principales contraintes était que la syntaxe de TeLa devait rester aussi proche que possible de la syntaxe de UML 1.4, telle qu'implantée dans l'outil UML Objecteering. En conséquence, nous ne prétendons pas que la syntaxe concrète actuelle pour les opérateurs locaux, tel que l'opérateur de boucle et l´opérateur de choix des diagrammes de séquence de TeLa, soit la plus adéquate.

Deux constructions particulièrement intéressantes introduites dans ce chapitre sont la construction « verdict explicite » et la construction « alternative par défaut explicite ». Nous montrons les propriétés qu'une description de test devrait satisfaire pour que ces constructions

soient bien définies. D´autres constructions intéressantes sont celles pour dénoter une valeur inconnue dans le paramètre d´un message envoyé par le SUT et celle pour dénoter l´affectation d´une telle valeur inconnue à une variable dynamique.

Un des aspects de TeLa présenté dans ce chapitre qui mérite une attention particulière est le traitement des notions de schéma de flôt de contrôle et de l'idée d'un composant actif ou passif. Nous interprétons la notion de passivité comme des restrictions sur les linéarisations autorisées. Quand on utilise une sémantique non-entrelacée, ces restrictions constituent un autre niveau sémantique qui doit être ajouté (optionellement) au dessus de la sémantique de base. Le schéma de flôt de contrôle est une propriété de composant, dénotée par une annotation sur le modèle de composants qui décrit l'architecture de test. Les principales constructions de TeLa dont l'interprétation est affectée par le schéma de flôt de contrôle sont le *focus bar* et la *coregion*. Par exemple, l'effet d'un *focus bar* à l'intérieur d'une *coregion* sur une ligne de vie qui représente un composant passif est similaire à celui de la construction *critical region* d'UML 2.0. Nous soutenons que nos définitions modélisent de façon adéquate la notion de passivité par rapport à l'interprétation :

- de la concurrence comme le degré de liberté d´ordonnancement qui reste quand on est obligé d'implanter cette concurrence par scheduling et non pas par des files d'exécution,
- des appels synchrones comme des appels bloquants pour l'émetteur,
- des *focus bars* comme des représentations de l'exécution d'une méthode.

2.5 Chapitre 5

Dans le cinqième chapitre nous définissons les éléments principaux de la sémantique formelle de TeLa.

La Section 1 présente des éléments d'une sémantique structurelle pour TeLa. Nous definissons formellement les fondements d'un modèle de composant qui est sous-jacent à notre langage à base de scénarios, en termes de « composants » avec des « ports » qui sont reliés par des « connecteurs ». Bien qu'il ne soit pas le propos de ce travail, ce modèle peut être vu comme la formalisation d'une partie du modèle de composants d'UML 2.0. Ensuite, nous précisons l´idée de la base structurelle (ensemble de lignes de vie) d'un scénario de description de test comme une série de vues sur un « snapshot » du modèle de composants, afin d'assurer que cette base reflète l'« architecture de test » et la « configuration de test ». Le modèle de composants sous-jacent à une description de test joue aussi un rôle important dans la généralisation des techniques de synthèse de test dans le contexte de test de composants.

La hiérarchie du modèle de composants fournit un cadre dans lequel on peut définir la composition/décomposition des lignes de vie, ainsi que des propriétés sur des composants qui affectent l'interprétation des diagrammes. Les deux propriétés de ce genre introduites dans cette thèse sont la sémantique de communication, qui peut être à base de messages ou à base d'événements, et le schéma de flôt de contrôle, qui peut être passif ou actif. La hiérarchie de composants facilite aussi la définition de propriétés locales telles que la « contrôlabilité locale » et peut être utilisée pour guider le choix de l'architecture de déploiement.

La Section 2 présente des éléments d´une sémantique comportementale pour TeLa, et en particulier, des éléments d´une sémantique non-entrelacée. Les sémantiques non-entrelacées sont très importantes pour le test distribué. En ce qui concerne la description de tests qui impliquent des données non-énumérés, dans le Chapitre 5 nous n´avons formalisé que la sémantique entrelacée. Cependant, une description informelle de la sémantique non-entrelacée dans le cas des données non-énumerées est donné dans le Chapitre 4.

Nous donnons une définition formelle de « verdict local » pour les modèles non entrelacés et nous définissons formellement comment les « verdicts globaux » sont obténues à partir de ces « verdicts locaux », en incluant le cas du « verdict inconclusif ». La dérivation des verdicts locaux et la dérivation de verdicts globaux à partir de ceux-ci a une importance primordiale dans le contexte du test distribué.

Ensuite, nous donnons une définition formelle de « verdict local implicite » et de la propriété de « complétude de test » pour les modèles non-entrelacés. La dernière propriété est la bien connue « complétude en entrée » du cas entrelacé, avec, en même temps, des conditions pour assurer que n'importe quelle exécution qui ne se bloque pas termine avec un verdict global. Nous définissons les propriétés qu'un modèle non-entrelacé doit avoir pour que les verdicts implicits soient bien définis. Une notion bien définie de verdict implicite assure que les descriptions de tests ont un niveau suffisant d'abstraction tout en assurant la propriété de complétude de test.

Pour le cas non-entrelacé, nous affaiblissons les définitions standards de déterminisme et de contrôlabilité, ce qui nous amène aux définitions formelles de plusieurs notions de déterminisme et de contrôlabilité. Le concept de déterminisme minimal est essentiel pour que le verdict implicit de *fail* soit bien défini. Les différentes notions de contrôlabilité nous permettent de définir la hiérarchie de cas de test suivante: « cas de test parallèle », « cas de test parallèle cohérent à l'extérieur», « cas de test parallèle cohérent », « cas de test parallèle centralisable à l'extériure » et « cas de test centralisable ».

Dans la Section 3, nous définissons formellement comment une sémantique à base de messages peut être définie comme une restriction d'une sémantique à base d'événements pour une certaine classe de descriptions de tests (celles qui sont RSC: réalisable avec la communication synchrone). Nous montrons comment une sémantique à base de messages peut être mélangée à une sémantique à base d'événements composant par composant. Dans plusieurs cas, l'utilisation d'une sémantique à base de messages évite d'encombrer les descriptions de test avec beaucoup de messages de synchronisation. Ceci est le cas pour la répresentation en TeLa de la sortie de l'outil de synthèse de test TGV (fondé sur un modèle entrelacé), comme présentée dans le Chapitre 6. Le fait que la sémantique restreinte soit plus proche de la sémantique des diagrammes de séquence des versions d'UML antérieurs à UML 2.0 peut être aussi très utile pour les concepteurs de tests qui sont familiarisés aves ces notations.

2.6 Chapitre 6

Dans le sixième chapitre, nous étudions l´utilisation de TeLa dans le synthèse de test. Un résumé du travail présenté ici a été publié dans [PicJarTra02]. Une application de la méthode décrite ici dans le contexte des lignes de produit a été publié dans [NebPicTra].

Les premiers travaux sur le développement de la méthode de synthèse de test à partir d´un modèle UML avec Umlaut/TGV sont présentés dans [JézGuePen98], [JérJézGue98] et [Gue01]. Le premier article propose un cadre pour intégrer la technologie de vérification et validation formelles dans le cycle de vie orienté-objet. Le deuxième présente ce cadre plus en détail, en décrivant le schéma global pour générer une API de simulation à partir d'un modèle UML afin de pouvoir utiliser les outils de vérification et de validation, en particulier, l'outil de synthèse de test TGV. Le troisième décrit comment ce schéma a été réalisé dans le simulateur Umlaut.

Dans cette thèse, nous étendons la méthode de synthèse à partir des modèles UML avec Umlaut/TGV, pour arriver à une intégration complète avec UML, premièrement, à travers l'utilisation d'objetifs de test et de cas de test basés sur les scénarios et, deuxièment, à travers l'échange de modèles XMI avec les AGL UML commerciaux. Nous suggérons aussi comment la méthode pour le test système pourrait être généralisée au test de composants.

Dans le cadre de ce travail, nous avons raffiné et amélioré le simulateur d'Umlaut, tout en clarifiant la sémantique qu´il implante et les restrictions que la dérivation de cette sémantique, à partir d´un modèle UML, impose sur ce modèle. Pour cela, nous avons bénéficié de l'évaluation et du test de la méthode et de l'outil que nous avons entrepris conjointement avec les membres du projet COTE qui jouaient le rôle d´utilisateurs, notamment France Télécom R&D. Un des objectifs principaux de COTE était d'étudier l'applicabilité de l'approche TGV à la synthèse de test dans le domaine UML.

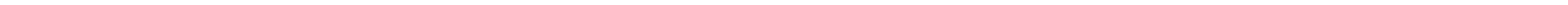
Ce chapitre vise aussi à documenter la méthode et l´outil originaux, ainsi que ses extensions, en montrant clairement les différentes phases et les différents choix de sémantique et d'implantation dans chaque phase. Nous avons inclu les notes d´implantation concernant le prototype actuel tout au longue du chapitre. Ces notes précisent les endroits où l'implantation actuelle est incomplète, ou bien, les endroits où il y a des difficultés à implanter la méthode telle qu'elle est présentée. Ainsi, ce document peut aussi servir comme un manuel technique pour l'outil. À la fin de chacune des principales sections, on récapitule nos principales propositions pour améliorer et étendre la méthode et l'outil déjà étendus dans le projet COTE.

Dans la Section 1, nous fournissons la définition des termes additionels (par rapport aux termes définis dans le Chapitre 2) dont nous aurons besoin dans ce chapitre. Dans cette section aussi, nous présentons une vue d'ensemble de la méthode, en la divisant en quatres parties principales. La première de ces quatre parties, qui concerne la dérivation d´un modèle formel (un système de transitions étiquetées) à partir d´un modèle UML de l´application, est présentée dans la Section 2. La partie suivante, qui concerne la derivation d'un modèle formel (un système de transitions étiquetées) à partir d'une représentation UML d'un objectif de test – dans le langage O-TeLa – , est présentée dans la Section 3. Le cœur de la méthode et de l'outil, la synthèse de test sur des modèles formels, est présenté dans la Section 4. La dernière partie, qui concerne le *mapping* entre la représentation en termes du système de transitions étiquetées d´entrée-sortie du cas de test produit, et une representation UML de ce même cas de test – dans le langage TeLa – , est présentée dans la Section 6. Nous illustrons la méthode décrite dans ce chapitre en faisant référence à un exemple de sistème de contrôle de trafic aérien simplifié.

2.7 Chapitre 7

Dans le septième chapitre, nous concluons en décrivant les contributions originales de cette thèse ainsi que les perspectives. Nous décrivons les contributions dans les domaines des langages de description de test fondé sur les scénarios, de la sémantique structurelle de TeLa, de la sémantique comportementale de TeLa, de l'utilisation de TeLa pour la synthèse de test et aussi des contributions originales dans le cadre conceptuel sur lequel est basé le travail. Les perspectives concernent la sémantique de TeLa, la syntaxe de TeLa, le support de TeLa par des outils, et les extensions de TeLa.

Chapter I : Introduction



1 Context and motivation

The testing phase is an extremely important part of any software development. The cost of this phase, even in non-critical software projects, commonly represents between a third and a half of all development costs. In areas where the cost of failure and outages is very high (this can be measured in different terms but the usual measure is hard cash), the costs of testing will be much greater than this.

As pointed out by Beizer [Bei90], there is a common myth according to which if programmers were really good, there would be no testing phase. While it is true that more rigour in earlier phases should reduce the costs of the testing phase, software testing is absolutely unavoidable. Moreover, currently, and for the foreseeable future, automatic code generation from a verified specification is only applicable in very limited circumstances. Testing is necessary in any engineering discipline and all the more so in a still very young discipline such as software engineering where it is particularly difficult to reduce the set of parameters to be verified during development, and tested at the end of it, to a reasonably-sized set. If the collapse of structures such as bridges and dams was a stimulus for progress in civil engineering, a good many more serious software disasters (hopefully not involving loss of life) will be needed before software engineering reaches a level of development at which the denomination engineering becomes indisputable.

An important way of reducing the high cost of testing and therefore, also, the high cost of software, is by automating aspects of the testing phase. One approach to doing so is by raising the level of abstraction at which the tests are described and automating the generation of platform-specific tests from these abstract tests; another is by automatically generating tests of an implementation from the specification of that implementation. In both of these techniques, the introduction of a more formal approach is of obvious value, and the language in which the tests are to be described plays a pivotal role.

1.1 Software testing

Testing is generally viewed as one of the set of techniques described by the ubiquitous term V&V (Validation and Verification).

1.1.1 What is verification and validation?

The usual distinction between validation and verification, apparently due to Boehm [Boe75] but adopted by the IEEE in their official definition [IEEE90], is as follows: verification attempts to answer the question “is the system being built right?” while validation attempts to answer the question “is the right system being built?”. That is, verification concerns the internal coherence of the application while validation concerns whether the application fulfils the requirements. According to another similar well-known definition, verification is an evaluation of whether a software development product complies with the requirements at the start of that development phase (or is coherent with the corresponding product of the previous phase), whereas validation is an evaluation of whether the final product complies with (user) requirements. Though useful and, in the case of the first definition at least, undoubtedly good mnemonics, these definitions are less than ideal. In fact, particularly in the case of the first definition, it is not obvious that the two definitions are mutually exclusive, see below.

1.1.2 What is testing?

Testing, on the other hand, is usually defined as the activity of looking for errors in the final implementation. Though the term "final implementation" is still subject to different interpretations, this definition is more useful for our purposes than the above definitions. Clearly, for all but the most trivial applications, correctness, that is, the absence of errors, cannot be demonstrated through testing. At the risk of being pedantic, we restate that the objective of testing is to find errors; it cannot be to show their absence.

Like many software terms, and indeed a large part of language, the term "testing" has recently fallen victim to marketing considerations. In the wake of widely-publicised, hugely-expensive software failures, positive perceptions of testing increase among the software-developer community and the software-user community at large, and the tendency to use the term "testing" as widely as possible, however inappropriate it may be, increases commensurately. In this document we will not use the term testing for any checking that is not performed on the implementation.

1.1.3 What are black-box testing and conformance testing?

By black-box or functional testing we will understand testing of the behaviour of the implementation without reference to the internal structure of the application, in particular, without reference to its source code. By conformance testing we will understand testing designed to check that an implementation conforms to a specification, preferably in terms of a well-defined conformance relation. Though testing is universally considered a validation activity, according to a strict reading of the above definitions of V&V, it is not obvious whether conformance testing is to be classified as verification or validation.

1.1.4 Abstraction and current software testing trends

In recent years, the benefits of abstraction and the idea of working at model level has finally begun to take root in the software-development community, witness the OMG´s "model-driven architecture" approach, a better name for which would be model-driven development. Graphical analysis and design languages are at the heart of these developments, in particular, the OMG-standard modelling language UML [OMG03]. A radical revision of this language, UML 2.0, is scheduled for release in mid 2003. Testing technologies used in industry are beginning to reflect these developments by taking advantage of analysis and design assets to introduce more automation into the testing phase.

1.2 Component testing

According to the definition of UML 2.0 [U2P03][1], which represents a relatively well-developed consensus view, a component, "is a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment." The main aspects of this definition are thus encapsulation and substitutability. A component is an instantiable

[1] As stated above, the UML 2.0 specification has not yet been released. In this document, when we refer to the UML 2.0 proposal, we are referring to the latest proposals of the U2 Partners consortium. This does not present any difficulties since we are only interested in the superstructure part of the standard and there is only one remaining proposal for this superstructure: that of the U2 Partners.

entity[2]. Also according to UML 2.0, "a component specifies a formal contract of the services that it provides and those that it requires from other components in the system. The required and provided services are defined through ports that are characterised by interfaces."

Conformance testing of a component, then, checks that it conforms to its specification in terms of services, where any communication with a component must be carried out via its ports. This is not just service testing, that is, checking a single provided service in isolation, but also checking dependencies and temporal orderings between services provided and required by the component and its peers. To emphasise the difference with the standard notion of contract that is generally associated to testing individual services, these dependencies and orderings that a component participates in and which must be checked were called contractual relations in [PicSanYel96].

Over the last few years, standard software applications have come to be increasingly distributed. This tendency began tentatively as the simple separation of the processing from the database and later also from the client, with communication taking place over a LAN or WAN, the buzzwords of the time being 2-tier and later 3-tier architectures. This tendency accelerated with the use of technologies such as Java+RMI, CORBA [OMG02b], DCOM/COM+, and the next buzzwords: n-tier and middleware. The tendency shows all the signs of increasing further with the latest wave of technologies (and buzzwords) based on distributed component technologies: J2EE, .NET, CCM [OMG02a], Web Services [W3C02], MDA, etc.

1.3 Telecom testing

Testing in the telecommunications area has certain peculiarities. The applications to be tested are frequently distributed and the testing must often be carried out partially, or fully, through some mediating system. Most communication is asynchronous and in most cases, control aspects are overwhelmingly more important than data aspects. Generally speaking, the systems used are reactive, that is, any activity they may have is provoked by their environment. Finally, many systems are critical, in the sense that failures can be very expensive.

The socio-economic context is also different in that heterogeneity of the infrastructure has been the norm for quite some time. Since the telecom monopolies have been dismantled, service heterogeneity and the number of actors sharing parts of the infrastructure has also increased markedly. As a consequence, conformance testing and certification is of great importance in the telecom field and a large amount of effort has been invested in the techniques and tools needed to carry it out.

This set of circumstances has led to a well-developed and well-established conceptual basis for conformance testing in the telecom field, see [ISO/IEC92a]. It has also led to a significant degree of formalisation of this conceptual basis, see [Pha94], [ITU-T97]. This formalisation has favoured the development of formal approaches to test generation from a specification, see, for example, [Tre96], [Jér02]. We will use the term test synthesis for this, to distinguish it from the generation of concrete tests from abstract tests. Of particular interest in our view is test synthesis from a specification guided by test objectives. Current formalisations of

---

[2] We use slightly different terminology to UML 2.0 in that we use the term "component" for what UML 2.0 calls "component instances" and the "components" of UML 2.0 are referred to in our component model as "component types".

conformance-testing concepts are based on interleaving models but attention is now turning to more general formalisations using non-interleaving or “true concurrency” models, see for example [Jar02].

The aspect of communicating systems that was referred to in the OSI model as the “application layer” has become far more elaborate in modern telecom systems and, just as in other areas of software, there is now a marked tendency towards the use of distributed component-based applications. This is an indication of the fact that the frontiers between telecom software and other types of software are becoming steadily more diffuse and, consequently, the distinction between the types of testing applied is also becoming less clear.

The development of an approach to testing in component-based systems, while building on the established bank of knowledge of the software testing field, see, for example, [Bei90], [Bin00], could clearly benefit from adopting and adapting the conceptual basis established in telecom conformance testing. Similarly, an attempt to formalise the conceptual basis of component-based testing should also strive to take advantage of the body of work done in this area in the telecom field.

1.4 Testing languages

Testing as a software domain has sufficient specificities for specialised languages to have been developed in which to write test descriptions. In this particular software domain, efficiency is not generally a crucial consideration but interactivity, portability and extensibility are sought-after qualities. Many testing programs consist of relatively simple operations performed a large number of times (with different data). These factors point to the use of interpreted languages and, indeed, scripting languages have often been used as software testing languages, e.g. use of Expect [NIS03] for testing applications with a command-line interface.

As mentioned above, the area of conformance testing has seen significant development in the telecom field in the last ten years or so. Part of this development has been the design of a conformance testing language, TTCN [ISO/IEC92b], currently in its third version. In the first version tests are described in a syntax resembling state tables with tree-like decision structures. The second version introduced distributed testing constructs and the third version, TTCN-3 [ETSI03a], is a fully-fledged programming language aimed at general communicating systems. The progression of the TTCN language through these three versions mirrors the developments in telecom software mentioned above. The change of name between versions two and three from the “Tree and Tabular Combined Notation” to the “Testing and Test Control Notation” and the fact that TTCN-3 has an associated graphical syntax GFT [ETSI03b] based on MSC [ITU-T99] are also significant developments.

The software landscape sketched in the previous paragraphs points to the need for a graphical component testing language. If such a language is to be widely integrated in software development projects, it should be based on the standard modelling language UML. The most adequate UML view on which to base such a language, particularly in the telecom domain is the sequence-diagram view (recall that TTCN-3 GFT is based on MSC). In our opinion, in order to avoid overloading the graphical syntax with large numbers of constructs, thereby losing much of the advantage of using a graphical syntax, this language should be situated at a higher level of abstraction than TTCN-3.

2 Objectives

This thesis concerns software component testing in general but the methods, techniques and tools discussed here are most adequate for systems in which the communication aspect is significant and which are not data-intensive. An important feature of this area of testing is the lack of adequate language support, particularly UML-integrated language support, as already mentioned. In the work reported on here, we address the issues of using a scenario-based test-description language for component testing of centralised and distributed applications and propose such a language, called TeLa (Test Language) [PicJarHeu01]. In designing TeLa we were obliged to find solutions to some non-trivial semantic problems. We then apply scenario-based test descriptions in the context of test synthesis from UML models.

We base our test description language on UML for the following reasons:

- A UML-based language can be smoothly integrated into a UML-based process and UML CASE tools.
- Using a UML-based language allows us to capitalise on the widespread familiarity with UML. This then facilitates not only the production of tests, but also the understanding of tests written by others, encouraging the use of tests as documentation.
- Use of UML-based test descriptions in a UML-based development manifests the relation between tests and other software development assets.

Among the UML views, the sequence-diagram view provides the most suitable basis for such a language since it is the only UML view in which temporal orderings can be clearly represented (in our opinion, collaboration diagrams with message numbering annotations do not satisfy this criterion). Sequence diagrams are also considered user-friendly and easy-to-understand (though, in fact, appearances can be deceptive).

We also sketch the development of a more generic scenario-based description language similar to TeLa to be used to describe test objectives, called O-TeLa. We then apply the test objective language O-TeLa and the test description language TeLa to the problem of the synthesis of test cases from UML models guided by test objectives.

The major part of the work presented here has been carried out by the author in the context of the COTE project [JarPic01] of the French RNTL national research programme. The participants in this project were France Télécom R&D, Gemplus, IRISA, LSR-IMAG and Softeam. In the COTE project, TeLa was conceived as the heart of a UML test environment involving both synthesis of TeLa test cases from O-TeLa test purposes as well as manual production of TeLa test cases. The COTE project also dealt with other aspects of TeLa such as its implementation in a UML CASE tool and the production of executable tests on different platforms from TeLa test descriptions. However, the author´s part in these latter aspects of the use of TeLa being limited, we do not discuss these issues in this thesis.

The work reported on in Chapter 2, Section 2.1 was carried out during the author´s stay in the group led by Gonzalo Léon Serrano at the *Departamento de Ingeniería de Sistemas Telemáticos* (Telematic Systems Engineering Department) of the *Universidad Politécnica de Madrid* (Madrid Technical University), Spain on a national research grant under the programme entitled "*Estancias Temporales de Científicos y Tecnólogos Extranjeros en España*" ("Temporary Posts in Spain for Foreign Scientists and Tecnologists").

Shortly after the COTE project commenced, the OMG published an RFP (Request For Proposals) for a UML Testing Profile. The consortium replying to this proposal published its proposals in March 2003 [UTP03]. Some of the author´s earlier work was provided as initial

input to this consortium and the author has had a limited participation in the work of this consortium. However, the work of the consortium has diverged from that presented here for a variety of reasons. In spite of this, significant commonalities remain and the main aspects of the two proposals could be made compatible.

3 Contributions

In the area of semantics for scenario-based test descriptions, the original contributions of this thesis concern firstly the structural semantics of such descriptions, in terms of an underlying component model, and secondly the behavioural semantics of such descriptions, in terms of partial-orderings. The formal definition of the key concepts required for a component-based, partial-order, test description language to be well-defined played a crucial role in guiding the construction of the language defined in this work. This fact notwithstanding, due to the complexity of the problem, we have not been able to define the full semantics in a completely formal fashion.

In the area of the definition of scenario-based test description languages, we have defined such a language, called TeLa, that includes a simplified easier-to-use sublanguage with its locally-defined operators. The contributions also concern the evaluation of the suitability for describing tests of existing scenario-based languages, and the testing-specific language constructs which must be added w.r.t. these other scenario-based langauges to obtain a test description language.

In the area of test synthesis, the area in which we have chosen to apply the scenario-based descriptions developed here, the contributions concern extending and generalising the methods and tools used.

Finally, we also define the conceptual basis of the work, materialised here in the form of a small glossary of terms.

4 Guide to the rest of the document

In Chapter 2 we present the conceptual background against which this work has been carried out.

In Chapter 3, we identify the choices made in designing our language and justify them, clarifying the semantic elements involved. We do so without losing sight of the objective of designing a UML-integrated language.

In Chapter 4, we present the TeLa language itself, introducing each of the constructs and illustrating the chosen syntax. For each construct, we identify the conditions needed for it to be well-defined and explain its informal semantics.

In Chapter 5, we formalise some crucial testing concepts in the semantic domain but we do not explicitly give the mapping from the syntax to this domain. These aspects concern the component model underlying a test description and the most important elements of the behavioural semantics of scenario-based test descriptions.

In Chapter 6, we present the use of the TeLa language (and the O-TeLa language) in test synthesis using Umlaut/TGV.

In Chapter 7, we conclude by summarising the contributions of this thesis, while pointing out the relevance of these contributions, and by describing the research lines and applications constituting the possible continuations of this work.

Chapter II : Conceptual Background

In this chapter we present the conceptual background to the work reported on in the rest of the document. In the first section, we briefly define the basic concepts we will need in the rest of the document. In the next section, we briefly summarise earlier work of direct relevance to the work of this thesis.

1 Basic Concepts

In this section we give more precise definitions of the terms used in this document. The definitions given in this section are our own. They represent an attempt to marry software testing terminology such as that of [Bei90], [IEEE90], [NIS96], [BSI98], OO testing terminology such as that of [Bin00] and telecom testing terminology such as that of [ISO/IEC92a] and [Tre96],[JarJér02]. An earlier version of some of these definitions was provided as initial input to the UTP (UML Testing Profile) consortium replying to the OMG´s Testing Profile RFP, see [UTP03].

1.1 The testing addressed in this document

1.1.1 Testing phases

1.1.1.1 UNIT TESTING

Testing of software units, that is, the smallest software components which it makes sense to test individually. In the case of object-oriented software, the logical units for unit testing are usually classes.

1.1.1.2 INTEGRATION TESTING

Testing of groups of software components, looking for errors in the interaction between these components. The integration can be carried out top-down, testing first the components which provide top-level services, or bottom-up, testing first the basic components. Clearly, performing the integration bottom-up helps to minimise the number of stubs needed.

1.1.1.3 SYSTEM TESTING

Testing of the whole software application, integrating all the groups of units tested in the integration testing phase.

1.1.2 Types of testing

1.1.2.1 COMPONENT TESTING

Testing of an individual software component. With a hierarchical component model, unit testing and system testing can both be viewed as types of component testing. Furthermore, part of the integration testing can also be viewed as the testing of successively larger components.

1.1.2.2 PASSIVE TESTING

Testing in which the tester observes the behaviour of the system under test during execution in its real environment and without emitting any stimulation, and emits a verdict as a result of comparing the observed behaviour with the expected behaviour. Frequently, the passive tester only emits a verdict if incorrect behaviour is observed. Passive testing of a software component is normally carried out *in situ*, that is, using the

real implementation of the components whose services the component under test requires. In this document, we are not concerned with passive testing.

1.1.2.3 ACTIVE TESTING

Testing in which a test controller sends stimuli to the system under test with the aim of exercising certain of its execution paths, observes its responses to these stimuli and emits a verdict as a result of comparing these responses to the expected ones. In the tests discussed in this document, the stimuli are either method invocations or signal exchanges.

1.1.2.4 WHITE-BOX (OR STRUCTURAL) TESTING

Testing in which knowledge of the internal structure of the system under test is used, usually in the form of an abstract representation of the source code. Coverage criteria, such as traversal of all the nodes or arcs of a graph, can be defined on these abstract representations. In this document, we are not concerned with white-box testing.

1.1.2.5 BLACK-BOX (OR FUNCTIONAL) TESTING

Testing in which interaction with the component is carried out exclusively through the interfaces defined in its specification and without reference to lower-level descriptions.

The language we define in this document is adapted to the black-box testing context. Description of grey-box tests, in which communications between elements of the system under test (SUT) may be observable (via software probes, for example), would require modifications of the test language developed here. For example, it would be useful to have a notation allowing the test designer to specify assertions on the values of parameters of intra-SUT messages. The definitions of component under test, system under test, test case etc. are also tailored to the black-box testing context and may well need to be modified in a different context.

1.1.2.6 CONFORMANCE TESTING

Testing that an implementation conforms to a specification; conformance testing therefore supposes *a priori* that the specification is correct. Evidently, in conformance testing, the specification serves as the oracle, where this is the name generally given to the functionality of deciding whether the test results are as expected. Conformance testing is normally black-box testing.

1.2 The different actors in a test case

1.2.1 The system under test

1.2.1.1 THE COMPONENT UNDER TEST (CUT)

The component which is the subject of testing is referred to as the *component under test* (CUT). We will use the terms CUT implementation and CUT specification where there is a need to distinguish these two levels.

1.2.1.2 THE SYSTEM UNDER TEST (SUT)

It may be impractical, or even impossible, to test the CUT without the presence of other software components whose implementation has not been derived for the purpose of testing. An example of when this situation may arise is in integration testing, notably if performed by testing successively larger units. Another example is when the CUT is embedded in other software which is costly or difficult to emulate, or for which neither source code nor specification is available, so that, in consequence, it must be tested *in situ*.

Thus, the test may involve components communicating with the CUT, and with other similar components, through interfaces/ports that are not accessible to the test software. The CUT together with the set of components whose communications are not *all* to be observed and controlled as part of the test is denoted the *system under test* (SUT). In this document we do not consider the grey-box testing case where some such communications can be observed but none of them can be controlled, e.g. using software probes. We will use the terms SUT implementation and SUT specification where there is a need to distinguish these two levels.

In order to maintain a precise notion of what is being tested, the situation where the SUT contains components other than the CUT clearly requires the hypothesis that the implementations of these other components are completely correct. The fact that they have already been tested may be enough for us to make this hypothesis. Notice that only two characteristics distinguish the CUT from the other components of the SUT: the intention of the designer to test the former and the correctness hypothesis for the latter. Though these two characteristics may be important in the test design, their only impact on the test execution is in the interpretation of a pass or fail verdict.

With regard to the black-box aspect of testing, in a black-box test, all interaction with the SUT is carried out exclusively through the interfaces defined in its specification and without reference to lower-level descriptions. This can still be the case even if the implementation under test is that of a CUT properly contained in the SUT implementation. Thus, in spite of the fact that an SUT-CUT distinction constitutes a hypothesis about the internal structure of the SUT, the latter is still the black-box of the testing. Similarly, with regard to the conformance testing, the specification which is assumed correct is that of the entire SUT, not just that of the CUT.

In most cases, the services required by the SUT of other software components are emulated by the tester, e.g. using stubs, see below. In such cases the CUT coincides with the SUT.

1.2.2 The test software

1.2.2.1 THE TESTER

We will refer to all the software in the test set-up which is not part of the SUT as the *tester*. From the SUT point of view, during the test, the tester plays the role of the SUT environment. The purpose of this software is to stimulate the SUT implementation, observe its responses and deliver a corresponding test verdict. We will use the terms tester specification and tester implementation where there is a need to distinguish these two levels. The tester may be divided into separate components at different hierarchical levels. At any hierarchical level, the component containing the functionality for delivering the test verdict will be called the *judge*. This component may, or may not, be the same component as the one which initiates the test.

As already stated, if the test involves using components whose implementations have not been derived for the purpose of testing but whose communications can *all* be observed, such components are considered to be part of the tester by virtue of the black-box testing supposition.

1.2.2.2 STUBS

A *stub* is usually defined as a skeletal or special-purpose implementation of a software module, used to develop or test a component that calls or is otherwise dependent on it[1]. In the testing context, stubs are purpose-built for testing and their outputs are therefore controlled (even if this may be at compile time rather than at execution time). Their inputs can also all be observed; however, if we choose not to implement the necessary code to do so, such stubs are then part of the SUT. Though such situations may arise in performing integration testing by testing components of greater and greater size, in the cases of interest here, stubs are part of the tester.

A typical situation in which the term stub may be used is when the component structure of the SUT environment part of a model of the entire application is used as the component structure of the tester, see the definition of the test architecture below. Another example is when special-purpose test modules are used to deal with each SUT invocation to its environment. This latter case is of particular interest since it may be possible to automatically generate such pure server stubs from the description of the test case in which they are involved.

1.3 Structuring concepts

1.3.1 Structure of the SUT

1.3.1.1 THE SUT COMPONENT INTERFACE

The *SUT component interface* groups interface instances of two kinds, where the types of these interfaces are defined in the test context:

- The instances of interfaces offered by the SUT to its environment (where we include any UML signals the SUT can receive from its environment). Between them, the interfaces contain all those SUT operations that may be invoked by the tester in the course of execution of the tests (including the set of signals that may be sent by the tester to the SUT).
- The instances of interfaces required by the SUT of its environment (where we include any UML signals the SUT can send to its environment). Between them, the interfaces contain all those tester operations that may be invoked by the SUT in the course of execution of the tests (including the set of signals that may be sent by the SUT to the tester).

We assume that the SUT component interface is usually structured in terms of ports, each with its multiplicity. If an SUT component model is available, e.g. as part of a specification of the entire application, the ports of the SUT interface are those ports of

[1] Note that this is the meaning of the term stub in testing and software development; in the context of RPCs, DCE, RMI, CORBA etc. it is used to mean a server proxy.

the component model which channel communications between the SUT and its environment. A port may contain a provided and a required part.

1.3.2 Structure of the tester

1.3.2.1 THE TEST CONTEXT AND TEST ENVIRONMENT

The *test context* is defined to be the collection of components and classes from which the tester is built. A test context may be shared by different test cases. The *test environment* is defined to be the execution environment of the test case or test cases.

1.3.2.2 THE TESTER COMPONENT INTERFACE

The *tester component interface* groups interface instances of two kinds, where the types of these interfaces are defined in the test context:

- The instances of interfaces offered by the tester to the SUT (including signals). Between them, the interfaces contain all those tester operations that may be invoked by the SUT in the course of the test.
- The instances of interfaces required by the tester of the SUT (including signals). Between them, the interfaces contain all those SUT operations that may be invoked by the tester in the course of the test.

We assume that the tester component interface is usually structured in terms of ports, each with its multiplicity. The tester component interface is defined as part of the test architecture.

1.3.2.3 THE TEST ARCHITECTURE AND TEST CONFIGURATION

The *test architecture* comprises the following four elements (see Chapter 5, Section 1 for more detail concerning the component model):

- the tester component type and its internal structure, defined using the elements of the test context; the "tester component interface" defined above comprises the set of ports of the tester components which are visible to the environment of the whole tester component (the unique instance of the tester component type),
- the SUT component type as a base-level component type; in the absence of an SUT component model, the "SUT component interface" defined above comprises the set of ports of the whole SUT component (the unique instance of the SUT component type),
- the connections, denoting client-server dependencies and therefore logical channels for method invocations, between the ports of the tester component interface and those of the SUT component interface,
- the communication architecture to be used for the connections between tester components and between the tester component interface and the SUT component interface.

The test architecture refers to the static (though it may include multiplicity information) structural information concerning the tester and its connections to the SUT. We will assume it is described using a component diagram, preferably based on a hierarchical component model defined in terms of ports and connectors between ports. It may also be annotated with information concerning the communication architecture, the control flow scheme etc., inside each component. A test architecture may be shared by different test cases.

There is a minimum of two levels of hierarchy in the test architecture in that there must be a single component, the whole tester component, which includes all the others (in order to be able to represent the tester as a single lifeline in our sequence diagrams, if required[2]). Similarly, in the test architecture, the SUT is considered to be a (non-standard) base-level component, whose ports are those of the "SUT component interface". In the simple object model case, the internal structure of the tester comprises only standard base-level components (objects); we suppose (contrary to systems such as CORBA) that any pure client must be encapsulated in an object.

The choice of test architecture will affect the process by which the global verdict is arrived at, and the mechanisms needed to verify the orderings specified in the test case. However, the level of abstraction of our test description language is such that this process and mechanisms are not included in test descriptions.

In the case where the SUT specification is part of a model of an entire application, the basis of the test architecture is defined by the component structure of the application, more precisely, by the part comprising the SUT environment and the SUT component interface. Such a test architecture will be called a default test architecture, w.r.t. the application specification. The functionality of coordinating the test may need to be completed and that of performing the task of the judge will need to be added, possibly inside a new component. In the case where the SUT specification is the entire model of the application (system testing), the components of the default architecture correspond to the external actors. This is the case treated in the application of Chapter 6.

Among the other possible test architectures, one of particular interest involves using a centralised tester in which one subcomponent makes all the invocations to the SUT and contains the judge functionality, while separate stub subcomponents deal with each of the invocations made by the SUT to its environment.

The *test configuration* is defined by the test architecture together with the dynamic information of the current state of the test components: currently existing links between objects and values of attributes (which includes current knowledge of SUT component IDs etc.). The initial configuration is the test configuration at the start of the test case.

1.3.3 Structure of the data exchanged between tester and SUT

1.3.3.1 DATA PARTITION

For large or infinite input data domains, it is clearly unrealistic to test each possible value. The most common way of reducing the number of values to be tested is to divide the input data domains into equivalence classes, under the hypothesis that the SUT behaviour does not depend on the particular member of an equivalence class that is used (the uniformity hypothesis). Commonly, also, a representative is chosen for each equivalence class. The set of equivalences classes for the input data, with or without representative values, is known as the *data partition*. In a given tester input, the data may be constrained, possibly by an accumulated set of constraints, in which case, the required equivalence classes partition the subset of the data which satisfies the constraints. Note that each tester output may require a different data partition.

---

[2] In particular, this possibility will be needed to represent the output of a test synthesis tool based on an interleaving model, such as TGV, see Chapter 5

There are many different techniques for generating the data partition from the specification or design model, in black box testing, and from the source code, in white box testing. These techniques therefore constitute the basis of the uniformity hypothesis. They are referred to as test data generation techniques.

1.4 Behavioural concepts

1.4.1 Dynamic creation

Both the tester and the SUT may create new components in their respective domains. However, the tester (respectively SUT) cannot create SUT components, (respectively tester components). Notwithstanding this restriction, the SUT (respectively tester) can implement creation methods which can be invoked by the tester (respectively SUT), assuming that the corresponding operation is offered in the interface of one of its components. We will require variables and constants taking values in the set of components in our test description language in order to model the communication of the identity of any dynamically-created SUT components or their ports, particularly knowledge of the creation of dynamically-created SUT components / ports to the tester.

1.4.2 General formulation of behavioural concepts

In Chapter 6, we give more specific formulations for the concepts of this section, and of some additional concepts, for the case where the underlying semantics is based on labelled transition systems and the case where the underlying semantics is based on partial orders.

1.4.2.1 ACTIONS

An action is a basic behavioural unit. When discussing UML representations, we will generally identify these actions with UML-actions.

1.4.2.2 CONTROLLABLE AND OBSERVABLE ACTIONS

The actions of the tester can be divided into controllable actions and observable actions. *Controllable actions* are actions describing tester outputs or tester internal actions; *observable actions* are actions describing tester inputs. Note that in this document, we do not use the term observable actions in the sense of actions that are not internal, as in, for example [Mil89]. We will also refer to transitions labelled by a controllable, resp. observable action, as controllable transitions, resp. observable transitions. Similarly for controllable events and observable events.

1.4.2.3 TEST VERDICTS

A *(test) verdict* is one of the following possible outcomes for a test case: *pass*, *fail*, *inconclusive* and *error*, see [ISO/IEC92a]. The first three verdicts concern the observed behaviour of the SUT implementation and the last verdict concerns correct execution of the test software.

In the enumerated data case, the meaning of the inconclusive verdict is w.r.t. a test objective. It is assigned when the behaviour of the implementation under test, while correct according to its specification, does not permit the test objective to be verified. In

the non-enumerated data case, the inconclusive verdict can also be assigned when the data values do not permit correct execution of the test but the exact origin of the problem (which data item received from the SUT) is not clear.

As stated above, the judge is the component responsible for assigning the global verdict. In some cases, the verdict may be derived in function of a set of *local (test) verdicts*, where a local verdict is a verdict derived by an individual test component in function of the behaviour it has observed. In such cases, the overall verdict derived by the judge is referred to as the *global (test) verdict*.

1.4.2.4 TEST RESULT

The *test result* is the behaviour actually performed in a test execution together with the verdict for that test execution. The behaviour actually performed in a test execution is the set of actions together with their parameter values either executed by the tester (controllable actions) or executed by the SUT and observed by the tester (observable actions), plus the ordering, and possibly timing, relations between them.

Note that we do not simply define the test result as the output of the SUT. In the presence of distribution and concurrency, such a definition is not sufficient, even if we view the input as the whole sequence of controllable actions and the output as the whole sequence of observable actions. This is since, due to the effects of concurrency, the SUT may exhibit observable non-determinism, i.e. it will not necessarily produce the same sequence of outputs for the same sequence of inputs. Moreover, orderings between controllable actions and observable actions are among the orderings we wish to check.

1.4.2.5 (CONFORMANCE) TEST CASE

A test case (meaning in this document a platform-independent test case, or abstract test case in the terminology of [ISO/IEC92a]) is a specification of an interaction between the tester and the SUT in which the tester stimulates the SUT via the “SUT component interface”, observes its responses at this interface, and assigns a verdict to the result of this interaction. The verdict is assigned in function of whether the test result is consistent with the SUT specification as defined by some conformance relation.

A test case is designed to exercise a particular execution or verify compliance with a specific requirement. The initial and final states of the tester in a test case correspond to the SUT being in well-defined and stable states according to its specification. A verdict must be associated to each final tester state. A fail or inconclusive verdict is obtained as early as possible from which it follows that, in the enumerated data case, a fail verdict or inconclusive verdict is always obtained immediately after the reception of a message from the SUT at the tester. In the non-enumerated data case, an inconclusive verdict can also be obtained immediately after the evaluation of a tester assertion or guard.

A test case may contain the specification of communications between different tester components (tester coordination messages) as well as between the SUT and the tester. It may also involve the creation of new objects or the destruction of existing objects by each of the two parties (tester & SUT) in their respective domains.

The generation of an executable test case for a particular platform from an abstract test case is normally a relatively easy task. An executable test case may contain extra messages, which are not in the alphabet of the specification, between those of the abstract test case. For example, if the test is implemented on a distributed platform, each such platform has its own internal messages.

2 Precursory work

In this section we discuss the methods, tools and techniques which have influenced the development of TeLa. We also discuss tool support for the testing process, the prime motivation for the definition of specific testing languages. We only discuss tools for test synthesis and test generation (defined below), since the work on TeLa was developed in the context of the COTE project [JarPic01] where both these types of tools were used. Of the two, we concentrate on the former, see Chapter 6.

Other types of tool support for the test process not discussed here are tools for test execution, some well known examples being the TETworks framework [OG03], the Expect tool [NIS03] the JUnit framework [BecGam98], [Goe01] and tools for test planning support, such as the tool based on the algorithm for minimising the stubs necessary in integration testing presented in [LeTJerJez00].

2.1 An approach to component testing

In this section we discuss component testing in the context of the component-based development methodology we developed in the OTELSO project (Eureka 1001), reported on in [PicSanYel96] and [PicSanSan96].

2.1.1 The OTELSO project

The goal of the OTELSO project was to define a methodology and implement a development environment for the specification, prototyping, implementation and testing of telecommunications applications on a standardised distributed-processing platform. In order to achieve this goal, the following tasks were undertaken:

- The study of the use of the formal techniques, notably SDL and LOTOS, alongside the use of object-based techniques and interface definition languages, notably CORBA IDL.
- The development of methodology guides and support tools for the testing and monitoring of distributed systems using graphical notations, notably MSC.
- The semi-automatic implementation on standardised distributed-processing platforms, notably CORBA (see [OMG02b] for the latest version of CORBA), via incremental prototyping techniques.
- The definition and implementation of a case study for demonstrating the development environment and the methodology: a distributed ATM system involving multiple banking consortia, banks and cashpoints located on a CORBA network.

The author was responsible for the part of this project concerned with testing, and therefore for the testing methods and tools developed in this project, as well as for the major part of the testing aspects of the OTELSO development methodology. We developed techniques for testing substitutable entities, or components, taking inspiration more from the Open Distributed Processing (ODP) framework [ISO95], rather than from the OMG´s rather limited Object Management Architecture (OMA) of the time.

Like the project definition, the proposals developed in the OTELSO project were very ambitious and, consequently, some of them were not implemented. This is particularly

true with respect to the manipulation of scenarios. For example, only very rudimentary event-ordering testing was actually implemented. However, due to the evolution of component and scenario technologies, many of these proposals are now more realistic than they were at the time the OTELSO work was carried out. In this sense, the methods and techniques discussed are more relevant now than when they were originally proposed, and an adaptation of these methods and techniques to a UML 2.0 context would, in our opinion, be of interest.

2.1.2 Component contracts in OTELSO

The OTELSO project relied heavily on the notion of contract, a notion which is fundamental to component-based development. The classic text in this field by the researcher who coined the phrase "design by contract" is [Mey88], [Mey97]. The usual notion of contract comprises invariants and pre- and post- conditions for the operations. The contracts of OTELSO contained both a provided and a required part and were used at different hierarchical levels as part of the development methodology.

The existence of contracts involving assertions can be used for testing purposes in several ways. They can be used in the testing phase for white-box/grey-box passive and active testing at both the integration and system level. They can also be used for non-regression testing of a component if its implementation changes and non-regression testing of its clients if it is used in a new context. These types of testing are the cornerstone of so-called self-testable, reusable components.

Contracts can also be used as the basis for black-box contractual conformance testing. In OTELSO, we investigated and implemented this type of testing for CORBA objects, by adapting the ADL language [SanHay94] and the corresponding ADLT tool to CORBA testing. Our proposal incorporated the possibility of either using stubs or of performing *in situ* testing, that is, using the real implementation of the components whose services the component under test requires, and monitoring their communications using software probes (implemented using the Orbix filter mechanism). In the former case, the stubs were actually situated in the same CORBA server as the tester program.

2.1.3 Contract scenarios in OTELSO

However, particularly in the telecom context, testing based on pre- and post- conditions of operations is not sufficient. Even for testing a single interface or port (we will refer to this as service testing), we need to be able to check invocation orderings and we would therefore like this to be part of the contract. For this reason, the methodology we developed in OTELSO allowed the addition of *contract scenarios*, described using the MSC´93 language, to the component contracts that we associated to ports[3]. We then investigated the derivation of event-ordering tests from these contract scenarios.

The scenarios were used to specify prohibited, optional and obligatory behaviour. We proposed to use the first as properties in passive testing, and the last two as test purposes or abstract test cases in active testing. As our testing architecture was built on top of a CORBA monitoring system using probes and a centralised monitor, also developed in OTELSO, we were obliged to address the problem of causal delivery of messages to the monitor in a distributed system. As each component of the architecture had a single

[3] In [PicSanYel96] and [PicSanSan96], we used the term interfaces rather than ports, for what were really interface occurrences.

port, we used a simple numbering scheme at each probe, together with a maximum message-delay supposition.

2.1.4 Contractual relations scenarios in OTELSO

However, again, particularly in the telecom context, contract-based testing derived from assertions and scenarios proved not to be sufficient. As our development methodology was based on a hierarchical component model, we used this model to extend the notion of contract further. As well as the contracts associated to the ports of a component, we associated a single contract to the component itself and included in this contract more scenarios, again specifying prohibited, optional and obligatory behaviour.

These additional scenarios were of two types: internal and external. The internal scenarios specified the communications between the ports of a component and its immediate subcomponents, and between these different subcomponents. The external scenarios were the projection of the internal scenarios of the parent component onto the messages involving that component. We referred to both types of scenario as *contractual relation scenarios*. Evidently, the external and the internal scenarios had to be coherent.

2.1.5 Scenarios in the OTELSO development process

In OTELSO, both assertions and MSCs were used throughout the development process, not just in the testing phase, and the development methodology considered two types of projection of scenarios. The projection of the internal contractual relation scenarios onto the messages involving an individual subcomponent to give the external contractual relation scenarios of that subcomponent, and the projection of the internal contractual relation scenarios onto an individual port of a subcomponent to give the contract scenarios of that subcomponent. The external contractual relation scenarios, in turn, form the basis of the internal contractual relation scenarios of the next level down in the component hierarchy. In the proposed top-down development, at each stage, the scenarios could be extended and completed.

As already mentioned, we proposed to use the contract scenarios and assertions of port contracts to derive service tests. Similarly, we proposed to use the external contractual relation scenarios of a component contract to derive component tests. The service tests derived from assertions were considered to be both part of the service tests and part of the component tests. We also began investigating how to incorporate the specification and execution of the assertion-based tests as part of the event-ordering based tests. Finally, the component hierarchy gave us a means of performing integration testing by testing successively larger components.

For more recent work on the use of MSCs in the development process see [Kru00], [MauRenWil01], [Bro02].

2.2 Formal approaches to conformance testing

As stated in Chapter 1, the testing phase absorbs a very large proportion of development costs. Automation of the tasks carried out in this phase holds out the prospect of significantly reducing these costs. Formalisation greatly facilitates automation, and in

the case of conformance testing, is essential in order to ensure that tests are correct, that is, they never reject a conformant implementation.

A formal definition of the notion of conformance relation is primordial in a formal approach to conformance testing. In conformance testing based on transition systems, a conformance relation is a test preorder, see [deNicHen84], [Abr87], in which observation is limited to the (externally visible) traces of the specification.

The models that have proved to be most applicable in testing distinguish the externally visible actions of the specification which are inputs from those which are outputs. We refer to such models as input-output models. Examples can be found in the Input-Output State Machines of [Pha94], the I/O Automata of [Lyn88], the Input-Output Transition Systems of [Tre96] and the Input-Output Labelled Transition Systems of [JerJar02] and of Chapter 6 of this document. These models also contain a third type of actions, the internal actions, which are susceptible to be hidden or abstracted away from, in order to show only the externally-visible behaviour. In many cases, there is no need to distinguish between different kinds of internal action so that the third subset often contains a single element, as in process algebra models.

To the inputs, resp. outputs, of the specification, correspond outputs, resp. inputs, of the tester. The input actions of the tester represent actions for which the tester environment, i.e. the SUT, has the initiative while the internal and output actions represent actions for which the tester itself has the initiative. The former kind of actions are termed *observable actions* while the latter are termed *controllable actions*.

We are interested in conformance relations on input-ouput models. The conformance relation used in [JarJer02], [Jer02] and in [Tre96] is the *ioco* relation. This relation supposes that the absence of visible activity (quiescence) – resulting from deadlocks, livelocks or waiting for input – is observable. Quiescence is considered to be a particular type of output though, in practice, tests detect this "output" by using timers. The *ioco* relation states that an implementation conforms to a specification if it cannot produce outputs which are unexpected w.r.t. the specification, after executing a trace of observable actions which is allowed by the specification (taking into account quiescence).

2.2.1 Formal approaches to test synthesis

In the 1980s, methods for synthesizing test cases from specifications based on Mealy machines were developed. A survey can be found in [LeeYan96]. However, these methods make rather restrictive hypotheses on the specification, and even more so on the implementation. In addition, due to the fact that the label on a Mealy machine transition contains both an input and an output, a Mealy machine is not the most suitable basis for modelling systems involving asynchronous communication, where, for example, a single input may lead to multiple outputs. Neither is it easy to map from specification languages such as SDL to Mealy machines.

For these reasons, in the 1990s, methods for synthesizing test cases from specifications based on input-output models were developed. For a survey of transition-system based methods, see [BriTre00]. These methods have begun to have some real application in industry. The methods most of interest in the context of this thesis are those that synthesize test cases from specifications guided by test objectives, using model-checking technology. In these methods, the synthesis is decoupled from the execution. Among these methods, in Chapter 6 we use one implemented in the TGV [JarJer02],

[Jer02] tool whose algorithms work on-the-fly. The use of on-the-fly algorithms means that arbitrarily large specifications can be treated. However, it also means that the synthesized test case is not necessarily minimal. More details of this kind of test synthesis can be found in Chapter 6.

Another approach developed in the 1990s is to synthesize tests from algebraic data type specifications, see [Gau95][LeGArn96].

2.3 Scenario languages in testing methods and tools

2.3.1 Scenario languages

The origins of using scenario languages to describe communicating entities lie in formalisms such as Lamport´s causality diagrams [Lam79]. Currently, the most well-known scenario langauge is the MSC language [ITU-T99], standardised by the ITU, which has gone through several versions over the last ten years. We will refer to the last three versions as MSC´93, MSC´96 and MSC´2000. For more details concerning the design of the current version, see [Ren99] and [Eng01].

For a survey of recent work on the use of MSCs and scenario languages, see [Pel02] and [HélJar01].

The object-oriented analysis and design language OMT [RumBlaPle91] also contained a rudimentary scenario language based on an early versions of the MSC notation and, seemingly, on the notions of [Lam86]. This language was imported into UML, the successor to OMT, where it was further developed [OMG01], [OMG03].

The idea of harmonising MSCs and UML sequence diagrams has been mooted for many years. More recently, this harmonisation began to take more concrete form, see [RudGraGra99] for example. The latest MSC standard has had significant influence on the sequence diagrams of the upcoming UML 2.0 standard, see [Hau01], [U2P03]. However, additional operators which risk seriously perturbing the semantics have been added to UML 2.0 w.r.t. MSC, see Chapter 3, Section 4.

2.3.2 Scenario languages in testing

2.3.2.1 SCENARIO LANGUAGES TO DESCRIBE PROPERTIES

The use of MSCs as properties on SDL models was already proposed in [Ek93], [AlgLejHug93]. An example of the use of MSCs as property descriptions in feature interaction detection, according to the detection framework of [ComPic94], can be found in [ComPicRen95].

2.3.2.2 SCENARIO LANGUAGES TO DESCRIBE TEST OBJECTIVES

A similar use of MSCs to describe test objectives was first proposed in [GraHofNah93], [Gra94], [Nah94], where these test objectives were used to guide the synthesise of test cases from an SDL model. The MSCs used were those of the MSC´93 standard which are much simpler than the MSCs of the current standard, involving no loops, alternatives or parallelism; they correspond roughly to the "basic MSCs" of the current standard.

2.3.2.3 SCENARIO LANGUAGES TO DESCRIBE/VISUALISE TEST CASES

The use of MSC´93 to describe test cases was also proposed in [Gra94], [Nah94], [GraHogNus95] and using the extended MSC´93 language of the Geode tool in [CavLeeMac97]. The use of timed versions of MSC´96 to describe test cases was proposed in, for example, [GraWal98], [Ren97] and [ObeKer99]. An extension of MSC´96 known as Hyper-MSCs for describing test cases was proposed in [EkkSchGra00b]. The use of MSC´96 to visualise test cases was proposed in [GraWal98]. The use of MSC´2000 to describe test cases is proposed in [SchRudGra00].

These efforts led to the development of a scenario-based graphical syntax for TTCN-3 [ETSI03b], a presentation of which can be found in [BakRudSch01]. A real-time extension to TTCN-3 is presented in [DaiGraNeu02] and further developed in [DaiGraNeu03].

The use of UML sequence diagrams to describe test cases was proposed in [EkkSchGra00]. An early version of TeLa, the test language based on UML 1.4 sequence diagrams presented in this thesis, was proposed in [PicJarHeu01]. More recently, a UML Testing Profile [UTP03], defining a testing framework based on UML 2.0 and making extensive use of UML 2.0 sequence diagrams was developed. A presentation of this profile is given in [SchDaiGra03].

2.3.3 Test generation, languages and tools

2.3.3.1 SCENARIO LANGUAGES IN TEST GENERATION

By test generation we understand the generation of a concrete test for a particular platform from a more abstract, platform-independent test. Evidently, the work on scenario-based test description has been accompanied by work on generation of concrete tests, e.g. in TTCN, from such test descriptions. Recently, there has been some work on generation of distributed tests from MSC-based test descriptions, e.g. [GrabKocSch99], [BakBriJer02], [DeuTob02].

Test generation from scenarios, particularly distributed test generation, is sufficiently non-trivial for some authors, such as [DeuTob02], to call these test descriptions test purposes. We prefer to reserve this term, and the term test objective, for descriptions allowing a greater degree of abstraction, even allowing for them to be selection criteria that contain only internal actions of the SUT specification, as is possible with the TGV tool.

2.3.3.2 TEST GENERATION TOOLS

If a test description language has a sufficiently high level of abstraction, it needs an associated test generation tool. As there are many such languages, there are many such test generation tools. We therefore only give some examples of tools performing test generation from a scenario-based test language.

Both the Autolink [SchEkGra98] and the Test Composer [GroJerKer99] tool can be used for test generation from MSCs, as well as for test synthesis. These tools are compared in [SchEbnGrab00]. The rather rudimentary (marketing jargon aside) tool Rational Quality Architect generating tests from UML sequence diagrams is presented

in [Rat01]. The ptk tool[4] for generating distributed tests from MSCs is presented in [BakBriJer02].

2.3.4 Test synthesis, languages and tools

2.3.4.1 SCENARIO LANGUAGES IN TEST SYNTHESIS

By test synthesis we understand the generation of a test case from a specification guided by a test objective. To our knowledge, the earliest work on test synthesis using scenario-based test objectives is that of [GraHofNah93], [Gra94] and [Nah94], as stated above. However, this synthesis from SDL specifications was not based on a formal notion of conformance. Since then, a more formal approach to test synthesis using scenario-based test objectives has been developed for SDL specifications, see [SchEbnGrab00], for example.

In [PicJarTra02] and in Chapter 6 of this document, we present a method for test synthesis from UML specifications based on the *ioco* theory of TGV [Jer02]. However, this involves a rather unnatural translation between non-interleaving models and interleaving models. The first steps towards developing a complete non-interleaving test synthesis method are presented in [Jar02]

2.3.4.2 TEST SYNTHESIS TOOLS

Early examples of test synthesis tools are TorX [Tre96], TGV [FerJarJer96], SAMSTAG [GraSchHog97], TVEDA [GroRis97], the first two being based on the *ioco* theory discussed above. Slightly more recent, commercial tools derived from these early examples and using scenario-based test objectives are Autolink [SchEkGra98] and Test Composer [GroJerKer99]; see [SchEbnGrab00] for a comparison of the two. A more recent version of TGV is presented in [JarJer02]. The work presented in Chapter 6 of this thesis lays the foundation for extending the TGV tool firstly, in order to be able to use as input scenario-based test objectives / test descriptions, and secondly, to component testing. A comparison of the algorithms used by the synthesis tools TorX, TGV and Autolink can be found in [Gog01].

---

[4] with its patented, though perhaps not novel, algorithms!

Chapter III : UML sequence diagrams as a basis for a test description language

In this chapter we discuss using scenarios as a basis for a test description language and, in particular, basing such a language on UML sequence diagrams. In Section 1, we discuss the use of UML as a basis for a test description language, which we call TeLa (Test Language) and establish a semantic basis. In Section 2, we introduce other general concepts of importance for defining a scenario-based test description language. In Section 3, we give an overview of, and a brief justification for, the main constructs we need to add to our test language with respect to UML 1.4 sequence diagrams. Finally, in Section 4, we give a brief assessment of UML 2.0 sequence diagrams, which have been defined since the work reported on here began.

1 Using UML for a Test Description Language

In this section, we discuss the issues involved in using UML as a basis for our test description language, which we call TeLa [PicJarHeu01]. First we deal with general considerations concerning the most suitable UML diagrams on which to base the language; in doing so, we briefly situate our approach w.r.t. other similar initiatives.

We start with the goal of using pre-UML 2.0 sequence diagrams [OMG01] [OMG03] as the basis for our test descriptions. We will refer systematically to UML 1.4 since, apart from a few minor corrections, the only difference between UML 1.4 and UML 1.5 is the addition of the action semantics. It is immediately apparent that these diagrams suffer from some serious ambiguities and that the first task towards accomplishing such a goal is to clarify their meaning. Thus, we are first lead to define the basics of the semantics of our test descriptions.

We examine the main difficulties in giving semantics to UML diagrams in general, and to UML 1.4 sequence diagrams in particular. We discuss the possible solutions to these problems and choose one such solution. We also indicate why we will interpret diagrams directly rather than passing via the UML 1.4 metamodel. We then examine the further difficulties which arise in giving semantics to UML sequence diagrams when these are to be viewed as test descriptions. Again we discuss possible solutions and choose one such solution. In each case where we choose a solution, we justify our choice with respect to the other possible solutions.

We then show how we can recover a semantics which is closer to UML 1.4 sequence diagrams as a specialisation of our chosen semantics and discuss the use of this feature.

1.1 Motivation for approach taken

Recall that we are interested in black-box testing and among the properties we wish to test are those concerned with correct ordering of the messages interchanged between the SUT and the tester, the latter playing the role of the SUT's environment. Among the different views used in UML modelling, interactions are clearly the most suited to describing message orderings and therefore the most adequate on which to base our test description language. Sequence diagrams are also considered to be especially user-friendly (as are related formalisms such as MSCs).

With simplicity and user-friendliness in mind, for our test language TeLa, we use only the sequence diagram, class diagram and component diagram views of UML with the overwhelming emphasis on the sequence diagrams. In contrast, the Test Profile defined by the consortium responding to the OMG RFP [UTP03] allows any UML diagram to be used in test descriptions and, in particular, proposes the use of state diagrams.

As explained below, in order to obtain a sufficiently-expressive test description language based on UML sequence diagrams their semantics must be clarified and their expressive power must be increased. In modifying UML 1.4 sequence diagrams, we do not attempt to maintain an equivalence between sequence diagrams, on the one hand, and collaboration diagrams with superimposed interactions (or “communication diagrams”, as their UML 2.0 equivalents are currently called), on the other. One reason for this is that such an equivalence inevitably involves the use of a sequence numbering system. Even assuming an adequate system can be devised to take into account guards,

loops, branching and explicitly-specified concurrency on a lifeline, the user-unfriendliness of such a system is inescapable.

The ETSI standardisation of TTCN-3 [ETSI03a] testifies to the current interest in MSC-like syntaxes for test languages. The latest version of the test language TTCN [ISO/IEC92b], which aims at a broader class of systems than its predecessors, comes equipped with a graphical syntax [ETSI03b] based on the MSC language [ITU-T99], [Ren99]. However, there are significant differences between the MSC/TTCN3 GFT and UML sequence diagrams. Moreover, in our opinion, an MSC-style graphical syntax is more suited to use in a language with a higher-level of abstraction than TTCN-3. In TeLa we allow numerous aspects which must be specified explicitly in TTCN-3 to remain implicit. Even if it were possible to introduce all the low-level detail of TTCN-3 into an MSC-style graphical language, the result would be almost unreadable and would lose the user-friendliness which such graphical languages are claimed to possess. In fact, in TTCN-3, the graphical language is incomplete and must be complemented by a significant amount of textual language.

We thus strive to maintain a certain coherence with GFT, while at the same time trying to remain as close as possible to the UML standard. In this respect, our approach could be considered to be more UML-integrated than that of [EkkSchGra00], for example. Having said that, it should be noted that the version of the UML standard which is under development is significantly different to the previous versions. For this reason, we have also kept an eye on the compatibility of our test description language with the draft versions of UML 2.0.

In spite of the desire to remain close to the UML standard, the range of behaviour we wish to treat makes the introduction of some new constructs inevitable, even w.r.t. UML 2.0. Where feasible, rather than extending the UML syntax, we propose adaptations of existing UML syntactic elements. We do not currently derive a mapping to the UML metamodel for each of the adaptations and extensions introduced, as this was not possible for the UML 1.4 metamodel and the work was carried out before any relatively-mature version of the UML 2.0 metamodel was available.

One of our original aims in starting this work was to define a UML Testing Profile. However, as it became clear that the UML 1.4 metamodel was not a suitable basis for our test description language, this goal was largely abandoned. After the start of work reported on here, a proposal was made at the OMG to standardise a UML testing profile based on the draft UML 2.0 proposals. The proposal of the consortium replying to the UML Testing Profile RFP has recently been made available [UTP03]. The author of the present document was able to contribute in a limited manner to this standardisation work, through the association of the IRISA laboratory with the French object-technology company Softeam, which is a member of this consortium. However, though much of the UML Testing Profile work is complementary to the work presented here, the scope is different. The UML Testing Profile addresses a wider range of issues but, due to time constraints, addresses them in less depth. Furthermore, the profile has a lower-level of abstraction than TeLa.

1.2 Semantics in the UML standard

We, of course, attempt to base the semantics of our test descriptions on the official UML semantics of sequence diagrams. However, the semantics of UML diagrams is

only informally sketched in the UML standard. Furthermore there are some consistency problems of a general nature even with this informal semantics.

Two routes to semantics are given informally in the UML 1.4 standard; a “1-stage” route and a “2-stage” route. By the 1-stage route we refer to the outline of the semantics of the actual diagrams (the concrete syntax) given in the UML Notation Guide document, §3 of the standard. By the 2-stage route we refer to the mapping to the metamodel given in the UML Notation Guide document together with the outline of the semantics of this metamodel (the abstract syntax) given in the UML Semantics document, §2 of the standard. The first problem is that these two routes to semantics are not necessarily consistent.

Another problem concerns the fact that the concrete syntax and the mapping to the metamodel would seem to have received less attention than the development of the metamodel itself. While it is generally recognised to be good practice to develop the abstract syntax of a language before developing the concrete syntax, the peculiarities of graphical languages such as UML mean that for such languages, the definition of the abstract syntax cannot be completely decoupled from that of the concrete syntax. The most important peculiarity in this respect is the constraints on the concrete syntax arising from the desire to use established graphical idioms, this being a consideration of prime importance for the usability of the language.

If the metamodel is defined without taking the concrete syntax into account sufficiently, this can lead to the situation in which some of the intended metamodels are impossible to describe using any concrete syntax which respects the established constraints. A case in point is the difficulty in defining a consistent mapping to the UML 1.4 metamodel for UML 1.4 interactions involving multiple threads. We should not lose sight of the fact that the mapping to the metamodel plays an important role in defining the semantics of the graphics and cannot be demoted to the status of an afterthought.

Another problem with the UML documentation is the ambiguities created by incomprehensible text, a problem apparently arising simply from a difficient proof-reading process. Past UML documents have contained phrases that would be serious contenders for the Plain English Campaign’s “Golden Bull Award”[1] or the Campaign’s more recent “Foot in Mouth Award”[2]. It is to be hoped that the upcoming UML 2.0 standard contains fewer candidates. Hopefully, also, all such phrases have been eliminated from this thesis!

1.3 UML 1.4 sequence-diagram semantics

The starting point for the semantics of TeLa is UML 1.4 sequence diagrams. The origins of these diagrams lie in early versions of the MSC notation and in Lamport´s causality graphs [Lam78] (themselves perhaps influenced by Hasse diagrams).

The semantics of these sequence diagrams is described using two relations, predecessor and activator, in a manner similar to the two relations of [Lam86]. In the case of procedural sequence diagrams, the mapping to a metamodel instance is relatively simple. The predecessor describes the ordering relation between outgoing invocations

---

[1] http://www.plainenglish.co.uk/goldenbull.html

[2] http://www.plainenglish.co.uk/footinmouth.html

on an instance and the activator describes the causality relation between an incoming invocation and consequent outgoing invocations made by the invoked method.

However, it is difficult to see how this scheme can be generalised to give a realistic semantics beyond procedural diagrams. The attempt to do so results in contradictory prescriptions of how to deduce activator and predecessor relations from the concrete graphical syntax. Parts of the standard seem to take the point of view that, in the presence of asynchronous messages, only the predecessor relation is to be used, while other parts clearly state that the semantics defined in terms of activator and predecessor is general. As an example of the second point of view, we cite the following quote from the UML Semantics document:

> "Thus, the predecessor's relationship imposes a partial ordering on the messages within a procedure, whereas the activator relationship imposes a tree on the activation of operations."

We now investigate in more detail the main problems with the semantics of UML 1.4 sequence diagrams and discuss possible solutions.

1.3.1 Semantics in terms of two relations between messages

In this section we discuss the difficulties inherent in the UML 1.4 approach to sequence diagram semantics in terms of two relations : predecessor and activator.

1.3.1.1 PREDECESSOR RELATING MESSAGES "WITHIN A PROCEDURE"

The predecessors of a message are defined in the UML Semantics document as "the set of messages that must be completed before the current message may be executed". Assuming that "the messages within a procedure" of the quote in the first part of this section are those that are emitted during that procedure, it is difficult to see how the ordering of asynchronous messages on the sender can be defined in terms of message completion. After all, in the general case, how can the sender know when an asynchronous invocation has "completed" so that the next message can be sent?

1.3.1.2 ASSUMPTION THAT CAUSAL FLOWS ARE DESCRIBED COMPLETELY

If the activator relation is the only way to relate messages emitted by different instances, sequence diagrams are not suitable for describing ordering properties in incomplete causal flows. This is true even for diagrams which do not involve any asynchronous messages.

The fact that the greater part of the ordering relations between messages is lost if all activation relations are not shown drastically limits the use of sequence diagrams at different levels of abstraction. This aspect is crucial for using sequence digrams in the software development process (for modelling use-cases etc.).

As regards testing, the obligation to show complete causal flows is a serious impediment to treating part of the application as a black box in a sequence diagram. This aspect is essential for a functional test language based on sequence diagrams which does not include an MSC gate type construct, see Section 1.4.2.1. The lack of abstraction which this obligation implies is also of crucial importance for representing test objectives, see Chapter 6, Section 3.

1.3.1.3 ORDERING SIGNALS W.R.T. OTHER MESSAGES

In UML, signals are distinguished from asynchronous invocations (asynchronous call actions are distinguished from send actions); unlike asynchronous invocations, they do not activate procedures and so the arrows representing them in sequence diagrams cannot have focus bars attached to the arrowhead end (or, equivalently, they cannot lead to another layer of nesting in the sequence numbering). They therefore terminate causal chains and can only be related to other messages (including those sent or received on the same lifeline as the signal) through a backward causal connection (i.e. a connection passing via activator relations). This is a severe restriction.

1.3.1.4 ORDERING ASYNCHRONOUS INVOCATIONS W.R.T. OTHER MESSAGES

It is not clear from the standard whether the use of focus bars (or, equivalently, nested sequence numbers) is allowed or desirable in the presence of asynchronous messages and/or active objects.

If focus bars are not used, according to the explanation of the mapping to the metamodel, no activator relations can be deduced between different instances. Thus, if the predecessor relation only relates messages "within a procedure" and the activator relation is the only way to relate messages emitted by different instances, without focus bars, invocations made by different instances are not related in any way!

In the literature, many examples of UML sequence diagrams involving only asynchronous messages and with no focus bars can be found, e.g. [ETSI01]. The interpretation seems to involve a total ordering on all the messages or a total ordering on the messages emitted or received on each lifeline. However, this interpretation is not usually made explicit and in fact, there is little to support it in the UML documentation apart from the ambiguous Fig. 3-56 of the standard. Furthermore, it still leaves open the question of how to interpret diagrams involving both synchronous invocations with focus bars and asynchronous invocations / signals without.

Even if focus bars are used, messages emitted spontaneously by active objects are not activated by another message and so the arrows representing them in sequence diagrams cannot be emitted inside a focus bar. They therefore initiate causal chains and can only be related to other messages (including those sent or received on the same lifeline as the spontaneously-emitted message) through a forward causal connection (i.e. a connection passing via activator relations). This is a severe restriction.

1.3.1.5 "SPLIT PERSONALITY" OF SEQUENCE NUMBERING ON ARROWS

The subnotation of the UML 1.4 sequence numbering system defined by the thread names and the explicit predecessors is apparently designed with the idea of relating messages emitted by different instances via the predecessor relation, contradicting other parts of the UML documentation, such as the quote given at the start of this section. In fact, this subnotation would seem to be designed for a semantics in which the predecessor relation is the only relation between messages. This impression is borne out by the example of Fig. 3-71 of [OMG01].

The rest of the sequence numbering notation is based on the two-relation semantics. The problem with this second subnotation, even in diagrams which do not involve asynchronous communication, is that it is only workable for complete causal flows, see above.

Given the fact that the sequence numbering notation is a hybrid of two notations with contradictory semantics, it is not surprising that a number of interpretations of it can be found in the literature, where, for example, it is used for denoting concurrency, branching or a mixture of both.

As an additional point, the sequence numbering system is particularly user-unfriendly and would probably be even more so if it were adapted to properly take into account guards, loops, branching and explicitly-specified concurrency on a lifeline (coregions) in sequence diagrams.

1.3.2 Semantics in terms of partial orders of messages

Formalisations such as that of [Kna99] do not attempt to address the difficulties discussed in the previous section and therefore do not help us in defining our test description language; like many analyses of UML sequence diagrams, they are of little use outside of the procedural diagram case. In looking for a solution, one option would be to explore the correspondence between the message flow graphs of [LadLeu95], with their precedence and communication relations, and UML sequence diagrams, with their precedence and activator relations. The simplest solution, however, is simply to remove the activator relation from the semantics.

We therefore suppose we have a semantics using only the predecessor relation between messages, i.e. in terms of partial orders of messages. Of course, we must allow predecessor relations between messages other than those sent in the same thread of a procedure.

The removal of the problematic activator relation solves most of the problems mentioned in the previous section at a stroke. However, the question immediately arises as to what is to be the role of the focus bar in the absence of an activator relation? This question is addressed in more detail in Chapter 4. Here we merely state that it serves to order messages if it falls inside the scope of an operator for explicitly specifying concurrency on a lifeline and, for synchronous invocations, to relate invocation to reply as well as to indicate that the caller is blocked / "waiting" while the call is treated, where appropriate.

The next question for our predecessor-only semantics is how are predecessor relations between messages to be inferred or, in UML terms, what is the mapping to the UML 1.4 metamodel? For a semantics defined in terms of predecessor alone, we can define three principle options for inferring predecessor relations from lifelines:

- *Minimalist interpretation*: lifelines imply no ordering, only a context and a direction in which predecessor relations between messages received and messages emitted can be imposed explicitly. A means to explicitly order certain messages emitted on the same lifeline may also be required. This interpretation leads to the simplest partially-ordered message semantics. Moreover, with this interpretation we can retain a notion of "message completion" in the definition of the default predecessor relations on a lifeline since, for these relations, the notion is a local one.

- *Maximalist interpretation*: lifelines imply an ordering of all messages emitted or received. A means to explicitly break this ordering in certain cases may be required. This is the most intuitive and user-friendly interpretation and is close to that of the interworkings of [MauWijWin93], though with differences such as the inclusion of synchronous invocations (in the object sense) and corresponding focus bars.

- *Causal interpretation*: lifelines imply an ordering of all messages emitted but not those received; predecessor relations between message received on a lifeline and messages emitted on that lifeline can be imposed explicitly, as for the minimalist interpretation. A means to explicitly order certain messages received on a lifeline, as well as to explicitly break the ordering for certain messages sent on a lifeline, may be required. This interpretation is related to the enforced order of [AluHolPel96], see also [MusPel99] except that we are ordering messages rather than events.

User-friendliness considerations point to the use of the maximalist interpretation which is the most intuitive. Furthermore, it suffers from fewer problems concerning the compatibility of orderings imposed on messages by different instances (though see Section 1.4.3.2).

1.3.3 Relation with the UML 1.4 metamodel

As discussed in Section 3, we need a language with branching and loops. Of course, on the introduction of branching and non-trivial loops, the simple metamodel for UML 1.4 interactions is insufficient. The introduction of loops would require the capacity to explicitly represent infinite-length partial orders. The introduction of loops and branching would require the capacity to explicitly represent infinite numbers of partial orders. Another problem with the interactions part of the UML 1.4 metamodel concerns that fact that stimuli can only be ordered through messages.

If we wished to define our semantics passing via the UML 1.4 metamodel, we would clearly need to extend this metamodel to avoid these problems. However, the interest of doing so is limited since the interactions part of the UML 2.0 metamodel is significantly different and does not suffer from these evident defects. However, the problem with using this latter metamodel is that it is not yet finalised (and no draft was available at the start of this work). We therefore do not pass via a metamodel to obtain our semantics; the semantics could be defined at a later date in terms of the UML 2.0 metamodel when this latter metamodel becomes stable.

1.4 UML 1.4 sequence diagrams for test descriptions

We now study the specific difficulties in using UML sequence diagrams as the basis of a test description language and discuss possible solutions. Our starting point is UML 1.4 sequence diagrams with the corrected semantics as proposed in the preceding section.

1.4.1 Problems with "partial-order of messages" semantics

In the previous section we have shown how many of the problems with the UML 1.4 semantics defined in terms of two different relations between messages can be solved by simply removing one of those relations. Having done so, we have then chosen the most intuitive way of inferring such relations from sequence diagrams which we have called the maximalist interpretation.

In UML 1.4 sequence diagrams, in order to show messages being sent to, or received from, the SUT, it must be represented explicitly by one or more lifelines. However, a sequence diagram which is a black-box test description is to be interpreted as a specification of the test software. Clearly, this software only implements the behaviour of the tester components, and, for each communication with the SUT, only the send/call

or receive actions performed by these test components. Obviously, it does not implement the corresponding receive and send/call actions performed by the SUT, even though these actions are explicitly represented in the diagrams.

For a test description, we must eventually obtain a semantics in terms of tester events only. Deriving this from a semantics as a partial-order of messages will not be easy, in particular, the derivation will require treating messages that involve the SUT differently to messages between tester components. Thus, a semantics of UML 1.4 sequence diagrams in terms of ordering of messages does not seem very suitable for a test description language which requires a semantics in terms of only tester events.

1.4.2 Semantics in terms of events

Three possible ways of achieving a semantics in terms of only tester events are as given below. For comparison, we point out that [ObeKer99] present three types of MSCs describing tests: one with tester lifelines and PCO lifelines, another with a single SUT lifeline and PCO lifelines and a third with a single SUT lifeline communicating with its environment.

1.4.2.1 "MSC-ORIENTED" SOLUTION

Communications with the SUT are modelled as communications between tester components and the tester environment, the connection to this environment being represented using MSC-style gates. An MSC-style event-based semantics is then used so that only tester events are specified. Concerning inter-tester communications, these could be specified either explicitly or by using gates. We consider the first solution clearer and more user-friendly. Non-local causality relations imposed by the SUT behaviour can be added either via MSC-style general orderings between different tester components, which pass via order gates and the tester environment, or by explicitly specifying the tester coordination messages needed to observe them.

1.4.2.2 "TTCN-3 –ORIENTED" SOLUTION

Communications with the SUT are modelled as communications between tester components and their TTCN-3 [BakRudSch01] GFT-style tester ports (a normal port is modelled as a lifeline which is entirely covered by an MSC coregion, "any" port is modelled as an MSC found message). An MSC-style event-based semantics is then used so that, again, only tester events are specified. Concerning inter-tester communications, these could be specified either explicitly or by using connecting ports (and possibly connectors [GraGraRud01]) as in GFT. We consider the first solution clearer and more user-friendly. Non-local causality relations imposed by the SUT behaviour can be added via MSC-style general orderings between the ports of different tester components or by explicitly specifying the tester coordination messages needed to observe them.

1.4.2.3 "UML-ORIENTED" SOLUTION

Communications with the SUT are modelled explicitly using SUT lifelines but the semantics is derived in two stages:

- In the first stage, the semantics of the diagram (in terms of either events or messages) is derived independently of its use as a test description, that is, without paying attention to which lifelines correspond to parts of the tester and which to parts of the SUT,

- In the second stage, the semantics in terms of tester events is derived by projection of the first stage semantics onto the tester components, that is, by reduction of the semantics w.r.t. SUT events.

Thus, though the specification is closed by representing SUT entities explicitly using lifelines, in the final semantics these lifelines only serve to impose extra orderings between the lifelines representing tester entities. As a consequence, different diagrams may have the same semantics, depending on whether or not we explicitly specify the tester coordination messages needed to observe the causal relations imposed by the SUT behaviour.

1.4.3 Semantics via projection onto tester lifelines

1.4.3.1 A TWO-STAGE EVENT-ORIENTED SEMANTICS

For TeLa, we choose the third of the possibilities discussed above for several reasons. Firstly, as already stated, the work reported on in this thesis was done in the context of the COTE project in which the aim was to implement minimal extensions to the syntax used in existing UML 1.4 tools, in particular, the Objecteering tool. The introduction of gates or ports would have been difficult to accomplish satisfactorily with only minimal changes to this syntax. However, there are other, less opportunistic reasons.

A more fundamental reason is that the introduction of an MSC-style gate or a GFT-style port construct introduces greater expressive power but complicates the semantics of TeLa, see [Hél01], for example. On the other hand, the absence of a port or gate construct means that diagrams involving a large number of lifelines cannot be split into several diagrams, and makes it difficult to reuse parts of diagrams in other diagrams. However, we introduce the notion of lifeline decomposition, see Chapter 4, Section 2.8, to help in these cases. With regard to the “TTCN-3 solution”, we take the view that the architectural information concerning tester ports is best presented in a component diagram, rather than in the sequence diagrams. Further, explicit communications between a component and its ports seems to indicate a communication architecture in which queues are associated to ports. We do not want to fix our communication architecture in this way. Again, we prefer to separate this information, specifying it as part of the test architecture.

Finally, we cite two pragmatic reasons which concern the user-friendliness and level of abstraction of a language using lifelines to represent SUT instances as opposed to a language using gates:

- It more clearly represents the relation of the interactions between the SUT and the tester described in the test case, to the interactions between the SUT and its environment described in the SUT specification, notably in the use-case scenarios.
- It more clearly shows the connections between the behaviour of different tester components. In particular, it is generally simpler and more intuitive to represent ordering relations established via message exchange with the SUT by specifying the message exchange which establishes them, rather than by specifying the ordering relations explicitly, or the synchronisations between tester components necessary to observe them.

We note in passing that in the absence of a gate or port construct, the importance of the TTCN-3 and UML 2.0 suspension region construct is reduced.

1.4.3.2 A TWO-STAGE EVENT-BASED SEMANTICS

We adopt a two-stage semantics derivation in which the second stage is a projection of the first and in which the second stage is defined in terms of tester events only. Greater compatibility with UML 1.4 and the UML 1.4 metamodel would be achieved by retaining the solution in terms of partial orders of messages, described in the last section, for the first stage of this derivation. However, an event-based semantics enables more accurate modelling of a larger range of systems than a message-based semantics. In particular, it is more suitable for modelling systems which may involve asynchronous communication and which may be distributed. Some of the communication patterns arising in this context are difficult or impossible to model with a message-based semantics.

A case in point is the difficulty of using an operator for explicitly specifying concurrency on a lifeline, in the style of the MSC coregion, with a message-based semantics. This operator is of particular importance in distributed-system modelling due to the inherent concurrency of such systems. The problem arises from the fact that the ordering is applied to entities that involve two lifelines, i.e. messages, while the cancellation of the default ordering is implemented through an operator whose scope covers a region of a single lifeline. This gives rise to problems in giving meaning to messages (or chains of messages) that are unordered at emission but ordered at reception or vice versa. That is, in the presence of applications of such an explicit concurrency operator, different lifelines may impose contradictory ordering requirements on the same messages. An example of this problem is given in Fig. 3-1, where we use the MSC notation for coregions.

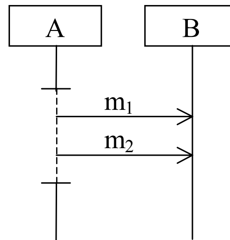


Figure 3-1: Contradictory ordering requirements on messages in a message-based semantics.

Moreover, if our second-stage semantics is to be event based, it would seem reasonable for our first-stage semantics to be also event-based. We therefore choose to use a two-stage semantics, in which:

- the first stage is defined in terms of SUT and tester events
- the second stage is a projection (via $\pi_{semantics}$) of this first-stage semantics onto the lifelines representing the tester and is defined solely in terms of tester events.

For reasons of homogeneity, we also wish to use the same semantic domain at both stages, in which case, the projection operation is algorithmically akin to the hiding of internal events and subsequent τ-reduction of [Jér02] [JarJér02], see Fig. 3-2.

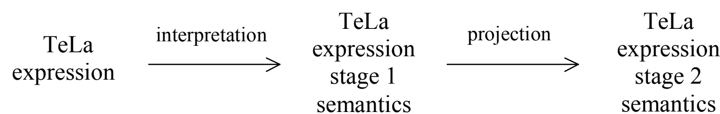


Figure 3-2: A two-stage semantics.

Though the choice of an event-based semantics constitutes another step away from the official semantics of UML 1.4 sequence diagrams, it turns out that it constitutes a step towards the semantics of UML 2.0 sequence diagrams, which is also to be event-based.

1.4.3.3 A TWO-STAGE SEMANTICS AND A TWO-STAGE SYNTAX?

We stated in Section 1.4.2.1 that we consider the solution of using a gate construct represents a more low-level view of a test case and our solution a more high-level user-friendly one. Following up on this remark, it would be of interest to investigate whether the projection of the two-stage semantics could be transposed to the syntactic domain. The syntactic projection could then be a part of the process of deriving the test software from TeLa expressions.

That is, suppose we have defined a mapping of TeLa to our semantic domain and the projection, $\pi_{semantics}$, in our semantic domain. Suppose we then define an extended language, say ν-TeLa, which includes a gate construct, together with a semantic mapping of this extended language to our semantic domain. Can we now define a projection, π_{syntax}, taking TeLa expressions which include SUT lifelines to ν-TeLa expressions which do not, in such as way as to make the diagram of Fig. 3-3 commute?:

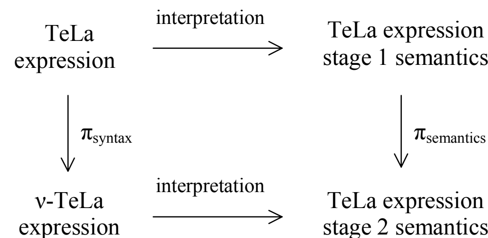


Figure 3-3: Two-stage syntax and two-stage semantics?

The ν-TeLa expressions produced should be close to TTCN-3 and could be produced as part of a translation to TTCN-3.

Since the formalisms that ν-TeLa should be strongly related to, namely the UML 2.0 and the UML Testing Profile are not yet finalised, the definition of the extended language is outside the objectives of this thesis. Though it certainly constitutes an interesting possibility for developing the work presented here, we will not discuss it further, apart from giving a few words of caution.

The notion of projection of HMSCs used in [Hél03] is more general than that used here in that it defines the projection onto subsets of events instead of sets of instances. An HMSC represents a family of partial orders generated from a finite set of motifs, the component bMSCs. [Hél03] studies the question of when the projection of such a family of partial orders can be decomposed as the sequential and parallel composition of a finite set of motifs. It turns out that projection can generate two types of causal motifs, namely unbounded crossings and crowns, which are not so decomposable. [Hél03] gives algorithms for deciding if a projection is likely to generate either of these motifs.

1.4.4 Choice of semantics via projection onto tester lifelines

We have argued for a language in which SUT lifelines can appear explicitly, and for a two-stage event-based semantics in which the second stage is obtained by a projection of the first stage, remaining within the semantic domain, onto a subset of the possible events, namely those occurring on tester components. There are several possible choices

for such a semantics. Which of them is the most suitable depends on the requirements. The natural topology of sequence diagrams, together with the idea of time flow on lifelines and messages, points to a partial-order, and therefore non-interleaving, semantics as the more natural choice, thus conserving the inherent concurrency. However, an interleaving semantics is clearly also possible.

Among our requirements is the need to generalise the controllability notion of centralised tests [Jér02] to the distributed case. The concept of controllability concerns the choices made by the tester and therefore means that the notion of choice is of some importance in the semantics.

In discussing the possible semantics, we consider the different semantics defined for a language with a similar syntax, namely MSCs.

1.4.4.1 INTERLEAVING SEMANTICS

In the interleaving setting, concurrency is not distinguishable from a diamond structure of sequential choices. Among the possible interleaving semantics it is clearly important to consider the official MSC'96 semantics [ITU-T98], in which MSCs are mapped to process algebra expressions, which are themselves given an operational semantics via deduction rules. A set-of-traces semantics can easily be derived from this process algebra semantics. As regards the system–behaviour axis of the classification of [SasNieWin96], the semantics of [ITU-T98] is perhaps more adequately classified as a system model, in which the structure is represented via process algebra expressions.

To ensure that the process algebra expressions are deterministic, uses of the MSC *alt* construct are mapped to uses of a delayed-choice operator rather than to uses of the usual process-algebra choice operator. The delayed-choice operator was introduced in [BaeMau94], where the authors argue that it is the most natural semantics for the MSC alternative construct. In their view, the fact that two branches of an MSC alternative can have a prefix with a common trace is a quirk of the MSC language and the interpretation of this situation as a non-deterministic choice is certainly not what the designer intends. Rather than prohibiting diagrams with an apparently non-deterministic interpretation, therefore, the official semantics determinises their semantics. According to the classification of [SasNieWin96], the official MSC semantics is therefore an interleaving, linear-time, system semantics.

The semantics chosen for UML 2.0 sequence diagrams is likely to be similar to the MSC semantics, though defined in terms of sets of traces and therefore a non-interleaving, linear-time, behavioural semantics according to the classification of [SasNieWin96].

1.4.4.2 NON-INTERLEAVING SEMANTICS

In non-interleaving setting, choice is distinguished from concurrency. The analogue of the interleaving-setting set-of-traces semantics, i.e. languages, is a semantics in terms of pomsets (partially-ordered multisets [Pra86]) i.e. concurrent languages. An example of this type of semantics for MSCs can be found in [KatLam98]. A similar semantics for MSCs in terms of partially-ordered families but passing via process algebra expressions can be found in [Hey98]. The pomset semantics of [KatLam98] is deterministic, in fact, a non-interleaving, linear-time, behavioural semantics according to the classification of [SasNieWin96]. The semantics of [Hey98] may or may not be deterministic and can be classified as a linear-time or branching-time, non-interleaving, semantics, which is behavioural, if we ignore the fact that it passes via process algebra expressions.

In branching-time, concurrent-language semantics, choice points must be inferred whereas in event structure models [Win87], they are clearly manifest in the form of conflicts. In terms of pomsets, if we have determinism[3], a conflict occurs when the composition of a pomset of the semantics with either of two actions gives another pomset of the semantics but composing it with both of them does not. If we do not have determinism, a conflict occurs when the two sets of maximal pomsets that can be obtained by extending the result of composing a given pomset with two actions are different. [HélJarCai02] propose an event structure semantics for MSCs. In the same way that [BaeMau94] argue that the flexibility in positioning choice points is a quirk of the MSC language to be abstracted away from, [HélJarCai02] argue that, in the partial-order setting, preserving non-deterministic choice points can be useful.

Ignoring complexity questions and supposing that all our semantic structures are well-behaved, a partial-order families type of semantics can be obtained from the event-structure semantics of [HélJarCai02] by extracting the set of configurations or prefixes of maximal partial-orders. A linear-time semantics can be obtained by determinising the event structure. Similarly, a set-of-traces semantics can be obtained from an event-structure semantics as the set of linearisations. Again ignoring complexity questions, instead of the corresponding language, we could use the automata that recognises it.

1.4.4.3 MINIMAL DETERMINISM FOR WELL-DEFINED FAIL VERDICTS

Thus, an event-structure semantics is the most distinguishing of these semantics and as already made clear, we will be interested in the event structure which is the result of projecting an event structure semantics such as that of [HélJarCai02] onto the tester lifelines.

In the usual case, we require the projected event structure to be deterministic, that is, no events in concurrency or in minimal conflict can have identical labels, in order to ensure that verdicts are well-defined. This guarantees, trivially, that all non-determinism is resolved before verdict assignment, i.e. in terms of a semantics in the style of the official MSC semantics, that a delayed-choice cannot occur on verdict assignment.

However, it is of interest to know what is the most general possible semantics for test descriptions and therefore to explore to what extent the determinism condition can be relaxed. This may be of interest in a higher-level specification where indefinite choices, i.e. choices in which the criteria for making the choice are not fully specified, may be used.

It is difficult to see any utility in allowing concurrent events labelled by the same action. We therefore reduce the set of models we wish to use to event structures in which no two events have this property. In the absence of a parallel operator, it is not difficult to give syntactic criterion for this property: in TeLa sequence diagram terms, it corresponds to prohibiting the occurrence of two events labelled with the same action inside a coregion.

First, we consider the semantics before projection, under the assumption that concurrent events labelled by the same action are not allowed. If such a semantics conserves SUT choice points (e.g. allowing the situation where two SUT lifeline events labelled with the same action are enabled events for two different alternatives of a choice) it is clearly

[3] In the usual sense for pomsets, not the “denotational determinism” of [Ren96] since all concurrent languages are denotationally deterministic.

too distinguishing, since the tester only has access to the information available at the "SUT component interface". As a reflection of this limitation, with the semantics defined via a projection onto tester components, a minimal conflict between events labelled by the same SUT action is projected onto one between two events labelled by (possibly different) observable actions, i.e. proper tester receptions. That is, SUT non-determinism either doesn´t affect the projected semantics or leads to non-determinism between events labelled by observable actions.

We now consider the semantics after projection, again under the assumption that concurrent events labelled by the same action are not allowed. In such a test description, we can distinguish two types of non-deterministic choices: choices (minimal conflicts) between events labelled by the same observable action and choices (minimal conflicts) between events labelled by the same controllable action. Clearly, the existence of the former prevents us from viewing observable actions as actions over which the tester has no control, that is, actions for which the SUT behaviour completely dictates the tester behaviour.

We define *minimal determinism* (min-det) as follows: a minimal deterministic test description is one in which:

- no two concurrent events are labelled with the same action
- no two events in minimal conflict are labelled with the same observable action

It is defined in terms of the projected semantics since, as mentioned above, the projection itself can introduce non-determinism between events labelled by observable actions.

The main interest in making this definition is that non-determinism between events labelled by controllable actions is included in the property of "controllability", i.e. minimal determinism + controllability $\Rightarrow$ determinism. Moreover, to ensure fail verdicts are well-defined and that we have a basis to discuss the notion of correctness of test cases, we only need to impose minimal determinism together with one additional constraint. That constraint is that any remaining non-determinism, due to a choice between events labelled by the same controllable action, is either resolved on a controllable action or leads to successful termination (implicit pass verdict) or the same explicit verdict on both branches. In terms of the semantics before projection, this says that all such non-determinism that is resolved is resolved by the tester, not by the SUT.

Note that minimal determinism together with this latter condition concerning the resolution of non-deterministic choices does not guarantee that all non-determinism is resolved before verdict assignment, i.e. it does not guarantee that a delayed-choice cannot occur on verdict assignment. Whether it does or not, depends on the use of verdicts in the language. It does guarantee, however, that in any delayed-choice between verdicts, none of the verdicts is a fail verdict, since we assume that a fail verdict can only be obtained on an event labelled by an observable action.

1.4.4.4 A SPECTRUM OF POSSIBLE SEMANTICS

For any of these semantics, as for the partially-ordered message semantics discussed in Section 1.3.2, there are different ways in which ordering relations between events could be inferred on lifelines. The three main possibilities discussed in the previous section when dealing with a partially-ordered message semantics all have their counterparts. Out of these three, we again choose the maximalist interpretation for the same reasons

as we chose it for the partially-ordered messages semantics in the last section, though we note the utility of the causal semantics discussed in [AluHolPel96] and [MusPel99].

In the presence of data, the above semantics have to be extended to cope with guards etc. This is particularly difficult in the partial-order case where dynamic variables must be scoped "architecturally", that is, with respect to components of the test architecture.

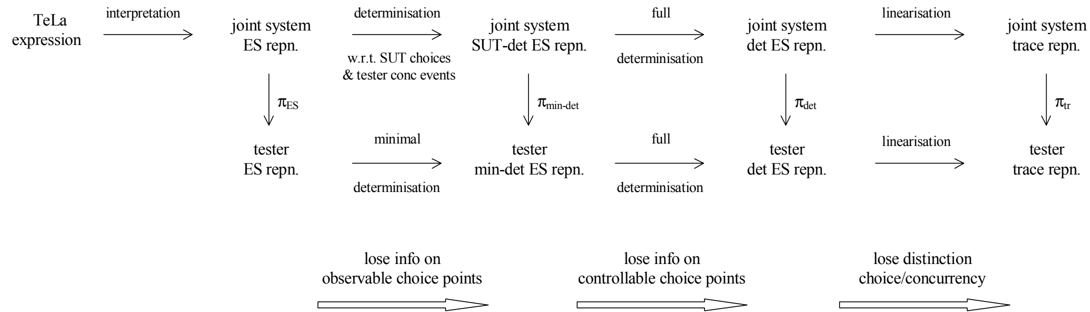


Figure 3-4: Part of the spectrum of possible semantics.

Fig. 3-4 shows the relation between some of the models used to discuss the semantics, from an event structure representation (ES repn.), through a minimally deterministic event structure representation (min-det ES repn.), a deterministic event structure representation (det ES repn.) to a set-of-traces representation. Recall that we are ignoring complexity questions and supposing that all our semantic structures are as well-behaved as necessary. The projection functions $\pi_{min-det}$ and π_{det} and π_{tr} include the removal of any non-determinism w.r.t. observable actions introduced by the projection, see next section.

The structures on the lower line are the projections of the structures described on the upper line onto the tester. On the upper line, the joint SUT-deterministic event-structure representation can be viewed as the determinisation of the joint event-structure representation w.r.t. SUT actions and actions on tester concurrent events (together with the additional condition discussed earlier). Similarly, on the lower line, the tester minimally deterministic event-structure representation is the determinisation of the tester event-structure representation w.r.t. proper observable actions and actions on tester concurrent events (together with the additional condition discussed earlier).

The area of a suitable formal semantics for MSC 2000, the well-established scenario language similar to TeLa, is widely-recognised to be still open. Furthermore, it is not the objective of this thesis to study this complex issue in depth. Apart from the restrictions we have justified so far in this section, therefore, we will take rather a hand-waving approach to semantics and, in particular, will not explore in detail the numerous decidability questions associated with the use of a scenario language such as TeLa. We will not choose any particular semantics but will instead discuss the constructs of TeLa in the context of the above spectrum of tester semantics from minimally-determinised event structures through determinised event structures to traces or automata recognising these traces.

1.4.5 Projection and non-local causality / non-determinism

The projection operation inevitably introduces non-local causality relations. How these relations are realised, e.g. addition of a global controller, via coordination messages etc. will depend on the implementation and in particular, whether or not the implementation

is distributed. The projection operation may also create non-local choices[4], and even non-determinism, between proper receptions. The term proper here refers to the fact that the sender is an SUT component. In a proper tester-emission action, the receiver is an SUT component. A proper tester action is either a proper tester emission or a proper tester reception.

The non-determinism created by the projection will need to be removed or syntactically prohibited. The non-local choices created by the projection operation will need to be resolved in some way before implementation. We allow non-local choices in our test descriptions without specifying how they are to be resolved. However, when making implicit verdicts explicit, non-local choices between proper receptions oblige us to add non-local causality relations, see Chapter 5, Section 2.3.3.2. Again, how these relations are realised will depend on the implementation.

Implementing non-local causality relations by explicitly specifying the tester coordination messages needed to realise them may create MSC race conditions – as they are described in [AluHolPel96] – that are not immediately apparent in the original test description. These race conditions are between receptions of coordination messages and receptions of messages from the SUT. One way of dealing with this problem, proposed in [GraKocSch99], is to suppose that the latency of the tester coordination messages is less than that of other messages, thus conditioning to a certain extent the execution of the test cases. In fact, also needed is the supposition that the time between the sending of a coordination message and its immediate predecessor event is also much smaller than the latency of non-coordination messages.

If the diagram is to be interpreted using the partial order of messages semantics, the race-condition problem does not arise. In fact, this interpretation may obviate the need for many, or all, of the coordination messages, depending also on the number of lifelines used to represent the SUT. This may also be true if the partial order of messages semantics is used for one or several components of the diagram, rather than for the whole diagram, see Chapter 5, Section 3.5. Use of this semantics for the whole tester, for example, is equivalent to supposing that the latency of tester coordination messages is not only less than that of other messages but is also significantly less than the time between any two events on a lifeline.

It should be pointed out that a projection from the semantics of the tester onto individual tester components is a different issue to the projection of a TeLa expression onto the tester lifelines. The former might be done in the context of deriving an operational semantics by synthesizing a set of communicating automata. In the context of a hierarchical component model, synthesis should be parameterisable in terms of which lifelines are grouped into a single automata.

---

[4] A non-local choice is a choice between actions which are not located on the same lifeline.

2 Other issues for our scenario-based test language

2.1 Level of abstraction of TeLa

When using a scenario language such as MSCs or UML sequence diagrams, it is important to be clear about the level of abstraction, in order not to confuse a scenario-based specification language with an execution-trace display language. Viewing a specification language in implementation-level terms can lead to misunderstandings. Having said that, there should, of course, be a clear relation between the traces of the specification and an execution trace.

For example, thinking in terms of the display of execution traces leads to difficulty in understanding the utility of a choice in which the conditions for choosing each branch are not specified, since there are not many circumstances in which one would want to really implement a non-deterministic choice. From a specification-language view, however, this construct is such a simply a choice described at a level of abstraction where the conditions for choosing each branch are not relevant. Such a construct is very useful at specification-level, particularly for the specification of concurrent systems. It may be used in order to leave the details of the choice to a later stage of software development or because it is the environment of the (part of the) system being specified that is to make the choice and the conditions on which it does this are unknown to the (part of the) system being specified. This type of choice is sometimes called non-deterministic. However, this term being confusing, we use the term "indefinite choice", reserving the term "non-deterministic choice" for a choice between identical actions.

Thinking in terms of the display of execution traces also leads to viewing an operator for explicitly specifying concurrency on a lifeline, such as the MSC coregion, as a specification of multi-threading, that is, of implementation-level parallelism. The specification-language view of an explicit concurrency operator, however, is that it merely specifies that the ordering of the elements in its scope is not constrained, a useful construct at specification level. An implementation may be conformant to this specification by implementing any one particular ordering of the possible orderings. It may not have to implement all of them.

The sequence-numbering system of UML 1.4 is also oriented towards execution-trace display. For example, how is it supposed to cope with guards? In the presence of all but the most trivial guards, one can only know the actual sequencing of messages at execution time.

By defining our semantics in terms of partial orders of events, we place our test description language TeLa firmly at the specification level. However, at the same time, we recognise that there is a legitimate demand for some lower-level information in a scenario language used in the object or component setting. In particular, the notion of control flow scheme, to which it is difficult to give any meaning outside of a trace-langauge setting, is of some importance in UML 1.4 but is lost in our partially-ordered event semantics. We therefore attempt to recover this semantic aspect as an extra semantic layer imposing restrictions on allowed linearisations. We do not formalise this extra semantic layer completely but outline the main characteristics in Chapter 4, particularly when discussing the focus bar construct.

Defining a test description language also requires taking a position on other abstraction issues. We allow several constructs to be used in such a way as to leave some of the information necessary for their implementation implicit. For example, test non-local choices, implicit verdicts and certain internal actions and guards all need some implicit synchronizations mechanisms in order to be implemented. Not specifying these synchronization mechanisms allows them to depend on the execution platform. That is, these mechanisms are considered to be part of the information that is added in deriving an executable test case for a specific platform from an abstract test case. Similarly, we do not explicitly specify the exact mechanisms by which a global verdict is derived from a local verdict, we merely attempt to ensure that this derivation is unambiguously defined semantically.

2.2 The internal structure of the tester and the SUT in TeLa

In this section we first present the requirements for the representation of the structure of the two entities involved in the test descriptions, the tester and the SUT. We then see how these requirements can be achieved using UML.

2.2.1 Components in TeLa

2.2.1.1 TESTER LIFELINES

The tester may be composed of several entities in a TeLa test description, even in the case where a centralised implementation is to be derived. This tester internal structure is a part of what has been defined as the test architecture. In TeLa, we require that the tester lifelines represent components from a hierarchical component model. The reason for a hierarchical component model is to define:

- a framework for lifeline composition and decomposition,
- a framework on which to specify component properties that influence the interpretation of TeLa test descriptions, notably the chosen control flow scheme information and the chosen communication semantics (messages or event),
- a flexible framework on which to define local versions of global properties such as controllability,
- a flexible framework on which to base deployment.

2.2.1.2 BASE-LEVEL COMPONENTS

We assume that a subset of the set of components of this hierarchy are *base-level components*. Components that are not base-level components can be decomposed in TeLa sequence diagrams, that is, they can be represented via multiple lifelines in another TeLa sequence diagram.

In an object-oriented system, the component structure is usually defined with the base-level components being the objects. In such cases, each object has a single port so we can unify the port identifier and the object identifier. We do not use tester lifelines representing ports, unless these are also base-level components, due to the problems this entails for defining the value of variables on lifelines and internal actions on lifelines.

2.2.1.3 SIMPLE OBJECT MODEL

A simple object model is treated as a three-level component hierarchy involving a single top-level component and below it, two second-level components, the SUT and the tester. In the simple object case, we assume that the ports on the SUT and tester components are in 1-1 correspondence with the ports on the base-level components they contain, so that the second-level port identifiers can be unified with the base-level port identifiers, themselves unified with the object identifiers.

2.2.1.4 SUT LIFELINES

In Section 1.4.3.1, we explain why we have adopted a language in which SUT lifelines figure explicitly rather than adopting other solutions such as using a construct similar to the MSC gate. However, we have not addressed the question of the relation between representing the SUT via multiple lifelines and the SUT internal structure. In black-box testing, the tester only has access to those aspects of the internal structure and state of the SUT which can be obtained through the "SUT component interface", but exactly what these aspects are needs closer examination.

Communication with the SUT inevitably requires some knowledge of port identifiers. The SUT may therefore be shown in TeLa sequence diagrams using multiple lifelines representing SUT ports. The SUT actions of the TeLa diagram are then represented on the corresponding lifeline. Knowledge of port identifiers may be available to the tester in the initial configuration or may be acquired by the tester during the course of the test. It is statically represented in the structure of the so-called "SUT component interface". In the simple object model, port identifiers coincide with object identifiers so, in a sense, knowledge of SUT ports gives information about SUT internal structure.

If the component structure of the SUT is available as part of its specification, we may instead represent the SUT actions of the TeLa diagram as being located on lifelines which themselves represent components of this component structure, rather than representing SUT ports. This is particularly useful for test descriptions which are derived from sequence diagrams of the design model of the application. As long as the orderings in the projected semantics are the same, any of these representations can be used. Whichever representation is chosen, any ordering involving SUT actions must be the same.

It is important to note that messages transporting invocations between different SUT entities cannot appear in TeLa descriptions, since they are not observable by the tester in black-box testing. Notwithstanding this restriction, as already stated, we allow synchronization messages carrying no data between SUT lifelines. These messages serve the function of the MSC general orderings between instances. In spite of the fact that these messages carry no data values, they are allowed to have parameters containing dynamic variable names. The meaning of this is that they vehicle knowledge about data values. This is done to ensure that the intended value for a dynamic variable used on a tester instance can be a value which has been passed to it from another tester instance via the SUT.

In Chapter 5, Section 1, we sketch a possible formalisation of the above requirements. We only formalise the part of a component model needed to more precisely describe these requirements. In the following section, we look at the relation between these requirements and the modelling capabilities of UML.

2.2.2 Accomplishing TeLa requirements with UML

We now study the compatibility of this view of the tester and the SUT with UML interactions.

In UML 1.4, lifelines represent objects in object-level interaction diagrams and collaboration roles in specification-level interaction diagrams. Collaboration roles can represent any type of classifier playing a role, in particular, components or interfaces. In UML 1.4, a collaboration role cannot be defined without also defining at least one "base classifier" for it[5]. To achieve this, it is sufficient to explicitly define a component or interface for each collaboration role represented by a lifeline.

UML 2.0 introduces the notion of StructuredClassifiers, classifiers having "internal structure", that is, being composed of "parts" linked via instances of Connectors. A classifier with internal structure is said to own its parts. Parts are instances of Properties, Property being a subtype of ConnectableElement, this being a subtype of NamedElement. Connector is a subtype of Association. The notion of internal structure is more flexible (and more well-defined, see [HenBar99]) than the UML composition relation. However, the exact relation of the two in a language that contains both, such as UML 2.0, is still problematic.

UML 2.0 also introduces the notion of Ports. A port is a feature of a classifier that specifies a distinct interaction point between that classifier and its environment or between the classifier and its internal parts. A port encapsulates a set of provided interfaces and/or a set of required interfaces of its classifier. We wish to use the concepts of internal structure, properties, connectors and ports to define a hierarchical component model. We can achieve hierarchy if a part of a component can be typed by another component.

Component is a subtype of Class, this being a subtype of EncapsulatedClassifier, itself a subtype of StructuredClassifier. Thus a component may optionally have internal structure. However, in the most recent version of UML 2.0 specification, it is not clear that a Component is a ConnectableElement. If this is not the case, the parts of a component cannot themselves be components and we cannot define a hierarchical component model.

A collaboration is a StructuredClassifier, and also a BehavioredClassifier, in which the cooperating entities of the collaboration are the parts. Not only the parts of a collaboration but also the connector instances of that collaboration are referred to as "roles". In the case of a collaboration, the type of a part or connector specifies the properties required of an instance in order for it to play the corresponding role (in consequence, the parts of a collaboration/interaction are often typed by interfaces), hence the name Properties.

An interaction is a description of the behaviour of its enclosing classifier, focusing on the passing of information between the ConnectableElements of the Classifier via Messages. In a similar manner to UML 1.4, the parts of a collaboration, or an interaction, can be played by instances of components or interfaces; role playing is

---

[5] This unnecessary and limiting restriction seems to have arisen from a confusion between typing a collaboration role and binding it in an occurrence of the corresponding collaboration. In UML 2.0, a role is typed (as a "part") by a classifier explicitly representing its properties so that the restriction has been removed.

modelled as the binding of roles to classifiers in a collaboration, or an interaction, occurrence.

In the presence of a component model, the TeLa view can be seen to be compatible with UML by viewing the lifelines as roles played by instances of components or ports. In the absence of a component model, lifelines must be viewed as roles played by either objects or by one of the two de facto components: the SUT and the tester. Therefore, in an interaction occurrence – which for TeLa means in an application of a test description – the following is true of the roles of the interaction:

- In the case of the tester:
  - In the simple object model case, either the roles are bound to objects of classes of the test context or there is only one role which is bound to the implicitly-defined component, the tester.
  - Otherwise, the roles are bound to instances of components of the test context in accordance with the test architecture component diagram (in UML 1.4, the components of this diagram must also be the base classifiers of the roles). The largest such component is the whole tester.
- In the case of the SUT:
  - In the simple object model case, either the roles are bound to the objects/port instances of the “SUT component interface” or there is only one role which is bound to the implicitly-defined component, the whole SUT.
  - Otherwise, the roles are bound to the port instances of the SUT component (those of the “SUT component interface”) in accordance with test architecture component diagram. However, the SUT specification may contain a component diagram showing SUT components that are subcomponents of the whole SUT component owning the ports of the “SUT component interface”. If such an SUT component model exists, the roles may instead be bound to the components of this model (the largest of which is the whole SUT).

As already stated, we are interested in test descriptions in the context of a hierarchical component model. As a consequence, a component may be represented by a single lifeline in one diagram and by several lifelines, one for each of its subcomponents, in another.

2.3 Guards in TeLa

In UML 1.4, guards apply to tester emissions. In MSCs, they also apply to internal actions. Contrary to UML 1.4 and MSC, TeLa semantics is given by projection onto tester lifelines. This raises the question of whether, in TeLa, guards can apply to proper tester receptions.

The well-definedness of the implicit alternative, see below, on parallel test cases (see Chapter 4, Sections 2.10.5.1 & 3.4.4.1 for a definition) relies on the fact that the guards used in TeLa activity diagrams (activity diagrams are used to link TeLa sequence diagrams in the way that HMSCs link basic MSCs) have no effect on tester receptions in their scope. Nor do they have any effect on SUT emissions in their scope. On the other hand, guards that are part of the label of messages sent from the SUT to the tester are interpreted as guards on the corresponding tester receptions. However, guards on tester receptions are not treated in the same way to those on tester emissions.

Guards on tester receptions are specific to the particular observable action to which they are associated. Contrary to the situation for events labelled by controllable actions, for events labelled by observable actions, the mutual exclusion of guards and the implicit verdict if guards are not verified, only concern observable actions with the same name, not all observable actions. Similarly, the anonymous variables, see Chapter 4, Section 2.1.1.1, that may be used in controllable action guards are context-dependent and would be ambiguous if not clearly referring to a particular controllable action.

Note that TeLa guarded receptions are not associated with input queue management in the sense that if the guard does not evaluate to true, the message is not consumed and remains at the head of the input queue. This implicit queue management is not easily incorporated into a scenario language (e.g. in the MSC standard, it is stated that message input denotes message consumption rather than message reception). In TeLa, any such queue management must be modelled explicitly via internal actions. Guards other than those which are part of the label of a message sent from the SUT to the tester have no effect on receptions, in spite of the projection semantics.

2.3.1 Guards in MSCs and UML 2.0

By comparison, it is a static requirement in MSCs that guards containing dynamic variables are associated to a single "ready instance" and the intended value of the dynamic variables is that held by the owning instance. Guards which do not contain dynamic variables may be associated to several instances but at least one of these must be a "ready instance". In this case, the message sending is guarded but other instances may also be guarded by the same guard (whatever that means!).

These rules have been taken over by UML 2.0 sequence diagrams. We observe that it is difficult to see how they can be reconciled with lifeline decomposition, which, nevertheless, is allowed in both MSC and UML 2.0 sequence diagrams. For this reason, rules in TeLa are more general but inevitably more complex, see Chapter 4.

As for TeLa, any attempt to introduce SUT instances and a TeLa-style two-stage semantics in MSCs or UML 2.0 would inevitably mean that guards covering SUT instances would have to be disallowed or treated differently.

2.4 Verdicts in TeLa

According to our definition of a test case in Chapter 2, Section 1.4.2.5, it specifies the stimulation of the SUT via the SUT joint interface, the observation of its responses at this interface, and the assignment of a verdict by the tester. How is verdict assignment to be described in TeLa?

The behaviour leading to a pass verdict and that leading to a fail verdict are in some sense complements of each other. In active testing, the behaviour leading to a pass verdict is, in general, the more specific of the two. Since less-specific behaviour is more difficult to specify in a scenario-based notation and would require the introduction of notations such as wildcards on language elements other than message parameter values, in active testing the behaviour leading to a pass verdict is generally more easily specified than the behaviour leading to a fail verdict. Another reason why the description of the behaviour leading to a pass verdict is more easily specified is that it is directly related to the behaviour described in the SUT specification which we suppose is described in an extensional manner.

Thus, a pass verdict can be easily dealt with in an intuitive manner by simply leaving it implicit on reaching the end of a description of correct behaviour, that is on executing a maximal partial order (pass is a universal property, whereas fail is an existential property)

Now we must address the problem of incorporating the description of incorrect behaviour leading to a fail verdict in our test descriptions. It is not obvious when the semantics of such a complement behaviour can be defined satisfactorily in the non-enumerated data, partial-order context, particularly due to the presence of possible non-determinism, indefinite tester choices etc.

A scenario-based description of both correct behaviour and its complement in the same diagram is particularly difficult given the range of expression using multiple test components, MSC-like alternatives, coregions etc. Therefore, though incorrect behaviour leading to a fail verdict must be explicitly treated in the test software implementation, the clarity and tractability of scenario-based test descriptions can be considerably increased by making such behaviour implicit in these descriptions. Furthermore, for the level of abstraction at which we wish to situate our test descriptions, we view implicit fail or inconclusive verdicts as more appropriate. The behaviour leading to an implicit verdict will then be made explicit in a lower-level description derived from such a test description, most likely in the form of an exception mechanism. The TTCN-3 approach using so-called “defaults” can be considered to be such a lower-level approach.

Similarly, we will also want to leave the way verdicts are propagated through the tester to the judge component implicit in TeLa test descriptions

2.4.1 Implicit verdicts in TeLa

In general, TeLa test descriptions describe only the correct behaviour, leaving the behaviour which leads to fail verdicts, and in the non-enumerated data case, inconclusive verdicts, implicit. That is, every test description includes an implicit “otherwise fail” and an implicit “on reaching the end, pass”; in the non-enumerated data case, it may also include an “otherwise inconclusive”.

With regard to the semantics, it is clear that the complement behaviour which leads to an implicit verdict is difficult to define in a non-interleaving semantics. However, it is perhaps less obvious that it is not trivial to define in an interleaving semantics either; it is not simply a matter of using the language complement.

According to the definition of a test case of Chapter 2, Section 1.4.2.5, a fail verdict can only be assigned following the reception of a message from the SUT at the tester. An inconclusive verdict can be assigned following either the reception of a message from the SUT at the tester or following the evaluation of an assertion or guard by the tester.

A (local) fail verdict is implicit if, at any time, a tester receive event which is not one of the possible receive events described by the test description takes place. This includes, for example, a receive event of the correct type by the correct component with incorrect parameter values. The implicit fail verdict is well-defined if the test description is well-formed, that is, if it is minimally deterministic and satisfies the condition concerning the resolution of remaining non-determinism, see Section 1.4.4.3.

In the non-enumerated data case, a (local) inconclusive verdict is implicit if, at any time, one of the concurrent tester guards does not evaluate to true or, in the case of

alternatives, if none of the guards on the different alternatives evaluates to true. For the implicit inconclusive verdict to be well-defined, the test description must not only be well-formed, it must also be essentially controllable, see Chapter 4, Sections 2.10.5.1 & 3.4.4.1, thus defining a parallel test case. Test descriptions which are not essentially controllable are not considered test cases but test descriptions which need refining into one or several test cases, analogous to the test graphs of [Jér02] [JarJér02].

Implicit fail verdicts also figure in the dataless MSCs used as test descriptions in [ObeKer99] but the conditions under which these verdicts are valid are not really explored.

2.4.2 Explicit default alternative in MSC

In MSCs, the "otherwise" alternative denotes an alternative guarded by the conjunction of the negation of the guards associated to the other alternatives (where an unguarded alternative is considered to have the guard `true`). It is not clear in the MSC standard if guarded non-local choices, where the guards involve dynamic variables, are allowed. If they are allowed, the conjunction of the negation of guards has to take into account that the same variable may have different values in different clauses of the conjunction (since different clauses may come from different instances). If they are not allowed, it would result in MSCs which are clearly isomorphic but not all legal. An example is given in Fig. 3-5 where the l.h.s would be legal but the r.h.s. would not.

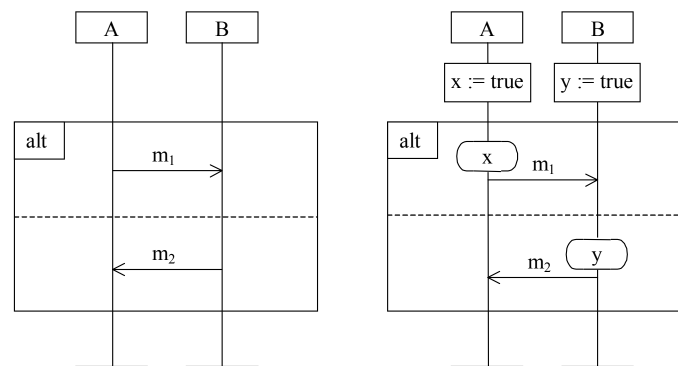


Figure 3-5: Isomorphic MSCs demonstrating the problem with restricting guarded non-local choices.

The MSC default alternative is not adopted as-is in TeLa since it does not distinguish between the effect of guards on observable and controllable actions.

2.4.3 Explicit verdict and explicit default alternative in TeLa

TeLa includes the possibility of specifying explicit verdicts on proper tester emission or proper tester reception events (that is emission to, resp. reception from, the SUT). In addition, the choice construct in TeLa includes the possibility of adding an explicit default alternative, for a restricted class of TeLa test descriptions. The question arises as to the relation between these two constructs, as well as that between these constructs and the TeLa implicit verdict.

The explicit default alternative is a global notion. For this reason, it can only be used on a restricted class of well-formed test descriptions, namely those that are essentially controllable and therefore define parallel test cases. These concepts are defined in Chapter 4, Sections 2.10.5.1 and 3.4.4.1. This restriction is to ensure that each choice has one of two types of default alternative. One type of default alternative is between

proper receptions, denoting a reception which is not among those of the choice, and one type is between proper emissions, denoting the situation where none of the guards of the alternatives of the choice are verified. See Chapter 4, Sections 2.10.3.1 & 3.4.4.2 for more details.

As already stated, the implicit fail verdict is a local notion of verdict that applies to any well-formed test description. As regards the explicit verdicts, the explicit fail verdict can only be placed on proper receptions, while the explicit inconclusive verdict can be placed on both proper tester receptions and, in the non-enumerated data case, on proper tester emissions.

The use of the explicit inconclusive verdict on a proper tester reception does not annul the existence of the implicit fail verdict applying to the same proper tester reception and can be used on any well-formed test description. On the other hand, since a fail verdict is implicit for any unspecified reception, the use of the explicit fail verdict on a proper tester reception involves joint use of the explicit default alternative since, otherwise, it is redundant. As the latter construct can only be used on a restricted class of test description, those that define parallel test cases, the same is true for this type of explicit fail verdict.

Similarly, since an inconclusive verdict is implicit on tester emissions (in parallel test cases) use of the explicit inconclusive verdict on a proper tester emission involves simultaneous use of the explicit default alternative since, otherwise, it is redundant. Thus, like the implicit inconclusive verdict, this type of explicit inconclusive verdict can only be used on a restricted class of test descriptions, those that define parallel test cases.

2.5 Message order as a special case of event order in TeLa

Procedural sequence diagrams describe traditional, centralised OO applications involving single flow of control and traditional method calls. Though the full range of sequence diagrams of the UML standard including asynchronous invocations etc. describes a wider range of applications, the message-based semantics which the UML 1.4 standard attempts to define is more appropriate to a centralised single-flow-of-control view of applications.

We capitalise on this idea by noting that the partially-ordered message semantics could even be used as a characterisation of this centralised aspect in object- or component-based systems. Though we have chosen the more general partial order of events viewpoint, we would like to be able to derive the partial order of messages viewpoint as a special case of the partial order of events viewpoint, in the way that centralised applications can be viewed as a special case of distributed applications. This builds on the work of [AluHolPel96] and [RobKheGro97] who study when an MSC can be considered to be an overspecification w.r.t. a given communication model. Along with [EngMauRen02], the authors also consider interpreting MSCs on architectures involving several communication models.

Normally, the messages treated according to a message-based semantics would not include messages between two components annotated as active, unless some type of rendez-vous communication is implied.

2.5.1 Uses of a partially-ordered message semantics

In many cases, the partial-order of event semantics is too loose and a partial-order of message semantics is sufficient. The mechanisms discussed here make it easy to restrict the semantics in such cases without having to explicitly clutter the diagrams with the coordination messages and local orderings necessary to explicitly implement the message semantics in the event semantics.

The purpose of defining the partially-ordered message semantics inside the partially-ordered event semantics is to enable us to easily choose between the two semantics and even to mix them. Again, being able to denote that a part of a diagram has the former semantics enables us to describe such mixing without having to explicitly clutter the diagrams with coordination messages.

The fact that sequence diagrams represent generalised interaction frameworks, as defined in Chapter 5, derived from a hierarchical component model provides us with the context in which to define the parts of the test description which are to use the partially-ordered message semantics. In a language where components cannot be decomposed such as TTCN-3 GFT, this context is lacking. For simplicity, here we will suppose that this information is provided as an annotation on the diagram describing the initial component snapshot, again as defined in Chapter 5, though the limitation on this semantic mixing that this method supposes may be significant in very dynamic test descriptions.

A restricted-semantics annotation on a component can also be considered as an implementation directive, indicating the advisability of implementing such a component in a centralised manner.

Finally, incorporating the partially-ordered message semantics inside the partially-ordered event semantics provides a bridge between UML 1.4 sequence diagrams and TeLa sequence diagrams / UML 2.0 sequence diagrams for developers already experienced in the former.

2.6 Data in TeLa

In the non-enumerated data case, we suppose that each base-level component has a data environment comprising values of the variables of the test description and that message receptions and assignments have an effect on this data environment. We do not define a complete TeLa data language but merely impose restrictions on what can be considered a suitable data language for TeLa.

One of the most important aspects to using data variables is that, though dynamic variables are global in scope, architecturally speaking, they are local. Though each such variable is owned by a base-level component, this component being the only one that can assign values to it, the value of a given variable held by one lifeline may be different to that held by another lifeline.

When using dynamic variables, the variable name may be prefixed by a base-level component in order to identify the intended value. If the variable is not so prefixed, assumptions are made about the intended value, often involving the assumption that several components share a common view of this value. The value of a variable held on a lifeline which does not represent a supercomponent of the owning component of that

variable is the value received in the last message in which that variable appeared in a parameter expression.

In addition, we allow the specification of assertions and guards that are to be evaluated by components which are not base-level components, even if this may involve implicit synchronizations of the subcomponents. In the presence of lifeline decomposition, identification of the component which is to evaluate a guard or assertion may require prefixing it with the component name.

More details concerning the use of dynamic variables in TeLa can be found in Chapter 4.

2.7 Time in TeLa

Timing notions are of great importance in testing though in some cases, e.g. timer on no response, they can be left implicit in an abstract test case description. For introducing timing notions in TeLa, we favour the use of time constraints, similar to those of UML 1.4 (but more well-defined!) rather than the use of explicit timer operations, similar to those of MSCs. A time constraint is a higher-level concept which can be implemented at a lower level using a timer. However, little work has been done on the use of these time constraints in TeLa and we therefore leave the introduction of such constructs for future work.

3 Extending UML 1.4 sequence diagrams

In this section we give a brief overview of, and justification for, the main modifications and additional constructs introduced in TeLa – w.r.t. UML 1.4 sequence diagrams – many of them being inspired by similar constructs in MSCs. In introducing them, we point out any corresponding constructs in the sequence diagrams of recent drafts of UML 2.0.

3.1 Sequential composition

We require a means of connecting sequence diagrams in order to represent a single test case using several diagrams, each diagram except the initial one being the continuation of the previous one. The sequential composition operator then enables compact representations of test cases involving branching, see below, via a notion of alternative diagram continuations.

The basic type of sequential composition we require is weak sequential composition, which does not imply any synchronisation between components on passing from one diagram to the next. Weak sequential composition is necessary for compositionality, that is, in order for the behaviour to be independent of exactly how we cut the representation into diagrams.

However, we also require a type of strong sequential composition, which does imply such a synchronisation, in order to describe a series of test cases chained together. This is since at the end of each test case a global verdict must be reached which implies a synchronization. In order to describe the sequential execution of separate test cases we must therefore have a means of denoting the synchronization which must take place at the end of every test case.

Note that the synchronization via projection onto tester lifelines means that our strong sequential composition is not truly global but applies to tester components only. No implicit synchronization between tester components and SUT components is implied; all tester-SUT communication must be explicitly described. Note also that it is a specification-language construct, the purpose of which is to facilitate descriptions at higher levels of abstraction; the synchronization so defined may be realised in different ways at the implementation level.

No similar constructs exist in UML 1.4. The UML 2.0 “seq” interactionOperator, inspired by the MSC inline operator of the same name, also implements weak sequential composition. The UML 2.0 “strict” operator could be used to implement strong sequential composition, except for the fact that it implies a synchronization between all lifelines, rather than just tester lifelines. Moreover, the operands of the strict operator would have to be the entire diagrams, making this operator rather cumbersome to use.

3.2 Internal actions

We need an internal action construct, inspired by the MSC construct of the same name, in order to specify events such as assignments and assertion evaluations on tester

lifelines. To facilitate lifeline decomposition we allow assertion internal actions to cross several instances. See Chapter 4 for more details.

No similar construct exists in UML 1.4. In UML 2.0, an "action occurrence" apparently figures in the graphical syntax, e.g. Fig. 8-137. The only explanation given in the text syntax is the following statement under the ExecutionOcurrence construct: "ExecutionOccurrences that refer to atomic actions such as reading attributes of a Signal (conveyed by the message), the Action symbol may be associated with the reception EventOccurrence with a line in order to emphasize that the whole action is associated with only one EventOccurrence (and start and finish associations refer the very same occurrence)." Needless to say that the circumstances in which this construct can be used are not at all clear from this text. Even its use in the case addressed, reception of a signal, is confused.

3.3 Synchronization messages (or virtual messages)

We introduce the notion of synchronization messages as a way of imposing ordering between different instances which does not involve a method invocation. This is a specification-language construct, the purpose of which is to facilitate descriptions at higher levels of abstraction; the orderings so defined may be realised in different ways at the implementation level.

In spite of the black-box testing context, we allow synchronisation messages between SUT lifelines simply as a way of increasing the readability of diagrams. Since the semantics of a test description is in terms of tester events only, as a test description, a diagram with intra-SUT synchronisation messages can always be transformed into an equivalent diagram in which there are no intra-SUT synchronization messages. The transformation will generally involve adding intra-tester synchronization messages.

We allow synchronization messages to carry knowledge of dynamic variable values (or constraints on values, in the symbolic case) between lifelines. This is denoted by using the dynamic variable name as a parameter of the synchronization message.

Currently, we consider that synchronization messages are similar to MSC general orderings and do not contribute events to the semantics. This means that they are not transitive. This policy could be reviewed.

No similar construct exists in UML 1.4. The "general ordering" construct of UML 2.0, inspired by the MSC construct of the same name, when applied between events on different lifelines, is similar to the TeLa synchronisation message construct. However, unlike TeLa synchronization messages, it does not allow the transfer of knowledge about dynamic variable values between lifelines.

3.4 Explicit concurrency

The use of the maximalist interpretation means that we need a construct to explicitly specify concurrency. To break the default ordering on lifelines, we introduce the coregion construct, inspired by the MSC construct of the same name. In the absence of loops, when combined with the local ordering construct (see below), it is as expressive as the explicit predecessors of the UML 1.4 arrow label notation. Unlike this UML 1.4 notation, it is unambiguous in the presence of loops, moreover, it is more user-friendly.

The interaction between the coregion and the focus bar is semantically rich in TeLa, see Chapter 4 for details.

No similar construct exists in UML 1.4. UML 2.0 has also introduced the coregion and, moreover, their syntax turns out to be similar to ours. The interaction between coregions and execution regions is similar to that between coregions and focus bars proposed here except that the impact of the control flow scheme is not treated in UML 2.0.

3.5 Local ordering

We wish to allow arbitrary extra ordering relations to be imposed between events on the same lifeline which fall inside a coregion. We introduce the local ordering construct in order to accomplish this.

No similar construct exists in UML 1.4. The general ordering construct of UML 2.0, inspired by the MSC construct of the same name, is similar to the one we propose here except that it is not restricted to events on the same lifeline. The UML 2.0 construct (like its MSC counterpart) thus covers both the TeLa synchronization message construct and the TeLa local ordering construct. We use two different constructs as we consider that it leads to clearer and thus more easily-readable diagrams, particularly when the possibility of using synchronization messages within the SUT is offered, as in TeLa.

3.6 Loops

We introduce a construct to allow the specification of general iterative behaviour. If a non-ambiguous syntax were defined for multi-object lifelines (this is a big “if”!), our sequence-diagram loop syntax could also be used for sending the same message, possibly with different parameter values, to a set of objects, in a similar way to the UML 1.4 recurrence construct.

The recurrence of UML 1.4 enables the same message to be sent to a set of objects. Concerning the “target” of the call/send action involved, [OMG01] speaks of the action “iterating over a set of target instances”. This obliges the use of multi-object lifelines, though the meaning of emissions and other receptions on such lifelines is ambiguous. Moreover, it is difficult to see how the notation could be clarified, particularly in the presence of asynchronous messages without associated focus bars and in the presence of indefinite choices. In [Kna99] it is stated that the formal semantics defined therein is flexible in this regard. However, this simply sidesteps the issue of finding a non-ambiguous syntax in the general case.

There is also a “presentation option” of UML 1.4 in which “a connected set of arrows may be enclosed and marked as an iteration”. However, its use in the presence of concurrency (e.g. in the case of alternative messages with non mutually exclusive guards) is not clear and the statements made about this in [OMG01] are confused. Furthermore, this construct can only describe limited iterative behaviour since if a finite number of iterations is not specified, there is no way to map it to the UML 1.4 metamodel. This is since the absence of a loop construct at the metamodel level means that loops must be unfolded in order to map them to the UML 1.4 metamodel.

The “loop” construct of UML 2.0, inspired by the MSC inline expression of the same name, is similar to the TeLa activity-diagram loop construct. However, as well as a

guard, MSC and UML 2.0 loops allow a minimum and maximum number of iterations to be specified. In TeLa, we also introduce the user-friendly TeLa sequence-diagram loop construct, which allows the specification of a subset of the loops specifiable with the TeLa activity-diagram loop construct.

3.7 Branching

We introduce a construct, which we will call choice, for specifying alternative behaviours. This construct allows for alternate behaviours which are not fully specified (without such an occurrence denoting concurrency, as in UML 1.4).

As can be observed from the old name of the widely-accepted telecom test description language TTCN: *Tree and Tabular Combined Notation*, in general, a test description will involve branching[6]. In black-box testing, the conditions for taking each of the branches at an SUT branch point may depend on details of the internal state of the SUT not known to the tester. This situation is particularly likely to arise if the SUT contains concurrency and/or distribution. Therefore, to describe the situation where the SUT may exhibit different possible behaviours, a test description language needs a mechanism for describing choices which may be indefinite. We say that a choice is indefinite if the criteria for making it are not fully specified.

In the case where the tester is at the origin of different possible behaviours, all branching should be fully specified (including the guards being mutually exclusive) if the tester is to be controllable, and therefore fully executable. This condition may be relaxed if the indefiniteness of the specification is to be resolved at execution time.

As discussed earlier in this chapter, UML 1.4 attempted to define a semantics based on complete causal flows (though outside of procedural diagrams, the attempt could not be said to have been successful!). It is logical that in such a context all choices are viewed as occurring at message emission. However, for a test language, the obligation to completely specify all causal flows, including those which pass through the SUT, runs contrary to a black-box testing philosophy. In accord with this philosophy and coherent with the projection semantics, we view all the choices of our test descriptions as tester choices. Our test descriptions therefore contain choices other than those between emissions of UML 1.4. Notwithstanding this restriction to tester choices, we may occasionally use a syntax where a sequence-diagram choice operator is shown as being located on an SUT instance simply for the purposes of simplifying the graphical representation.

The branching construct shown in Fig. 3-57 of the UML standard and introduced as a “presentation option” is not suitable for our test description language for the following reasons:

- *Limited expressiveness*: only choices between message emissions in which each emission is guarded are allowed. In the case where the choices are between messages sent from the SUT to the tester, we cannot specify indefinite choices.
- *Incompletely-specified choice ambiguity*: it is not stated that the disjunction of the guards is equivalent to true, yet nothing is said of the meaning in the case where none of them is verified. In fact, the same problem arises with a single guarded

[6] To reflect its expanded aims it is now called the Testing and Test Control Notation

message: it is not clear what the semantics is supposed to be if the guard is not verified. Two possible semantics for such situations are deadlock or continuation without sending the message, but for the latter option, we also have the problem discussed in the next point.

- *Complete causal flow assumption*: if causal flows are not completely specified (which is only guaranteed in the procedural diagram case), representing different alternatives on the same diagram leads to ambiguity since it cannot be discerned which messages belong to which alternative.
- *Branching/concurrency confusion*: for mutually-exclusive guards it denotes alternative behaviours; otherwise, depending on the value of the guards at execution time, it may denote alternative behaviours or it may denote concurrency. A language construct which mixes concurrency and branching is not very useful for specification. Particularly when we consider that with any non-trivial data language it is undecidable whether a choice denotes exlusively choice or sometimes choice and sometimes concurrency, depending on data values.

The "alt" operator of UML 2.0, inspired by the MSC inline expression of the same name, is similar to the TeLa activity-diagram choice construct. However, in TeLa, we also introduce the user-friendly TeLa sequence-diagram choice construct, which allows the specification of a subset of the choices specifiable with the TeLa activity-diagram choice construct.

3.8 Focus bars/suspension regions

In TeLa, as in UML 1.4, the use of focus bars is obligatory for synchronous invocations where they serve to relate request and reply. For asynchronous invocations, focus bars are optional. By comparison, UML 1.4 is rather vague about the use of focus bars with asynchronous invocations.

In TeLa, apart from connecting request to reply in synchronous invocations, the main role of focus bars is to impose ordering if they fall inside the scope of a coregion. In this role, they are thus simply equivalent to a particular set of local ordering relations. Exactly which local ordering relations are denoted by a focus bar and the interaction of this ordering role with the notion of "passive"/"active" object is discussed in Chapter 4.

In TeLa, we do not use MSC-style suspension regions since we prefer to model control-flow scheme constructs using an additional semantic layer. Whether this layer is required or not is to be denoted via annotations on the test architecture component diagram. Hence, in TeLa, whether a synchronous call is blocking or not depends on the control flow annotations associated to the component represented by the emitting lifeline, and those associated to its subcomponents.

The meaning of focus bars (or suspension regions) is not trivial in the presence of lifeline composition/decomposition. In the same way as with guards, see Section 2.3 above, this issue is not addressed in MSC nor in UML 2.0 sequence diagrams. However, in MSC, this lack of treatment is not of any semantic importance since control flow constructs officially have no semantics!

In UML 2.0, focus bars can be placed on lifelines and correspond, at the metamodel level, to execution occurrences. An executionOccurrence is defined by two events, a start event and a finish event. From the current UML 2.0 documentation, it is not clear if

the execution occurrence construct can be used more widely than for focus bars. It is also not clear if focus bars are to be used systematically with synchronous, or even with asynchronous, messages arrows.

3.9 TeLa textual language

3.9.1 Syntax of expressions

In UML descriptions, no specific notation is prescribed for expressions, such as guard expressions, action expressions (on state machine transitions), expressions calculating method parameter values, etc. in order to leave open the possibility of using different expression languages. In MSCs, this idea is made a little less vague by defining a framework in which different languages could be used but which permits some general reasoning about expressions in the language.

Though a specific textual language, based on OCL, was defined for TeLa in the COTE project, in this document we take the same position as UML and MSC, though less precisely defined than in MSC.

3.9.2 Variables

TeLa contains static variables, which parameterise the whole specification, at two levels: at the level of the whole specification, and, in the case of two-tier scenario structures, also at the level of the individual sequence diagrams. TeLa also contains system dynamic variables and component dynamic variables. System dynamic variables are owned by a single base-level component and can be used in the parameters of messages.

3.9.3 Syntax of message-arrow labels

The TeLa notation for message-arrow labels uses the guard notation as in UML 1.4 and the notation for a method name and parameter values as in UML 1.4. As we do not wish to use the sequence numbering notation for the reasons given earlier, the TeLa notation does not use the dot notation (for activation), the slash notation (for explicit predecessors), the thread names notation (for concurrency), or the recurrence notation (for simple iterations), nor does it use the return value notation (for showing the value returned by an synchronous call on the invocation message arrow).

In TeLa, a syntax for showing the value returned by an invocation on the return message arrow is needed. A special notation is also needed to distinguish the case where an exceptional value is returned. In addition, a syntax is needed for showing the value returned in parameters of the invocation which are of the *out* or *inout* kind on the return message arrow. Finally, a special syntax is used to indicate the sending of an unknown value by the SUT to the tester, and a special syntax is used to indicate the assignment of such an unknown value to a dynamic variable.

Where lifelines do not represent the lowest level of components in the component hierarchy (usually objects), as we will see later, the labels of the communication events must include information about the ports involved in the communication. Similarly, the labels of internal action events must include information about the owning component. For communication events, the required information can be obtained by using a syntax

for the message arrow labels that includes the name of the originating and target ports of the message. For proper reception events (i.e. those for which the sender is the SUT), it may often be the case that only the receiver port information is available.

In particular, this extended message arrow label syntax is needed in order to guarantee an adequate representation of the output of the Umlaut/TGV tool in the TeLa language. The only representation of this output that can be guaranteed in all cases is one with only two lifelines: the SUT and the tester.

3.9.4 Syntax of internal action contents

A syntax is also required for use with the internal action construct we have introduced. An assertion internal action contains a boolean expression. An assignment internal action contains a series of assignments either each on a new line or separated by commas. A creation internal action contains the name of the creator and created subcomponents and the creation parameters. An escape internal action contains a procedure call or the assignment of a function value to a dynamic variable, for which the procedure/function is external. Such “trap doors” should satisfy certain semantic restrictions in order for the semantics of the language not to be perturbed by them.

4 UML 2.0 sequence diagrams

In Sections 1.3 and 1.4, we discussed at length the suitability of UML 1.4 sequence diagrams for describing tests. Though based on UML 1.4 sequence diagrams, the test description language we have defined borrows many constructs from MSC. Since UML 2.0 sequence diagrams also borrow many constructs from MSC, on introducing the constructs of TeLa which are novel w.r.t. UML 1.4 sequence diagrams, in Section 2, we also related them to UML 2.0 sequence diagrams.

The starting point for our test language was UML 1.4 since UML 2.0 was very immature at the time this work began. However, now that UML 2.0 is almost ready for release and since the recently released UML Testing Profile is based on UML 2.0, in the light of our work, we briefly discuss here the semantics of UML 2.0 sequence diagrams.

4.1 Semantics via an abstraction/conformance relation?

As for UML 1.4, the abstract syntax, or metamodel, of a UML 2.0 sequence diagram is called an interaction. According to the UML 2.0 specification, the semantics of a single UML 2.0 interaction (not that of a pair of such interactions!) is stated to be a set of "valid traces" and a set of "invalid traces". It is stated explicitly that the union of valid traces and invalid traces does not necessarily constitute the trace universe. Clearly, then, a notion of trace universe also plays a part in the semantics, though exactly what part, is unclear! It seems reasonable to assume that the (semantic counterparts of the) actions of the interaction are contained in the set of atomic actions used to construct the traces of the trace universe.

The description of the semantics given in [U2P03] apparently betrays a confusion between the following two possibilities for the set of traces that an interaction is a denotation of:

- a set of traces constructed from the actions represented in that interaction (we will use the term "explicit-model" semantics)
- a set of traces constructed from actions of some unspecified universe of actions, that contains the actions represented in the interaction, where the traces represent executions of the interaction (we will use the term "set-of-possible-models" semantics)

The latter view of semantics is illustrated by the *neg*, *ignore*, *consider* and *assert* interaction operators and the *state invariant*. In the definition of these constructs, but not in that of the other constructs of the language, interactions are apparently viewed as properties that any actual execution must or may satisfy. The *ignore*, *consider* and *assert* operators, in particular, affect how the selection of the traces from the nebulous trace universe is performed.

The "set-of-possible-models" semantics is similar to the way MSCs are used in, for example, [Ek93], [AlgLejHug93], [ComPicRen95]. It is also similar to the way MSCs are used to describe test objectives in [GraHogNah93] and [SchEkGra98][7]. In the third

[7] The test objectives of [Jer02], [PicJarTra02] and Chapter 5 of this document are more general since they may involve actions of the specification which do not appear in the test, i.e. the relation between the test objective and the test is more than just an abstraction relation.

article cited, properties are divided into positive properties, i.e. properties that any selected trace must have, and negative properties, properties that any selected trace must not have. The test objective scenarios of [PicJarTra02] and of Chapter 6 of this document are similarly divided into positive and negative scenarios. In such a scheme, the negation of a positive property, or positive selection criterion, gives a negative property, or negative selection criterion. Ideas of this nature could be at the origin of the *neg* operator.

However, in all the cited articles, the universe in which the intended/selected traces reside is clear, that is, the set of intended models of the property is restricted. For example, in [Ek93], [AlgLejHug93] the denoted traces belong to the set of traces of an SDL specification and in [PicJarTra02] the selected traces belong to the set of traces of a UML specification. In the UML 2.0 sequence-diagram case, the set of intended models, i.e. the universe in which the selected traces reside, is not discussed.

An example of a scenario language for which a "set-of-possible-models" semantics is used without restricting the set of intended models, that is, without discussing the universe in which the selected traces reside, is the "live sequence charts" of [DamHar01]. Though the semantics of these charts is actually given in terms of finite state machines, it could have been given in terms of selected execution traces. There is some similarity between the contentious UML 2.0 constructs and the constructs of these charts, notably the "live sequence chart" constructs enabling parts of a chart to be labelled as optional or mandatory. There are also significant differences, such as the fact that the authors of [DamHar01] use this mechanism to encode control structures such as alternatives, unlike in MSC and UML 2.0 sequence diagrams, where these control structures are given as operators. However, the semantics of "live sequence charts" is given in terms of a single finite state machine and if the semantics were given in terms of traces, it would be in terms of a single set of traces: those selected.

Therefore, if a "set-of-possible-models" semantics is required and the "valid traces" and the "invalid traces" are selected execution traces, why is the set of "invalid traces" not simply the complement of the set of "valid traces"? Clearly, it is not intended to be the case, so that there must be execution traces which are neither "valid" nor "invalid". What is the meaning of such traces?

The definition of the *neg* operator is consistent with interactions being properties, but with the "valid traces" and "invalid traces" being simply the trace expressions of the properties, not the execution traces selected by these properties. If this is the case, the semantics is not really defined as properties. Instead, it is an "explicit-model" semantics which is then used (as both positive and negative selection criteria) to select execution traces, in a similar way to [Ek93], [AlgLejHug93], [ComPicRen95] or Chapter 6 of this document. There is then some abstraction relation between an interaction trace and the execution traces it selects.

This is not without its problems since using a language as complex as UML 2.0 sequence diagrams (which imports all the richness of MSC 2000 and adds a few more operators, for good measure!) as a property language is likely to run into difficulties, particularly concerning compositionality.

However, before we can conclude that the semantics of UML 2.0 sequence diagrams is an "explicit-model" semantics, and that the "valid traces" and "invalid traces" are simply the trace expression of the properties, we note that the definition of the operators *ignore*, *consider* and *assert* is inconsistent with this view. The definition of these operators makes clear that the "valid traces" and "invalid traces" are execution traces

according to a "set-of-possible-models" semantics. These operators appear to have been influenced by the interactive method of defining test objectives in the TestComposer and Autolink tools, see [SchEbnGra00] for example. Thus, we cannot avoid addressing the problem of giving meaning to invalid execution traces as well as to execution traces that are neither valid. This in a "set of possible models" semantics for which the universe of actions is apparently any universe which contains the actions of the interaction.

Perhaps the intention is that the semantics contain within it the notion of conformance relation and the idea of demonstrating conformance. In testing involving a centralised tester and enumerated data, it is usually said that an inconclusive verdict must be relative to a test objective. In this thesis, we use inconclusive verdicts for testing involving a possibly distributed tester and non-enumerated data, without this being relative to a test objective. Perhaps the execution traces which are neither "valid" nor "invalid" are supposed to correspond to traces which would produce an inconclusive verdict. That is, it cannot be demonstrated that such traces are conformant and it cannot be demonstrated that they are not conformant. However, a conformance relation is normally specified as a relation between two well-defined semantic models. Without such a relation, it is difficult to see how anything can be "demonstrated"!

Even without taking into account the problems with the *ignore, consider* and *assert* constructs, it is difficult to judge if the pair-of-trace-sets semantics is viable, since the rules for deriving new pairs of trace sets from combined trace sets are not given. For example, if (a,b) represents a set of valid and a set of invalid traces, $\bullet$ denotes sequential composition and . trace concatenation, is it the case that $(\{a\},\{b\})\bullet(\{c\},\{d\}) = (\{a.c\}, \{b\}\cup\{a.d\})$?

At this point, we conclude that in the absence of further explanation on the part of the authors of the UML 2.0 interactions document, further analysis degenerates into pure speculation. However, in our opinion, due to the addition of the operators cited above, the semantics of the proposed language cannot be defined coherently.

4.1.1 Separating semantics from abstraction/conformance relation

Putting aside the issue of whether the current semantics can be made consistent, there seems to be little justification for combining the notion of semantics of an interaction and that of an abstraction or conformance relation w.r.t. to execution traces. On the other hand, there is every reason to separate these two concepts. Even if there is a need to mix syntactic constructs which reflect the two viewpoints, which is certainly debatable, a more sensible approach would be to define the semantics in two stages, as follows.

First, define an unambiguous semantics in terms of the set of traces composed exclusively from the actions represented in the interaction, without the constructs that concern an abstraction/conformance relation. This is already, in itself, a difficult enterprise. Second, define a semantics for interactions which involve the additional constructs according to an abstraction/conformance relation between the set of traces defined by the first stage semantics and the traces which are considered to represent an execution of the interaction in some other universe. Separating the issues of semantics and abstraction/conformance relation also allows flexibility in choosing the exact abstraction/conformance relation required, without compromising the semantics of the basic constructs.

This two-stage approach would still require strong well-formedness conditions to ensure that the use of the constructs reflecting the abstraction-relation / conformance-relation view is restricted to situations where they have a clear meaning. See the following sections for examples of situations where, with the current definitions, they do not. As defined currently, these constructs are problematic.

We now take a closer look at the contentious constructs: those whose semantic definition apparently incorporates some abstraction or conformance relation.

4.2 Abstraction/conformance relation constructs

4.2.1 The *neg* interaction operator

Invalid traces can be defined explicitly using the *neg* interaction operator.

The semantics of this operator is clearly intended to be an operator turning valid traces into invalid traces and vice versa. However, it is far from clear how it is to be composed with the other operators of the language. For example, what is the meaning of a state invariant, a duration constraint or an *assert* operator in the scope of a *neg* operator?

As stated above concerning the separation of the concept of semantics of an interaction and that of an abstraction/conformance relation, if the context requires the specification of invalid traces, with a clear semantics of interactions in terms of traces, this can be achieved by the designer simply labelling some complete interactions as specifying invalid traces rather than valid ones. This is the case for the existential, universal, exact and negated interpretations of MSCs in [Kru00], for example. The incorporation of the concept of negation of a behaviour into the language itself as an operator, without defining a proper compositional semantics, only serves to create serious confusion.

4.2.2 The state invariant

The state invariant defines a constraint on a lifeline. If the constraint is true, the trace is a valid trace and if it is false, the trace is an invalid trace.

As for the *neg* operator below, the use of a parallel operator with one operand involving a state invariant does not have an obvious meaning. In fact, with any non-trivial data language, it is likely to be undecidable at compile time whether such a combined fragment denotes an interleaving of valid traces or an interleaving of valid and invalid traces. The latter would appear to be meaningless.

4.2.3 The *ignore*/*consider* and *assert* interaction operators

Though not stated in [U2P03], one assumes that an *ignore* instruction for a message type has priority over a *consider* instruction for the same message type in the surrounding scope and vice versa.

The *ignore* interaction operator is used to define a set of message types that can occur between those specified in valid executions. Apparently, then, the ignore operator defines new actions of the semantic universe as well as an abstraction/conformance relation, according to which valid executions can contain these actions in any position. Apparently, the set of actions explicitly specified via messages plus the set of actions specified in an *ignore* clause does not define the total set of actions. If a valid trace can

contain other actions which are not in the alphabet of the interaction, is there a need for an *ignore* operator, apart from to cancel the effect of a *consider* or an *assert* operator? If a valid trace cannot contain other such actions, is there a need for an *assert* operator?

The *consider* interaction operator is used to define the set of message types that should not be ignored. From Fig. 8-156 of [U2P03], it would seem that message types not appearing in the interaction can be “considered” using the *consider* operator. How are actions involving such a message type to be considered dealt with? That is, if a trace involves the sending and receiving of a message of this type in the scope of such a *consider* expression, is it invalid? If so, why is there a need to use an *assert* operator in Fig. 8-156? If not, in what way is such a message type to be “considered”? The mystery deepens with the explanation on page 396, according to which messages to be “considered” are handled in some manner by the running system but ignored by the specification!

The meaning of the *assert* interaction operator is that the trace of the operand of the assertion is the only valid trace. Again, this “meaning” is far from clear. Is this not already covered by having no operator or by using the *consider* operator? What is the meaning of a *neg* interaction operator in the scope of an *assert*? What about an *ignore* in the scope of an *assert*?

What is the meaning of an expression “ignoring” a message type *a* in parallel with an expression involving message type *a*, or with an expression that “considers” message type *a*, or with an expression that asserts the exchange of a message of type *a*? What is the meaning of a *neg* or *ignore* expression in the scope of an *assert* operator?

The abstraction/conformance relation is not at all clear from these operators. Is the default to ignore actions not explicitly specified, that is, can a valid trace have any actions between those explicitly specified as long as they are not from the alphabet of those explicitly specified? If so, why is there a need for an *ignore* operator? Or is the default to consider actions that are not explicitly specified? If so, why is there a need for a *consider* or an *assert* interaction operator? Needless to say that a significant amount of confusion surrounds the definition of these operators.

Finally, what is the meaning of a combined fragment whose interaction operator is *ignore*, *consider* or *assert* in parallel with another fragment? For example, what is the meaning of an expression “ignoring” a message type *a* in parallel with an expression involving message type *a*, or with an expression that “considers” message type *a*, or with an expression that asserts the exchange of a message of type *a*?

The introduction of the *ignore*, *consider* and *assert* operators creates even more difficulties than that of the *neg* operator.

4.3 Other problems with UML 2.0 sequence diagrams

4.3.1 Scope of interaction fragments

The scope of interaction fragments does not seem to be limited in any way, e.g. Fig 8-134 shows such a scope which crosses a message. The question arises as to the meaning of, for example, a loop combined-fragment whose scope includes the reception of a message but not its emission (or vice versa). Can a simple set of syntactic restrictions be defined?

4.3.2 The *strict* and *critical region* constructs

A *strict* interaction operator designates that the combined fragment represents a strict sequencing between the behaviours of the operands. The *critical region* interaction operator designates that the traces of the operand cannot be interleaved with any other occurrences. In both these cases, there is an ambiguity concerning lifeline decomposition, which may lead to more messages becoming explicit.

The meaning of an interaction in which an *ignore* interaction operator, a *neg* interaction operator or a state invariant occurs in the scope of a *strict* operator or a *critical region* operator is also far from clear. What about such a combined fragment whose scope includes a *critical region* or a use of the *strict* interaction operator? For example, what is the meaning of a *strict* expression or a *critical region* in parallel with an expression containing some of the same messages?

4.3.3 Meaning of sequence numbering

The sequence numbering system has not been abandoned in spite of the problems that its use supposes in the presence of loops, choices, coregions, guards etc., and in spite of the impossibility of equivalence between interaction diagrams and collaboration diagrams with superimposed interaction (or communication diagrams as they are now called) with the sequence numbering system in its current form.

4.4 Concluding remarks on UML 2.0 sequence diagrams

Part of language design should involve ensuring that all, or at least most, expressions that can be written in the language have an obvious meaning. This aspect does not seem to have been given sufficient importance in UML 2.0 sequence diagrams. While many of the problems of UML 1.4 sequence diagrams have been solved by basing the language on MSC, the UML 2.0 constructs which are new with respect to MSC introduce a plethora of new ambiguities, opening up a veritable Pandora’s box of impenetrable expressions, and, in so doing, seriously obfuscate the semantics.

Chapter IV : A scenario-based test description language for component testing (TeLa)

In this chapter we present the TeLa language and give an informal presentation of its event-based semantics. This is not done in the style of a user manual or a reference manual but more in the style of a tutorial.

We do not intend this chapter to be a complete language design, rather a detailed discussion of the issues involved in defining a scenario-based test language. Most of the constructs proposed, particularly for the one-tier scenario structures, were responses to concrete needs and desires expressed in the COTE project. In general, we have not questioned the basis of these needs and have assumed them to be well-founded. Instead, we analyse the proposed constructs to clearly show the restrictions necessary for them to be well-defined. In a more complete language design, we would perhaps eliminate some of these constructs if the restrictions are seen to be too constraining. Similarly, we sometimes offer several alternatives to specify the same behaviour. In a more complete language design, some of these alternatives would be eliminated. However, we consider that the decision as to which constructs should remain in a more polished language is a decision which should be taken after more consultation with tool vendors and users.

The choice of concrete syntax was also heavily influenced by the objectives of the COTE project. One of the principal objectives was for this syntax to be as close as possible to UML 1.4 syntax, as implemented in the Objecteering UML tool. Thus, we do not claim that concrete syntax for local operators, such as the TeLa sequence-diagram loop operator and TeLa sequence-diagram choice operator, is the most adequate.

Regarding the use of the language, only part of the language presented in this chapter was actually used to describe tests in the COTE project.

In the first section we present an overview of TeLa. In Section 2 we present the constructs used in TeLa sequence diagrams. In Section 3 we present the constructs used in TeLa activity diagrams.

1 Overview of TeLa language

1.1 One-tier or two-tier scenario structures

The full TeLa language is a two-tier language in which TeLa activity diagrams – based on UML activity diagrams – can be used to link TeLa sequence diagrams – based on UML sequence diagrams – much in the way that HMSCs can be used to link MSCs. We prefer this syntax to using an equivalent of MSC in-line operators in spite of the fact that it is the latter which is presented as the main syntax in UML 2.0 sequence diagrams. The main reason for this is that it leads to a simpler-to-use sublanguage which guarantees fundamental technical properties, notably concurrent controllability, see Section 2.10.5.1. Another reason is that the fact that inside the scope of an in-line operator, lifeline ordering is not respected (due to the vertical sequencing of different operands) can be confusing.

In accordance with our aim of progressive presentation of the required language constructs, we first define a language based exclusively on UML sequence diagrams before showing the need for the extra tier based on UML activity diagrams.

Such a progressive presentation also introduces the two tiers of the language in order of importance in the sense that, for most applications, the sublanguage defined by the one-tier part of TeLa is sufficient. This sublanguage is designed to be simpler and more intuitive than the full language but as a consequence it is less expressive. In this respect, the relation between the full two-tier language and the one-tier sublanguage is different to the relation between HMSCs and MSCs with inline expressions. In the latter case, as for UML 2.0 sequence diagrams and UML 2.0 interaction graphs, the two tiers have the same expressiveness[1]. In TeLa, the main complexity contained in MSCs is absent from the one-tier sublanguage. As this complexity is not always apparent to the user, in our view, this constitutes an advantage of the TeLa approach over the MSC/UML2.0 approach. The discussion of Section 3.4.4 concerning the non-trivial restrictions that must be imposed on a two-tier test description in order for it to define a test case throws some light on this complexity.

The main feature of TeLa which leads to the one-tier sublanguage being simpler is the locally-defined (that is, on a single lifeline) loop and choice constructs. Locally-defined operators were seen as user-friendly and were demanded by the CASE-tool vendor in the COTE project. The use of these one-tier constructs is restricted in order to guarantee that they can be rewritten in terms of two-tier constructs, without the need to introduce a parallel operator in the two-tier diagrams. The fact that they can be rewritten then means that, without loss of generality, we can restrict the sequence diagrams linked via TeLa activity diagrams to TeLa elementary sequence diagrams, where a TeLa elementary sequence diagram is defined as a TeLa sequence diagram containing no locally-defined loops and choices.

[1] For the sublanguages with no guards, more types of loops can be described with MSCs than with HMSCs.

1.2 Limited parallelism

MSC, TTCN-3 GFT and UML2.0 constructs which are significant by their absence in TeLa are an explicit parallel operator (though, or course, parallelism exists between different lifelines of a diagram) and the notion of sequence diagram gates/ports. The reason for this absence is that, in the presence of loops, their introduction would significantly complicate the semantics.

In the absence of loops and gates, if two expressions to be placed in parallel involve two sets of components which together comprise part of an interaction framework of the component model (roughly speaking, they are disjoint, no component of one is a subcomponent of a component of the other), use of a parallel operator simply denotes a type of union. In this case, the expression involving a parallel operator is equivalent to one which does not involve a parallel operator. Again in the absence of loops and gates, if the two expressions involve two sets of components which do not have the above disjointness property, a test description involving a use of the parallel operator is equivalent to one involving only the use of coregions and local orderings.

In our estimation, most types of test currently used can be specified without a parallel operator and gates. This certainly applied to the tests of the COTE project. Nevertheless, as a consequence of the lack of these constructs, it is difficult to use TeLa for describing some types of distributed testing scenarios, such as those involving concurrent choices. If the demand for these constructs becomes more evident, they could be added to the current language, but any such addition should be done while taking care with regard to the semantic implications. In any case, in our opinion, an analytic approach to a sequence-diagram based test language first involves clarifying the most important and most well-defined constructs before extending the language to include more complex constructs.

1.3 Other significant omissions from the language

Like MSCs, there is no multi-cast to an arbitrary number of receivers in TeLa. The multi-cast construct of UML 1.4 sequence diagrams (involving the “recurrence” and the “multi-object”) gives rise to serious ambiguities concerning the meaning of communications shown as originating or terminating in multi-objects. In TeLa, we do not wish to introduce constructs which are clearly ill-defined. It is possible that in the context of a hierarchical component model, multi-cast messages could be unambiguously defined, c.f. the assumptions about the transmission of knowledge about system dynamic variable values of Section 2.2.4.5. However, do not study this issue here.

Again as for MSCs and UML 2.0 sequence diagrams, by the nature of sequence diagrams, it is not possible to specify the creation of an arbitrary number of components. Again, the use of an underlying hierarchical component model and a creation internal action may permit unbounded component creation to be specified in certain cases. However, we do not explore this issue further here. As in MSCs and UML 2.0 sequence diagrams, the number of components to be created is fixed at compile time.

Finally, we reiterate that, as for MSCs, the event denoted by a message arrow touching a lifeline is the consumption of the corresponding message. If the receiver has a queue, the event denoted is not the arrival of a message in this queue. We further suppose that there is no access to any input queue without consumption. Guards on messages received from the SUT

at the tester are assertions on parameter values. If no such assertion evaluates to true, a fail verdict results; the message cannot be left in some input queue.

1.4 The underlying component model

We suppose that underlying a TeLa test description is a specification of a test description component model and an initial snapshot of this component model, see Chapter 5, Section 1. If none is specified, the simple object model is assumed and the test context is completely determined by the initial lifelines, along with the identification of which lifelines belong to the tester and which to the SUT.

A component model describes a structure of components owning ports which are connected via connectors. Usually, the smallest type of component, a base-level component, is an object. Base-level components have a single port and we therefore identify them with this port. Events are labelled by actions which refer to ports. We call the base-level component that owns the originating port of an invocation the emitting component. We call the (not necessarily base-level) component that owns the target port of an invocation the receiving component. Since ports are owned by components, events can also be viewed as being owned by components.

The simple object model is a special case in which there are two levels of hierarchy (apart from the top-level component which includes all others). These are the whole tester and whole SUT component level, and the base-level (the objects/ports)

The existence of an underlying hierarchical component model provides a framework in which to define the notion of composition/decomposition of lifelines and in which to define component properties which influence the interpretation of TeLa test descriptions. We stipulate that the same level of decomposition (the same interaction framework) must be used throughout a one-tier or two-tier scenario structure.

The notion of decomposition is analogous to that of MSCs or to the part decomposition of UML2.0 and is a feature which does not exist in the TTCN-3 GFT and is not really treated in the UML Test Profile currently under standardisation.

Currently, we suppose that any component properties are denoted via annotations on the initial snapshot diagram defining the test context, even though this supposition is rather limiting. The two properties discussed here are the control flow scheme (the property of being active or passive) and the property of internally using a message-oriented semantics or an event-oriented semantics. We assume, for simplicity, that all dynamically-created components are passive and that all dynamically-created components use the semantics of their supercomponent.

The first property is that of the control flow scheme. Control flow scheme restrictions add an extra layer of semantics which we will refer to as implementation directives. If we choose to add this layer of semantics, it imposes extra well-formedness conditions on TeLa sequence diagrams, notably concerning the use of focus bars. Passiveness is understood in terms of implementation directives concerning:

- the implementation of concurrency using scheduling rather than execution threads
- the implementation of synchronous calls as blocking the sender
- the interpretation of focus bars as representing method body executions

Base-level components are passive unless there is an annotation to the effect that they are active. Other components are active if they contain an active subcomponent and passive otherwise. More details can be found in the sections concerning the TeLa constructs whose interpretation is affected by this property, in particular, the focus bar.

The second property is that of communication semantics (events or messages). This concerns the ordering relations which are to be inferred between the events located on the lifeline or lifelines that represent a given component. Components use the event-oriented semantics unless there is an annotation to the effect that they use the message-oriented semantics. More details can be found in Chapter 3, Section 2.5 and Chapter 5, Section 3.

1.5 Verdicts

Semantically, verdicts are annotations to events (in the non-interleaving semantics)[2] and are therefore local. In general, TeLa verdicts are implicit and are derived as follows:

- if the behaviour completes as shown, the verdict is pass,
- if an unspecified reception from the SUT occurs, the verdict is fail,
- in the non-enumerated data case, if one of the concurrent guards of the tester does not evaluate to true, the (local) verdict is inconclusive; similarly, if none of the alternative guards of the tester evaluate to true.

All implicit verdicts are local, however, for pass and fail verdicts, the passage from local to global is simple. As fail is an existential notion, a single local fail implies a global fail. As pass is a universal notion, a local pass on all concurrent branches implies a global pass, this is simply the result of reaching the end of the behaviour without any other verdict having been derived.

However, the semantics of the inconclusive verdict is more delicate. The meaning is that the component which derives the verdict performs no more actions while the other components of the interaction framework continue as far as possible to see if they can derive a fail verdict. See Chapter 5, Section 2.1.3.2 for more details. The process of communicating the local verdicts to the entity that is responsible for the global verdict is not explicitly modelled in TeLa.

We also allow a limited use of explicit verdicts, for the situation in which this is more convenient. The use of any explicit verdict other than an inconclusive verdict on reception of message from the SUT involves restrictions on the test description in order for it to be well-defined.

The existence of implicit verdicts means that in TeLa we do not have the notion of defaults, as in TTCN. Using implicit verdicts instead of defaults allows a more abstract view of test behaviour and is also more flexible, e.g. it can accommodate the notion of decomposition of lifelines.

---

[2] In an interleaving semantics, they would be annotations to transitions, not special states.

1.6 TeLa textual language / data language

In the COTE project [JarPic01], a textual language based on OCL (Object Constraint Language, a language which complements the UML and is defined in the same UML standard [OMG03], [OMG03]) was defined for TeLa. In this document, we do not define such a language. We merely discuss the essential elements it must contain and the restrictions which would need to be imposed on it in order for it to be used as the TeLa textual language. Clearly, it must be able to refer to objects of the model.

1.7 TeLa test descriptions

A TeLa test description consists of a set of specification-level parameters or static variable declarations and a parameterised TeLa scenario structure. A scenario structure can be one-tier or two-tier, in which case we also describe the test description as one-tier or two-tier.

A two-tier scenario structure comprises a TeLa activity diagram in which each node contains either a set of sequence-diagram level parameter (static-variable) declarations and a TeLa diagram reference to an elementary sequence diagram or a symbol indicating the empty sequence diagram. The empty sequence diagram is a sequence diagram with no events and is simply used to keep the notation compatible with the UML activity diagram notation.

A one-tier scenario structure comprises a set of TeLa sequence diagrams linked via the TeLa diagram referencing mechanism.

A TeLa diagram reference is a construct in which a sequence diagram is referenced by name; clearly, any sequence diagram which is to be referenced must be named. TeLa diagram references are allowed in two different contexts in TeLa sequence diagrams, see Section 2.3, and one in TeLa activity diagrams, see Section 3.2.

2 TeLa sequence diagrams and 1-tier scenario structures

In the previous chapter, we gave an overview of TeLa test descriptions, and hence TeLa sequence diagrams, in terms of partial orders of events or messages, we discussed what these diagrams represent in terms of testing and clarified the main differences between TeLa sequence diagrams and UML 1.4 sequence diagrams. In this section, we introduce the syntax of TeLa sequence diagrams and give an informal description of the intended semantics. In discussing the semantics, we sometimes need to discuss both the semantics before and after projection. The question of the semantics of TeLa is discussed in more detail in Chapter 5.

A TeLa sequence diagram may be named (in order to be referenced) and may be accompanied by a set of sequence-diagram level parameter declarations, as stated at the end of the previous section.

2.1 Basic constituents of TeLa sequence diagrams

2.1.1 Variables

2.1.1.1 INTRODUCTION

TeLa is a statically-typed language. There are three kinds of typed variables in TeLa:

- *Static variables*, which parameterise whole TeLa activity diagrams or whole TeLa sequence diagrams and whose value is fixed for the entire activity or sequence diagram on instantiation of that diagram. Static variables can be declared at two levels as described above.
- *Dynamic variables*, whose value can be changed in the course of a TeLa test description by an assignment. Each dynamic variable is owned by a particular base-level component. There are two kinds of dynamic variable:
  - *system dynamic variables*, whose scope is global to all components.
  - *component dynamic variables* which are of type integer and cannot be used in message parameters thereby restricting their architectural scope to a single component; a typical use of such a variable is as a loop counter.

  Dynamic variables are considered to be declared on first use (the type of system dynamic variables is known from the corresponding class diagrams).
- *Anonymous variables*, which refer to the parameters of a message from the SUT to the tester. They occur either in a guard on such a message or in an internal action attached to the arrowhead of such a message. The scope of these variables is a single message so that the same anonymous variables can be reused for different messages.

Among the types, we need a type which is a supertype of all the component types used in the underlying component model and a type which is a supertype of all the port types used in the underlying component model. We can then use variables and constants of these types to model the communication of component and port identifiers, notably in the case of dynamically-created components.

2.1.1.2 SYNTAX

For anonymous variables, we use the syntax $_{1}$, $_{2}$ etc. where the subscripts refer to the position in the parameter list.

2.1.1.3 WELL-FORMEDNESS CONDITIONS

System dynamic variables must be owned by a tester base-level component. No system dynamic variables can be owned by an SUT component. Reading and writing of SUT public attributes by the tester should be modelled as synchronous get and set operations.

Component dynamic variables cannot be used in message parameters; they can only be used in internal actions on a lifeline representing the component that owns them or a supercomponent of it.

Anonymous variables can only refer to the parameters of a message received by the tester from the SUT.

2.1.1.4 INFORMAL SEMANTICS

Different components do not necessarily share the same knowledge of the value of a given system dynamic variable.

Though static variables also contribute, it is the need for anonymous variables, or of proper-reception assigned variables, see later, that makes it impossible to avoid a symbolic treatment of data. This reflects the fact that a test description must often deal with the situation in which the SUT can send any one of an infinite set of values and the tester behaviour may depend on which value is sent.

2.1.2 Message arrows

2.1.2.1 INTRODUCTION

As in UML 1.4 and UML 2.0, in TeLa there are three types of message arrow: a continuous-line unfilled arrow used to represent an *asynchronous message* (invocation or signal), a continuous-line filled arrow used to represent a *synchronous invocation message* and a discontinuous-line unfilled arrow used to represent both a *synchronous return message* and a *component-creation message*.

We treat a component-creation message as a particular type of synchronous invocation, the corresponding return message being the sending of the created component identifier to the creator component (as in the UML 1.4 sequence diagrams of the Objecteering UML CASE tool).

The syntax of the arrow labels is discussed in Section 2.2. Messages between tester ports or between SUT ports are known as *coordination messages* (tester coordination messages or SUT coordination messages respectively). Asynchronous coordination messages with the message name “sync”, see the section on message arrow labels, are called *synchronization messages*.

2.1.2.2 SYNTAX

Fig. 4.1 shows the three different types of message arrow.

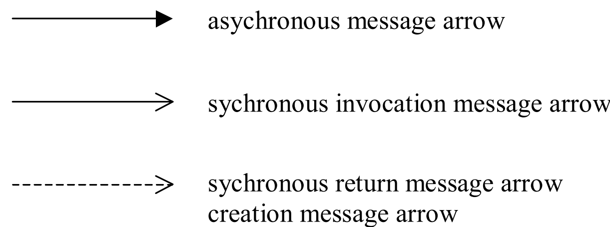


Figure 4-1: The three different types of message arrow

2.1.2.3 WELL-FORMEDNESS CONDITIONS

A synchronous invocation arrow must be associated to a synchronous return arrow in the opposite direction.

The only types of SUT coordination messages allowed are synchronization messages.

2.1.2.4 INFORMAL SEMANTICS

All message arrows except synchronization and creation messages denotes two events, the send event and the corresponding receive event, and an ordering relation between them. If the message is exchanged between lifelines representing subcomponents of a component in which the message semantics applies, further ordering relations are inferred, see Chapter 3, Section 2.5 and Chapter 5, Section 3. The events are labelled by the direction of the event (send or receive), by the name of the message and by the originating port (necessarily a port of a base-level component) and the target port. They may also be labelled by a guard and by a set of associated assignments.

As in MSCs, the receive event of a message, for all messages except synchronization messages and creation messages, denotes message consumption; if the diagram is implemented on a communication architecture where the message is queued after reception, the reception and queueing occur before the TeLa denoted receive event.

Creation messages denote a single event. Synchronization messages do not denote any events, their meaning is similar to that of the MSC general ordering construct, see Section 2.2.4.4 for more details.

Unguarded synchronization messages with no parameters play the role of non-local MSC general orderings, that is, general orderings between events on different lifelines. They denote ordering relations between the predecessors of the virtual event corresponding to the sending of the synchronization message, and the successors of the virtual event corresponding to the reception of the synchronization message. For the semantics of guarded synchronization messages or synchronization messages with parameters, see Section 2.2.

2.1.3 Lifelines

2.1.3.1 INTRODUCTION

As in UML 1.4, in TeLa, arrows are shown linking vertical lines, or lifelines. There are two types of lifelines, tester lifelines, which represent tester components, and SUT lifelines, which represent either SUT components or SUT ports.

In the case where they represent tester or SUT components, the name given to the lifeline can be the full name or the abbreviated name. The full name is the name of a component followed by a colon followed by the name of the component type of the component. The whole tester

and whole SUT component are exceptions to this rule in that they are simply labelled by the name tester and SUT respectively. In the case where a lifeline represents a component, let this component be called the lifeline component.

In the case where lifelines represent SUT ports, the name given to the lifeline is the name of an SUT port. In the case where a lifeline represents a port, let this port be called the lifeline port.

In the simple object model, there are two levels of components, the whole SUT and whole tester components, and the base-level components or objects. In the simple object model, a tester lifeline represents either the whole tester component or it represents a tester object (= tester port). Similarly, in the simple object model, an SUT lifeline represents either the whole SUT component or it represents an SUT object (= SUT port).

We do not distinguish graphically lifelines representing passive and active components since the property of passiveness or activeness is viewed as an additional semantic layer which we may, or may not, wish to add.

2.1.3.2 SYNTAX

Each SUT lifeline is capped by a rectangle with the name of the component/port appearing inside this rectangle; each tester lifeline is capped by an actor symbol with the name of the component appearing below this symbol. Hence, the two types of lifeline are distinguished by the use of an actor symbol or a component symbol rather than by the use of a stereotype as in [UTP03], see Fig. 4.2.

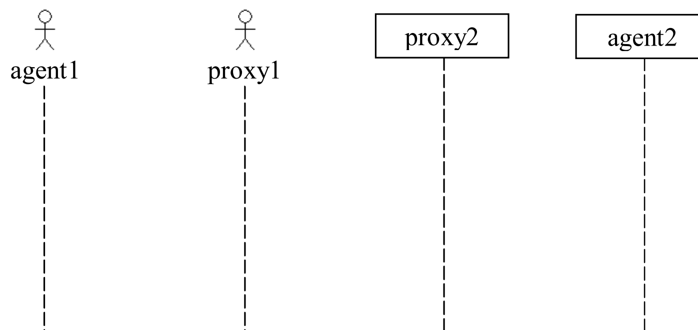


Figure 4-2: Tester and SUT lifelines.

2.1.3.3 WELL-FORMEDNESS CONDITIONS

The events located on a lifeline representing a component are owned by that component or by one of its subcomponents. The events located on a lifeline representing an SUT port refer correctly to that port (for emissions it is the originating port, for receptions, the target port). This is part of what is defined as structural consistency, see Chapter 5, Section 1.6.

The SUT must be modelled as a set of components or as a set of ports but not a mixture of the two.

Internal actions, see Section 2.4, are not allowed on lifelines representing SUT ports. The only type of internal action that is allowed on lifelines representing SUT components are creation actions.

2.1.3.4 INFORMAL SEMANTICS

The semantics of a lifeline is that it imposes a total ordering on the events in its scope unless they are contained in a coregion. A coregion specifies a lack of ordering relations between the events located on the lifeline component which are inside its scope, see Section 2.5.

2.2 Message arrow labels

2.2.1 Introduction

The constituents of a label on a message arrow are as follows:

- *A guard*, i.e. a boolean expression involving static and dynamic variables. In the case of a guard on a message emitted by the tester, the boolean expression may be preceded by a component name and a colon. Guards are optional. Notice that their meaning is different on message received by the tester from the SUT to other messages, see below.
- *The name of the base-level port which is the originator of the message and of the port which is the target of the message*. In the simple object model, target ports are also base-level ports (objects). Originator and target port names are obligatory on all tester coordination messages and on messages from the tester to the SUT, unless they can be deduced from the context, see below. As regards messages from the SUT to the tester, on some systems, only the target port can be specified. Thus for messages from the SUT to the tester, depending on the type of system being modelled, either one or two ports must be specified, unless they can be deduced from the context. If SUT lifelines represent components, port information may, or may not, be specified for intra-SUT messages; if it is specified, it serves only to restrict the allowable lifeline decompositions.
- *The name of the message*. That is, the operation (or signal) name, for invocations (or signals), the operation name followed by a colon, for return messages, the word “create”, for creation messages, and the word “synch”, for synchronization messages. A message name is obligatory on all messages.
- *The message parameters*: message parameters are optional though they must be coherent with the formal parameters of the operation or signal, for invocations, with the formal creation parameters for creation messages, and with the return value and the out or in/out parameters, for synchronous return messages. In all cases except for synchronous return messages, the syntax used for the message parameters is a list of expressions separated by commas.

2.2.1.1 OMITTING THE PORT INFORMATION

The originating, resp. target, port can be deduced from the context for those message arrows that originate, resp. terminate, in a lifeline which represents a base-level component. The case where SUT lifelines represent ports, rather than components, is a special case. Hence, in a diagram in which all lifelines represent base-level components (objects/ports), all port information can be omitted from the messages.

The target port for synchronous invocation return messages can also be deduced from the context since it must be the same as the originating port of the invocation message. Hence, in the case of synchronous invocation return messages from the SUT to the tester in systems where the originating port cannot be specified, no port is specified. In other systems, only the originating port is specified on synchronous invocation return messages.

2.2.1.2 COORDINATION MESSAGES

Messages between two tester components or between two SUT components are called *coordination messages*. Coordination messages with the message name "sync" are called *synchronization messages*. Synchronization messages are simply used to impose orderings between events on different lifelines in a way that is similar to non-local MSC general orderings.

We allow both intra-tester and intra-SUT synchronization messages, even if this leads to redundancy, in order for the user to be able to choose the simplest representation for each case.

2.2.1.3 OWNING COMPONENT OF A GUARD

In the case of a guard on a message emitted by the tester, if the boolean expression of the guard is preceded by a component name and a colon, this component is the owning component of the guard. If the boolean expression of the guard is not preceded by a component name and a colon, the lifeline component is the owning component of the guard. In a simple object model, all guards are owned by a base-level component (object).

In the case of a guard on a message received by the tester from the SUT, the owning component of the guard is the owning component of the target port of the message.

Under the projection semantics, a guard on an SUT synchronization message leads to a guard, or a clause of a guard, on one or more receptions from the SUT for which the owning component is defined as above.

2.2.1.4 SYSTEM DYNAMIC VARIABLE VALUES

Recall that each system dynamic variable is owned by a base-level component but a value of a system dynamic variable can be held by any base-level component. Thus, we must ensure that there is no ambiguity in the intended value of such a variable at any use of it.

A system dynamic variable used in a guard may be preceded by a base-level component name followed by a dot. If this is the case, the intended value of the variable is that held by the specified base-level component. If this is not the case, the intended value is as explained in the well-formedness conditions below.

The intended value of a system dynamic variable used in a message parameter is that held by the owning component of the originating port. If this port information is not available as may be the case for messages received from the SUT or intra-SUT messages, the intended value is that held by the emitting lifeline component, for tester coordination messages, and the receiving lifeline component for messages from the SUT to the tester.

2.2.2 Syntax

Some examples are given in Fig. 4-3 and Fig. 4-4.

In Fig 4-3, SUT lifelines represent components. We assume that we know from the associated component model that the two proxies are base-level components so that it is not necessary to use two port names in messages exchanged with them. *user* is the originating port of the synchronous invocation *find*. *findit* and *add_log* are asynchronous invocations. *peers*, *location* and *y* are system dynamic variables.

In Fig. 4-4, The diagram on the r.h.s is closer to the implementation and demonstrates that the implementation of ordering relations introduced by the SUT may introduce race conditions

between tester coordination messages and messages from the SUT. In [GrabKochSchHog99], it is proposed that this be resolved by supposing that tester coordination messages have lower latency.

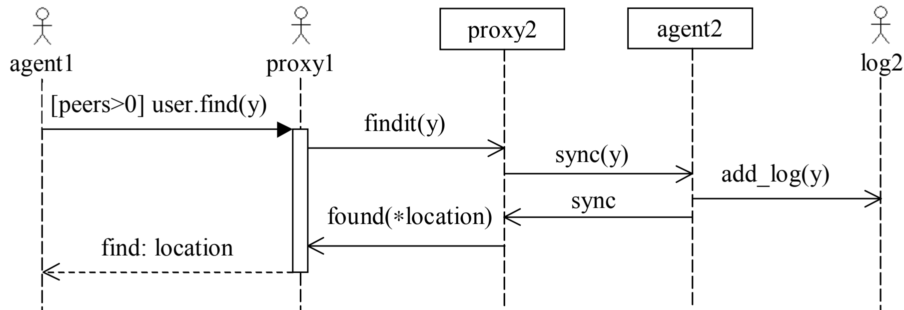


Figure 4-3: A TeLa sequence diagram showing the use of dynamic variables, guards, ports, message parameters and proper-reception assigned variables.

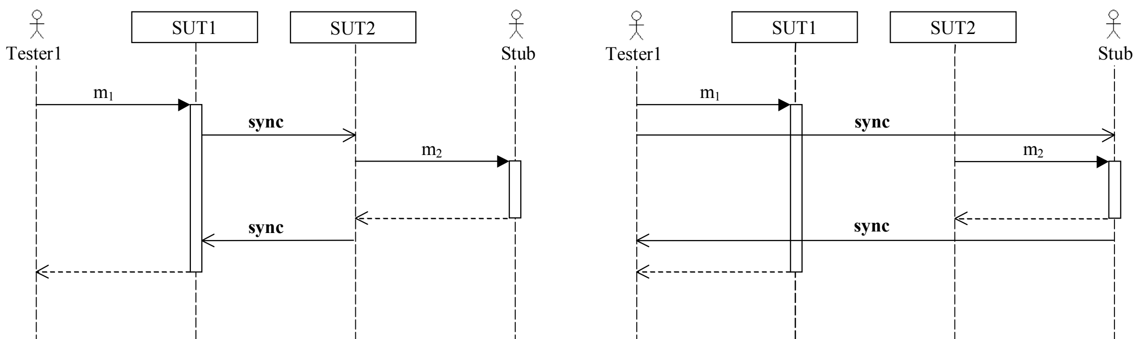


Figure 4-4: TeLa diagram showing two equivalent ways of using synchronization messages (intra-SUT and intra-tester messages respectively) to specify that the tester expects to receive invocation m_2 from SUT2 after it has invoked m_1 on SUT1.

2.2.2.1 "OUT" AND "IN/OUT" RETURN VALUES IN SYNCHRONOUS INVOCATIONS

Synchronous return messages must also take into account the values returned in "out" or "in/out" parameters. In the syntax used, the expressions for the out and in/out parameter values appear as a list separated by commas inside parentheses and the return value precedes these values, outside the parentheses. The order of appearance of the "out" or "in/out" return values is the order of appearance of the corresponding parameters.

Hence, if there are no "out" or "in/out" parameters, the return value appears without parenthesis immediately after the message name and the colon. If there are "out" or "in/out" parameters, these appear after the return value inside parentheses and separated by commas, e.g. *get_last* : *last* (*top*, 4), where *get_last* is the name of the synchronous invocation, *last* and *top* are variables.

2.2.2.2 ARBITRARY VALUES RETURNED BY THE SUT

The expressions used in the message parameters of messages sent from the SUT to the tester extend the expressions which can be used in the message parameters of other messages with two new constructs as follows:

- The * construct: this syntax denotes the transmission of an unknown value.
- The *x construct, where x is a dynamic variable owned by the component represented by the lifeline at the receiving end of the message or a subcomponent of it: this syntax

denotes an unknown value and the assignment of this unknown value to *x*. We will refer to *x* as a *proper-reception assigned variable*.

- In the case where there are “out” or “in/out” parameters, if on a synchronous invocation return, after the colon, there is a single wildcard, it applies to both the return value and the “out” / “in/out” return values. Otherwise the wildcard can be applied specifically to one of these parameters, e.g. *get_last*: ∗ (4, ∗).

The use of the ∗ construct is reserved for messages sent from the SUT to the tester; if we wish to describe the tester sending an arbitrary value to the SUT, a static variable should be used for this purpose. The ∗*x* construct is an alternative to using the ∗ construct and an assignment internal action, see below.

2.2.2.3 EXCEPTIONS

We also need a special notation for the case where a synchronous invocation returns an exception in place of a return value. For this, we replace the colon with a sharp sign, e.g. *get_last# *error_type*

2.2.3 Well-formedness conditions

2.2.3.1 PORT INFORMATION

The originating port of any message emitted on a lifeline must be owned by the component represented by that lifeline or by a subcomponent of it. The target port of any message received on a lifeline must be owned by the component represented by that lifeline or by a subcomponent of it.

If specified explicitly rather than being left implicit, the target port of a synchronous invocation return message must be the originating port of the invocation message.

2.2.3.2 COMPATIBILITY WITH TEST CONTEXT AND COMPONENT MODEL

The names of the messages must be coherent (type-compatible) with the structure of the ports in the underlying component model, that is, the interfaces of which they are composed. The name of an asynchronous message must be the name of an operation required at the originating port and offered at the target port or of a signal produced at the originating port and accepted at the target port. The name of a synchronous message must be the name of an operation required at the originating port and offered at the target port. The name of a return message must be the name of an operation required at the target port and offered at the originating port followed by a colon etc.

2.2.3.3 SYNCHRONIZATION MESSAGES

Synchronization messages are considered to be type compatible with any port. Any expression used in the parameter of a synchronization message must comprise a single dynamic variable. The only SUT coordination messages allowed are synchronization messages.

2.2.3.4 GUARDS

The owning component of a guard must be a subcomponent of the lifeline component.

The owning component of a guard on a message emitted from the tester must be a supercomponent of the base-level component owning the originating port of that message.

If a system dynamic variable used in a guard is preceded by a base-level component name and a dot, the identified component must be a subcomponent of the component owning the guard. If a dynamic variable name used in an assertion or in a creation expression is not preceded by a component name, all base-level subcomponents of the component owning the guard must share a common view of its value.

2.2.4 Informal semantics

2.2.4.1 MESSAGE NAMES AND PARAMETERS

The exchange of a message between two components is interpreted as either a synchronous invocation, an asynchronous invocation or signal exchange, a synchronous invocation return, or a component creation. The message name is interpreted as the name of this invocation or signal, for the first two types of message, and the name of the corresponding invocation for the third type. For the last type, the standard message name “create” is used. The parameters are interpreted as the parameters of the invocation or signal for the first two types, and as the return values together with the values of the *out* and *inout* parameters for the synchronous invocation returns. The parameters of the creation message are interpreted as the parameters passed to the created component on creation.

2.2.4.2 GUARDS ON TESTER EMISSIONS

For a guarded message emitted by the tester the guard of the message arrow label is interpreted as a guard on the corresponding emission action, that is, semantically, it is part of the corresponding controllable event. Part of the semantics of guards on tester emissions is the implicit inconclusive verdict, as follows. In the following, we suppose that an unguarded message is guarded by the condition *true*.

If the guarded message is not one of several alternative messages that can be sent by the tester and the guard is not verified, an inconclusive verdict results. This inconclusive verdict is local to the component owning the guard.

If the guarded message is one of several alternative messages that can be sent by the tester and none of the guards of these alternatives are verified, an inconclusive verdict results. This inconclusive verdict is local to the component owning the choice.

In both cases, the semantics is the same as that obtained if the guard is placed in an assert internal action occurring immediately before the emission action, see below.

Alternative message emissions by the tester arise with the use of the TeLa sequence-diagram choice construct introduced in Section 2.10. However, more general alternative behaviours can arise with the use of the activity-diagram choice construct in two-tier scenarios. The semantics of guarded messages in the case of this more general alternative is defined in Section 3.4.

2.2.4.3 GUARDS ON PROPER TESTER RECEPTIONS

If any message received by the tester from the SUT has a data expression as the value of one of its parameters, semantically this in fact denotes a boolean expression that is to be conjoined to any existing guard. This boolean expression equates the anonymous variable corresponding to that parameter to the expression given as the parameter value.

For a guarded message received by the tester from the SUT, the guard of the message arrow label is interpreted as a guard on the corresponding tester reception action, not on the corresponding SUT emission action. That is, the guard is part of the corresponding observable

event. Moreover, we assume that the guard constrains only the anonymous variables. Part of the semantics of guards on proper tester receptions is the implicit fail verdict, as follows. In the following, we suppose that an unguarded message is guarded by the condition *true*.

If the guarded message is not one of several alternative messages that can be received by the tester, then if a message with the same name and signature as that of the message arrow label in question is received and the guard is not verified, a fail verdict results.

If the guarded message is one of several alternative messages that can be received by the tester, then if a message with the same name and signature as that of the message arrow label in question is received, and none of the guards on any of the alternative messages having that message name and signature are verified, a fail verdict results.

In both cases, the semantics is the same as that obtained if the guard is placed in an assert internal action attached to the arrowhead of the incoming message, see below.

Alternative message receptions by the tester arise with the use of the TeLa sequence diagram choice construct introduced in Section 2.10. However, more general alternative behaviours can arise with the use of the activity diagram choice construct in two-tier scenarios. The semantics of guarded messages in the case of this more general alternative is defined in Section 3.4.

2.2.4.4 GUARDS ON SYNCHRONIZATION MESSAGES

Synchronization messages define ordering relations between the predecessor(s) of the virtual send event and the successor(s) of the virtual receive event. If they have parameters, they also denote the communication of knowledge about dynamic variable values, see below.

Synchronization messages may be guarded. For intra-tester synchronization messages, these guards have the same meaning as those on other types of message emitted by the tester. For intra-SUT synchronizations, through the projection semantics, these guards are in fact guards on tester reception events.

2.2.4.5 MESSAGE PARAMETERS AND VALUES OF SYSTEM DYNAMIC VARIABLES

Message arrows between lifelines serve to pass knowledge of system dynamic variable values between the components represented by lifelines, or rather between the base-level components which are subcomponents of them. If such a variable is used in a message parameter expression, knowledge of its value is considered to be transmitted by the communication.

Suppose we have a use of a system dynamic variable, e.g. in a guard. We define the intended value of the variable in this use to be that assigned to it in the assignment internal action that is the “most recent predecessor” of the event in question. The “most recent predecessor” is the one that is reached via the most recently received communication (recursively).

When a message is sent to a port on a component that is not a base-level component, it is assumed that all the knowledge of system dynamic variable values contained in the communication is transmitted to all the base-level subcomponents of this component. This policy could perhaps be refined, supposing the knowledge is only transmitted to the components connected to this port, though we do not investigate this possibility here.

In the symbolic case, it is knowledge of the constraints on a system dynamic variable´s value that is passed between components via communications. Constraints are accumulated along paths through the partial orders defining the semantics of the system. The global constraint at a particular point of an execution is defined as the conjunction of the constraints on each of

the concurrent paths. For example, for a consistent test description execution, the conjunction of the accumulated constraints at each of the pass-annotated events must not be equivalent to false.

Due to the way we define the intended value of a system dynamic variable, we must allow synchronization messages to have parameters in order to use them to transmit knowledge about dynamic variable values where required. Since we allow multiple SUT lifelines, this also applies to SUT synchronization messages.

2.3 TeLa sequence diagram references

2.3.1 Introduction

As stated in Section 1.7, TeLa diagram references are allowed in two different contexts in TeLa sequence diagrams; we will refer to these two types of diagram reference as lifeline-anchored references and message-anchored references. A message to which a message-anchored reference is associated is called a link message.

TeLa diagram references are used in TeLa sequence diagrams to denote sequential composition. In the case of message-anchored references, the composition is always weak sequential composition. However, in the case of lifeline-anchored references, we may want to denote strong sequential composition with respect to the tester components, that is, sequential composition involving a synchronization between all the tester components. Strong sequential composition is novel with respect to MSCs.

As shown in Section 2.10, message-anchored references are used to implement alternative diagram continuations. They allow the use of a syntax for the TeLa sequence diagram choice construct which is close to the “presentation option” branching mechanism of UML 1.4, generally perceived as user-friendly.

TeLa diagram references can also be used to implement a joining of alternative branches in the sense that an identical reference to a diagram can be made in several different diagrams representing the different alternative branches. The interpretation is that the referenced diagram is the continuation of each of the alternatives, that is, that the alternative behaviour is common after a certain point.

From a tooling point of view, the diagram referenced could be implemented as hyperlinks to the corresponding diagram, a mechanism similar to that proposed in [EkkSchGra00b].

2.3.2 Syntax

For lifeline-anchored references, see Fig. 4-5 l.h.s., we use a syntax taken from the MSC reference construct. For message-anchored references, see Fig. 4-5 r.h.s. we use a syntax taken from a presentation option of UML 1.4. In message-anchored references, a send event of the referenced diagram appears in the referencing diagram and it is to this send event that the reference is attached.

Our use of lifeline-anchored references is much more restricted than the MSC use of MSC references (or UML 2.0 use of the equivalent), due to the absence of gates in TeLa and to the absence of reference expressions other than sequence-diagram names. TeLa lifeline-anchored references always cover all the lifelines of an interaction. The diagram references in TeLa

activity-diagrams, see Section 3.2, are similar to the lifeline-anchored diagram references in TeLa sequence diagrams.

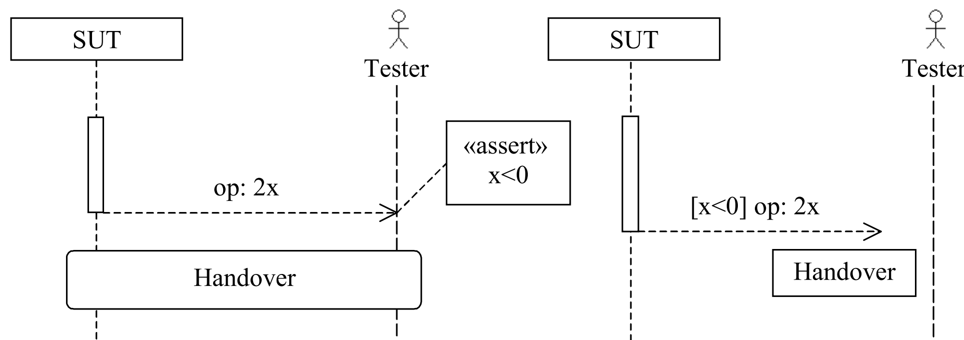


Figure 4-5: lifeline-anchored references (left) and message-anchored references (right), where *Handover* is the name of a TeLa sequence diagram.

Our use of message-anchored references is not the same as the use of the UML 1.4 "presentation option" of the standard from which the syntax is taken. In the standard, this syntax was proposed as a kind of simplified gate construct, used for dividing a sequence diagram horizontally, placing some lifelines of an interaction on one diagram and some on another. Moreover, on the referenced diagram, we do not use the matching syntax, in which the receiver half of an arrow is shown, in order to be able to reference a given sequence diagram via a message-anchored reference or via a lifeline-anchored reference. However, as for the UML 1.4 "presentation option", the link message is repeated in the referenced diagram, except in the case of the "default alternative", presented in Section 2.10.3.1 and 2.10.3.2.

The message arrow label on the repeated message in the referenced diagram does not need to include the guard, if one exists. Usually, then, the guard on the repeated message only appears in the referencing diagram. Similarly, if the link message is a synchronous invocation return, the synchronous invocation focus bar only appears on the referencing diagram, not in the referenced diagram. If the link message is any other message emitted from inside the scope of a focus bar, the focus bar in the referencing diagram is shown without the bottom side of the rectangle and the focus bar in the referenced diagram is shown without the top side of the rectangle.

The syntax we use for a lifeline-anchored reference denoting strong sequential composition is to precede the reference with a horizontal bar (covering all the lifelines of the interaction) representing a synchronization between tester components, as shown in Fig. 4-6.

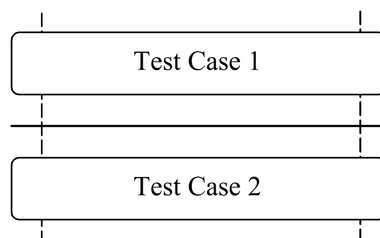


Figure 4-6: Strong sequential composition in TeLa

2.3.3 Well-formedness conditions

We do not allow TeLa diagram references to be used recursively. In particular, we do not allow a sequence diagram reference to contain the name of the referencing diagram. Detecting such cycles could be done using the Tarjan algorithm for calculating simply connected components [Tar72].

2.3.4 Informal semantics

Semantically, weak sequential composition implies ordering relations between the last event on a lifeline of the referencing diagram and the first event on the same lifeline of the referenced diagram, for each of the lifelines which the two diagrams have in common.

Strong sequential composition implies ordering relations between all the tester events of the referencing diagram (and any preceding diagrams) and all the tester events of the referenced diagram (and of any succeeding diagrams). One way of realising this ordering in the semantics is by explicitly introducing an event shared by all the tester lifelines. Strong sequential composition does not imply any synchronization between the tester and the SUT; all tester-SUT communication must be explicitly described.

2.4 Internal actions

2.4.1 Introduction

The internal action construct is used to describe a tester component performing any other operation besides sending or receiving a message. The stereotypes «assign», «assert», «create» and «escape» are used for variable assignments, assertion evaluations, subcomponent creation and escape to another language respectively. In the enumerated data case, only the «create» and «escape» internal actions are used.

2.4.1.1 ALLOWED INTERNAL ACTIONS IN TELA

An assignment internal action is used to assign values defined by expressions to dynamic variables. A single assignment internal action may contain assignments to several such dynamic variables. In the usual case, all variables assigned to in an assignment internal action belong to the same base-level component and the internal action is attached to a single lifeline. If this is not the case, the internal action may be attached to a single lifeline or multiple lifelines, depending on the level of decomposition. The semantics of an assignment internal action attached to a single lifeline at the point where a message is received from the SUT (we will say it is attached to an SUT-message arrowhead) is slightly different to that of other assignment internal actions, see below.

An assertion internal action is used to verify that a boolean expression involving static, dynamic and anonymous variables holds. It may be attached to a single lifeline or to multiple lifelines. The semantics of an assertion internal action attached to a single lifeline at the point where a message is received from the SUT (we will say it is attached to an SUT-message arrowhead) is different to that of other assertion internal actions, see below.

Recall that each dynamic variable is owned by a base-level component but a value of a system dynamic variable can be held by any base-level component. Thus, we must ensure that there is no ambiguity in the intended value of such a variable at any use of it. A system dynamic variable value used in an assertion or on the r.h.s of an assignment may be preceded by a

base-level component name followed by a dot. If this is the case, the intended value of the variable is that held by the specified base-level component. If this is not the case, the intended value is as explained in the well-formedness conditions below.

An escape internal action is used to insert code from an external language. It either takes the form of an assignment, to a system dynamic variable, of the result of a call to an external function, or the form of a call to an external procedure. A create internal action is used to model the creation of a subcomponent.

2.4.1.2 OWNING COMPONENT OF AN INTERNAL ACTION

For an assertion or assignment internal action attached to multiple lifelines, let the components that are represented by one of the lifelines to which the internal action is attached be called the *internal action components*.

Recall that the component containment hierarchy is a tree. For an assertion internal action attached to multiple lifelines, the smallest component containing all the internal action components is said to own the internal action. For an assertion internal action attached to a single lifeline at the point where a proper reception action occurs, the component owning the target port of the message is said to own the internal action. For an assertion internal action attached to a single lifeline and not at the point where a proper reception occurs, if a subcomponent of the lifeline component is not specified explicitly as the owning component (its name followed by a colon appears before the assertion), the assertion belongs to the lifeline component.

For an assignment internal action attached to multiple lifelines, the smallest component containing all the internal action components is said to own the internal action. For an assignment internal action attached to a single lifeline at the point where a proper reception action occurs, the component owning the target port of the message is said to own the internal action. For an assignment internal action attached to a single lifeline and not at the point where a proper reception occurs, the smallest component containing the set of base-level components owning a variable on the l.h.s. of one of the assignments is the owner of the internal action.

For a creation internal action, the smallest component contaning the created and creator component is said to own the internal action.

An escape internal action is owned by the component represented by the lifeline to which it is attached.

In the simple object model, all internal actions are owned by a base-level component (object). The information about the owning component is considered to be part of the action labelling each of the four types of component-internal event.

2.4.1.3 DECOMPOSTION OF CERTAIN INTERNAL ACTIONS

Certain assignment internal actions can be decomposed into an exchange of tester coordination messages. The necessary prerequisite is that the dynamic variables on the r.h.s. of the assignment(s) are all owned by the same base-level component c, while the dynamic variable on the l.h.s. is owned by another base-level component d. In this case, the assignment internal action can be decomposed into the exchange of a tester coordination message from c to any supercomponent of d which does not include c.

In a similar way, a creation action can be decomposed into a message exchange – the creator sending a creation message to the created component, using the UML syntax for this.

2.4.2 Syntax

In the graphical syntax used here, an internal action is presented as a stereotyped note attached to a particular point on one or several lifelines. Notes stereotyped «assert» contain boolean expressions; each line is a new clause to be conjoined with the clauses of the other lines. Notes stereotyped «assign» contain assignment expressions. Notes stereotyped «create» contain creation expressions. Notes stereotyped «escape» contain escape expressions.

Figs. 4-7 and 4-8 show alternative syntaxes for proper receptions. All these alternative syntaxes are legal, since we hold the view that more user experience is needed before deciding which of these options to prohibit. The first figure shows that the use of expressions in message parameters does not obviate the use of anonymous variables. p_1 and p_2 are ports, u, v, x, y are system dynamic variables, $_{1}$, $_{2}$, $_{3}$, $_{4}$ are anonymous variables. Note that in the enumerated data case, neither internal actions, nor guards, nor the any-value notation nor the proper-reception assigned variable notation are used.

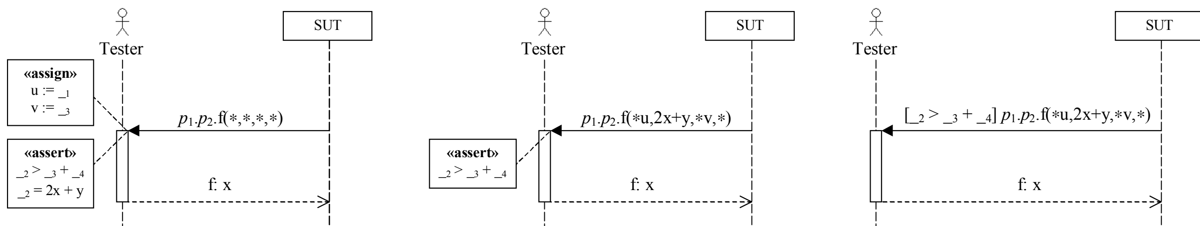


Figure 4-7: Three alternative syntaxes for the same behaviour showing the use of assign and assert internal actions attached to SUT-message arrowheads, proper-reception assigned variables, anonymous variables and guards.

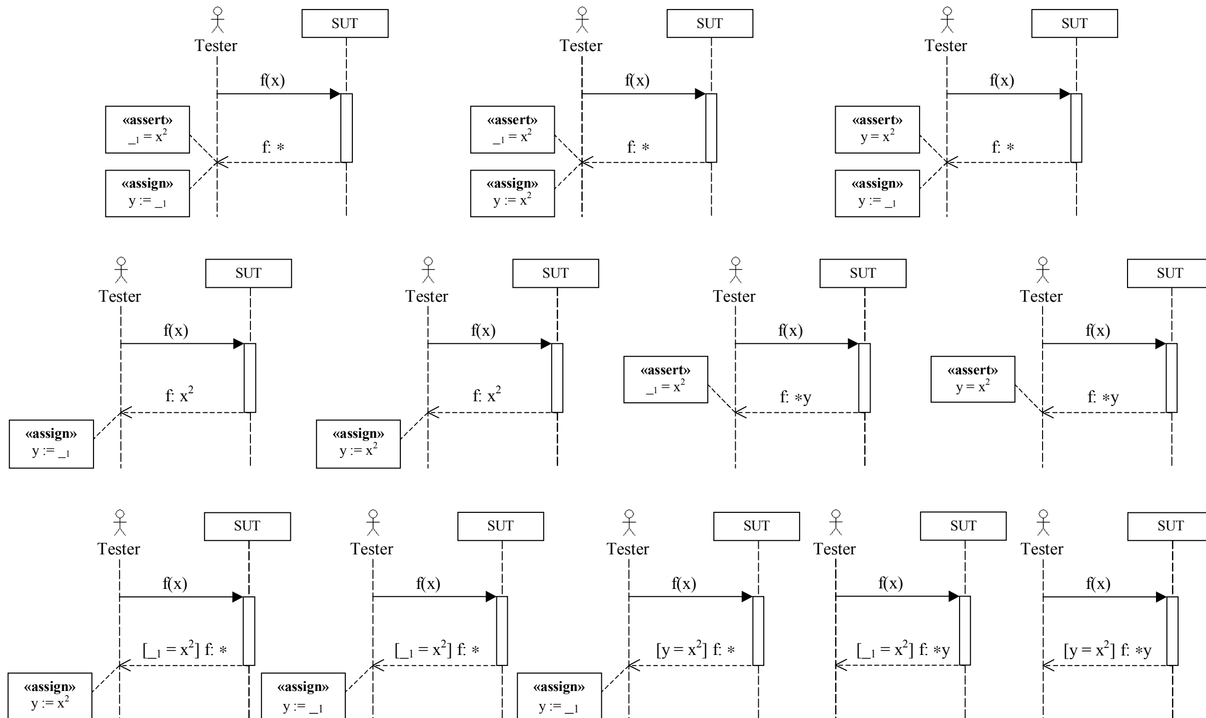


Figure 4-8: Twelve alternative syntaxes for the same behaviour showing the use of assign and assert internal actions attached to SUT-message arrowheads, proper-reception assigned variables, anonymous variables, and guards.

2.4.3 Well-formedness conditions

All internal actions except the «create» internal action, can only be placed on lifelines representing tester components. The «create» internal action can also be placed on a lifeline representing an SUT component but not on a lifeline representing an SUT port.

If an internal action is attached to multiple lifelines, it cannot fall within the scope of a coregion on any of them. If an internal action is attached to multiple lifelines, the components represented by these lifelines must be subcomponents of a single component (and, of course, be members of an interaction framework, see Chapter 5, Section 1.3.5).

The first two parameters of a create internal action must be names of components (the first being that of the creator component and the second being that of the created component).

2.4.3.1 OWNING COMPONENTS

The owning component of a create internal action, an assertion internal action or an assignment internal action that is attached to a single lifeline must be a subcomponent of the component represented by the lifeline. In the case of the assignment internal action, this implies that each dynamic variable on the l.h.s. of an assignment is owned by a base-level subcomponent of the component owning the internal action. Each dynamic variable on the l.h.s. of an assignment in an assignment internal action attached to multiple lifelines is also owned by a base-level subcomponent of the component owning the internal action.

2.4.3.2 VARIABLES USED IN ASSERTIONS AND ASSIGNMENTS

An assertion internal action attached to an SUT-message arrowhead can involve static, dynamic and anonymous variables. All other assertion internal actions can only involve static and dynamic variables. An assignment internal action attached to a message arrowhead must involve at least one anonymous variable on proper reception-assigned variable in the expression on the r.h.s. of each assignment. All other assignment internal actions can only involve static and dynamic variables. Creation actions can only involve state and dynamic variables.

2.4.3.3 DYNAMIC VARIABLE VALUES IN ASSERTIONS AND ASSIGNMENTS

If a dynamic variable name used in a creation expression, an assertion or on the r.h.s. of an assignment is preceded by a base-level component name and a dot, the identified component must be a subcomponent of the component owning the internal action.

If a dynamic variable name used in an assertion or in a creation expression is not preceded by a base-level component name, all base-level subcomponents of the component owning the internal action must share a common view of its value.

If a dynamic variable name used on the r.h.s. of an assignment is not preceded by a component name followed by a dot, the value of this variable held by the owner of the dynamic variable on the l.h.s. of that assignment must be inferred.

2.4.3.4 EQUIVALENCE OF SYNTAX INVOLVING GUARDS AND INVOLVING ASSERTIONS

An assertion internal action cannot occur inside a coregion.

2.4.4 Informal semantics

2.4.4.1 ASSERTIONS

Semantically, an assertion internal action attached to an SUT-message arrowhead denotes a guard on the corresponding proper reception action. We will assume that this guard constrains only the anonymous and/or proper-reception assigned variables of the corresponding message. If this is the only action available to the tester and the guard is not verified, a fail test verdict will be obtained. Thus, such an internal action does not define a controllable event but, instead, is part of an observable event.

Any other type of assertion internal action denotes either an assert component-internal event, that is, an action with a guard and a name but with no other content, or the guard of the following controllable event. In both cases, if this is the only alternative action available to the tester and the guard is not verified, an inconclusive test verdict will be obtained. For an assertion internal action whose assertion involves dynamic variables from different base-level components, a synchronization between these base-level components is implied.

2.4.4.2 ASSIGNMENTS

The r.h.s. of an assignment of an assign internal action that is attached to an SUT-message arrowhead may include anonymous variables. If the r.h.s. of an assignment is a single anonymous variable, the notation is an alternative to the use of a proper-reception assigned variable in the corresponding message parameter. Assignments from assign internal actions attached at any other point in a diagram cannot include anonymous variables. Semantically, an assignment internal action attached to an SUT-message arrowhead denotes a set of assignments involving the message variables, that is involving values sent by the SUT.

All assignment internal actions are interpreted as an assign component-internal event. An assignment to dynamic variable x defines a change in the data context associated to the base-level component which owns x. For an assignment internal action involving several assignments, the order of appearance of these assignments in the component-internal event is exactly that of the syntax. An assignment internal action comprising assignments to variables from different base-level components involves the synchronization of these components in order to realise the assignments specified.

2.4.4.3 COMPONENT CREATION AND ESCAPE

Semantically, a creation internal action is a creation component-internal event. Semantically, an escape internal action is an escape component-internal event. Both events are placed in the partial orders according to their predecessors and successors on the lifeline on which they appear.

2.4.4.4 RELATION BETWEEN GUARDS AND ASSERTIONS

In general, in system specification there is a clear difference between an assertion and a guard: a guard specifies that an execution path is not feasible if the expression does not evaluate to true, whereas an assertion specifies that an exception will be raised if the expression does not evaluate to true. However, in a test description and in the presence of implicit verdicts, this distinction is blurred.

Consider, for example, a guarded proper tester reception event that is not in conflict with any other events (that is, it is not one of several possible alternative behaviours). If this event is enabled in an execution, and an event corresponding to the reception of a message on the same port with the same name and the same signature is received, if the guard is not satisfied

a fail verdict results. Thus, a kind of exception is raised. More details of the semantics of guards and assertions can be found in Sections 3.4 and 3.5.

2.5 Coregions

2.5.1 Introduction

A coregion is used to break the default ordering, that is, to introduce concurrency on lifelines[3]. Concurrency may arise naturally on a lifeline representing a non base-level component due to parallelism of its subcomponents. It may also arise when we do not want to restrict the behaviour of the SUT to one particular ordering, for example, to test a multicast operation to be performed by the SUT. Notice that a coregion merely states that the lifeline in question imposes no ordering requirements, not that the events in question are not ordered. They may be ordered via messages, as are the events: sending of the invocation of *get-time* to *Stub1* and the reception of the corresponding reply from *Stub1* in Fig. 4-9.

If a lifeline is part of a component on which the partially-ordered message semantics is to be used, the role of the coregion is different, see Chapter 5, Section 3. In particular, the use of this semantics means that some coregions have no effect.

If the test description is to define a fully-executable test case, the only events from the same base-level component which are allowed in the scope of a coregion must be receptions, since for a fully executable test case, the tester behaviour must be fully specified.

2.5.2 Syntax

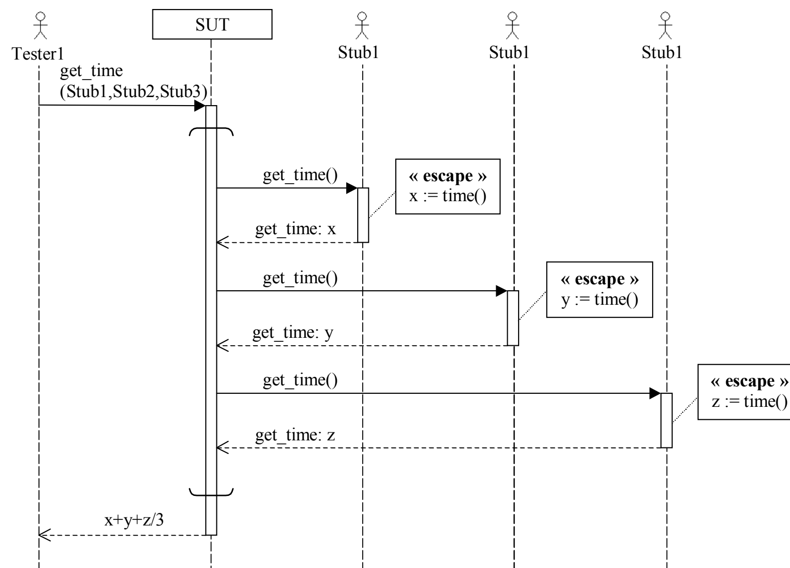


Figure 4-9: A coregion used to explicitly specify a multicast to three components in which ordering is unimportant. The parameters of the message *get_time* sent by *Tester1* contain the constants *Stub1*, *Stub2* and *Stub3*.

[3] The introduction of this operator therefore pre-supposes we are not using the minimalist interpretation discussed in Chapter 2, Section 3.3.2.

In the current graphical syntax, a coregion is presented as a pair of vertical braces, as shown in the example of Fig. 4-9. The syntax used does not connect up the braces since the coregion may be annuled, or partially annuled (see Section 2.7) by a focus bar appearing in its interior.

In this example, we can suppose that the ports in question are those of base-level components (objects) and are therefore identified with these base-level components. *x*, *y* and *z* are system dynamic variables. The internal action is an escape internal action, rather than an assign internal action, since the function *time()* is an external function, that is, it is not part of the TeLa language. Note that if the SUT is passive in this example, the restrictions concerning calling regions discussed in Section 2.7.4.3 mean that no event (such as the sending of another *get_time*() invocation) can occur between the sending of a *get_time*() invocation and the reception of the corresponding reply.

2.5.3 Well-formedness conditions

A coregion must be contained in a single TeLa sequence diagram (i.e. coregions are not allowed to cross diagram references). This condition is imposed to simplify the semantics and to allow the equivalence between one-tier and two-tier TeLa test descriptions to be easily demonstrated.

A coregion may not overlap with that of a focus bar if the lifeline component is passive. Any such coregion is effectively annuled by the restrictions imposed by passiveness (see well-formedness conditions and semantics of focus bars). This does not prohibit a focus bar on a passive lifeline from containing a coregion or from being contained in the scope of a coregion.

The scope of a coregion may not overlap with that of a suspension region (a calling region on a lifeline representing a passive component). This does not prohibit a calling region from containing a coregion or from being contained in the scope of a coregion.

We do not allow identically-labelled events in the scope of a coregion in order not to introduce pointless non-determinism.

2.5.4 Informal semantics

A coregion cancels the ordering effect of a lifeline. The only ordering between events falling inside a coregion is that defined via messages (including synchronous calls) or by explicit local orderings, where these may be represented by the local ordering construct or by focus bars.

As stated in Section 2.7.4, a focus bar or calling region inside a coregion cancels the effect of the coregion for the events in its scope, except any events which may be inside another coregion contained in the scope of that focus bar or calling region. More details concerning the semantics of focus bars or calling regions inside coregions is given in Section 2.7.4.

The coregion construct lends itself to confusion between implementation-level parallelism, where the meaning is in terms of multiple thread/processes, and specification-level parallelism, where the meaning is simply that the ordering is not constrained. To clarify this point, consider that specification-level parallelism may be implemented as implementation-level parallelism, i.e. via multiple threads, or it may be implemented using a single thread and choosing an order from among the possible orders.

We attempt to cater for both these interpretations by considering that the basic interpretation is specification-level parallelism while allowing the passive/active component annotations on

tester components to indicate how this specification-level parallelism is meant to be implemented. We may even allow these annotations to be placed on SUT components if we wish to be able to impose such implementation restrictions on the SUT.

2.6 Local orderings

2.6.1 Introduction

Local orderings are curved discontinuous lines connecting events on the same lifeline. They are used to impose orderings between events that fall inside a coregion. They are essential for conserving orderings under lifeline composition.

2.6.2 Syntax

Figure 4-10 shows the syntax of the local ordering construct. Notice that there no ordering is denoted between the reception of messages m_1 and m_2, nor between the emission of message m_3 and m_4, nor between the reception of m_1 and the emission of m_3.

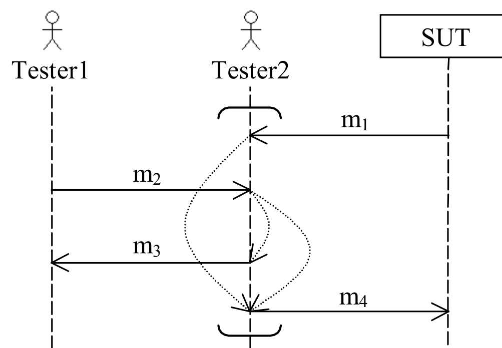


Figure 4-10: Local orderings specifying that the reception of message m_2 precedes the emission of messages m_3 and m_4 and that the reception of message m_1 also precedes the emission of message m_4.

2.6.3 Well-formedness conditions

A local ordering can only be used to connect two events falling inside the same coregion.

2.6.4 Informal semantics

Local orderings play the role of local MSC general orderings, that is, general orderings between events on the same lifeline. They are used to impose orderings between events within a coregion.

2.7 Focus bars

2.7.1 Introduction

In TeLa, vertically-aligned rectangles called focus bars must be attached to the end of synchronous invocation arrows and may optionally be attached to the end of asynchronous

invocation arrows. In the case of a synchronous invocation, the focus bar is terminated by the return message of that synchronous invocation[4].

Note that we allow focus bars associated to intra-SUT synchronization messages. This is to allow the definitions of nested invocation and callback, see below, to take into account causal chains passing through the SUT.

2.7.1.1 OWNING COMPONENT OF A FOCUS BAR

Given a focus bar, let the invocation represented by the message arrow to which the focus bar is attached be called the *focus bar invocation*, the port which is the target of the focus bar invocation be called the *focus bar port* and the owning component of the focus bar port be called the *focus bar component*. The focus bar component is said to own the focus bar. In the simple object model, the owning component of a focus bar is always a base-level component (object).

By structural consistency, the focus bar component is necessarily a subcomponent (proper or otherwise) of the lifeline component. An event is in the scope of a focus bar if it is vertically coincident with it. Due to lifeline composition/decomposition, focus bar scope does not necessarily coincide with the idea of activations. For example, an emmision event can be ordered with respect to the events of a focus bar without being emitted by (a subcomponent of) the focus bar component. Let the events in the scope of the focus bar be called the *focus bar events*. By structural consistency, if the lifeline component is the focus bar component, all focus bar events are owned by the focus bar component or a subcomponent of it.

2.7.1.2 CALLING REGIONS

The connection between a synchronous invocation message and the corresponding return message is denoted by connecting the synchronous invocation message reception event and the return message emission event on the receiver lifeline using a focus bar. The vertical space on the sender lifeline bounded by the synchronous invocation emission event and the return message reception event also defines a scope which we call the *calling region* (though we do not use any special syntax to denote it).

Given a calling region, let the bounding synchronous invocation and corresponding return be called the *calling region invocation* and the *calling region return* respectively. Let the component owning the originating port of the calling region invocation (the emitting component) be called the *calling region component*. Let the bounding events of the calling region, together with any events falling inside the calling region be called the *calling region events*. If the lifeline component on which the calling region occurs is a passive component, the calling region is said to define a *suspension region*. A calling region on a lifeline representing an active component is not a suspension region.

2.7.1.3 NESTED INVOCATIONS AND CALLBACKS

An invocation emitted from within the scope of a focus bar whose originating port is owned by the focus bar or a subcomponent of it is said to be a first-level *nested invocation* of the focus bar invocation. If an intra-SUT synchronization message has an associated focus bar, then any message emitted inside the scope of this focus bar is said to be a first-level nested invocation of the focus bar invocation. A first-level nested invocation is a nested invocation

[4] Note that we also allow synchronous invocations from the tester to the SUT with no return arrow in order to model the situation in which the SUT does not reply due to some internal error.

and a nested invocation of a nested invocation is also a nested invocation, i.e. the property of being nested is recursive. Notice that on composition of a lifeline containing a (level 1) focus bar and another lifeline on which a nested invocation of the (level 2) focus bar invocation is emitted, in the composed diagram, the nested invocation emission will be in the scope of the (level 1) focus bar represented in the composed diagrams. Focus bars may be conserved under lifeline composition by using auto-invocations. If the notion of passive/active components is used, we oblige this conservation on passive components.

A nested invocation i_2 of an invocation i_1 is said to be a *base-level callback* or *object callback* of i_1 if

- i_2 is received inside the calling region defined by i_1
- the target port of i_2 is the originating port of i_1

A nested invocation i_2 of an invocation i_1 is said to be a *callback* or *focus bar callback* of i_1 if:

- i_1 is a first-level nested invocation of a focus bar invocation,
- i_2 is received inside the calling region defined by i_1 (this calling region being in the scope of the corresponding focus bar),
- the target port of i_2 is owned by the focus bar component or by a subcomponent of it.

2.7.1.4 NESTED AND COINCIDENT FOCUS BARS

Focus bars which overlap vertically on a lifeline may, or may not, overlap horizontally. If they do not overlap horizontally, they are said to be coincident but independent. If two focus bars overlap horizontally, one of them must be vertically contained in the other and the inner one is said to be a nested focus bar of the outer one. If a set of focus bars are nested, an event may be in the scope of several focus bars. In such a case, an event is said to be in primary scope of the outermost focus bar at that vertical position. By well-formedness, if a focus bar is nested inside another, the inner focus bar invocation is a nested invocation of the outer focus bar invocation.

2.7.2 Syntax

The focus bar is represented as a vertical rectangle attached to the end of a message arrow. Figs. 4-11, 4-12 and 4-13 illustrate the interaction between the focus bar and the coregion.

Classifying Tester2 of Fig. 4-11 as a passive component is to say that in any linearisation, the events $!m_1$ and $!m_8$ cannot occur between $?m_2$ and $!m_3$, nor between $!m_3$ and $!m_4$, nor between $!m_4$ and $!r_2$, nor between $?m_5$ and $!m_6$, nor between $!m_6$ and $!m_7$ (where ! denotes emission and ? denotes reception). See Section 2.7.4 for more details.

Classifying Tester2 of Fig. 4-12 as a passive component is to say that in any linearisation, the events $!m_1$ and $!m_8$ cannot occur between any of the other events, and the event $!m_7$ cannot occur between $?m_5$ and $!m_6$ and also that m_5 is necessarily a callback. See Section 2.7.4 for more details.

In Fig. 4-13, note the use of focus bars attached to intra-SUT synchronization messages in order to allow causal chains to pass through the SUT.

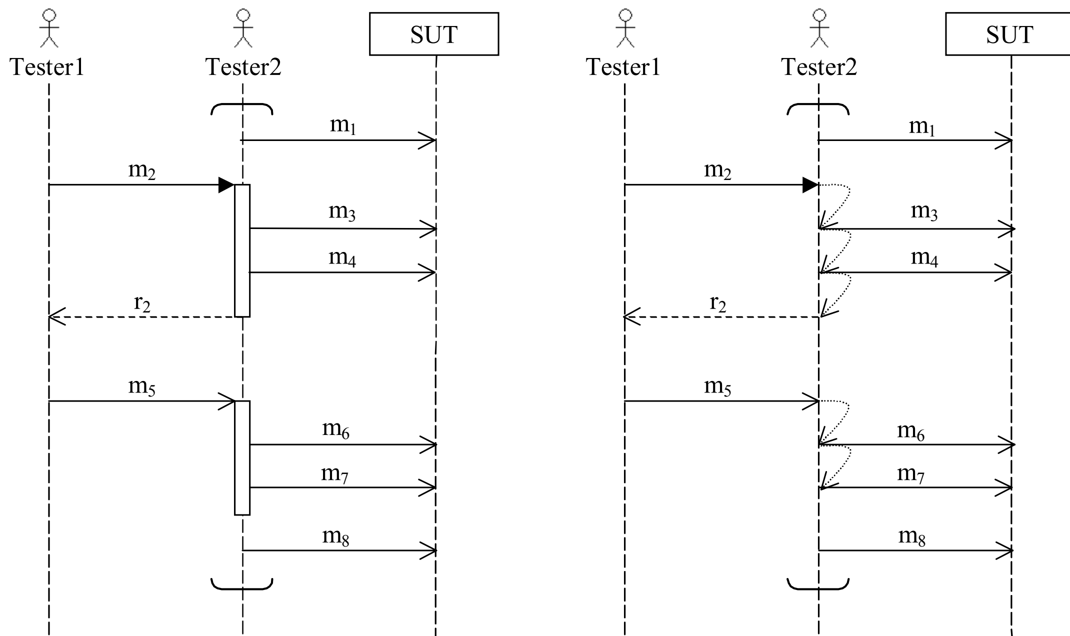


Figure 4-11: Focus bars associated to synchronous and asynchronous invocations falling inside the scope of a coregion (l.h.s) and the equivalent orderings denoted using the local ordering construct (r.h.s).

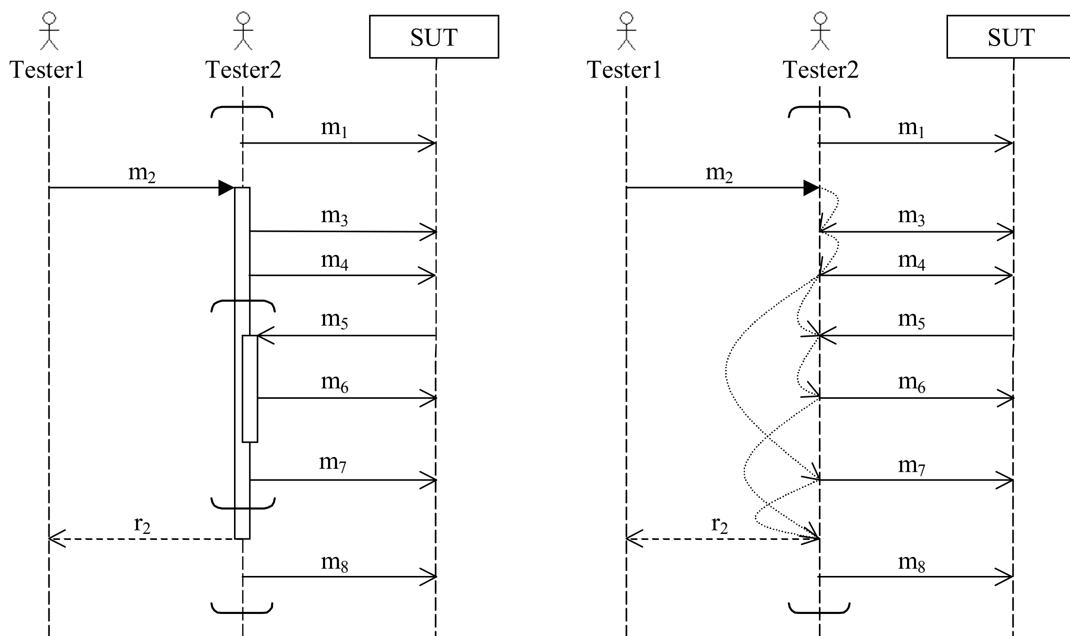


Figure 4-12: A more complicated example of nesting of focus bars and coregions (l.h.s) and the equivalent orderings denoted using the local ordering construct (r.h.s).

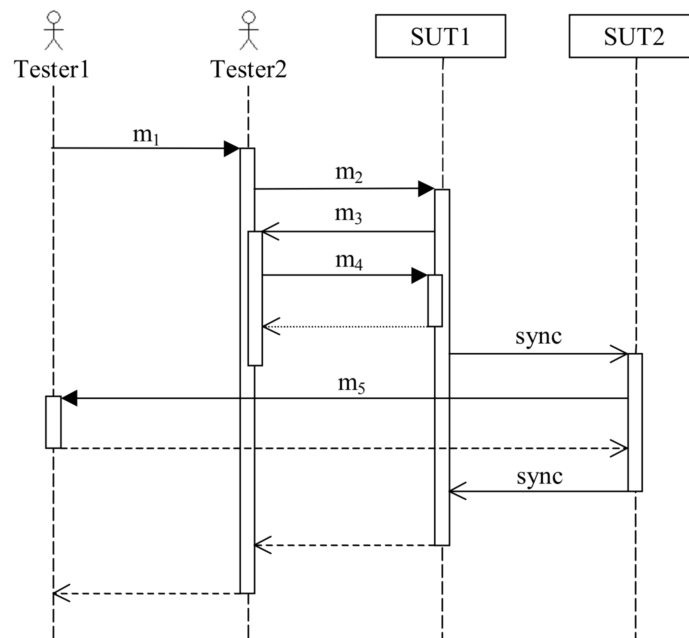


Figure 4-13: Nested invocations and callbacks. To simplify the example, we suppose that all lifelines represent base-level components so that there is no need for explicit port information.

2.7.3 Well-formedness conditions

2.7.3.1 OWNING COMPONENTS AND NESTING

The return message of a synchronous invocation must be owned by the focus bar component or by a subcomponent of it.

If a focus bar f_2 with focus bar invocation i_2 is nested inside another focus bar f_1 with focus bar invocation i_1, then i_2 must be a nested invocation of i_1. In fact, in the case where the lifeline on which the focus bar appears represents a passive component, i_2 is necessarily either an auto-invocation or a callback of a nested invocation of an invocation emitted in the scope of f_1.

2.7.3.2 FOCUS BARS ON PASSIVE LIFELINES

If control flow scheme annotations are used in the component model, we also have the following well-formedness conditions w.r.t. the passiveness notion:

- If a lifeline component is passive, all incoming invocations must have associated focus bars and all outgoing invocations must be in the scope of a focus bar. Thus, on a passive lifeline, all calling regions are inside focus bars.
- If two lifeline components are passive, then any composition of these lifelines (see Section 2.8) must conserve all focus bars (by means of auto-invocations that may be synchronous or asynchronous). This then ensures that on passive lifelines, all events in the primary scope of a focus bar are owned by the focus bar component or one of its subcomponents.
- Coincident but independent focus bars cannot occur on a passive lifeline.
- If a lifeline component is passive, no focus bar occurring on it can have incoming messages in its scope that are not callbacks of a nested invocation of the focus bar

invocation (in fact, due to point 2 above, we need only discuss first-level nested invocations).

2.7.4 Informal semantics

A focus bar attached to a synchronous invocation arrow and terminated by a synchronous return arrow serves to relate the request to the reply. A focus bar attached to either a synchronous an asynchronous invocation that falls inside a coregion serves to impose a total ordering between the events in its scope. A calling region that falls inside a coregion also serves to impose a total ordering between the events in its scope. Thus, in this respect, a focus bar is a shorthand for a set of local orderings. Coincident but independent focus bars denote the same ordering relations as a single focus bar occupying the combined vertical scope of the coincident focus bars.

In addition to denoting ordering, the focus bar is the principle TeLa construct concerned by control flow scheme annotations (i.e. the property of activeness/passiveness). The way focus bars impose orderings and the way they are affected by the property of passiveness is coherent with their interpretation as representing method bodies (or activations).

2.7.4.1 BLOCKING OF PASSIVE COMPONENTS ON SYNCHRONOUS INVOCATION

If the emitting component of a synchronous invocation is passive, it is considered to be blocked between the sending of the invocation and the reception of the corresponding reply. The same applies to any passive subcomponent or supercomponent of it. Clearly, for a synchronous invocation emitted in the scope of a focus bar, if the focus bar component is passive, it is among the blocked components.

The blocked component(s) may be unblocked by the reception of a base-level callback of the synchronous invocation, the unblocking lasting for the duration of the treatment of this callback. We also consider that the blocked component(s) may be unblocked by the reception of a focus bar callback that is not a base-level callback, if the following three conditions are satisfied: the synchronous invocation is emitted in the scope of a focus bar, the originating port of the synchronous invocation is the focus bar component or a subcomponent of it, and the focus bar component is passive. Again, the unblocking lasts for the duration of the treatment of this callback. Thus, the only events owned by a passive supercomponent of the emitting (base-level) component that can occur in a calling region are receptions and replies to callbacks.

2.7.4.2 PASSIVE FOCUS BAR COMPONENTS AND COREGIONS

Suppose we have a coregion containing, or overlapping, a focus bar. That is, some or all of the events in the scope of the coregion are in the scope of the focus bar but there are events in the scope of the coregion which are not in the scope of the focus bar. The property of passiveness of the focus bar component is read as an implementation directive imposing a relation between the focus bar events of the coregion and the other events of the coregion. The relation between these two types of events is expressed as a constraint on the allowed linearisations of all events in the scope of the coregion.

Let the events which are in the scope of the coregion, but not in the scope of the focus bar, and which are owned by a passive sub- or super-component of the focus bar component, be called the restricted events. The constraint on linearisations is as follows: in any allowed linearisation, no restricted event can occur between two focus bar events. Notice that interleaving with events of the coregion owned by active subcomponents of the lifeline

component is not constrained. See Figures 4-11 and 4-12 and the explanatory text of Section 2.72 for an example.

2.7.4.3 PASSIVE CALLING REGION COMPONENTS AND COREGIONS

Suppose we have a coregion containing, or overlapping, a calling region. If the calling region is inside a focus bar which, in turn, is inside a coregion, the case is already covered by the previous section, so suppose the calling region is not inside a focus bar. In a similar way to the situation for focus bars, the property of passiveness of the calling region component is read as an implementation directive imposing a relation between the calling region events of the coregion and the other events of the coregion.

Let the events which are in the scope of the coregion but not in the scope of the calling region, and which are owned by a passive super-component of the calling region component, be called the restricted events. The constraint on linearisations is as follows: in any allowed linearisation, no restricted event can occur between two calling region events. Notice that interleaving with events of the coregion owned by active subcomponents of the lifeline component is not constrained.

2.8 Lifeline composition

2.8.1 Introduction

The existence of an underlying component model provides the structure necessary to define lifeline composition and decomposition, see the definition of interaction framework in Chapter 5, Section 1.3.5 and 1.4.3.

The coregion and local ordering constructs are essential in order to preserve the orderings on a composed lifeline, i.e. at the interface of the component which the composed lifeline represents.

Lifeline composition also presents a challenge for the use of focus bars. However, we have defined focus bars in such as way as to permit their unambiguous use in the presence of lifeline composition (see, in particular, Section 2.7.3.2). As mentioned earlier, the same level of decomposition must be used throughout a one-tier or two-tier scenario structure.

2.8.2 Syntax

Two examples of lifeline composition are shown below. Both Figs. 4-14 and 4-15 show two views of the same behaviour, one showing more detail than the other. In both figures, on the r.h.s., the component Tester 2 is represented, while on the l.h.s. its two subcomponents Tester2a and Tester2b are represented.

Fig. 4-14 shows the need for local ordering relations; the required ordering relations cannot be imposed using two coincident but independent focus bars (since $?m_1$ and $!m_3$ are not ordered).

Classifying Tester2 of Fig. 4-15 as a passive component is to say that m_3 cannot occur between m_1 and r_1 in any linearisation, thus imposing the relation $!r_1 < ?m_3$, effectively annulling the coregion.

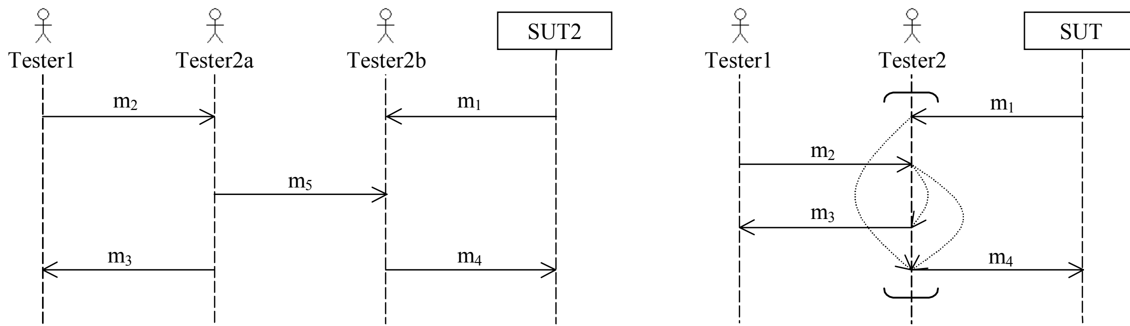


Figure 4-14: The TeLa sequence diagram on the r.h.s. is obtained from the TeLa sequence diagram on the l.h.s by composing the two lifelines Tester2a and Tester2b.

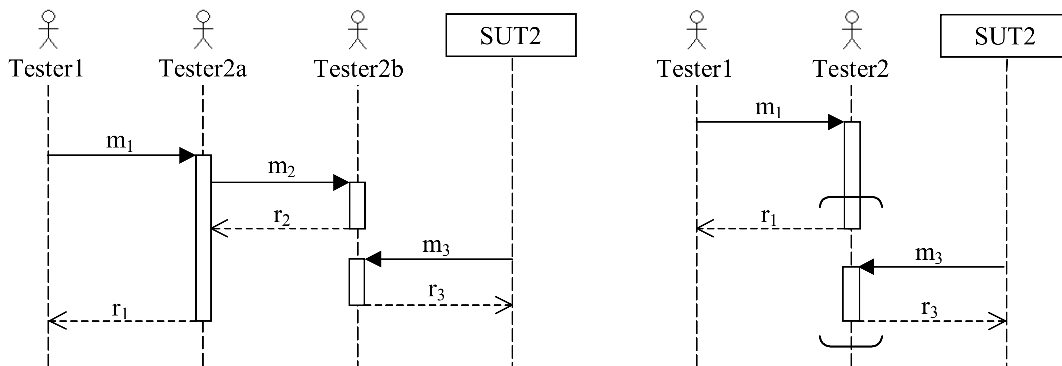


Figure 4-15: The TeLa sequence diagram on the r.h.s. is obtained from the TeLa sequence diagram on the l.h.s by composing the two lifelines Tester2a and Tester2b.

2.8.3 Well-formedness conditions

The set of lifelines of the composed and decomposed diagram both define an interaction framework, see Chapter 5, Sections 1.3.5 and 1.4.3.

2.8.4 Informal semantics

One diagram is a composition of another if the interaction framework of the first diagram is a decomposition of that of the second diagram, see Chapter 5, Section 1, and the ordering and conflict relations for the events which appear in both diagrams are the same.

2.9 Sequence diagram loops

2.9.1 Introduction

We wish to introduce a locally-defined loop construct to provide a user-friendly, if rather low-level, means of specifying simple loops in a single sequence diagram. However, the global meaning of locally-defined loops is not necessarily clear, except for procedural sequence diagrams (diagrams with a single flow of control and using only synchronous invocations). In particular, if the scope of a loop is only defined on one lifeline, how do we know its scope on other lifelines when asynchronous invocations do not necessarily have an associated focus bar and there may be more than one control flow? How do we detect illegal casual chains which

start in the scope of a loop and finish on the lifeline of the loop but outside the scope of the loop (or vice versa)?

One way to solve this problem is to explicitly specify the global scope by delimiting a region encompassing several lifelines (of course, no messages may cross this scope). This is the solution adopted in the MSC standard for the loop in-line expression.

We adopt a slightly more complicated solution that involves recursively defining the notion of projection of the loop scope on other instances, see below. This second solution is more in keeping with our desire to use operators with local scope in TeLa sequence diagrams, even if this means that TeLa sequence diagram loops are less expressive than TeLa activity diagram loops. More complicated loops can be described using TeLa activity diagrams.

In TeLa sequence diagrams with no so-called "crowns", see Chapter 5, Section 3.1, that is, sequence diagrams which have the RSC property [ChaMatTel96], we can define the projection of the loop scope onto other lifelines as follows:

- If the event corresponding to the emission, resp. reception, of a message is inside the loop, the event corresponding to the reception, resp. emission, of that same message is also inside the loop.
- If a lifeline contains two events which are in the loop, any events vertically between these two events are also in the loop. If the events are located on a lifeline other than the one on which the loop is defined, the vertical space between these two events is considered to define the projection of the scope of the loop onto this lifeline.

The projection of the loop scope is therefore defined by the two valid cuts of the diagram (that is, cuts which bisect all lifelines without bisecting a message) which include the loop scope and the minimum number of events. This can be calculated using Tarjan´s algorithm for calculating simply connected components of a graph [Tar72], see also [HélMai00].

In general, all loops should be placed on tester lifelines. However, in the same way as we allow intra-SUT synchronizations, we also allow loops on SUT lifelines, in case this is more convenient. Recall that though the SUT system dynamic variables are not allowed in TeLa, we have not disallowed SUT component dynamic variables, which can therefore be used as loop counters for such loops.

2.9.1.1.1 Owning component of a loop

The owning component of the loop is the smallest component which contains all the following components as subcomponents:

- the owning component of the originating port of each of the messages emitted inside its scope
- the owning component of the target port of each of the messages received inside its scope
- the owning component of the internal actions inside its scope

Clearly, if these component are not all the same component, tester coordination internal to the component owning the loop is necessary to implement it. In the simple object model, all loops are owned by a base-level component (object).

2.9.2 Syntax

Graphically, a sequence diagram loop is presented as a self-invocation of the method `for(a;b;c)`, which thus defines the scope of the loop construct. `a`, `c` are any textual expressions legal inside internal actions and `b` is a boolean expression; See Fig. 4-16 for an

example in which the test fails if the size of any of the queues 0, 1, 2 and 3 is less than *max_size*.

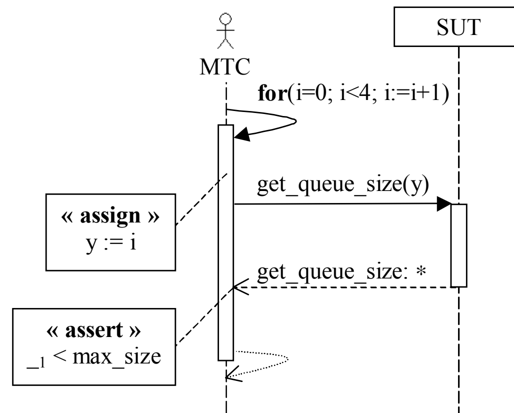


Figure 4-16: An example of the use of the TeLa sequence diagram loop construct, where *i* is a component dynamic variable, *y* is a system dynamic variable and $_1$ is an anonymous variable.

2.9.3 Well-formedness conditions

The sequence diagram loop cannot be used in diagrams which are not RSC (see Chapter 5, Section 3.1) [ChaMatTel96] in order for the notion of projected scope, see above, to be well-defined.

If a system dynamic variable value used in a loop guard is not preceded by a base-level subcomponent name followed by a dot (where this base-level subcomponent is a subcomponent of the loop component), all the base-level subcomponents of the component owning the loop share a common idea of its value.

The scope of a loop on any lifeline (that is including the projected scope) cannot overlap with any other scope (operator, focus bar, calling region). This does not prohibit a loop containing another scope or being contained within another scope, though see the condition below concerning coregions. Applying this rule to calling regions on procedural diagrams (where loops may involve more than two lifelines) eliminates the possibility of causal flows starting and finishing on the lifeline of the loop, with start event inside the scope of the loop but with finish event outside the scope of that loop (or vice versa).

The scope of a loop must be entirely contained in a single sequence diagram.

Though a coregion may occur inside the scope of a loop, a loop may not occur inside the scope of a coregion. This latter restriction ensures that each use of a TeLa sequence-diagram loop is equivalent to a use of a TeLa activity-diagram loop. One apparent way to relax this restriction would be to state that the events in the scope of a loop that falls inside a coregion are ordered w.r.t. each other in the same way as the events in the scope of a focus bar that falls inside a coregion. However, there would be no equivalent TeLa activity diagram for such a TeLa sequence diagram without the introduction of an explicit parallel operator between TeLa activity diagrams, something that we have tried to avoid.

2.9.4 Informal semantics

In the expression `for(a;b;c)`, `a` is any internal action occurring immediately before the scope of the loop on the lifeline, `b` is a boolean expression evaluated on each iteration to see if

the loop body should be executed[5], and `c` is any internal action occurring immediately before the end of the scope of the loop on the lifeline. `for(a;;c)` defines a loop with unspecified termination condition (rather than being infinite).

2.10 Sequence diagram choices

2.10.1 Introduction

The TeLa sequence diagram choice construct provides a user-friendly means of specifying choices in a sequence diagrams. A sequence diagram choice is used to model a situation in which the tester may perform one of several possible message receptions or one of several possible message emissions.

Though semantically all choices are tester choices, due to the semantics by projection onto tester lifelines, syntactically, the choice operator may be situated either on a tester lifeline or on an SUT lifeline.

The TeLa sequence diagram choice makes use of message-anchored TeLa sequence diagram references. Different sequence diagrams are used to denote the different possible ways another primary sequence diagram can be continued. Several messages on the referencing diagram, one for each of the possible continuations, are then used as link messages to the referenced diagrams.

2.10.1.1 THE NOTION OF LOCAL CHOICE

We say that a TeLa sequence diagram choice is *local* if the messages constituting the different alternatives of the choice are represented as being emitted by the same (tester or SUT) lifeline, whether it is the emissions, or the receptions, of these messages which are in the scope of the choice operator. Otherwise we say the choice is *non-local*.

The notion of local choice concerns the semantics before projection onto tester instances. As a consequence, it is not the most useful locality notion for our test description language. Below, we define a notion of locality that concerns the semantics after projection onto tester lifelines.

2.10.1.2 CHOICES BETWEEN MESSAGES RECEIVED ON THE SAME LIFELINE

For a language with a semantics based on the use of complete causal flows, all choices should be viewed as occurring at emission and therefore a choice operator whose scope can only include emissions, and which can therefore only specify local choices, should be sufficient. Hence, the “presentation option” branching construct of UML 1.4 was of this nature.

TeLa, however, is a black-box test description language. The semantics by projection onto tester lifelines means that choices involving alternative messages sent from the SUT to the tester in fact denote choices between the corresponding receptions of these messages at the tester. If the scope of a sequence-diagram choice operator could only include emissions and we wished to describe a choice between reception, at the tester, of messages sent from *different* SUT lifelines, we would be obliged to add SUT-internal synchronization messages, as in Fig. 4-27 and 4-28. While we allow this possibility, we take the view that obliging it,

[5] Note that evaluation of this guard never leads to a verdict since the exit from the loop is guarded by its negation so an event from one of the these two branches is always fireable.

that is, obliging the specification of inter-SUT communication, contradicts the black-box nature of the SUT.

We therefore allow the scope of the sequence-diagram choice operator to contain either message emissions or message receptions but not both. A choice operator whose scope can include message receptions makes non-local choices inevitable but, as already mentioned, this notion of locality is, in any case, of little relevance to our test language. More importantly, such an operator allows greater flexibility for the test designer, even if it leads to diagrams which are perhaps slightly less intuitive, such as those of Figs. 4-22 and 4-24. It also enables us to deal with the problem of denoting alternative responses to a synchronous invocation, providing the means to ensure that both messages are emitted from the bottom of the focus bar, as in Figs. 4-21 and 4-23.

The restriction of TeLa sequence diagram choices to choices between either message receptions or message emissions makes it easier to ensure syntactically that a test description is “controllable”, see Section 2.10.5.1. More complicated choices, such as choices between internal actions, can be described using TeLa activity diagram choices.

2.10.1.3 MORE LOCALITY NOTIONS

The concepts defined in this section are generalised to the TeLa activity diagram choice construct in Section 3.4.1.2.

2.10.1.3.1 Tester-internal, SUT-internal, hybrid and proper choices

We say that a TeLa sequence diagram choice is *tester internal* if all the messages which constitute the alternatives of the choice are both emitted and received by the tester. The notion of tester internal is the same for both the semantics before projection, and that after projection, onto tester instances.

We say that a TeLa sequence diagram choice is *SUT internal* if all the messages which constitute the alternatives of the choice are both emitted and received by the SUT. The notion of SUT internal concerns the semantics before projection onto tester instances. Under the projection semantics, an SUT-internal choice will become a choice between tester actions.

The only SUT-internal choices we allow are local, see next section. We do not assume that tester internal choices are local, though the under-specification that the use of non-local tester internal choice implies would be somewhat unusual.

We say that a choice is *proper* if either all the messages constituting the alternatives of the choice are sent from the SUT to the tester or are all sent from the tester to the SUT. Note that if a choice is neither tester internal nor SUT-internal, it is not necessarily proper. This is since choices may involve both coordination messages and proper messages (that is messages which are not coordination messages). Such improper choices will be called *hybrid* choices.

2.10.1.3.2 Test local and SUT local choices

We say that a choice is *test local* if the messages constituting the different alternatives are either all emitted or all received by a particular tester instance and *test non-local* otherwise. The notion of tester local is the same for the semantics before projection, and that after projection, onto tester instances. However, it does depend on the level of decomposition, c.f. the notion of maximally test local choice of Section 3.4.1.2.

Similarly, we say that a choice is *SUT local* if the messages constituting the different alternatives are represented as being either all emitted or all received by a particular SUT instance. The notion of SUT local concerns the semantics before projection onto tester

instances (though in some systems, information concerning the sending port is available at the tester so the notion may be recoverable after projection). Under the projection semantics, an SUT-local choice will become a choice between tester actions that is not necessarily test local

Since the scope of the sequence diagram choice construct is defined on a single lifeline and the choice is either between emissions or receptions on that lifeline, all choices which can be described with it are either test local or SUT local. In particular, a tester internal choice is necessarily test local, but not necessarily local, while an SUT internal choice is necessarily SUT local, but not necessarily local. A choice which is neither test local nor SUT local cannot be represented in a TeLa sequence diagram and must be represented using a TeLa activity diagram.

2.10.1.3.3 Test local choice: the appropriate locality notion for testing

The appropriate locality notion for the testing context is not the local choice but the test local choice. The use of the semantics by projection onto tester instances reflects the fact that TeLa is a black-box testing language. In such a language, the important property for a choice between messages sent from the SUT is whether or not it is test local; whether or not it is local (and therefore SUT local) is of much less importance in black-box testing. In any real implementation all choices are local; however, test non-local choices between messages received from the SUT will often arise in black-box testing. The property of locality is only really of importance when it coincides with that of test locality, i.e. for tester emissions.

Note that the notion of test local depends on the level of decomposition. See Section 3.4.1.2 for a notion that does not depend on decomposition: maximally test local choice.

In TeLa sequence diagrams, a test local choice is either a local choice between emissions on a tester lifeline or a choice, which may or may not be local, between receptions on a tester lifeline.

As for test non-local choices, the principal interest is in test non-local choices involving messages emitted by the SUT. However, we do not prohibit test non-local choices between messages emitted by the tester; such a choice is to be interpreted as (rather unusual) under-specification of the tester. Such non-local choices will need to be resolved by other mechanisms such as additional synchronizations or a global controller before a test description involving such a choice can be fully executable. However, since TeLa involves implicit verdicts, the same is also true of test non-local choices between tester receptions, see Chapter 5, Section 2.3.3.2 for more details.

The only SUT-internal or hybrid SUT-local choices we allow are those whose scope contains only emissions. In the hybrid case, some of these emissions are of SUT synchronisation messages. In the SUT internal case, they are all of SUT synchronisation messages. Such SUT local choices are subject to additional well-formedness constraints in order to ensure that the tester choice which they denote (i.e. after projection) is deterministic and is a choice between tester receptions.

2.10.1.4 THE NOTION OF TEST CASE AND CHOICES BETWEEN TESTER EMISSIONS

To ensure that a TeLa test description defines a test case, see Section 2.10.5.1, in the enumerated data case, only those sequence-diagram choices that denote a choice between tester receptions can be used. Recall that, due to the semantics by projection, such a choice operator may actually show a choice between SUT emissions. Activity-diagram choices, see Section 3.4, on the other hand, can be between tester emissions, even in the enumerated data case, though the resulting test description cannot denote a test case.

In the non-enumerated data case, sequence-diagram choices between tester emissions can also be used. To ensure that a TeLa test description defines a test case, see Section 2.10.5.1, the messages of such a choice must be guarded.

The meaning of guards is different for receptions from the SUT than for other types of message, as explained in the previous sections concerning message labels and internal actions.

2.10.1.5 INDEFINITE CHOICES

Choices for which the criteria for making the choice are not fully specified are termed *indefinite choices*. All other choices are termed definite choices. We reserve the term non-deterministic choice for an indefinite choice between messages with identical messages names and parameter values.

Unlike the unwieldy semantics of branching in UML 1.4, the interpretation of an indefinite choice in TeLa does not allow concurrent execution of alternative events in an execution where more than one alternative event of a choice is fireable. In both TeLa sequence diagram choices and TeLa activity diagram choices, only one alternative of the choice can ever be executed. Indefinite choices are simply interpreted as a form of under-specification, except for non-deterministic choices between observable actions which are considered invalid. An indefinite choice between controllable actions means that the information in the specification is not sufficient to guarantee that only one of the alternatives is fireable in any execution.

2.10.1.6 OWNING COMPONENT OF A CHOICE

In a choice between tester emissions, the owning component of a choice is the smallest component containing the following subcomponents:

- the owning component of each of the guards on the messages emitted inside its scope (the owning component of the originating port of that message is necessarily a subcomponent),
- the owning component of the originating port of each of the unguarded messages emitted inside its scope.

Clearly, if these components are not all the same base-level component, tester coordination internal to the component owning the choice is necessary to implement it. In fully-guarded choices between emissions (in TeLa, to ensure controllability, we impose the condition that all choices between emissions should be fully-guarded), only the last message of a choice can be unguarded.

If the choice is not test local (and is thus to be interpreted as under-specification of the tester) the owning component of the choice may be the whole tester component. If the choice is test local, then in the simple object model, the owning component of the guard and of the originating port of each of the emitted messages (and therefore of the choice) is the same base-level component, which is represented by the lifeline.

In a choice between tester receptions, the owning component of a choice is the smallest component containing the following subcomponents:

- the owning component of the target port of each of the messages received inside its scope.

Clearly, if these components are not all the same component, tester coordination internal to the component owning the choice is necessary to implement it.

If a choice is not test local, the owning component of the choice may be the whole tester component. If the choice is test local, then in the simple object model, it is owned by a base-level component (object).

2.10.1.7 CHOICES SPECIFIABLE WITH SEQUENCE DIAGRAM CHOICE OPERATOR

We now discuss the different types of choice which can be specified with the TeLa sequence diagram choice operator. We do so from a tester point of view since, thanks to the projection semantics, a TeLa sequence diagram specifies only tester behaviour.

Clearly, for a choice between messages which are all emitted by the tester (so ruling out hybrid SUT-local choices) if the choice is local, it is test local. Such a local choice may or may not be proper and may or may not be SUT local. If the choice is non-local, it is either SUT local and proper or test local and tester internal. Choices between messages emitted by the tester are dealt with in Sections 2.10.2.1 and 2.10.2.2 below.

For a choice between messages which are all received by the tester in the semantics before projection (so ruling out hybrid SUT local choices), the choice may be test local whether or not it is local. If the choice is not tester internal and not SUT local it must be non-local (but test local). We do not allow hybrid choices between messages which are all received by the tester in order to obtain the property of controllability, see below. Therefore, if the choice is non-local and not tester internal, it is necessarily proper. Choices between messages received by the tester are dealt with in Sections 2.10.2.3 and 2.10.2.4 below.

The remaining choices are SUT local and either SUT internal or not proper. We impose the condition that, under the projection semantics, such a choice must denote a deterministic choice between tester receptions. These choices are dealt with in Section 2.10.2.5 below.

As stated above, choices which are neither SUT local nor test local cannot be described using the TeLa sequence-diagram choice operator. If required, they must be described using the TeLa activity-diagram choice operator. However, if such a choice is between messages sent by the SUT, it can be turned into a semantically-equivalent hybrid SUT-local choice or an SUT-internal choice by the addition of intra-SUT synchronisation messages, see Section 2.10.2.5 below, due to the use of the semantics by projection onto tester lifelines.

2.10.2 Syntax

Graphically, the choice between several alternatives is presented as a self-invocation of the method `choice`, which thus defines the scope of the choice construct. As stated above, the connection between the alternative messages of the choice and the consequent ensuing behaviours is made using message-anchored TeLa sequence diagram references.

We now present systematically the different types of choice which it is possible to describe with TeLa sequence diagrams. The examples of TeLa sequence diagram constructs in previous figures of this chapter show only diagram fragments. However, the use of sequence diagram references requires the use of complete diagrams, in order to show how diagrams are named. Thus each TeLa sequence diagram shown in the figures of this section is enclosed in a box with an indexing tab in which its name appears.

2.10.2.1 TEST LOCAL CHOICES BETWEEN MESSAGES SENT BY THE TESTER

Such choices are necessarily local though they may be tester internal (their scope includes only tester coordination messages), hybrid (their scope includes some tester coordination messages) or proper.

2.10.2.1.1 General case

Such choices are described using a sequence-diagram choice operator – whose scope includes the alternative emissions – on the *emitting tester lifeline*. Such a choice is either tester internal

see Fig. 4-17, hybrid see Fig. 4-18, proper and SUT non-local see Fig. 4-19 or proper and SUT local see Fig. 4-20.

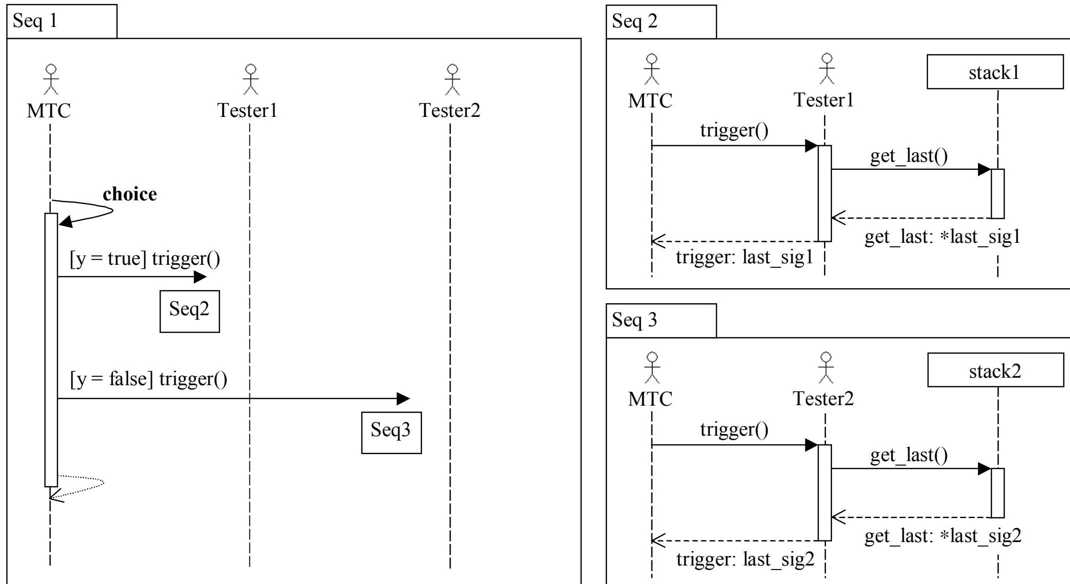


Figure 4-17: A tester-internal, local choice

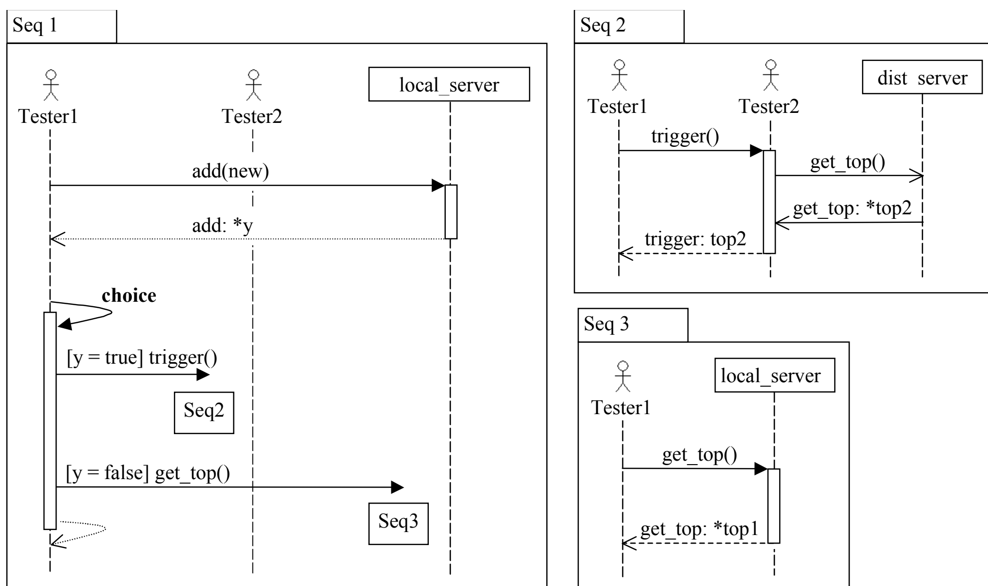


Figure 4-18: A test local, SUT non-local, hybrid choice.

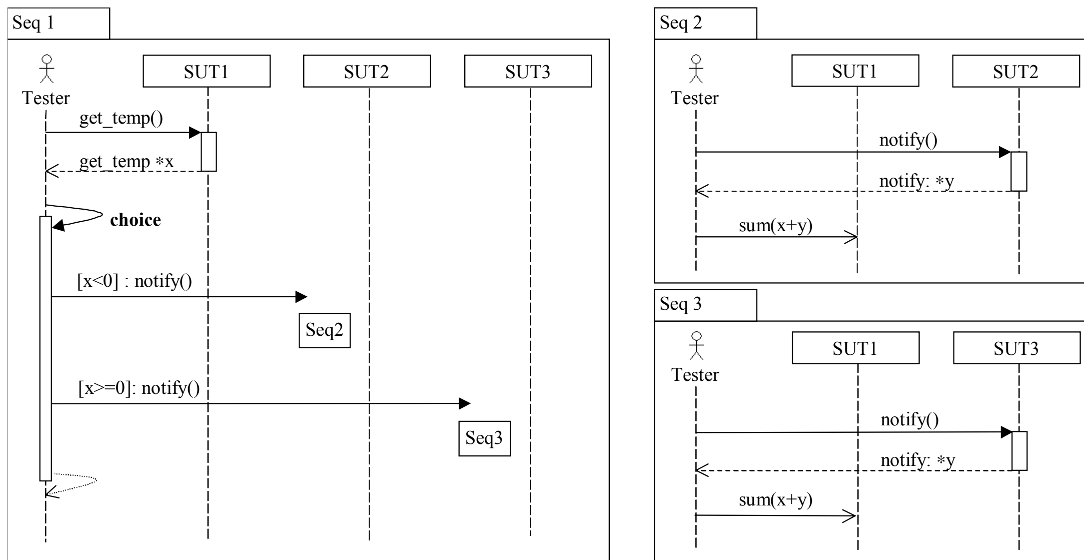


Figure 4-19: A fully-guarded, test local, SUT non-local, proper choice.

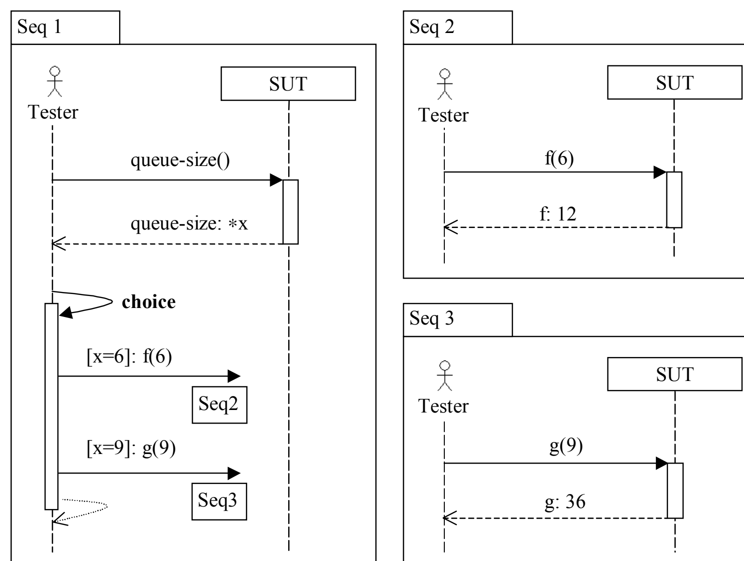


Figure 4-20: A fully-guarded, test-local, SUT-local, proper choice between messages emitted by the tester.

2.10.2.1.2 Alternative for (test) local / SUT local choices between synchronous returns

Due to the projection semantics, such choices can also be described using a sequence-diagram choice operator – whose scope includes the alternative receptions – on the *receiving SUT lifeline*. This is done for presentation reasons in the case of choices between different tester responses to a synchronous invocation from the SUT, see Fig. 4-21. Note that alternative responses to a synchronous invocation are shown as originating at the same point: the bottom of the corresponding focus bar.

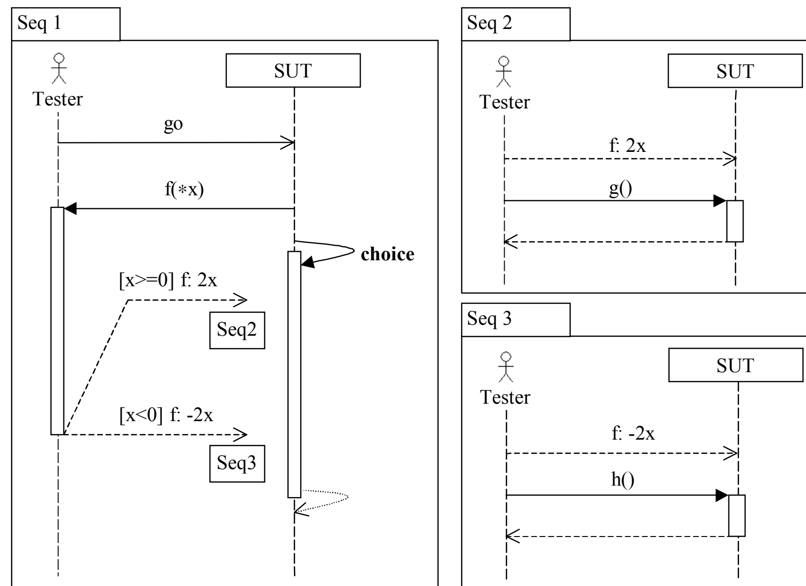


Figure 4-21: A fully-guarded, test local, SUT-local, proper choice between return messages to a synchronous invocation made by the SUT. In this case, for presentation reasons, the choice operator is placed on the SUT lifeline.

2.10.2.2 TEST NON-LOCAL CHOICES BETWEEN MESSAGES RECEIVED BY THE SUT

Such choices are necessarily SUT local, non-local and not tester-internal. They are also proper, since we do not allow hybrid SUT-local choices whose scope contains receptions of SUT synchronization messages. Test non-local, SUT-local, proper choices between messages received by the SUT are to be interpreted as under-specification of tester behaviour.

2.10.2.2.1 (Test) non-local, SUT-local, proper choices

Such choices are described using a sequence-diagram choice operator – whose scope includes the alternative receptions – on the *receiving SUT lifeline*, see Fig. 4-22.

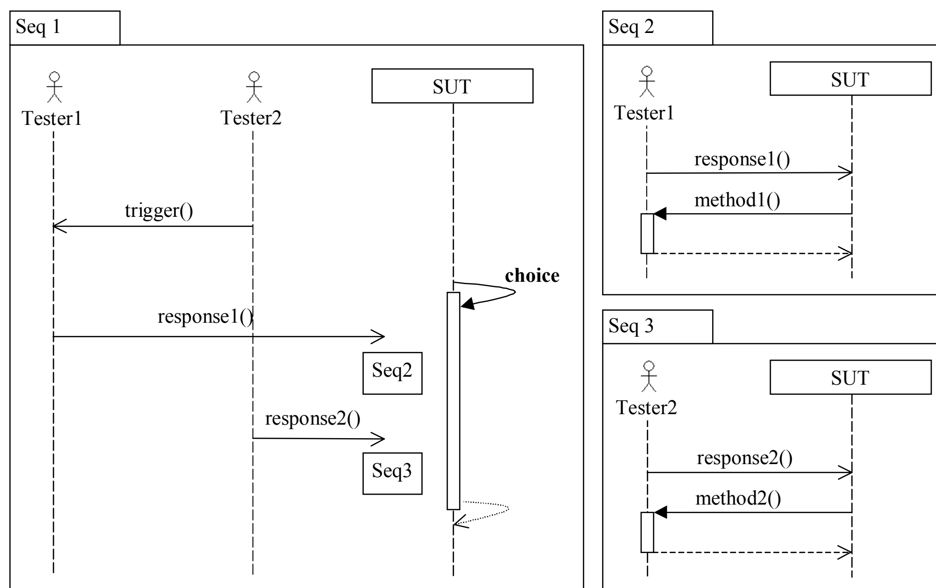


Figure 4-22: A test non-local, SUT-local, proper choice between messages emitted by the tester. Such a diagram is to be interpreted as under-specification of the tester.

2.10.2.3 TEST LOCAL CHOICES BETWEEN MESSAGES RECEIVED BY THE TESTER

If such choices are non-local, they may be tester internal (their scope includes only tester coordination messages) or proper; recall that hybrid test-local choices between messages received by the tester are prohibited in order to ensure controllability. Tester-internal, non-local choices are to be interpreted as a rather unusual under-specification of tester coordination. If they are local, they are either tester internal (in which case, they are covered in Section 2.10.2.1) or they are SUT local and proper.

2.10.2.3.1 Test-local choices not covered in Section 2.10.2.1

Such choices are described using a sequence-diagram choice operator – whose scope includes the alternative receptions – on the *receiving tester lifeline*. We illustrate the (SUT) local, proper case, see Fig. 4-23, and the non-local, proper case, see Fig. 4-24, and the non-local tester-internal case, see Fig 4-25. Notice the crucial difference between the example of Fig. 4-20 and that of Fig. 4-23. In the latter, if the value returned by the SUT is neither 6 nor 9, an implicit fail verdict is obtained. In the former, if the value returned by the SUT is neither 6 nor 9, an implicit inconclusive verdict is obtained.

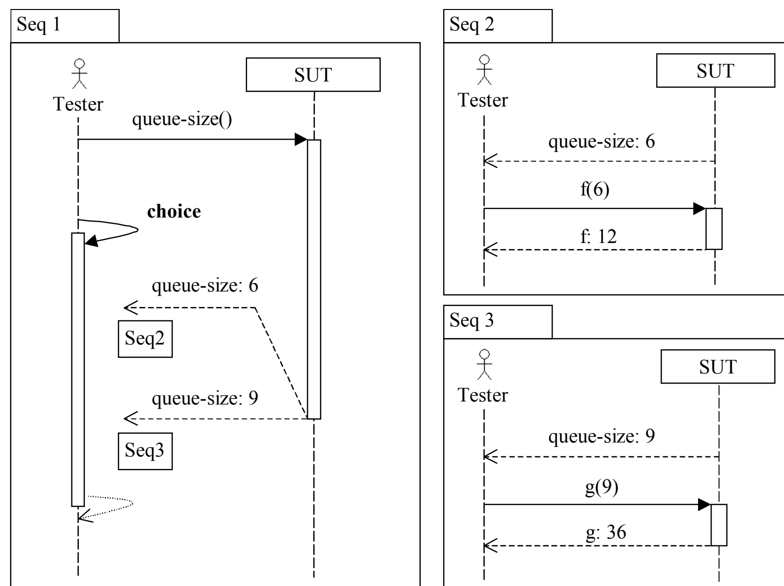


Figure 4-23: A test local, SUT-local, proper choice between messages received by the tester.

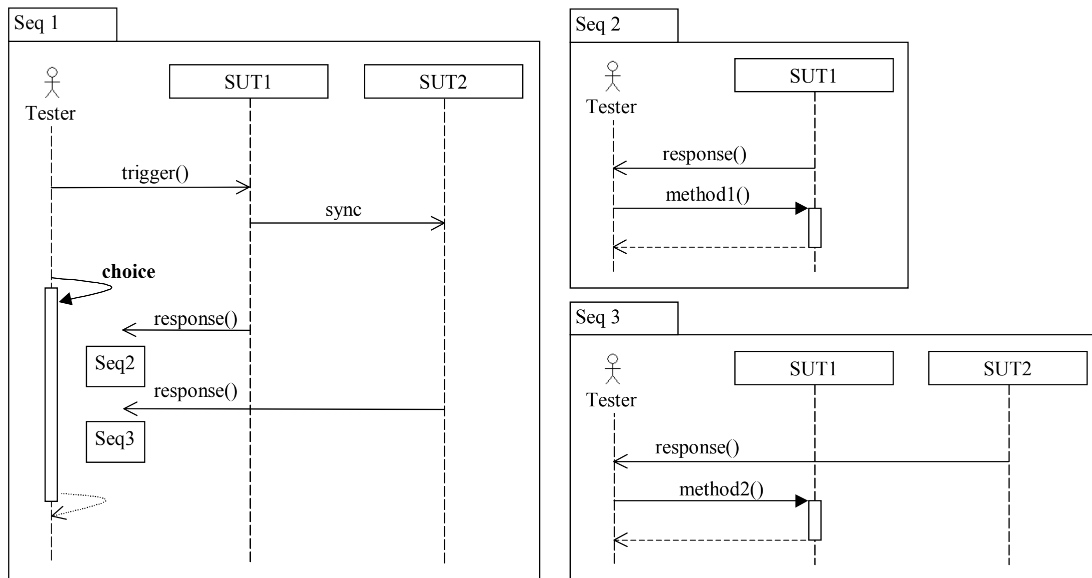


Figure 4-24: A test local, SUT non-local, proper choice between messages received by the tester.

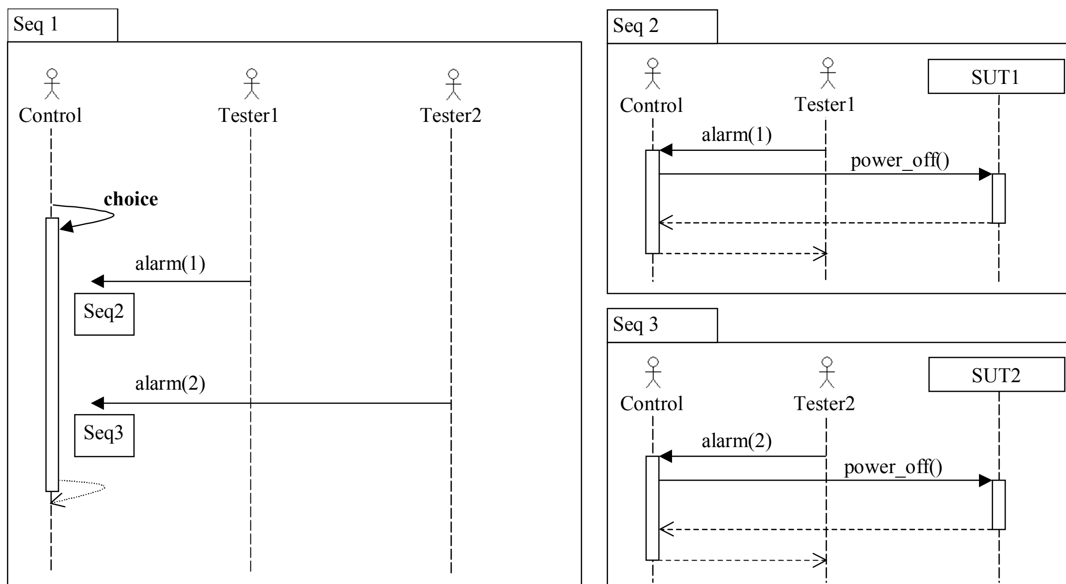


Figure 4-25: A tester-internal, non-local choice. Such a diagram is to be interpreted as a (rather unusual) under-specification of tester coordination.

2.10.2.4 TEST NON-LOCAL CHOICES BETWEEN MESSAGES SENT BY THE SUT

Such choices are necessarily SUT local, local and not tester-internal. They are also proper, since we have stipulated a choice between messages sent by the SUT. Hybrid SUT local choices whose scope contains both emissions of messages to the tester and emissions of SUT synchronization messages are dealt with in case 5.

2.10.2.4.1 Test non-local choices which are (SUT) local

Such choices are described using a sequence-diagram choice operator – whose scope includes the alternative emissions – on the *emitting SUT lifeline*, see Fig 4-26.

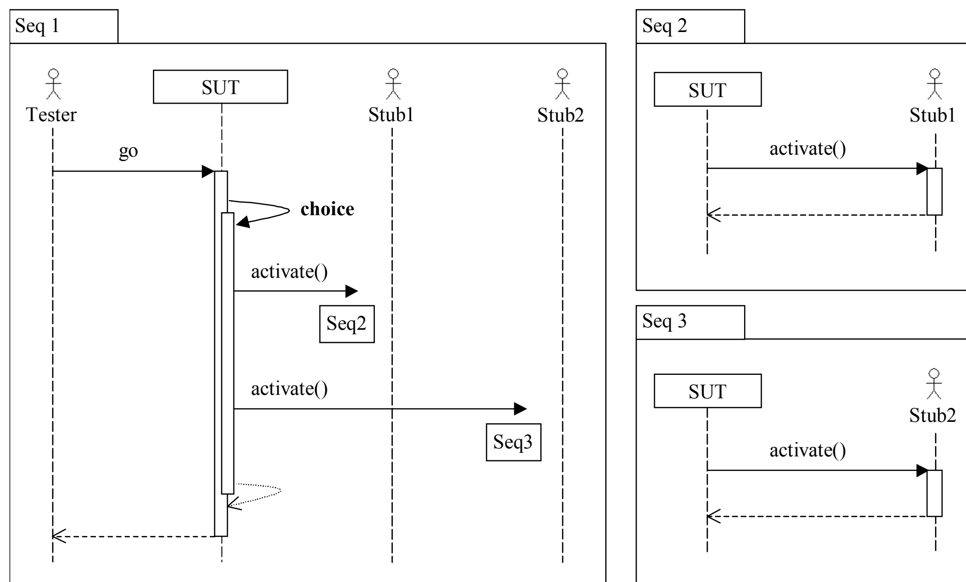


Figure 4-26: A test non-local, SUT-local, proper choice between messages received by the tester.

2.10.2.5 OTHER TEST NON-LOCAL, SUT-LOCAL CHOICES

We also allow certain types of SUT-local choices whose scope includes emissions of SUT synchronisation messages as an extra facility. Such choices are either SUT internal or hybrid. If, under the projection semantics, such a choice becomes a choice that is not between tester receptions or is between tester receptions but is non-deterministic, it is statically incorrect. If, under the projection semantics, such a choice is equivalent to a test local choice, it is preferable to use the test local version.

2.10.2.5.1 SUT-local choices which are SUT-internal or not proper

Such choices are described using a sequence diagram operator – whose scope includes the alternative emissions – on the emitting SUT lifeline. We illustrate both the SUT-internal, SUT-local case, see Fig. 4-27, and the hybrid SUT-local case, see Fig. 4-28.

Fig 4-27 also shows knowledge of the value of system dynamic variable *x* being transferred from SUT3 to SUT1 or SUT2, and knowledge of the system dynamic variables *y* and *z* being transferred in the opposite direction. The intra-SUT transfer of knowledge about *x* ensures that the value known to SUT3 (assumed to have been received earlier from some tester component) is the value expected by the tester as a parameter of the invocation *b* or *c*. Under the projection semantics, the guards on the SUT emissions will become guards on the tester receptions, that is, assertions on expected message parameter values.

In fact, the test description of Fig. 4-28 has exactly the same semantics as the test description of Fig. 4-24.

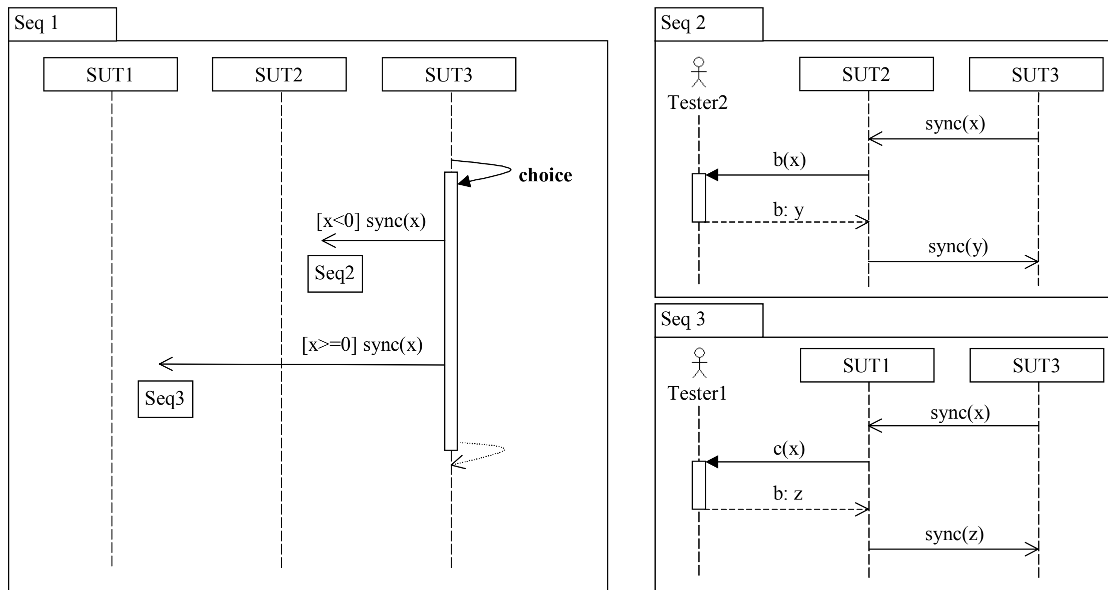


Figure 4-27: An SUT-local, SUT-internal choice denoting a deterministic choice between tester receptions (under the projection semantics).

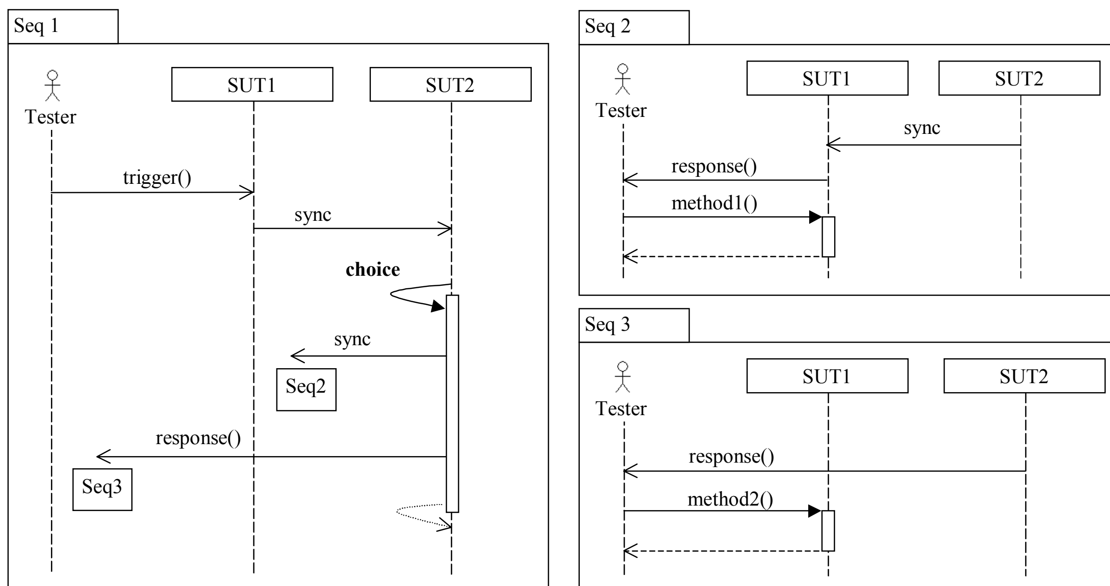


Figure 4-28: An SUT-local, hybrid choice denoting a deterministic choice between tester receptions (under the projection semantics).

2.10.3 Extensions to the basic choice

2.10.3.1 EXPLICIT DEFAULT ALTERNATIVE

An explicit default alternative can be specified to cover the case where none of the other alternatives of a sequence-diagram choice are possible. Though defined locally, the explicit alternative is a global concept and we must impose restrictions on its use, see well-formedness conditions below.

2.10.3.2 SYNTAX OF THE EXPLICIT DEFAULT ALTERNATIVE

In the case of a (test) local choice between guarded messages emitted by the tester, an explicit default alternative is represented graphically as the sending of an asynchronous message labelled `else`[6], from inside the scope of the choice operator on the tester lifeline. It does not in fact represent the sending of any message by the tester, simply the case where none of the guards evaluates to true, see Fig. 4-29 for an example. It may be shown as being received on any SUT lifeline, the meaning is the same.

In the example of Fig. 4-29, if $\neg(x=2y) \wedge \neg(x=0)$ the *ready* message is sent. If x=y=0, the *stop* message is sent. Thus, the TeLa sequence diagram choice is interpreted as a case statement. No inconclusive verdict can be reached in the choice.

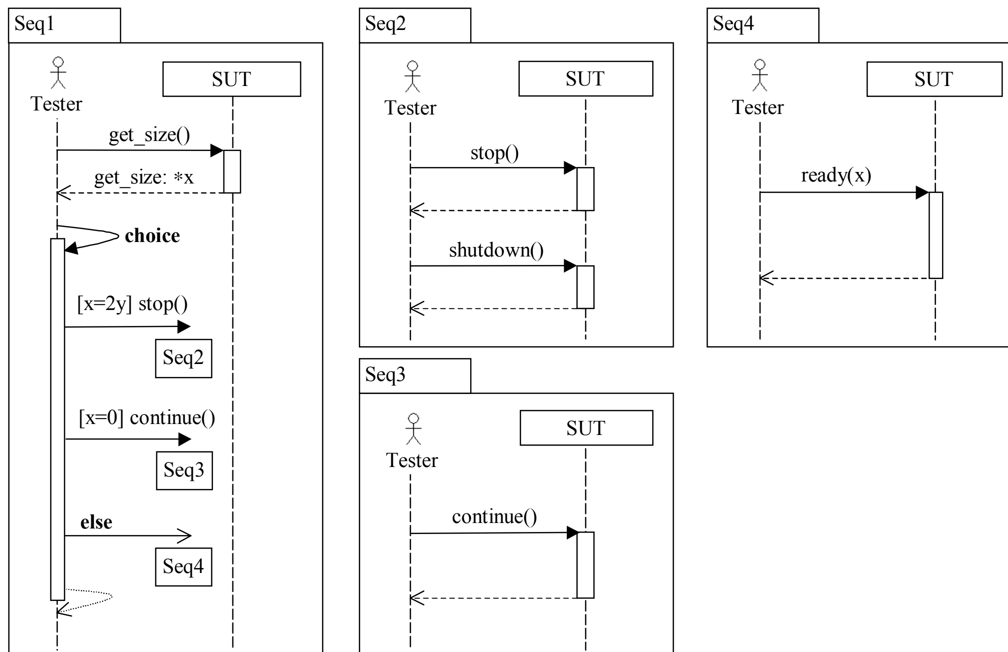


Figure 4-29: A choice between messages emitted by the tester which includes an explicit else alternative.

If an explicit default alternative is not present in a choice between tester emissions, in the case where none of the guards evaluates to true an inconclusive verdict is obtained, see Fig. 4-30 for an example.

In the example of Fig. 4-30, the implicit else alternative means that if $\neg(x=2y) \wedge \neg(x=0)$ an inconclusive verdict is reached.

[6] The word `else` is shorter than the word `otherwise` and is also the word used for this purpose in UML activity diagrams.

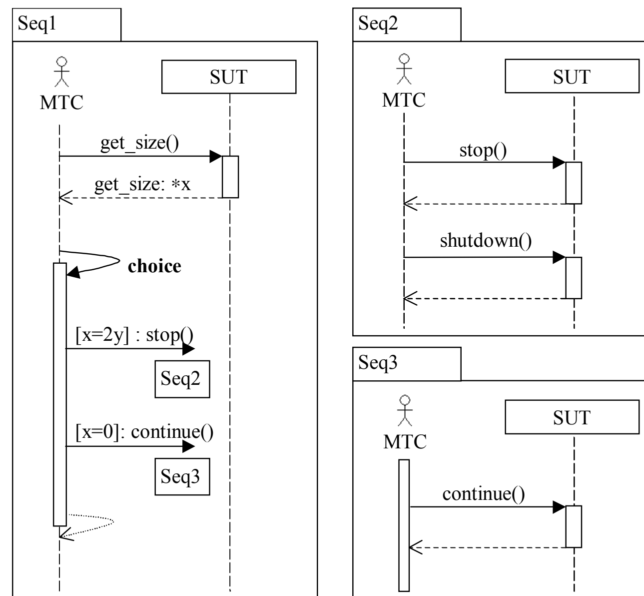


Figure 4-30: A choice between messages emitted by the tester which does not include an explicit else alternative.

In the case of a test local proper choice between messages for which the tester is the receiver, an explicit default alternative is also represented as an asynchronous message labelled `else` received inside the scope of the choice operator on the tester lifeline. It does not necessarily represent the reception of an asynchronous message at the tester, simply of any message that is not among those specified, see Fig. 4-31 for an example. It may be shown as being emitted on any SUT lifeline, the meaning is the same. In Fig 4-31, no fail verdict can be reached in the choice.

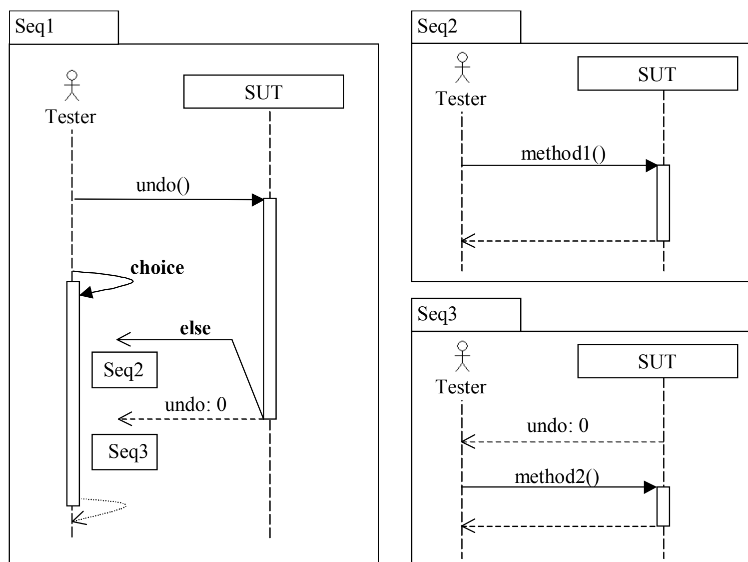


Figure 4-31: A test local choice between messages received by the tester which includes an explicit else alternative.

In an explicit alternative is not present in a choice between tester receptions, in the case where a message not among those specified is received a fail verdict is obtained. Fig. 4-32 shows an example in which the implicit else alternative means that if the SUT returns a value other than 0 or 1, a fail verdict results.

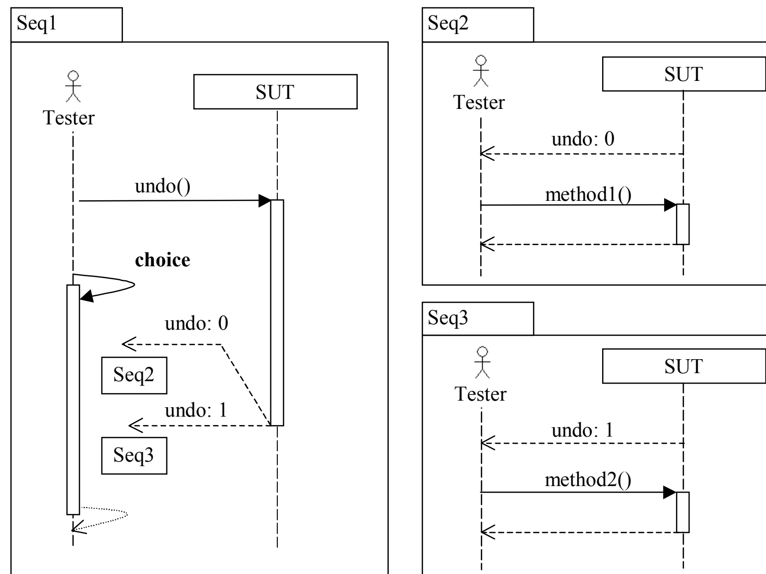


Figure 4-32: A test local choice between messages received by the tester which does not include an explicit else alternative.

In all cases, the asynchronous message labelled “else” only appears on the referencing diagram, not on the referenced diagram.

As already made clear, the most important type of test non-local choices to consider are those between messages for which the tester is the receiver of messages from the SUT. The only statically-correct choices of this type are proper, as in the example of Fig. 4-26. The only statically-correct SUT-internal and hybrid, SUT-local choices also denote proper choices between messages for which the tester is the receiver. An explicit default alternative for these choices takes the form of an asynchronous message labelled `else` sent from inside the scope of the choice operator on the SUT lifeline. It does not necessarily represent the reception of an asynchronous message at the tester, or the sending of any asynchronous message by the SUT, simply the reception at the tester of any message that is not among those specified. It may be shown as being emitted on any SUT lifeline; the meaning is the same.

In these test non-local choices, if an explicit default alternative is not present, then in the case where a message not among those specified is received, a fail verdict is obtained.

However, test non-local choices between messages for which the tester is the emitter are also possible. Such choices are necessarily proper and represent under-specification of the tester. An explicit default alternative for these choices takes the form of an asynchronous message labelled `else` received inside the scope of the choice operator on the SUT lifeline. It does not in fact represent the sending of any message by the tester, simply the case where none of the guards evaluates to true. It may be shown as being received on any SUT lifeline; the meaning is the same.

If an explicit default alternative is not present, in the case where none of the guards evaluates to true, an inconclusive verdict is obtained.

Notice that for test non-local choices, in both the explicit and implicit default alternative cases, the implementation of the default alternative requires the resolution of this non-locality, e.g. by synchronizations or an implicit global controller, see Chapter 5, Section 2.3.3.2.

2.10.3.3 ALTERNATIVE LEADING TO AN EXPLICIT VERDICT

A message of an alternative can be shown as explicitly leading to the termination of the behaviour and the derivation of a verdict.

In a choice between messages emitted by the tester, an explicit verdict is necessarily a local inconclusive verdict (arrived at by the sender). We assume that in such a choice, the explicit default alternative is also used and that it does not also lead to an explicit inconclusive verdict, otherwise the choice is equivalent to one without an explicit verdict or explicit default alternative.

In a choice between messages received by the tester from the SUT, an explicit verdict can be a local fail verdict or a local inconclusive verdict (arrived at by the receiver). In the case were the explicit fail verdict is used, we assume that the explicit default alternative is also used and that it does not also lead to an explicit fail verdict, otherwise the choice is equivalent to one without an explicit verdict or explicit default alternative. In the case where the explicit inconclusive verdict is used, the explicit default alternative may, or may not, be used; if it is not used, the implicit fail verdict still exists.

As for implicit verdicts, we do not explicitly specify the communication of the local verdict to the component responsible for the global verdict. Nor do we need to specify test component termination. Also, as for implicit verdicts, an inconclusive verdict is not definitive since it can still degrade into a fail verdict.

An explicit pass verdict is superfluous since it is implicit on reaching the end of any specified behaviour which does not end in an explicit inconclusive or fail verdict.

2.10.3.4 SYNTAX OF ALTERNATIVE LEADING TO EXPLICIT VERDICT

Graphically, alternatives leading to explicit verdicts in a choice operator use a syntax which is similar to that of the message-anchored sequence-diagram reference. The construct has the same form as such a reference except that one of the keywords *fail* or *inconclusive* is used in the place of a TeLa sequence diagram name.

An example involving a choice between messages received by the tester is given in Fig. 4-33. In this example involving an explicit inconclusive verdict, since the explicit default alternative has not been used, the default verdict still exists: if a value different to 0 or 1 is received the verdict is fail.

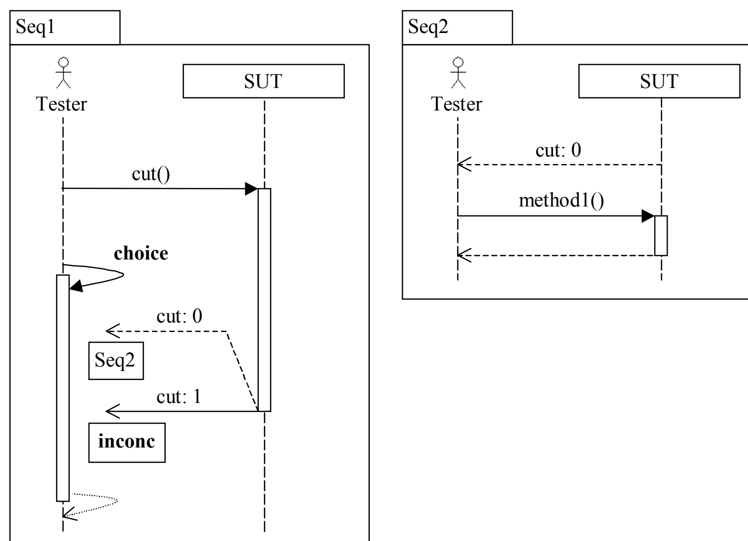


Figure 4-33: Choice between messages received by the tester with alternative leading to an explicit verdict.

An example involving a choice between messages sent by the tester is given in Fig. 4-34. In this example, if “done” is true, the value 0 is returned and the behaviour terminates with an inconclusive verdict. If “done” is not true and “dusted” is true, the value 1 is returned and the behaviour continues in diagram Seq2. Finally, if neither “done” nor “dusted” is true, the behaviour continues in diagram Seq3 (in which case, the returned value is represented in diagram Seq3 but not in diagram Seq1).

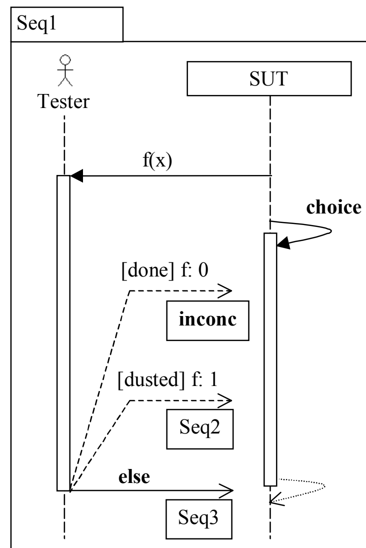


Figure 4-34: Choice between messages emitted by the tester with an alternative leading to an explicit verdict and an explicit default alternative. (Diagrams Seq2 and Seq3 not shown)

2.10.3.5 ALTERNATIVE(S) CONTINUED ON THE SAME DIAGRAM

A presentation option for the choice operator is to represent the behaviour of one of the alternatives in the same diagram where the choice is made, thus reducing the number of diagrams needed. In the case where there is only one acceptable behaviour, this also provides a means of clearly distinguishing the pass case from the explicitly-specified inconclusive or fail verdict case.

We can generalise the alternative continued on the same diagram to a join of alternatives in the case where a choice contains explicit verdicts. In this construct, all the alternatives which do not lead to an explicit (or implicit) verdict have the same continuation, that is, share the same behaviour after the choice.

This construct is similar, but not identical to the MSC 2000 *Esc* in-line operator and to the UML 2.0 interactions *Break* interaction operator.

2.10.3.6 SYNTAX OF ALTERNATIVE CONTINUED ON THE SAME DIAGRAM

Graphically, this presentation option is realised by representing one of the messages in the scope of the choice using an arrow stretching from the emitting lifeline to the receiving instance, i.e. without having an associated TeLa sequence diagram reference. The continuation of the behaviour for that alternative is then the behaviour shown in the same diagram below the choice operator. See Fig. 4-35 for an example.

For the generalisation to a join of alternatives, all the alternatives that do not lead to an explicit verdict are shown using arrows which reach all the way from the emitting instance to the receiving lifeline, i.e. without having an associated TeLa sequence diagram reference (e.g. see the return messages with value 1 and 2 in Fig. 4-36). The continuation of the behaviour

for these alternatives is then the behaviour shown in the same diagram below the choice operator.

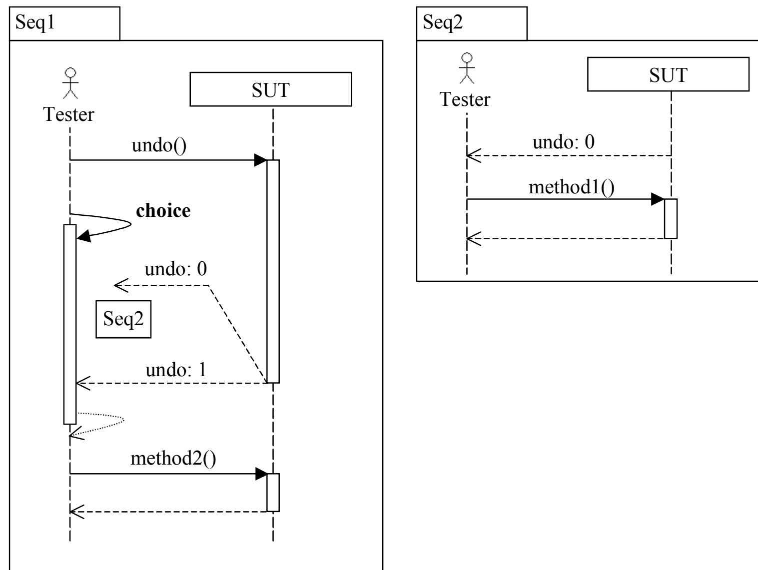


Figure 4-35: Choice with alternative continued on the same diagram (same behaviour as that represented in Figure 4-30).

In the example of Fig. 4-36, if the SUT returns the value 0 an inconclusive verdict is obtained, if the SUT returns the value 1 or 2 (the behaviour is the same), method2() is invoked, otherwise a fail verdict is reached.

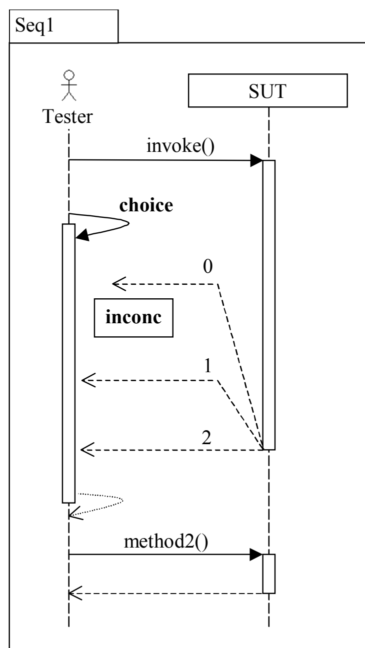


Figure 4-36: Choice with several alternatives continued on the same diagram (thus involving a join).

2.10.3.7 CHOICE WITH BLOCK STRUCTURE

We also allow the use of the choice operator with a block structure. For certain simple cases, this enables us to represent on one diagram, choices which would otherwise have to be represented using several linked diagrams. It also enables us to represent, using a one-tier

scenario structure, behaviour which would otherwise require a two-tier scenario structure, i.e. use of TeLa activity diagrams. For example, it enables us to describe some choices between internal actions.

On the other hand, as for the sequence diagram loop construct, the global meaning of this locally-defined construct is not necessarily clear, except for procedural sequence diagrams (diagrams with a single flow of control and using only synchronous invocations). In particular, if the scope of a block is only defined on one lifeline, how do we know its scope on other lifelines when asynchronous invocations do not necessarily have an associated focus bar and there may be more than one control flow? How do we detect illegal casual chains, i.e. chains of events starting and finishing on the lifeline of the choice, with start event in the scope of a block but with finish event outside the scope of that block (or vice versa)?

One way to solve this problem is to explicitly specify the global scope by delimiting a region encompassing several lifelines as the scope of each block (of course, no messages may cross this scope). We adopt a slightly more complicated solution that involves recursively defining the notion of projection of the block scope on other instances, see below. This second solution is more in keeping with our desire to use operators with local scope in TeLa sequence diagrams.

In TeLa sequence diagrams with no so-called “crowns”, see Chapter 5, Section 3.1, that is, sequence diagrams which have the RSC property [ChaMatTel96], we can define the projection of the block scope onto other lifelines as follows:

- If the event corresponding to the emission, resp. reception, of a message is inside the scope of a block, the event corresponding to the reception, resp. emission, of that same message is also inside the scope of that block.
- If a lifeline contains two events which are in the scope of a block, any events vertically between these two events are also in the scope of that block. If the events are located on a lifeline other than the one on which the block is defined, the vertical space between these two events is considered to define the projection of the scope of the block onto this lifeline.

The projection of the block scope is therefore defined by the two valid cuts of the diagram (that is, cuts which bisect all lifelines without bisecting a message) that include the block scope and the minimum number of events. This can be calculated using Tarjan´s algorithm for calculating simply connected components of a graph [Tar72], see also [HélMai00].

2.10.3.8 SYNTAX OF CHOICE WITH BLOCK STRUCTURE

Graphically, a choice with block structure is represented by using two or more self-invocations of the method `block` inside the scope of the choice operator as shown in Fig. 4-37.

Unfortunately, there is no suitable syntax for expressing the very common case of a choice with block structure in which the choice is based on different return values of a synchronous invocation. Such a choice must be described using the preceding constructs.

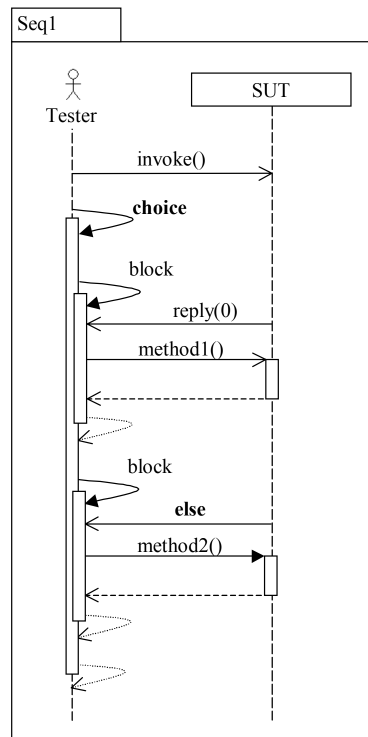


Figure 4-37: Choice with block structure (and explicit default alternative).

2.10.4 Well-formedness

2.10.4.1 ENSURING (ESSENTIAL) CONTROLLABILITY

The scope of any TeLa sequence-diagram must contain either a set of emissions on a lifeline or a set of receptions on a lifeline and must be entirely contained within a single TeLa sequence diagram.

All choices between tester emissions must be fully-guarded, that is, each message emitted in the scope of the choice operator, except possibly the last, must have a guard.

Test non-local choices between messages received by the tester must be proper.

The scope of hybrid SUT-local choices and of SUT-internal choices must contain only message emissions (in the hybrid case, some of these emissions are of SUT synchronisation messages, in the SUT-internal case, they are all of SUT synchronisation messages).

The tester choice denoted by a hybrid SUT local choice or an SUT-internal choice under the projection semantics must be a deterministic choice between tester receptions.

2.10.4.2 EXPLICIT DEFAULT ALTERNATIVE AND EXPLICIT VERDICT

Without any extra assumptions, the explicit default alternative is only allowed in coherent parallel test cases, see Sections 2.10.5.1 and 3.4.4.1. That is, if a choice between tester emissions involves a default alternative, any events concurrent with the entire choice must be controllable events. Similarly, it a proper choice between tester receptions, or a choice between SUT emissions, involves a default alternative, any events concurrent with the entire choice must be observable events. With assumptions about priorities between the different types of message, these restrictions can be relaxed, see Section 3.4.4.2.

Explicit verdicts, except the explicit inconclusive verdict in a proper choice between messages received by the tester, can only be used in coherent parallel test cases. This follows from the fact that use of such an explicit verdict also implies use of the explicit default alternative.

2.10.4.3 RELATION WITH OTHER SCOPES

The scope of a choice must be entirely contained in a single TeLa sequence diagram

The scope of a choice cannot overlap with any other scope (operator, focus bar, call region). Furthermore, the scope of a choice, other than a choice with block structure, may not contain any other scope. This does not prohibit a choice from being contained within another scope (except that of a choice!). However, the following restriction limits the choices that can be contained in a coregion.

2.10.4.4 CONCURRENT CHOICES

A very limited amount of concurrency is permitted between sequence-diagram choices. Any concurrent choices must satisfy the following restrictions:

- Only one of the concurrent choices has one, or more, alternatives continued on the same diagram and only one of the concurrent choices uses sequence diagram references other than explicit verdicts. These restrictions ensure that the continuation of the behaviour is well-defined. Concerning the second of these restrictions, in fact, due to the following restriction, the only explicit verdicts that can be used are inconclusive verdicts on reception at the tester of a message from the SUT.
- None of the concurrent choices uses the explicit default alternative. This restriction ensures that the one-tier test description is equivalent to a two-tier one; in two-tier TeLa test descriptions the default alternative is global.
- Either all of the concurrent choices involve tester emissions or all of the concurrent choices involve tester receptions (to ensure concurrent essential controllability, see Section 2.10.5.1).

2.10.4.5 DYNAMIC VARIABLES IN MESSAGE GUARDS

The rules concerning the value of variables used in guards are the same as those for assertion internal actions. That is, a variable may be preceded by a base-level component name followed by a dot. In this case, the intended value is that held by the specified component. Otherwise, it is assumed that all the base-level subcomponents of the owning component share a common view of its value.

2.10.4.6 CONDITIONS SPECIFIC TO THE CHOICE WITH BLOCK STRUCTURE

The choice with block structure cannot be used in diagrams which are not RSC (see Chapter 5, Section 3.1 and [ChaMatTel96]) in order for the notion of projected scope, see above, to be well-defined.

The scope of a choice block on any lifeline (that is including the projected scope) cannot overlap with any other scope (operator, focus bar, call region). A choice block can contain another scope or the entire choice with block structure may be contained within another scope. Applying this rule to calling regions on procedural diagrams (where loops may involve more than two lifelines) eliminates the possibility of causal flows starting and finishing on the lifeline of the choice, with start event inside the scope of a block but with finish event outside the scope of that block (or vice versa).

2.10.5 Informal semantics

The TeLa sequence diagram choice operator specifies several alternative behaviours. The events in the scope of the choice (the alternatives) are either all emission events or all receptions events. Other events can be concurrent with the choice, though concurrent choices are subject to the restrictions given above in the well-formedness conditions.

Whether the scope of the choice operator contains emission or reception events, the meaning is that only one of the messages concerned will be emitted and subsequently received and the behaviour will then continue as defined for that alternative.

In the case of controllable actions, the value of the static and dynamic variables decides which of the different alternatives of the choice is to be fired. In the case of observable actions, the name of the message and the value of the anonymous variables, both provided by the SUT, decide which of the alternatives of the choice is to be fired. In an executable test description, only one alternative of a choice can be fireable at run-time.

2.10.5.1 DETERMINISM AND CONTROLLABILITY

A TeLa choice is said to be indefinite if at least one of the alternatives of the choice (in the event structure semantics, one of the minimal events in the corresponding configuration) is not guarded or if the guards are not mutually exclusive. It is said to be non-deterministic if it is indefinite and, in addition, the message names on the transitions concerned are identical.

2.10.5.1.1 No indefiniteness in choices between tester emissions

A choice between controllable events, such as a choice between messages emitted by the tester, is a choice which is to be resolved by the tester. An indefinite choice between controllable actions means that the information in the specification is not sufficient to guarantee that, at any time and in any execution, the tester has only one fireable alternative. For the test description to represent a test case, such choices can be neither non-deterministic nor indefinite since a test description with such an indefinite choice cannot be fully executable. Unfortunately, with any non-trivial data language, demonstrating that a set of guards are mutually exclusive is an undecidable problem. Moreover, we do not want to prohibit the possibility of overlapping guards on controllable events in all test descriptions, one reason being that we may wish to deal with this aspect at execution time.

We choose to resolve this problem differently for TeLa sequence diagram choices and TeLa activity diagram choices. For the latter, we provide a mechanism to explicitly assign priorities to alternatives in order to turn indefinite choices between controllable events into definite choices. Not doing so leaves open the possibility of dealing with indefinite choices at execution time (if only by signalling a run-time error). For TeLa sequence diagram choices, with which we are concerned in this section, we simply treat a choice between tester emissions as a case statement, that is, we suppose there is an implicit priority defined by vertical position. Evidently, this also eliminates the possibility of non-determinism arising in such choices.

2.10.5.1.2 No non-determinism in proper choices between tester receptions

A choice between between observable events, i.e. a choice between messages received by the tester from its environment, i.e. the SUT, is a choice which is to be resolved by this environment. Recall that the guards on such messages do not have the same interpretation as the guards on tester emissions, since they cannot imply a description of some aspect of the SUT internal state. The tester’s knowledge of the SUT internal state is limited to that which can be obtained via the “SUT component interface”. A description of the aspects of the SUT

internal state on which the emission of each of the alternative messages depends cannot form part of the test. Similarly, choices between different unguarded actions simply indicate that the SUT can send any one of several messages. However, a proper choice between receptions that involves two messages with identical labels and overlapping guards is a choice which will not necessarily be fully resolved by the SUT and must be disallowed. Therefore, proper choices between receptions can be indefinite but not non-deterministic. Similar considerations apply to choices between SUT emissions since, under the projection semantics, they also denote choices between tester receptions.

In a similar way to the case for indefiniteness of emission choices, we choose to resolve the problem of non-determinism of proper choices between receptions differently for TeLa sequence diagram choices and TeLa activity diagram choices. For the latter, we provide a mechanism to explicitly assign priorities to alternatives. Not doing so leaves open the possibility of dealing with non-deterministic choices at execution time (if only by signalling a run-time error). For TeLa sequence diagram choices, with which we are concerned in this section, we simply treat a choice between tester receptions as a case statement, that is, we suppose there is an implicit priority defined by vertical position.

2.10.5.1.3 TeLa sequence diagram choices that ensure (essential) controllability

By definition, then, no TeLa sequence diagram choice between tester emissions can be indefinite and no TeLa sequence diagram proper choice between tester receptions can be non-deterministic.

First, let us assume that the test description is such that, for any TeLa sequence diagram choice, no events are concurrent with the emission or reception events of the choice. Clearly, in this case if all choices of a one-tier TeLa test description involving messages received by the tester are proper, the test description is deterministic and *controllable* and therefore defines a *centralisable test case*. With the restrictions imposed on SUT local choices which are hybrid or SUT-internal, these choices also denote deterministic choices between receptions at the tester of messages from the SUT, so test descriptions which include such choices also define test cases. Thus, to ensure controllability, we only need to prohibit hybrid test local choices between receptions since these are choices involving both controllable and observable actions.

Well-formed choices between tester emissions should be fully guarded, that is, each of the alternative messages, except possibly the last, should have a guard. This is since if one alternative is not fully guarded, the meaning is that it has the guard *true* and none of the alternatives below it will ever be explored. Similarly, in choices between tester receptions in the non-enumerated data case, if more than one reception of the choice has the same message name and number of parameters, all such messages should be guarded, except possibly the last.

Now, let us first assume that the TeLa test description is such that there are events which are concurrent with those of a sequence-diagram choice. Given the nature of sequence diagram choices explained above (and given that hybrid test local choices between receptions are prohibited) there are no choices between observable and controllable events. Hence, even a test description in which there are events that are concurrent with a choice is still *essentially controllable* (and, if minimally deterministic, defines a *parallel test case*).

An essentially-controllable test description is said to be *properly concurrently controllable* (and, if minimally deterministic, to define a *externally coherent parallel test case*) if no event labelled by a proper controllable action is concurrent with an event labelled by an observable action. Necessary requirements for this are:

- for all hybrid or proper choices between tester emissions, none of the concurrent events are labelled by observable actions,
- for all proper choices between tester receptions, none of the concurrent events are labelled by proper controllable actions.

An essentially-controllable test description is said to be *concurrently controllable* (and, if minimally deterministic, to define a *coherent parallel test case*) if no event labelled by a controllable action is concurrent with an event labelled by an observable action. Necessary requirements for this are as above, with the first requirement strengthened to all choices between tester emissions (i.e. including tester-internal choices) and the second requirement strengthened to all controllable actions (including tester-internal actions).

An essentially controllable test description is said to be *controllable* (and, if minimally deterministic, to define a *centralisable test case*) if no event labelled by a controllable action is concurrent with any other event. Necessary requirements for this are as above with the first requirement strengthened to exclude all concurrent events.

More details about non-determinism and controllability in TeLa can be found in Chapter 5, Section 2.2.

2.10.5.2 VERDICTS

2.10.5.2.1 Implicit verdicts

In this section, we only discuss implicit verdicts concerning messages in the scope of a choice. Implicit verdicts concerning messages that do not fall in the scope of a choice were discussed in Section 2.2.4.2 and 2.2.4.3.

For a choice between receptions of messages by the tester from the SUT, if an explicit default alternative is not present, an implicit default alternative leading to a fail verdict is inferred. An implicit verdict is well-defined whether or not there are events which are concurrent with the choice. Recall that the allowed hybrid SUT-local choices and SUT-internal choices also denote such proper tester choices. The implicit fail verdict is local but also global, since fail is a universal property.

For a choice between messages emitted by the tester, if an explicit default alternative is not present, an implicit default alternative leading to an inconclusive verdict is inferred. An implicit verdict is well-defined whether or not there are events which are concurrent with the choice. The implicit inconclusive verdict is local to the owning component of the choice; it is not definitive and can degrade into a fail verdict due to activity which is concurrent with the activity of the component owning the choice.

2.10.5.2.2 Explicit verdicts

Concerning explicit verdicts, as stated above, the only explicit verdict which does not also involve the use of the explicit default alternative (and therefore the restrictions applicable to the use of this alternative) is the explicit inconclusive verdict on tester receptions. This verdict is local to the owning component of the target port of the message. All other explicit verdicts are local to the owning component of the choice.

3 TeLa activity diagrams and 2-tier scenario structures

The syntax of two-tier TeLa test descriptions is based on that of UML 1.4 activity diagrams. However, the semantics is completely different to UML 1.4 activity diagram semantics. A two-tier scenario structure comprises a TeLa activity diagram in which each node contains either a set of sequence-diagram level parameter (static-variable) declarations and a TeLa diagram reference to an elementary sequence diagram or a symbol indicating the empty sequence diagram. The empty sequence diagram is a sequence diagram with no events and is simply used to keep the notation compatible with the UML activity diagram notation.

The full language using two-tier scenarios is very rich and can consequently describe many behaviours which cannot easily be interpreted as test cases. Though we wish to allow far more general descriptions than those which describe executable test cases, we still need to impose certain restrictions in order for verdicts to be unambiguously defined. In particular, for a test description to be well-defined, we impose *minimal determinism* together with the condition that any remaining non-determinism is either resolved on a controllable action or leads to successful termination (implicit pass verdict) or the same explicit verdict on both branches, and we impose conditions on choices involving guarded controllable actions, see the notion of *normal form*. We view the most general type of legal test description as analogous to the “test graphs” of [JarJer02], [Jer02].

In order for a test graph to define a test case, we impose further restrictions, in particular, those concerning *controllability*. However, even a test description we describe as a test case is not fully executable since:

- We may wish to leave implicit the resolution of choices with alternatives guarded by overlapping guards, in order to be able to choose how this is to be implemented (e.g. by using some execution-level mechanism). In consequence, we do not oblige the use of priorities between alternatives of a choice, in the case where guards may be overlapping.
- We may wish to leave implicit the resolution of synchronizations, in order to be able to choose how this is to be implemented (by using a centralised implementation, by using a global controller, by using explicit synchronization messages, via shared variables, etc.). In consequence we do not oblige all choices to be test local nor all guards, assignments, assertions to be local to a base-level component.

We still view the language of well-defined test descriptions as a test description language, rather than a test objective description language. In a language for describing test objectives, further abstraction is required, e.g. wildcards on method names in message arrow labels, wildcards on values in parameters of messages sent by the tester etc., see Chapter 6, Section 1.2.3.5.

3.1 Overview of syntactic elements

3.1.1 UML 1.4 activity-diagram syntax used

The features of UML activity diagrams that are used in TeLa activity diagrams are as follows:

- *Action states*: in TeLa, the “action” of an “action state” (we also use the term *node*) is either the name of a TeLa elementary sequence diagram or a dash, representing the empty diagram, i.e. the diagram containing no events. Nodes with empty sequence diagrams are

used in loops with exit conditions and in consecutive choices in order to respect UML activity diagram syntax.

- *Transitions*: in TeLa, transitions between action states denote weak sequential composition of the corresponding sequence diagrams.
- *Decisions*: in TeLa, as in UML, the diamond-shape decision syntax is used for both choice and "merge"[7], with only the choice case admitting guards. However, in TeLa, guards may only be attached to the lines emanating from the diamond shape in a choice (not to the incoming lines to a diamond shape). The structure of the activity diagram choice guards is discussed in Section 3.4.2.1; A choice may lead straight to another choice via an empty sequence diagram.
- *Initial (pseudo)state*: in TeLa the state-diagram initial state symbol is used to denote the starting point of a 2-tier scenario.
- *Loops*: in TeLa we allow transitions to form loops. By using an empty sequence diagram, such loops may commence with a choice, allowing us to model loops with exit conditions.
- *Synchronization bars*: in TeLa, a limited use of is made of state-diagram synchronization bars; they are only used with one source state and one destination state. This syntax is used to denote strong sequential composition of all tester lifelines of the TeLa elementary sequence diagram referenced in the "source" state and the TeLa elementary sequence diagram referenced in the "destination" state, in the case where the default weak sequential composition is not required, e.g. on composition of complete test cases.

3.1.2 UML 1.4 activity-diagram syntax not used

The principal features of UML activity diagrams which are not used in sequence connector diagrams are as follows:

- *Concurrency*: for simplicity, we only allow concurrency at the sequence diagram level (via the coregion construct and the inherent concurrency between instances). As for MSCs [ITU-T99], the sequence diagram concatenation mechanism does not imply a synchronisation point so any concurrency between events on different instances is not truncated at concatenation points.
- *Nested graphs (subactivity states)*: nodes which contain subgraphs are currently disallowed purely on the grounds of simplicity.
- *Final (psuedo)state*: the state diagram final state symbol is not used in TeLa since only one final state is allowed in UML activity diagrams (leading to readability problems).
- *Organisation of responsibilities (swimlanes)*: this concept is not useful in this context.
- *States with input and output (object flow)*: this concept is not useful in this context.
- *Control indications (signal sending & receipt, deferred event)*: this concept is not useful in this context.

---

[7] The use of the term "merge" in UML 1.4 reflects the fact that in UML 1.4 activity diagrams, as in UML 1.4 sequence diagrams, in the presence of overlapping guards, the same operator denoted choice or concurrency depending on data values. From here on we will use the term "join".

3.1.3 Elementary sequence diagrams

A TeLa elementary sequence diagram is a TeLa sequence diagram satisfying the following restrictions:

- it contains no sequence-diagram choice operators
- it contains no sequence-diagram loop operators.
- it contains no sequence-diagram references
- it contains no assertion internal actions and no guards on tester emissions if these fall within the scope of an activity-diagram choice.

and with the following extensions:

- it may contain explicit verdicts, using the message-anchored sequence-diagram reference style notation, subject to the restrictions given in Section 3.5.3 below.

This is an extension since these verdicts have only been defined so far for an alternative of a sequence diagram choice.

3.2 TeLa activity-diagram reference

3.2.1 Introduction

The activity-diagram reference is used to reference a TeLa elementary sequence diagram from a node of a TeLa activity diagram.

3.2.2 Syntax

The syntax consists of simply placing the name of the referenced TeLa sequence diagram in the required node of the TeLa activity diagram. See Fig. 4-38 for an example.

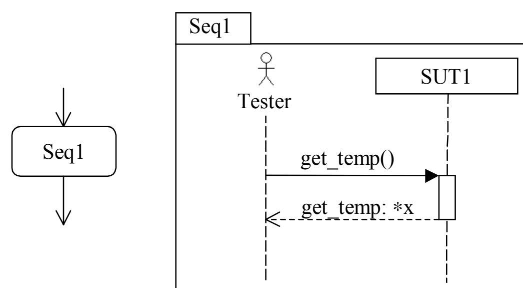


Figure 4-38: Activity diagram node (l.h.s.) referencing TeLa sequence diagram (r.h.s.).

3.2.3 Well-formedness

The referenced TeLa sequence diagram must be elementary.

3.2.4 Informal semantics

The referenced TeLa sequence diagram is composed with the diagram referenced in the previous state, see sequential composition.

3.3 Sequential composition (weak and strong)

3.3.1 Introduction

A transition from an activity diagram node N_1 to an activity diagram node N_2 denotes the weak sequential composition of N_1 and N_2. A transition from an activity diagram node N_1 to an activity diagram node N_2 which passes via a synchronization bar denotes the strong sequential composition of N_1 and N_2.

3.3.2 Syntax

Weak sequential composition is denoted by using the normal UML activity diagram transition between nodes. Strong sequential composition is denoted by adding the UML activity-diagram synchronization bar, that is by using a transition from N1 to the synchronization bar and another transition from the synchronization bar to N2. See Fig. 4-39 for an example.

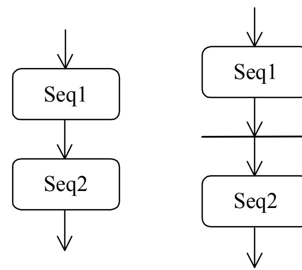


Figure 4-39: Weak (l.h.s.) and strong (r.h.s.) sequential composition

3.3.3 Well-formedness

Each transition connects one of the following:

- an activity-diagram node to another activity diagram node
- an activity-diagram node to a synchronization bar
- a synchronization bar to an activity-diagram node.

Each synchronisation bar is connected to a single activity diagram node in each direction.

3.3.4 Informal semantics

A transition from an activity diagram node N_1 to an activity diagram node N_2 denotes the weak sequential composition of each of the partial orderings denoted by a path through the activity diagram from the intial state to node N_1 and the partial ordering denoted by the TeLa sequence diagram referenced in node N_2.

The weak sequential composition of A and B, where A and B are two partial orderings obtained from TeLa sequence diagrams S_A and S_B, can be thought of as the partial ordering obtained by glueing the top of the lifelines of S_B to the bottom of the lifelines of S_A (before projection), for all the lifelines the two diagrams have in common.

TeLa sequence diagrams specify the behaviour of the tester and, in spite of the use of SUT lifelines, the semantics of TeLa concerns only tester events. As stated in Chapter 3, this semantics is obtained by projection onto tester lifelines. Clearly, then, the synchronization of

strong sequential composition also applies only to tester lifelines, it does not imply the synchronization of the SUT. All tester-SUT communication must be explicitly described.

A transition from an activity diagram node N_1 to an activity diagram node N_2 which passes via a synchronization bar denotes the strong sequential composition of each of the partial orderings denoted by a path through the activity diagram from the intial state to node N_1 and the partial ordering denoted by the TeLa sequence diagram referenced in node N_2.

The strong sequential composition of A and B, where A and B are two partial orderings obtained from TeLa sequence diagrams S_A and S_B, can be obtained by ordering all the events of A w.r.t. each of the the first events on each lifeline of B.

3.4 TeLa activity-diagram choice

3.4.1 Introduction

The activity-diagram choice construct is used to model a situation in which one of several possible behaviours can occur. TeLa activity-diagram choices are not restricted to choices between message emission events or between message reception events. Neither is a choice restricted to one in which each alternative has only one enabled / fireable event; several concurrent events may be enabled / fireable in a single alternative. Furthermore, there may be events that are concurrent with the enabled events of all the alternatives.

An alternative of a TeLa activity-diagram choice can be explicitly guarded. A guard on a TeLa activity-diagram choice only concerns controllable events. A guard on an alternative is a conjunction of local guards, each of which has an owning component. A local guard only applies to (we will say "covers") controllable events which are located on its owning component or a subcomponent of it; it has no effect on controllable events on other components. The notion of covered by a guard is defined more precisely below.

A TeLa activity-diagram choice may have a default alternative. If this is the case, the requirements on the diagram in order for it to be well-defined are stricter than those on a diagram without a default alternative, see below.

Optionally, each alternative of a TeLa activity-diagram choice can be given a priority which serves to ensure that only one alternative has a fireable event in the case where:

- guards associated to controllable events are not mutually exclusive,
- guards associated to events labelled by the same observable action are not exclusive.

The interpretation of a choice in which guards are not mutually exclusive and priorities are not assigned is that the choice is indefinite, i.e. not fully specified, but that only one alternative can be taken in any execution (c.f. the UML 1.4 equivalent which may denote choice or concurrency depending on data values).

3.4.1.1 READY, ENABLED AND FIREABLE EVENTS OF AN ALTERNATIVE

An event is a *ready event* for an alternative of a TeLa activity-diagram choice if it is a possible first event on that alternative (without taking into account guard evaluations). Note that several concurrent events may be ready for an alternative and that a reception event can never be a ready event since the notion of ready event is concerned with the semantics before projection onto tester lifelines.

An event is a *syntactically-ready event* for an alternative of a TeLa activity-diagram choice if it is a ready event for that alternative and is not a ready event for any alternative of any other TeLa activity-diagram choice. If a choice is local, see below, all ready events for all alternatives are also syntactically ready.

A controllable event is an *enabled event* for an alternative of a TeLa activity-diagram choice if it is a ready event for that alternative. An observable event is an *enabled event* for an alternative of a TeLa activity-diagram choice if it has a chain of predecessors, all of which are SUT events, leading to a ready event. That is, it is enabled in the semantics after projection onto the tester lifelines.

An event is a *syntactically-enabled event* for an alternative of a TeLa activity-diagram choice if it is enabled for that alternative and is not enabled for any alternative of any other TeLa activity-diagram choice. If a choice is test local, all enabled events for all alternatives are also syntactically enabled. In Fig. 4-41, the event ?*m*4 (reception of message *m*4) is enabled but not syntactically-enabled on the r.h.s. alternative of the upper of the two choices. It is enabled and syntactically enabled on the r.h.s. alternative of the lower of the two choices.

If a controllable event is enabled but not syntactically enabled, it may be guarded by local guards from several different choices. If this is the case, the owning components of these local guards will not necessarily coincide. Moreover, another controllable event that is enabled in the same alternatives is not necessarily covered by the same guards, that is it may be covered by some but not by others. To avoid defining complex compatibility conditions between such local guards, we simply restrict to test descriptions in which all enabled controllable events are syntactically-enabled.

A two-tier test description is said to be in *semi-normal form* if all enabled controllable events are syntactically-enabled. The restriction to test descriptions in semi-normal form facilitates the syntactic detection of the essential controllability needed for the use of the default alternative. It would be further facilitated if we also restricted to test descriptions with no enabled but not syntactically-enabled observable events. However, test non-local choices between observable actions are an inevitable feature of using the projection semantics so that choices with enabled but not syntactically-enabled observable events are also likely to arise quite frequently.

We reserve the term *fireable* for the execution level; an event is fireable in a given configuration if it is enabled in that configuration and its guard evaluates to true in that execution.

3.4.1.2 LOCAL AND TEST LOCAL CHOICES

A TeLa activity-diagram choice is said to be a local choice if all ready events on all the alternatives are located on the same lifeline. It is said to be non-local otherwise. Clearly, this notion concerns the semantics before projection onto tester lifelines.

A TeLa activity-diagram choice is said to be a *test local choice* if all enabled events on all the alternatives are located on the same lifeline. It is said to be test non-local otherwise. Clearly, this notion concerns the semantics after projection onto tester lifelines.

The notions of local choice and test local choice are therefore relative to an interaction framework and susceptible to change on decomposition. A choice is said to be a *maximally local choice*, resp. *maximally test local choice*, if it is local, resp. test local, for the maximal decomposition.

3.4.1.3 OWNING COMPONENT OF A CHOICE

As for TeLa sequence diagram choices, the owning component of a choice (we assume it is in semi-normal form) is the smallest component which has the following as subcomponents:

- the owning component of each of the local guards,
- the owning component of the originating port of any tester emission actions which are enabled in any of the alternatives,
- the owning component of any internal actions which are enabled in any of the alternatives,
- the owning component of the target port of any tester reception actions which are enabled in any of the alternatives.

The owning component of a choice is the owning component of any implicit verdicts derived in that choice, though since a local fail verdict is also a global fail verdict, this is only of any importance in the case of the implicit inconclusive verdict. The latter verdict can only be derived in the non-enumerated data case and for a test description that it essentially controllable, see Section 3.4.4.1.

Note that a choice is test local iff the owning component of that choice is represented by a single lifeline.

3.4.1.4 GUARDS ON ALTERNATIVES OF A CHOICE

Suppose a two-tier scenario structure is in semi-normal form. An unguarded alternative of a choice of such a two-tier scenario structure is said to be implicitly guarded if one of the following is true:

- it has an enabled event labelled by a tester assertion internal action,
- it has an enabled event labelled by a tester emission action for which the arrow label on the corresponding message includes a guard.

Thus, assertion internal actions are in fact treated as guards. There are no real assertions in TeLa, in the sense of expressions that do not influence the choice of execution path but which, if executed, result in an exception. This is by the nature of testing, where guards themselves can already result in a type of exception: the implicit verdict.

For a two-tier scenario structure in semi-normal form, an implicit guarded alternative is semantically equivalent to an explicit guarded alternative. We therefore suppose that any explicitly guarded alternative of a choice satisfies the following:

- no enabled event of the alternative is labelled by a tester assertion,
- no enabled event of the alternative is labelled by a tester emission for which the arrow label on the corresponding message includes a guard.

A two-tier scenario in semi-normal form and for which all the choices also satisfy the above condition is said to be in *normal form*. Again, the problem if we allow alternatives to be both explicitly and implicitly guarded concerns the fact that an event may be covered by two local guards whose owning component is not the same.

The guards of a TeLa activity-diagram choice are tester guards that apply to tester emissions or tester internal actions. They have no effect on tester receptions. Therefore, proper tester receptions (observable actions) that are enabled for an alternative may have associated assertions or message arrow guards. The above restriction does not affect guarded intra-SUT messages either since under the projection semantics these guards become guards on proper tester receptions.

We do not attempt to turn observable action guards into activity-diagram guards in the same way as is done for controllable actions, since such guards are specific to the particular observable action to which they are associated. Contrary to the situation for controllable actions, for observable actions, the mutual exclusion of guards and the implicit verdict if guards are not verified, only concern observable actions with the same name, not all observable actions. Similarly, the anonymous variables that may be used in controllable action guards are context dependent and would be ambiguous if not clearly referring to a particular controllable action.

An event *e* which is labelled by a controllable action *a* other than an assertion and which is enabled for an alternative *A* of a choice *C* is said to be unguarded in that alternative if:

- Alternative *A* is not guarded
- Alternative *A* is guarded by guard *G* but:
    - *a* is an internal action whose owning component is not a subcomponent of the owning component of any of the local guards of which *G* is composed.
    - *a* is a tester emission action and the owning component of its originating port is not a subcomponent of the owning component of any of the local guards of which *G* is composed.
- The corresponding message arrow label does not include a guard (we include this situation for completeness, though in a test description in normal form it cannot arise).

Otherwise, the event *e* is said to be guarded for alternative *A* of choice *C*. In the normal form case, we also say that *e* is covered by the guard of *A*.

3.4.1.5 FOCUS BARS

In a TeLa activity-diagram choice, a focus bar is allowed to begin before the choice and end after the choice. This means that the top half of the focus bar appears in one diagram while the bottom half appears in each of the diagrams constituting the different alternatives. The use of focus bars straddling choices in this way is inevitable, for example, in modelling the situation where the SUT may respond to a synchronous invocation in several possible ways.

3.4.2 Syntax

3.4.2.1 GUARDS AND OWNING COMPONENTS

The guard on an alternative of a sequence diagram choice takes the form of a conjunction of local guards. The conjunction may be presented without logical conjunction symbols by writing each clause of the guard on a new line.

Each local guard is a boolean condition involving static and dynamic variables (but not anonymous variables) preceded by a component name and a colon; this component is the owning component of the local guard. The component name and the colon can be omitted in the case where there is only one tester lifeline. If the component name is omitted, the component represented by the unique tester lifeline is the owning component of each of the local guards.

The dynamic variables used in a local guard may, or may not, be preceded by a base-level component name and a dot.

3.4.2.2 GRAPHICAL SYNTAX

Graphically, a Tela activity-diagram choice is represented using the UML activity diagram decision construct. Alternatives can be guarded and a default alternative can be given using the keyword `else` in place of a guard. An example of a sequence connector choice in which each alternative has only one minimal event and that minimal event is a controllable event is given in Fig. 4-40.

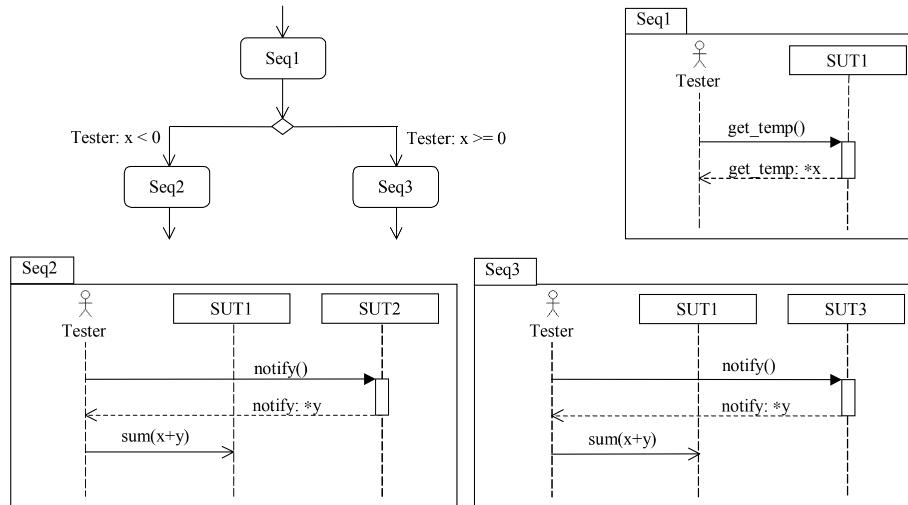


Figure 4-40 : A TeLa two-tier scenario structure showing a local choice and comprising a TeLa activity-diagram with three nodes, together with the corresponding sequence diagrams. This scenario structure represents the same behaviour as the one-tier scenario structure of Fig 4-19.

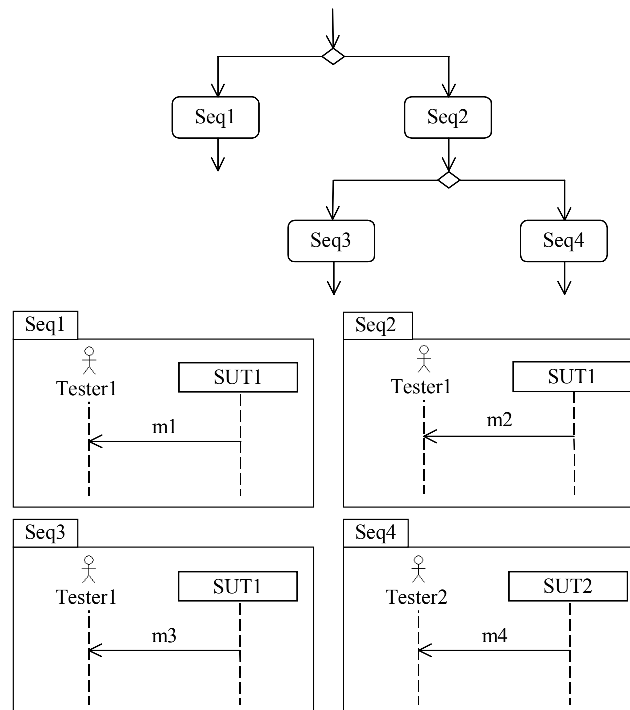


Figure 4-41: TeLa activity-diagram choice illustrating the notion of enabled and syntactically enabled.

3.4.3 Well-formedness

3.4.3.1 GUARDS

The owning components of the local guards of which the guard of an alternative is composed must be members of an interaction framework of the component model of the test description (see Chapter 5, Section 1). That is, they are disjoint in the sense that none of the components is a sub or supercomponent of another. The particular interaction framework used must be the same for all the different alternatives of the choice.

For each dynamic variable used in a local guard:

- if the variable name is preceded by a base-level component name and a dot, the intended value is that held by this component; moreover, this component must be a subcomponent of the component owning the local guard in which the variable is used,
- otherwise, all the base-level components of the owning component of the local guard must share a common view of the value of that variable.

3.4.3.2 DEFAULT ALTERNATIVE

If a choice has a default alternative, that choice must be essentially controllable, that is, if a controllable event is enabled in one alternative, no observable event can be enabled in any other alternative unless it is in all of them.

3.4.4 Informal semantics

A TeLa activity-diagram choice describes a situation in which one of several behaviours is possible. Whether the choice is fully specified or not, in the sense that it cannot be guaranteed that only one of the alternatives is fireable in any execution, only one of the alternatives can be chosen in any execution. Choices that are not fully specified are termed *indefinite choices*.

If an alternative is unguarded, it is considered to be guarded by the expression "true" owned by the whole tester component.

3.4.4.1 DETERMINISM AND CONTROLLABILITY

In this section, we will only discuss the semantics after projection onto the tester lifelines. We will say that two events which are enabled in different alternatives of a choice are in minimal conflict.

A test description is *minimally deterministic* if no concurrent events have identical labels and no observable events in minimal conflict have identical labels. A test description that is not minimally deterministic is considered not well-defined. We also suppose for well-definedness that any remaining non-determinism, due to identically-labelled controllable events in minimal conflict, is resolved by controllable events (delayed choices are resolved by the tester).

If, in addition, the test description is such that there are no minimal conflicts involving an observable event and a controllable event, we say that the test description is *essentially controllable* and describes a *parallel test case*. If, in addition, the test description is such that no proper tester emissions (i.e. to the SUT) is concurrent with a proper tester reception (i.e. from the SUT), we say that the test description is *properly concurrently controllable* and describes a *externally-coherent parallel test case*. If, in addition, the test description is such that no tester coordination action or internal action is concurrent with a proper tester reception, we say that the test description is *concurrently controllable* and describes a

coherent parallel test case. Finally, if, in addition, the test description is such that no controllable event is concurrent with any other controllable event we say the test description is *controllable* and describes a *centralisable test case*.

In the non-enumerated data case, different controllable events may be enabled in different alternatives of a choice. However, if the test description is to describe a parallel test case, all such controllable events must be guarded (in the sense defined above). For more details on determinism and controllability in TeLa, see Chapter 5, Section 2.2.

We leave for future work the study of how a test description can be completed in order to turn it into a parallel test case, as well as the study of how a parallel test case can be made externally coherent, coherent or centralised by resolution of tester concurrency e.g. by adding synchronizations.

3.4.4.2 DEFAULT ALTERNATIVE

The default alternative is a global notion which can only be given meaning in a centralisable test case, an externally centralisable test case (see Chapter 5, Section 2.2) or a coherent implementation of a parallel test case.

Suppose the test description describes an coherent parallel test case. The meaning of the default alternative for a choice in which all enabled events are observable events is that of an unspecified reception from the tester. In the non-enumerated data case, the meaning of the default alternative in which all enabled events are controllable events is that none of the guards on the specified alternatives evaluates to true.

Now suppose the test description describes an externally-coherent parallel test case. If we suppose that the implementation gives priority to tester coordination messages and tester internal actions (we must suppose the latency of tester coordination messages is less than that of other messages), the default alternative is also meaningful. The meaning of the default alternative for a choice in which all enabled events are observable events, or tester coordination messages / tester internal actions that are concurrent with all of such observable events, is again that of an unspecified reception from the tester.

Finally, suppose the test description describes a parallel test case. If we suppose that the implementation gives priority firstly to tester coordination messages and tester internal actions and secondly to observable actions, the default alternative is also meaningful for any parallel test case. The meaning of the default alternative for a choice in which all enabled events are observable events, or controllable events that are concurrent with all such observable events, is again that of an unspecified reception from the tester.

3.5 Implicit and explicit verdicts

3.5.1 Introduction

3.5.1.1 IMPLICIT VERDICTS

In the non-enumerated data case, the implicit inconclusive verdict is derived if any one of a set of guards on concurrent controllable events does not evaluate to true, or if none of a set of guards on alternative controllable events evaluates to true. In the former case, the inconclusive verdict is local to the component owning the guard. In the latter, the inconclusive verdict is

local to the component owning the choice. We assume that all other components continue their execution to see if a fail verdict can be derived.

The meaning of guards on controllable actions not evaluating to true also merits further explanation, given the complexity of the guards described previously. In this calculation, local guards on events which are concurrent are disjoined whereas local guards on events which are in conflict are conjoined. It should also be noted in carrying out this calculation that that the same dynamic variable may have different values in concurrent or conflicting local guards.

The implicit fail verdict is derived if, at any time, an unexpected proper reception occurs.

The meaning of “unexpected” in the phrase “unexpected proper reception” merits further explanation. It takes into account the name of the action (as in an interleaving semantics), the signature of that action and the guard (as in the non-enumerated data interleaving case), as well as the port on which the action occurs (as in the non-enumerated data partial-order semantics case). Again, local guards on events labelled by the same observable action (where this notion includes the signature and the owning port) that are concurrent are disjoined whereas local guards on events labelled by the same observable action that are in conflict are conjoined. Again, it should also be noted in carrying out this calculation that that the same dynamic variable may have different values in concurrent or conflicting guards.

3.5.1.2 EXPLICIT VERDICTS

TeLa elementary sequence diagrams, those that can be referenced in TeLa activity diagrams, can also contain explicit verdicts.

3.5.2 Syntax (explicit verdicts)

The syntax used is the same as that used for the explicit verdicts occurring as alternatives of sequence diagram choices.

3.5.3 Well-formedness

The explicit inconclusive verdict on proper tester reception is allowed anywhere in a sequence diagram.

In the non-enumerated data case, the implicit inconclusive verdict can only be derived for a test description that is a parallel test case.

Explicit verdicts, other than the explicit inconclusive verdict on proper tester reception, are only legal if the event to which they are attached is enabled in an alternative of a choice with an explicit default alternative. The explicit default alternative requires that the test description be at least essentially controllable. If the choice is between alternative controllable events, the explicit verdict must be an inconclusive verdict and the explicit default alternative must not also lead to an explicit inconclusive verdict. If the choice is between observable events and the explicit verdict is a fail verdict, the explicit default alternative must not also lead to an explicit fail verdict.

3.5.4 Informal semantics

The implicit fail verdict is well-defined for any well-defined test description. In the case where alternative receptions are possible, if the test description describes a coherent parallel test case, the implicit fail verdict can be viewed as an implicit (global) default alternative.

In the case where none of a set of alternative tester guards evaluates to true, if the test description describes a coherent test case, the implicit inconclusive verdict is well-defined and can be viewed as an implicit (global) default alternative. If the test description describes a parallel test case that is not coherent, the implicit inconclusive verdict can be made well-defined by stating that if a component derives an inconclusive verdict and a sub or super component derives an implicit fail verdict, the result is a fail verdict. With this priority scheme, the implicit verdict can again be viewed as an implicit (global) default alternative.

In the case of the explicit verdict, the corresponding event is annotated with the verdict. In an execution, if the corresponding action occurs in a configuration where that event is enabled, the local verdict is derived. How a global verdict is derived from this local verdict is described in Chapter 5, Section 2.1.2.2.

3.6 TeLa activity-diagram loops

3.6.1 Introduction

We allow sequences of transitions to form loops. Loops with an exit involve a TeLa activity diagram choice.

3.6.2 Syntax

A loop with an exit condition requires use of a node with an empty sequence diagram (denoted with a dash) in order to respect the UML activity diagram syntax. See Figure 4-42 for an example of a sequence connector loop in TeLa.

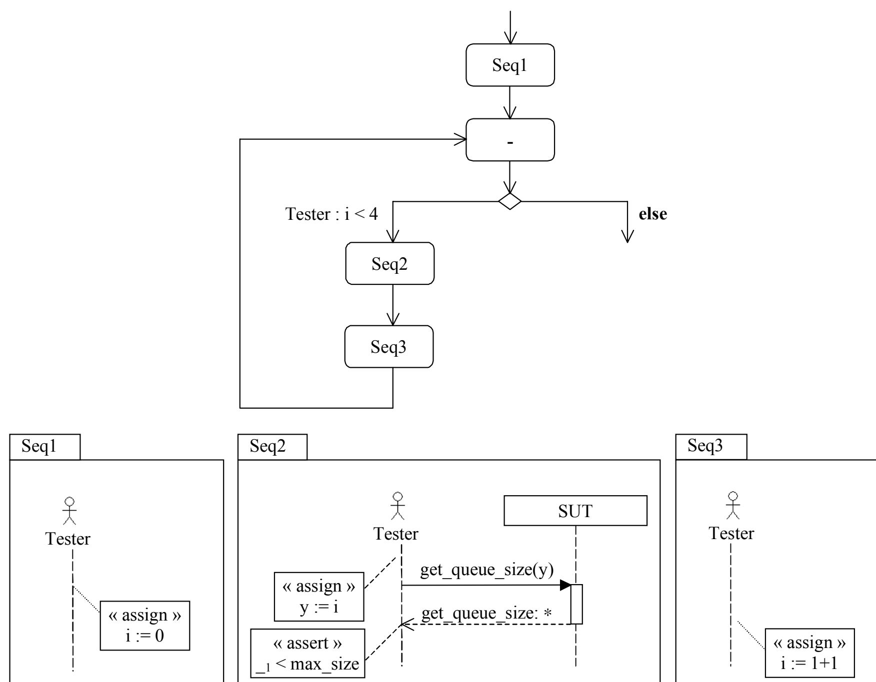


Figure 4-42: A TeLa two-tier scenario structure showing a loop with specified exit condition and comprising a TeLa activity diagram with four nodes, together with the corresponding sequence diagrams, one of which is empty. This two-tier scenario represents the same behaviour as the one-tier scenario structure of Fig. 4-16.

3.6.3 Well-formedness

If the conditions for re-entering and exiting the loop are specified, the exit condition must be the negation of the re-entry condition. Usually, the default alternative branch is used to ensure this.

3.6.4 Informal semantics

A loop permits the execution of a behaviour a specified or unspecified number of times.

The loop guard is always considered to be part of a controllable event. Even if, initially, there is a single ready event and it is an observable event, the diagram is not equivalent to a diagram in which the guard is moved to a guard on the observable event.

Chapter V : Elements of a formal semantics for a scenario-based test description language for component testing

In this chapter, we some of the main elements of a formal semantics for TeLa. In Section 1, we define the component model underlying TeLa test descriptions. In Section 2, we present elements of a behavioural semantics. In Section 3, we show how a message-based semantics can be mixed with an event-based semantics on a component by component basis.

1 Elements of structural semantics

1.1 Component structures and component models

1.1.1 Component structures

We define a component structure $\Im$ as a tuple:

$$(X, \Pi, I, \leq_X, mult_{(X,X)}, ownsport, mult_{(X,\Pi)}, portint, connect)$$

where:

- X is a finite set of component types containing a distinguished subset $X_\perp$, the set of base-level component types, and a distinguished element $x^\top \notin X_\perp$, the top-level component type; in the usual case, the elements of are $X_\perp$ are classes. Π is a finite set of port types and I a finite set of interface types.
- $\leq_X$ is a reflexive, transitive, anti-symmetric binary relation, known as the *contains* relation, between elements of X satisfying:

  $$\forall\alpha \in X,\ x^\top \leq_X \alpha \quad \text{and} \quad \forall\alpha \in X_\perp,\ \{\beta \in X \mid \alpha \leq_X \beta \} = \{\alpha\}.$$

  $\leq_X$ defines a containment hierarchy among the component types. We denote the irreflexive reduction of $\leq_X$ by $<_X$ and call it the *properly_contains* relation. We denote the transitive, irreflexive reduction of $\leq_X$ by $<<_X$ and call it the *has_part* relation (it signifies immediate containment). $(X,<<_X)$ is a tree with root $x^\top$ and leaves $X_\perp$.

- $mult_{(X,X)}$: $X \times X \mapsto \mathbb{N} \times \mathbb{N}$ is a function associating a pair of natural numbers, ($min_{(\alpha,\beta)}$, $max_{(\alpha,\beta)}$) satifying $max_{(\alpha,\beta)} \geq init_{(\alpha,\beta)}$, to each pair of component types, (α, β), that is consistent with $\leq_X$, i.e.:

  $$mult_{(X,X)}(\alpha, \beta) \neq (0, 0) \ \Leftrightarrow\ (\alpha, \beta) \in <<_X$$

  The tuple ($min_{(\alpha,\beta)}$, $max_{(\alpha,\beta)}$) defines the *multiplicity* of the "child" component type β in the "parent" component type α, i.e. the specification of the initial and maximum[1] number of instances of β that can be parts of any instance of α. We assume that:

  $$\forall\alpha \in X{-}X_\perp,\ \exists\, \beta\in X,\quad mult_{(X,X)}(\alpha, \beta) = (1, n) \text{ for some } n \geq 1,$$

  This condition ensures that every instance of a component type contains an instance of a base-level component type.

- *ownsport*: $X \mapsto \Pi$ is a surjective mapping that associates a non-empty set of port types to each component type. In the usual case, each base-level component type is related to a single port type, i.e.

  $$\alpha\in X_\perp \ \Rightarrow\ ownsport(\alpha) = \{p\} \text{ for some } p \in \Pi.$$

  The top-level component type $x^\top$ may, or may not, have associated ports.

[1] We demand an upper bound to the number of components (instances of component types) of each type to ensure that the number of components is bounded.

- $mult_{(X,\Pi)}$: $X \times \Pi \mapsto \mathbb{N} \times \mathbb{N}$ is a function that associates a pair of natural numbers, ($min_{(\alpha, p)}$, $max_{(\alpha, p)}$) satisfying $max_{(\alpha, p)} \geq init_{(\alpha, p)}$, to each pair of component types and port types (α, p) and that is consistent with the *ownsport* mapping in the sense that:

  $$mult_{(X,\Pi)}(\alpha, p) \neq (0, 0) \;\Leftrightarrow\; p \in ownsport(\alpha).$$

  The tuple ($min_{(\alpha,p)}$, $max_{(\alpha,p)}$) defines the *multiplicity* of port type p in component type α, i.e. the specification of the initial and maximum number of instances of p that can be parts of any instance of α. In the usual case:

  $$\alpha \in X_{\perp} \;\wedge\; ownsport(\alpha) = p \;\Rightarrow\; mult_{(X,\Pi)}(\alpha, p) = (1,1)$$

  i.e. the multiplicity of the single port type owned by a base-level component type is (1,1). However, this is not true for non-standard base-level component types, see below.

- *connect* is a relation between pairs of component types and port types satisfying:

  $$\begin{aligned}((c_1, p_1), (c_2, p_2)) \in connect \;\Leftrightarrow\;& p_i \in ownsport(c_i),\ i = 1,2 \\ \wedge\;& (\, c_1 <<_X c_2 \;\vee\; \exists\, c_3 \in X\ (c_3 <<_X c_1 \;\wedge\; c_3 <<_X c_2)\,) \\ \wedge\;& \text{the constituent interfaces of } p1 \text{ and } p2 \text{ are compatible}\end{aligned}$$

- *portint*: $\Pi \mapsto I$, is a surjective mapping taking each port type to the set of interface types of which it is composed.

The two clauses $c_1 <<_X c_2$ and $c_3 <<_X c_1 \wedge c_3 <<_X c_1$ correspond to the “delegation connectors” and “assembly connectors” of UML 2.0, respectively. We do not define the notion of interface compatibility here. Such a definition would require well-formedness conditions for each of the two types of connectors (assembly and delegation). However, for the objective of defining an underlying component basis for TeLa sequence diagrams, we do not need this level of detail.

A base-level component type α for which the number of elements of *ownsport*(α) is one and the multiplicity for this port type in α is (1,1) is called a *standard base-level component type*. Any other base-level component type is called a *non-standard base-level component type*. Non-standard base-level component types are introduced in order to model components whose internal structure is unknown, in particular, with the SUT of black-box testing in mind.

A component structure can be defined via a set of diagrams, each representing a component type α and showing the immediate sub-component types of α, each with a multiplicity, the port types of α, each with a multiplicity, and the port types of the immediate sub-component types of α. The diagram also shows lines representing *connect* relations, each line either connecting port types of distinct sub-component types of α, or a port type of α and a port type of one of its immediate sub-component types.

Given a component structure $\Im$, as above, instances of elements of Π, the set of port types of $\Im$, are called Π-typed ports, or simply *ports*, where Π is clear. Instances of elements of X, the set of component types of $\Im$, are called X-typed components, or simply *components* where $\Im$ is clear. A set of X-typed components will be called an X-typed component set or simply component set. Instances of the *connect* relation between ports will be called *connectors*.

1.1.2 Component models

A *component model* Ω is defined as comprising the following elements:

- A component structure, $\Im$, as defined above

- A set of rules for each component type α defining how to create/destroy ports and connect relations on creation/destruction of an immediate subcomponent[2], i.e. an instance of a component whose type β satisfies $\alpha <<_X \beta$. Some or all of these rules may be generic to some or all of the component types of the component structure. We assume that ports or connect relations are only created/destroyed on creation/destruction of immediate subcomponents, except for ports of non-standard base-level components and connectors involving such ports. We will not formalise these rules here.
- The rule stating that in any instantiation of the component structure, there is always exactly one instantiation of the top-level component $x^\top$. Notice that the top-level component is not covered by any multiplicity restriction.

In the component models we use here, we further assume that for each component type, among the port types related by the *ownsport* mapping is a distinguished port type called the creation port type, which is the destination of any creation messages of a component of that type. This is to enable component creation to be modelled as the sending of a message from the creator component to the created component as in UML sequence diagrams.

The definitions of component structure and component model are intended to be compositional in the sense that the top-level component type of a given component model can be used as the internal structure of a non top-level component type of a larger component model.

1.1.3 Component-model snapshots

A snapshot S of a component model Ω, as above, is set of components, ports and connectors, together with relations that are instances of the *contains*, *ownsport* and *connect* relations, which satisfy the multiplicity constraints between their types and the rules of the component model, and in which components and ports have identity. That is, a component-model snapshot is a legal instance of a component model.

More formally, a component-model snapshot is a tuple ($\underline{S}$, π_S, $\leq_S$, $ownsports_S$, $connects_S$) where $\underline{S}$ is a set of components typed by the elements of X, π_S is a set of ports typed by the elements of Π, and the relations $\leq_S$, $ownsports_S$, and $connects_S$ satisfy:

$\forall\, c, c_1, c_2 \in \underline{S}\,,\ \forall p, p_1, p_2 \in \pi_S\ :$

$c_1 \leq_S c_2 \ \Leftrightarrow\ \text{type}(c_1) \leq_S \text{type}(c_2)$, so that $(\underline{S}, <<_S)$ is a tree
where $<<_S$ is the transitive, irreflexive reduction of $\leq_S$

$p \in ownsports_S(c) \ \Leftrightarrow\ \text{type}(p) \in ownsport(\text{type}(c))$
and $ownsports_S(c1) \cap ownsports_S(c2) = \varnothing,\ \ \forall\, c1, c2 \in \underline{S}$ s.t. $c1 \neq c2$

$((c_1, p_1), (c_2, p_2)) \in connects_S \ \Leftrightarrow\ ((\text{type}(c_1), \text{type}(p_1)), (\text{type}(c_2), \text{type}(p_2))) \in connect$
$\wedge\ \ p_1 \in ownsports_S(c_1),\ p_2 \in ownsports_S(c_2)$

$min_{(\text{type}(c),\, \alpha)} \leq \#(\alpha \text{ in } c) \leq max_{(\text{type}(c),\, \alpha)}$,
where $\#(\alpha \text{ in } c)$ is the no. of components $c' >>_S c$, s.t. $\text{type}(c') = \alpha$

$min_{(\text{type(c)},\, \kappa)} \leq \#(\kappa \text{ in } c) \leq max_{(\text{type(c)},\, \kappa)}$,
where $\#(\kappa \text{ in } c)$ is the no. of ports $p \in ownsports_S(c)$, s.t. $\text{type}(p) = \kappa$

and where the connect relations are in accordance with the rules of the component model.

---

[2] A more general model would be obtained by substituting proper subcomponent for immediate subcomponent.

For any snapshot, the contained elements of a component according to $\leq_S$, $<_S$ and $<<_S$ are known as *subcomponents*, *proper subcomponents*, and *immediate subcomponents* or *parts*, respectively. If $p \in ownsports_S(c)$ for $c \in \underline{S}$, $p \in \pi_S$, we say that the component owns the port in that snapshot.

The top-level, resp. a base-level, component is an instance of the top-level, resp. a base-level, component type. We use the notation $c^{\top}$, resp. $c_{\perp}$, for the top level, resp. any base level, component of a snapshot. We use the notation $C_{\perp}$, for any component set of a snapshot whose elements are all base-level, components.

The relation of components with the active object notion is as follows. Certain base-level components are denoted active components and then a component is active if at least one of its subcomponents is an active component.

1.1.4 Dynamics of component models

We define a *component animation* of a component model as a sequence of component snapshots of that component model in which adjacent elements satisfy (recall that components have identity):

- a component present in one snapshot but not in the next snapshot must have been explicitly destroyed
- a component present in one snapshot but not in the previous snapshot must have been explicitly created

Notice that the multiplicity requirements ensure that when a non base-level component is created, at least one subcomponent must also be created. We define a *component development* of a component model as a set of component animations of that component model which is coherent, according to some criteria which we do not define here. We suppose that the set of component animations of a component development associated to a test description share an initial snapshot. Two component developments are isomorphic if there is a renaming of component and port identities taking one to the other.

We assume that a newly-created component is given a unique identifier. If the creation message syntax is used in a TeLa sequence diagram, we show this identifier being returned in a return message to the creating component. If creation of a component implies creation of subcomponents, these are created successively, the parent creating the child and being aware of its identity. If component creation takes place in a loop, on leaving the loop, the variable representing the returned component identity has the value of the last component created.

Given a component development, the set consisting of the snapshots that are involved in at least one component animation of the component development is called the *snapshot universe*. Note that a snapshot universe may contain distinct snapshots which are isomorphic but which differ in the component or port identities. The set of components which take part in at least one snapshot of the snapshot context is called the *component universe* of the component development. We assume that each component in the initial snapshot has a distinct identity, and also that each component created in a component development has a distinct identity w.r.t. to any component which exists prior to its creation in any component animation of the component. The set of ports which take part in at least one snapshot of the snapshot context is called the *port universe* of the component development. We assume that each port in the initial snapshot has a distinct identity, and also that each port created on creation of a component has a distinct identity w.r.t. any port which exists prior to its creation in any

component animation of the component development. Thus: $ownsports_S(c1) \cap ownsports_{S'}(c2) = \varnothing$, $\forall c1 \in S, c2 \in S'$ s.t. $c1 \neq c2$, $\forall S^{\Delta} \in S$

If Δ is a component development, we denote the snapshot universe by S^{Δ}, the component universe by C^{Δ} and the port universe by P^{Δ}.

1.2 Test description component models

At the basis of a Tela sequence diagram is a component model, which we refer to as the test description component model. This component model constitutes the structural foundations underlying the diagram. In all component models underlying TeLa sequence diagrams, the first level of component types under the top level contains two distinguished component types with multiplicity (1,1): the tester component type and the SUT component type. The corresponding components will be referred to as the whole-tester component and the whole-SUT component.

1.2.1 Tester internal structure

The internal structure of the tester component type, together with the ports of the SUT component type and the connectors between the tester component type and the SUT component type, is that defined in the test architecture.

The internal structure of the tester specified in the test architecture may contain hierarchy. If it does not, the subcomponents of the tester component type are all base-level components which are classes of the tester class diagram. This is the simple object model in which the internal structure of the tester component type is a default one defined according to the class diagram of the tester.

In the case where a closed component model of the entire application – SUT + SUT environment – is available, the test architecture component model may, or may not, be derived from the SUT-environment part of the application component model (e.g. by addition of a controller module with verdict capability).

1.2.2 SUT internal structure

The SUT component type may, or may not, contain internal structure. In the simplest case, it does not contain internal structure and is therefore a base-level component type of the test description component model, as in the test architecture. This base-level component is standard, or non-standard, according to whether it owns one port type with multiplicity (1,1), or several port types. In the simple object-model case, the ports of the whole SUT component are in 1-1 correspondence with those of the standard base-level components (= objects) it contains and can be unified with them, see below.

A component model of the SUT may be available, most likely as part of a component model of the entire application. If it is, in the test description component model we can define the internal structure of the SUT component type to be that defined by this SUT component model, instead of defining the SUT to be a base-level component type, as is the case in the test architecture component model. However, we may still choose not to represent this internal structure in TeLa sequence diagrams via multiple SUT lifelines. Whether or not the SUT internal structure is represented explicitly in the diagram, the so-called “SUT component interface” contains all the SUT ports referenced by any actions of the test. The set of ports of

the whole SUT component, on the other hand, coincides with that of the "SUT component interface" if the SUT internal structure is not represented explicitly, but may be a proper subset of it, if the SUT internal structure is represented explicitly.

If we do not explicitly represent the SUT internal structure in TeLa sequence diagrams, we may represent it either as a single lifeline or as a set of lifelines representing ports.

If we do choose to explicitly reflect the SUT internal structure in TeLa sequence diagrams via multiple lifelines, we are apparently assuming that the SUT implementation is organised according to this component model but, in fact, this is for representational purposes only. As stated earlier, such a representation can be particularly useful for test descriptions derived from sequence diagrams of the application design model.

Whether we show the components inside the whole SUT component or whether we simply show the ports of this component, does not affect the semantics obtained by projection onto tester instances, as long as the causality relations between tester actions are the same. Recall also that, contrary to the situation for tester base-level components, SUT base-level components (including the whole SUT component if it is a base-level component) cannot own dynamic system variables in TeLa test descriptions. Note that SUT lifelines represent ports or components but never a mixture of the two.

1.2.3 Simple object models as special case

Each standard base-level component (= object, in usual case) of a snapshot always has a single port, so we can unify the identifiers of standard base-level components with those of their ports. If the underlying component model of a test description is a simple object model, it has two levels of component hierarchy apart from the top-level component: the tester + SUT level, and the base level.

In the simple object model, the ports on either of the two second-level components – the whole tester component and whole SUT component – are in 1-1 correspondence with the ports on the standard base-level components they contain. We can therefore unify the identifiers of the ports at the two levels, thus also unifying the identifiers of the ports on second-level components with the identifiers of the standard base-level components (= objects) they contain.

1.2.4 Dynamics of test description component models

Given a TeLa test description we assume that the initial snapshot is specified via a component diagram. In the simple object model case, it is sufficient to list the base-level components. In this case, the lifelines of the TeLa sequence diagrams represent base-level components and the initial snapshot is considered specified by the initial set of lifelines (supposing that SUT and tester lifelines are clearly distinguished).

We suppose that from a TeLa test description we can extract a set of linearisations. If we take the reduction of these linearisations w.r.t. any actions which are not creation or destruction actions, together with the initial snapshot, we define the component development of the test description. If this component development is not permitted by the component model (e.g. violation of multiplicity constraints), the component development is said to be inconsistent. Otherwise, it is said to be consistent.

We need to handle the dynamic aspect of components (snapshots and component developments) since, for example, the number of lifelines representing a component of a given type used in a diagram may be legally greater than that permitted by the multiplicity

constraints. We also wish to ensure than no action can be specified as belonging to a port on a component that has not yet been created or has already terminated, whether or not the component in question is explicitly represented via a lifeline.

1.3 Cuts and interaction frameworks

If SUT lifelines do not represent ports, the set of lifelines of a sequence diagram, or those of a set of sequence diagrams, should define an interaction framework for the component development, as defined below. This interaction framework is not necessarily a partition of any of the snapshots that it cuts.

1.3.1 Relations between sets of partially-ordered elements

In this section we look at some of the relations induced by a partial-order relation on sets of elements of the partial order. In our case, the sets of elements are sets of components from a snapshot. For $C, D \subseteq \underline{S}$ and $c, d \in \underline{S}$, and for $S \in S^{\Delta}$ we define:

C above d and *C below d:*

$$C \upharpoonleft_S d = \{c \in C \mid c \leq_S d\} \quad \text{and} \quad C \downharpoonleft_S d = \{c \in C \mid d \leq_S c\}$$

C not below d and *C not above d:*

$$C \nmid\!\downharpoonleft_S d = \{c \in C \mid \neg(d \leq_S c)\} \quad \text{and} \quad C \nmid\!\upharpoonleft_S d = \{c \in C \mid \neg(c \leq_S d)\}$$

C above D and *C below D:*

$$C \upharpoonleft_S D = \cup_{\{d \in D\}} C \upharpoonleft_S d = \{c \in C \mid D \downharpoonleft_S c \neq \varnothing\} \quad \text{and} \quad C \downharpoonleft_S D = \cup_{\{d \in D\}} C \downharpoonleft_S d = \{c \in C \mid D \upharpoonleft_S c \neq \varnothing\}$$

C not below D and *C not above D:*

$$C \nmid\!\downharpoonleft_S D = \cap_{\{d \in D\}} C \nmid\!\downharpoonleft_S d = \{c \in C \mid D \upharpoonleft_S c = \varnothing\} \quad \text{and} \quad C \nmid\!\upharpoonleft_S D = \cap_{\{d \in D\}} C \nmid\!\upharpoonleft_S d = \{c \in C \mid D \downharpoonleft_S c = \varnothing\}$$

Note that:

$$C \nmid\!\downharpoonleft_S D = \varnothing \iff C \downharpoonleft_S D = C \qquad C \nmid\!\downharpoonleft_S D = C \iff C \downharpoonleft_S D = \varnothing$$

$$C \nmid\!\upharpoonleft_S D = \varnothing \iff C \upharpoonleft_S D = C \qquad C \nmid\!\upharpoonleft_S D = C \iff C \upharpoonleft_S D = \varnothing$$

If $D \downharpoonleft_S C = D$ (all of D is below C), we say that C *covers* D and write $C \langle\langle_S D$.

If $C \langle\langle_S D$ and $C \downharpoonleft_S D = \varnothing$ (all of D is below C and none of C is below D), we say that C *caps* D and write $C \preccurlyeq_S D$

If $C \langle\langle_S D$ and $C \subseteq D$ we say D *develops* C and write $C \sqsubseteq_S D$

1.3.2 Snapshot extension

Let $S, S' \in S^{\Delta}$, we say that S' *develops* S and write $S \sqsubseteq_{S'} S'$, if:

- $\underline{S} \subseteq \underline{S}'$. Since all snapshots of S^{Δ} are trees that share the same root: $\underline{S} \subseteq \underline{S}' \Rightarrow \underline{S} \sqsubseteq_{S'} \underline{S}'$
- $\pi_S \subseteq \pi_{S'}$
- The relations $\leq_S$, *ownsports*$_S$ and *connects*$_S$ are the relations $\leq_{S'}$, *ownsports*$_{S'}$ and *connects*$_{S'}$ restricted to S

The rules concerning the creation and destruction of ports mean that the first condition is sufficient to guarantee the second except in presence of non-standard base-level components.

(S^Δ, $\sqsubseteq$) is a partial order. We define the downward extension in S^Δ, or simply *extension*, of a snapshot S as $S\downarrow = \{S' \in S^\Delta \mid S \sqsubseteq_{S'} S'\}$

1.3.3 Cuts

Let $S \in S^\Delta$ and $\underline{S}_\perp$ be the set of base-level components of S. A surjective function $\aleph: C^\Delta \mapsto \{0,1\}$ is said to be a *cut* of S if the following are true:

- $\forall c \in C_S^\aleph$, $C_S^\aleph\downharpoonright_S c = c$
  i.e. if a component is in $C_S^\aleph$, none of its proper descendants are in $C_S^\aleph$ (maybe itself).
- $C_S^\aleph \langle\langle_S \underline{S}_\perp$,
  i.e. every base-level component in $\underline{S}$ has an ancestor in $C_S^\aleph$.

where $C^\aleph = \aleph^{-1}(1)$, the *characteristic set of* $\aleph$ and $C_S^\aleph = \aleph^{-1}(1)|_S = C^\aleph \cap \underline{S}$ is the *characteristic set of* $\aleph$ *for S*.

If $\aleph$ is a cut of S we also say $\aleph$ cuts S or that $C_S^\aleph$ cuts S. In fact, the first of the two conditions above means that, if $\aleph$ cuts S then $C_S^\aleph \langle\langle_S \underline{S}_\perp \Leftrightarrow C_S^\aleph \preccurlyeq_S \underline{S}_\perp$ so we could have used the relation $\preccurlyeq_S$ instead of the relation $\langle\langle_S$ in the second condition.

1.3.4 Partitions and base-level cuts/partitions

Let $Cuts^\Delta$ be the set of functions $\aleph : C^\Delta \mapsto \{0,1\}$ which are cuts of some snapshot in S^Δ. Given a snapshot $S \in S^\Delta$, we define an equivalence relation $\equiv_S$ on $Cuts^\Delta$ by regarding two cuts as equivalent if they agree on S, i.e. $\aleph 1 \equiv_S \aleph 2$ if $C_S^{\aleph 1} = C_S^{\aleph 2}$, that is, their characteristic sets for S are the same. Denote by $[\aleph]_S$ the equivalence class of the cut $\aleph$ under $\equiv_S$. The equivalence class $[\aleph]_S$ has a unique representative $\aleph(S)$ obtained by using $C_S^\aleph$ as characteristic set, i.e. $C^{\aleph(S)} = C_S^\aleph$. $\aleph(S)$ is said to be the *partition* of S defined by any member of $[\aleph]_S$ on S. We also say that $\aleph(S)$ partitions S (and sometimes that $C_S^\aleph$ partitions S).

$\aleph$ is called a *base-level cut* of S, if it cuts S and all elements of $C_S^\aleph$ are base-level components. For any snapshot S, the partition of S defined by putting $C^\aleph = \underline{S}_\perp$, the set of base-level components of S, is always a base-level partition of S. The top-level partition is the trivial partition obtained by putting $C^\aleph = \{c^\top\}$. The entire set of base-level components of S^Δ, denoted $S_\perp^\Delta$, is a base-level cut of any $S \in S^\Delta$.

1.3.5 Interaction frameworks

We define the reach in S^Δ of $\aleph \in Cuts^\Delta$ as follows:

$$Reach(\aleph) = \{ S \in S^\Delta \mid \aleph \text{ is a cut of } S \}$$

We define the set of *interaction frameworks* for the component development of which S^Δ is the snapshot universe as follows:

$$Frames^\Delta = \{\aleph \in Cuts^\Delta \mid Reach(\aleph) = S^\Delta\}$$

Notice that an interaction framework may contain more components of a given component type than is permitted in any snapshot by the multiplicity constraints.

1.3.6 Minimal snapshots for a cut

If $\aleph$ is a cut of $S \in S^\Delta$, we say that the snapshot S' *develops S below* $\aleph$ and write $S \sqsubseteq_\aleph S'$ if:

- $S \sqsubseteq_{S'} S'$
  i.e. S' develops S (and, in particular, $C_S^{\aleph} \subseteq \underline{S}'$)
- $\underline{S} \nmid \lfloor_S C_S^{\aleph} = \underline{S}' \nmid \lfloor_{S'} C_S^{\aleph}$
  i.e. the components of S and S' which are not above $C_S^{\aleph}$ are the same

We define the (downward) extension of a snapshot $S \in S^{\Delta}$ below $\aleph \in Cuts^{\Delta}$ as:

$$S\downarrow^{\aleph} = \{ S' \in S^{\Delta} \mid S \sqsubseteq_{\aleph} S' \}$$

Clearly, $\aleph$ cuts $S \Rightarrow \aleph$ cuts S', $\forall S' \in S\downarrow^{\aleph}$.

A snapshot S_0 is minimal for $\aleph \in Cuts^{\Delta}$ if:

- $\aleph$ cuts $S_0 \;\wedge\; S' \sqsubseteq_{\aleph} S_0 \;\wedge\; \aleph$ cuts $S' \;\Rightarrow\; S' = S_0$

Let the set of snapshots which are minimal for $\aleph$ be denoted $minsnap(\aleph)$. If $\aleph$ cuts $S \in S^{\Delta}$ then $Reach(\aleph(S)) = \cup_{S \in minsnap(\aleph(S))} S\downarrow^{\aleph}$, where $\aleph(S)$ is the partition of S defined by $\aleph$.

1.3.7 Decomposition of cuts

Let $\aleph 1$, $\aleph 2$ cut $S \in S^{\Delta}$. We say that $\aleph 2$ *decomposes* $\aleph 1$ w.r.t. S or $\aleph 2$ is a decomposition of $\aleph 1$ w.r.t. S, and write $\aleph 1 \preccurlyeq_S \aleph 2$ if:

- $C_S^{\aleph 1} \langle\langle_S C_S^{\aleph 2}$
  i.e. $C_S^{\aleph 2} \lfloor_S C_S^{\aleph 1} = C_S^{\aleph 2}$, in other words, all of $C_S^{\aleph 2}$ is below $C_S^{\aleph 1}$

Since $\aleph 1$ and $\aleph 2$ are cuts of S, $C_S^{\aleph 2} \lfloor_S C_S^{\aleph 1} = C_S^{\aleph 2} \;\Rightarrow\; C_S^{\aleph 1} \lfloor_S C_S^{\aleph 2} = \varnothing$ (in other words, if all of $C_S^{\aleph 2}$ is below $C_S^{\aleph 1}$, none of $C_S^{\aleph 1}$ is below $C_S^{\aleph 2}$), i.e.:

$$\aleph 1 \preccurlyeq_S \aleph 2 \;\Leftrightarrow\; C_S^{\aleph 1} \preccurlyeq_S C_S^{\aleph 2}$$

so we could have used the relation $\preccurlyeq_S$ between characteristic sets for S instead of the relation $\langle\langle_S$ in the definition of decomposition.

If $\aleph 1$, $\aleph 2$ cut $S \in S^{\Delta}$ and $\aleph 2$ *decomposes* $\aleph 1$ w.r.t. S, then the reach of the partitions $\aleph 1(S)$ and $\aleph 2(S)$ defined by $\aleph 1$ and $\aleph 2$ are related as follows:

$$\forall\, S \in minsnap(\aleph 1(S)),\;\; \exists\, S' \in minsnap(\aleph 2(S))\,,\quad S' \in S\downarrow^{\aleph 1(S)}$$

Let $\aleph 1$, $\aleph 2 \in Frames^{\Delta}$. We say that $\aleph 2$ *decomposes* $\aleph 1$ or $\aleph 2$ is a decomposition of $\aleph 1$, and write $\aleph 1 \preccurlyeq \aleph 2$ if:

- $\aleph 1 \preccurlyeq_S \aleph 2 \quad \forall S \in S^{\Delta}$

If $\aleph 1 \preccurlyeq \aleph 2$ and $c \in C_S^{\aleph 1} - C_S^{\aleph 2}$ then we say that c is decomposed (into subcomponents) in $\aleph 2$.

The decomposition relation defines a partial order on $Frames^{\Delta}$. In fact, multiplicities being bounded, $(Frames^{\Delta}, \preccurlyeq)$ is a finite lattice with a maximum and mimimum element defined by the characteristic sets $\{c^{\top}\}$ and $S_{\perp}^{\Delta}$, where the latter is the entire set of base-level components of S^{Δ}.

1.3.8 Containing element of an interaction framework

Since snapshots are trees, if $S \in Reach(\aleph)$ and $c \in \underline{S} \lfloor_S C_S^{\aleph}$ then $C_S^{\aleph} \lceil_S c$ is a singleton. That is, if a component of a snapshot is contained in an element of the characteristic set of a cut, it

is not contained in any other element of that characteristic set. For a component set $C \subseteq C^{\Delta}$ and a cut $\aleph \in Cuts^{\Delta}$, define $Reach(\aleph)|C$ by:

$$Reach(\aleph)|C = \{ S \in Reach(\aleph) \mid C \subseteq \underline{S} \}$$

Then, by the nature of component structures:

$$S, S' \in Reach(\aleph)|\{c\} \Rightarrow c \in \underline{S} \downharpoonright_S C_S^{\aleph} \Leftrightarrow c \in \underline{S}' \downharpoonright_{S'} C_{S'}^{\aleph}$$

that is, a component cannot be contained in an element of the characteristic set of a cut for one snapshot and not be contained in an element of the characteristic set of that cut for another snapshot. Furthermore, the properties of component developments ensure that:

$$S, S' \in Reach(\aleph)|\{c\} \Rightarrow C_S^{\aleph} \upharpoonright_S c = C_{S'}^{\aleph} \upharpoonright_{S'} c$$

that is, if a component c is contained in a certain element of the characteristic set of a cut for one snapshot of S^{Δ}, it is contained in the same element of the characteristic set of that cut for any snapshot in $Reach(\aleph)|\{c\} \subseteq S^{\Delta}$. Finally, if $\aleph \in Frames^{\Delta}$:

$$\forall c \in C^{\Delta}, \forall S \in S^{\Delta}, \ c \in \underline{S} \Rightarrow C_S^{\aleph} \upharpoonright_S c \neq \varnothing \ \vee \ C_S^{\aleph} \downharpoonright_S c \neq \varnothing$$

that is, if any component of the component universe is present in any snapshot of the snapshot universe, then it is either above or below the characteristic set of the interaction framework for that snapshot.

For each $\aleph \in Frames^{\Delta}$, we can therefore define the function $\rho_{\aleph} : C^{\Delta} \mapsto C^{\aleph} \cup \{c^{\top}\}$ taking a component to the element of the characteristic set of the interaction framework that contains that component in some snapshot, if there is such a snapshot, and to the top-level component if not:

$$\begin{aligned} \rho_{\aleph}(c) &= c', \ \text{if } \forall S \in Reach(\aleph)|\{c\}, C_S^{\aleph} \upharpoonright_S c = \{c'\}, \\ &= c^{\top}, \ \text{if } \forall S \in Reach(\aleph)|\{c\}, C_S^{\aleph} \upharpoonright_S c = \varnothing \end{aligned}$$

Note that:

$$\rho_{\aleph 1}(c_2) = c_1 \in C^{\aleph 1} \quad \Leftrightarrow \quad \exists \aleph 2 \text{ s.t. } \aleph 1 \preccurlyeq \aleph 2 \text{ and } c_2 \in C^{\aleph 2}.$$

That is, a component is contained in a component which is a member of the characteristic set of an interaction framework iff there is a decomposition of that interaction framework for which it is a member of the characteristic set.

1.4 Generalised cuts and generalised interaction frameworks

If SUT lifelines can represent ports, the set of lifelines of a sequence diagram, or those of a set of sequence diagrams, should define a generalised interaction framework for the component development, as defined below. This generalised interaction framework is not necessarily a generalised partition of any of the snapshots that it cuts.

1.4.1 Generalised cuts

Let $S \in S^{\Delta}$, with component set $\underline{S}$ and port set π_S. Let $\underline{S}_{\perp}$ be the set of base-level components of S. A surjective function $\chi: C^{\Delta} \cup P^{\Delta} \mapsto \{0,1\}$ is said to be a *generalised cut* of S if the following are true.

- $C_S^{\chi} \cap ComP_S^{\chi} = \varnothing$
  No component selected by χ has ports selected by χ.
- $\forall c \in Com_S^{\chi}, \ \ Com_S^{\chi} \downharpoonright c = c$

If c is selected by χ or has ports selected by χ, this is not the case for any of its proper descendants.

- $Com_S^\chi \preccurlyeq \underline{S}_\perp$
  Every base-level component of the snapshot has an ancestor in S (maybe itself) that is selected by χ or has ports so selected.

where $C^\chi = \chi^{-1}(1) \cap C^\Delta$, $P^\chi = \chi^{-1}(1) \cap P^\Delta$ are the characteristic sets of χ

$C_S^\chi = C^\chi \cap \underline{S}$, $P_S^\chi = P^\chi \cap \pi_S$ are the characteristic sets of χ for S

$ComP_S^\chi = \cup\{p \in P_S^\chi\}\ ownsports_S^{-1}(p)$ is the set of components of S with ports selected by χ

$Com_S^\chi = C_S^\chi \cup \mathrm{Com}P_S^\chi$ is the set of components of S selected by χ or with ports selected by χ

The last two conditions simply state that the function $\aleph(S)$: $C^\Delta \mapsto \{0,1\}$ with characteristic set $\mathrm{C}^\aleph = \aleph(S)^{-1}(1) = Com_S^\chi$ is a partition of S, as defined in Section 1.3.4. For χ a generalised cut, $\aleph(S)$ is called the covering partition of χ for S. Notice that for a generalised cut we do not oblige $P_S^\chi = ownsports_S(ComP_S^\chi)$, i.e. if one of the ports of a component of S is selected by the generalised cut, we do not demand that all the ports of that component in S are selected by the generalised cut.

1.4.2 Generalised partitions and base-level generalised cuts/partitions

Let $GCuts^\Delta$ be the set of functions $\chi : C^\Delta \cup P^\Delta \mapsto \{0,1\}$ which are generalised cuts of some snapshot in S^Δ. Given a snapshot $S \in S^\Delta$, we define an equivalence relation $\equiv_S$ on $GCuts^\Delta$ by regarding two generalised cuts as equivalent if they agree on S, i.e. $\chi 1 \equiv_S \chi 2$ if $C_S^{\chi 1} = C_S^{\chi 2}$ and $P_S^{\chi 1} = P_S^{\chi 2}$, that is, their characteristic sets for S are the same. Denote by $[\chi]_S$ the equivalence class of a generalised cut χ under $\equiv_S$. We can define a unique representative of the equivalence class $[\chi]_S$, denoted $\chi(S)$, by defining its characteristic sets as follows:

$$C^{\chi(S)} = C_S^{\chi(S)} \quad \text{and} \quad P^{\chi(S)} = \cup\{S \in S\downarrow^{\aleph(S)}\}ownsports_S(ComP_S^\chi)$$

where χ is any member of $[\chi]_S$ and $\aleph(S)$ is the covering partition of χ for S[3]. $\chi(S)$ is said to be the generalised partition of S defined by any member of $[\chi]_S$ on S (we also sometimes say that $C_S^{\chi(S)} \cup \cup\{S \in S\downarrow^{\aleph(S)}\}ownsports_S(ComP_S^\chi)$ is a generalised partition of S). Trivially, a cut of S is a generalised cut of S and a partition of S is a generalised partition of S (for which $\chi(p) = 0\ \forall p \in P^\Delta$).

If $\aleph$ cuts $S \in S^\Delta$ and if $\underline{S}_\Phi \neq \varnothing$, where $\underline{S}_\Phi$ is the set of non-standard base-level components of S, then for any $C_\Phi \subseteq \underline{S}_\Phi$, putting $C^\chi = C_S^\chi = C_S^\aleph - C_\Phi$ and $P^\chi = P_S^\chi = \cup_{c \in C\Phi}\ ownsports_S(c)$ defines a generalised partition χ of S. Non-standard base-level components are logical candidates for the set $ComP_S^\chi$, that is, to be decomposed into ports. An example is to be found in the whole SUT component, which may be represented as a non-standard base-level component.

A generalised cut χ of $S \in S^\Delta$ is called a *base-level generalised cut* of S, if all the elements of the characteristic set of the covering partition for S, Com_S^χ, are base-level components. For

[3] If an element of $ComP_S^\chi$ has more immediate subcomponents in some extension of S than it does in S then it may also have more ports in that snapshot than it does in S. We must therefore use $S\downarrow^{\aleph(S)}$ if we require a generalised partition of S to also be a generalised partition of any extension of S, as is the case for partitions.

each snapshot S, for any $C_\perp \subseteq \underline{S}_\perp$, putting $C^\chi = \underline{S}_\perp - C_\perp$ and $P^\chi = \cup_{c\in C_\perp} ownsports_S(c)$ defines a base-level generalised partition of S.

In the case of a simple object model in which the port identifiers of the non base-level components are unified with the identifiers of the base-level components they contain, all generalised partitions are also viewed as partitions (since port IDs and base-level object IDs are unified).

1.4.3 Generalised interaction frameworks

For $\chi \in GCuts^\Delta$, we define $Reach(\chi)$ in S^Δ and $GFrames^\Delta \subseteq GCuts^\Delta$ in an analogous way to $Reach(\aleph)$ and $Frames^\Delta$ for cuts. $GFrames^\Delta$ is the set of *generalised interaction frameworks* for the component development of which S^Δ is the snapshot universe.

1.4.4 Minimal snapshots for a generalised cut

If χ is a generalised cut of $S \in S^\Delta$, then we say that the snapshot S' *develops* S *below* χ and write $S \sqsubseteq_\chi S'$ if $S \sqsubseteq_{\aleph(S)} S'$, where $\aleph(S)$ is the covering partition of χ for S. We can then define the (downward) extension of a snapshot S below a generalised cut, $S{\downarrow}^\chi$, in the same way as for a cut. Then, χ is a generalised cut of $S \Rightarrow \chi$ is a generalised cut of S', $\forall S' \in S{\downarrow}^\chi$. The notion of minimal snapshot for a generalised cut is defined in the same way as for a cut and, similarly, we denote by *minsnap*(χ) the set of minimal snapshots for the generalised cut χ. As for cuts, if χ is a generalised cut of S then $Reach(\chi(S)) = \cup_{S\in minsnap(\chi(S))} S{\downarrow}^\chi$, where $\chi(S)$ is the generalised partition defined by χ.

1.4.5 Decomposition of generalised cuts

Let $\chi_1, \chi_2 \in GCuts^\Delta$. For $S \in S^\Delta$, as previously we define:

$$P_S^{\chi i} = \chi_i^{-1}(1) \cap \pi_S, \quad C_S^{\chi i} = \{\chi_i^{-1}(1) \cap \underline{S}, \quad i = 1,2$$

$$ComP_S^{\chi i} = \cup\{p \in P_S^{\chi i}\}\ ownsports_S^{-1}(p), \quad i = 1,2$$

$$C^{\aleph(S)i} = Coms_S^{\chi i} = C_S^{\chi i} \cup ComP_S^{\chi i} \quad i = 1,2$$

We say that χ_2 decomposes χ_1 w.r.t. S or that χ_2 is a decomposition of χ_1 w.r.t. S and write $\chi_1 \preccurlyeq_S \chi_2$ if:

- $C^{\aleph(S)1} \preccurlyeq_S C^{\aleph(S)2}$
  i.e. the covering partition of χ_1 covers (and caps) the covering partition of χ_2.
- $P_S^{\chi 1} \subseteq P_S^{\chi 2}$
  i.e. the set of ports selected by χ_1 is a subset of the set of ports selected by χ_2.

If, in addition, we have that $c \in ComP_S^{\chi 1} \cap ComP_S^{\chi 2} \Rightarrow (ownsports_S(c) \cap P_S^{\chi 1}) = (ownsports_S(c) \cap P_S^{\chi 2})$ i.e. if some ports of a component are selected by χ_1, exactly the same ports of that component are selected by χ_2, we say that χ_2 is an *owned-ports preserving decomposition* of χ_1 w.r.t. S.

If χ_1, χ_2 are generalised cuts of S, then the reach of the generalised partitions $\chi_1(S)$, $\chi_2(S)$ defined by χ_1 and χ_2 are related as follows:

$$\forall S \in minsnap(\chi_1(S)),\ \exists S' \in minsnap(\chi_2(S)), \quad S' \in S{\downarrow}^{\chi 1(S)}$$

Let $\chi_1, \chi_2 \in Frames^\Delta$. We say that χ_2 decomposes χ_1 or that χ_2 is a decomposition of χ_1 and write $\chi_1 \preccurlyeq \chi_2$ if:

- $\chi_1 \preccurlyeq_S \chi_2 \quad \forall S \in S^\Delta$

If $\chi_1 \preccurlyeq \chi_2$ and $c \in C_S^{\chi 1} - C_S^{\chi 2}$ then we say that c is decomposed (into subcomponents or ports) in χ_2. The decomposition is owned-ports preserving if it is owned-ports preserving $\forall S \in S^\Delta$.

The decomposition relation defines a partial order on $GFrames^\Delta$ with a maximum element defined by the characteristic sets $\{c^\top\}$ and $\varnothing$ and minimum elements defined by the characteristic sets $\varnothing$ and $Ports_\perp$, where the latter is any set of ports of base-level components s.t. each base-level component contributes at least one port. If there are no non-standard base-level components, all base-level components have a single port and the partial-order $(GFrames^\Delta, \preccurlyeq)$ has a unique minimal element.

1.4.6 Containing element of a generalised interaction framework

Let $\chi \in GFrames^\Delta$. Let $\aleph(S) \in Cuts^\Delta$ denote the covering partition of χ for $S \in S^\Delta$. By the nature of snapshots and component developments, we can define the interaction framework $\aleph(\chi) \in Frames^\Delta$ by:

$$\aleph(\chi)(c) = \aleph(S)(c), \text{ where } S \in S^\Delta \text{ is any snapshot s.t. } c \in \underline{S}$$

We say that $\aleph(\chi)$ is the covering interaction framework of χ. Given $\chi \in GFrames^\Delta$, we can then define the function $\rho_\chi : C^\Delta \mapsto C^{\aleph(\chi)} \cup \{c^\top\}$, taking a component to the element of the characteristic set of the covering interaction framework $\aleph(\chi)$ of χ that contains that component in some snapshot, if there is such a snapshot, and to the top-level component, if not, by $\rho_\chi(c) = \rho_{\aleph(\chi)}(c)$.

Note that:

$$\rho_{\chi 1}(c_2) = c_1 \in C^{\aleph(\chi 1)} \quad \Leftrightarrow \quad \exists \chi_2 \text{ s.t. } \chi_1 \preccurlyeq \chi_2 \text{ and } c_2 \in C^{\aleph(\chi 2)}$$

where $\aleph(\chi_1)$, resp. $\aleph(\chi_2)$, is the covering interaction framework of χ_1, resp. χ_2. That is, a component c_2 is contained in a component $c_1$of the characteristic set of the covering partition of χ_1 iff there is a decomposition, χ_2, of χ_1 s.t. c_2 is an element of the characteristic set of the covering partition of χ_2.

1.5 Structural semantics of TeLa sequence diagrams

1.5.1 Test description interaction frameworks

As discussed previously, the test description component model underlying a TeLa test description is either that of the test architecture or that of the test architecture together with the SUT internal structure. This component model, together with the initial snapshot and the creation/destruction behaviour of the test description defines a component development. We assume this component development is consistent with the component model. We assume that the test description component model contains two distinguished component types, namely the tester component type and the SUT component type, both of multiplicity (1,1). Every component snapshot contains precisely one whole tester component and one whole SUT component.

The structure of a sequence diagram comprises a set of vertical lines, or lifelines, each labelled with a component or port name. Given a sequence diagram, we define a diagram-cut as a continuous line extending from its left border to its right border which does not bisect any message arrows (c.f. the notion of "valid cut" of [HélLeM01]. To be structurally consistent with a test description component development, the set of lifelines bisected by any diagram-cut must represent a generalised partition of a snapshot of that component development. If this is the case, then the set of lifelines of a single sequence diagram, or those of a set of sequence diagrams, defines a generalised interaction framework for the component development (but not necessarily a generalised partition of any of the snapshots of that framework).

1.5.1.1 TeLa sequence diagrams with a single SUT lifeline

In a TeLa diagram, if the SUT is represented via a single lifeline, each diagram-cut corresponds to a partition of a snapshot of the test description component development. The set of lifelines of the diagram defines an interaction framework (a specific type of generalised interaction framework) of the test description component model. Thus, the characteristic set of the covering partition of this interaction framework comprises the whole SUT component and one or several tester components, depending on the level of decomposition chosen.

Any internal structure of the SUT component type is not represented in the diagram via lifelines, whether or not it exists. However, whether it exists or not is important to how the diagram can be decomposed into another diagram, see Section 1.5.3. Recall that if it does not exist, the SUT component type is a base-level component type, standard, or non-standard, according to whether it owns one port type with multiplicity (1,1), or several port types.

1.5.1.2 TeLa sequence diagrams with multiple SUT lifelines

A TeLa diagram in which the SUT is not represented via a single lifeline, can be of two types.

1. If the SUT is represented via multiple lifelines, each representing a component, then, as in the previous case, each diagram-cut corresponds to a partition of a snapshot of the test description component development. The set of lifelines of the diagram again defines an interaction framework of the test description component model. The characteristic set of the covering partition of this interaction framework comprises several SUT components (the SUT is decomposed into components) and one or several tester components, depending on the level of decomposition chosen. Clearly, this representation presupposes the existence of an SUT component model defining the internal structure of the SUT component type.

2. If the SUT is represented via multiple lifelines, each representing a port, each diagram-cut corresponds to a generalised partition of a snapshot of the test description component development (which, in this case, is not a partition). The set of lifelines of the diagram defines a generalised interaction framework (which, in this case, is not an interaction framework). The characteristic sets of the generalised interaction framework comprises exactly the set of ports of the whole SUT component which constitutes the so-called "SUT component interface", and one or several tester components, depending on the level of decomposition chosen.

   Any internal structure of the SUT component type of the test description component model is not represented in the diagram, whether or not it exists. If it does exist, then the choice of a generalised partition in which the whole SUT component is decomposed into ports is a choice to explicitly represent this component as opaque to further analysis in spite of the existence of this SUT component model.

1.5.1.3 TESTER LIFELINES IN TELA SEQUENCE DIAGRAMS

Notice that the tester lifelines always represent components. The only type of diagram in which the tester lifelines can be viewed as representing ports is when each lifeline represents a standard base-level component, since in this case component IDs coincide with prot IDs. In particular, this will be the case in the simple object model. Though the definitions of generalised partition allow any component to be represented via its ports, we only use this possibility for the whole SUT component, never for any tester component, since the architecture of the tester is known to the test designer.

1.5.2 Finiteness of snapshots

In sequence diagrams, dynamic creation must be specified explicitly. Clearly, then, the fact that a diagram has a fixed finite number of lifelines means that it can only *explicitly* represent a fixed, finite number of components. However, the use of a hierarchical component model and of lifelines representing elements of cuts of snapshots from this model means that a TeLa sequence diagram could, in theory, represent an unbounded number of components. Though this is an interesting feature we do not make use of it here and we have defined multiplicities in such a way as to impose an upper limit on the number of possible components in a snapshot, in order to simplify the presentation.

1.5.3 Decomposition of diagrams

We have defined the notion of decomposition of partitions with the idea of using MSC-style decomposition in TeLa sequence diagrams, according to the hierarchy defined by the component model. The notion of decomposition gives us the possiblity of choosing the level of structural abstraction at which we wish to describe a test in a TeLa sequence diagram.

As described above, the set of lifelines of a TeLa sequence diagram defines a generalised interaction framework of the corresponding component development. The decomposition relation between generalised interaction frameworks lifts to a decomposition relation between sequence diagrams describing the same test. For obvious reasons, we discount the trivial partition which selects only the top-level component (the component which includes the whole SUT component and the whole tester component).

Regarding internal actions, though we have not included the following details in the semantics we have defined here, we note for future extension, the following details. A lifeline owning an assertion internal action can sometimes be decomposed into multiple lifelines and the sending of a coordination message between two of them. A lifeline owning a creation internal action can be decomposed into multiple lifelines including the creator and the created component and the sending of a creation message from creator to created.

A particular case of interest concerns the decomposition of a diagram in which the whole SUT component is represented via a single lifeline and in which the decomposition is applied to this component. If an SUT component model does not exist, the SUT component can only be decomposed into ports. If an SUT component model does exist, the SUT component can be decomposed either into ports or into components depending on whether or not we wish to show the internal structure of the whole SUT component.

1.6 Structural consistency of TeLa sequence diagrams

The set of lifelines of a TeLa sequence diagram defines a generalised interaction framework w.r.t. a component development. We now investigate the relation of the actions to the (generalised) partition defining this interaction framework.

1.6.1 TeLa test description actions

The alphabet of actions Σ is the disjoint union of the set of tester actions Σ^T and the set of SUT actions Σ^{SUT}, each the disjoint union of the set of component-internal actions ($\Sigma^T{}_{int}$ and $\Sigma^{SUT}{}_{int}$), reception actions ($\Sigma^T{}_?$ and $\Sigma^{SUT}{}_?$) and emission actions ($\Sigma^T{}_!$ and $\Sigma^{SUT}{}_!$). Both reception and emission actions can be of three types: *synchronous invocation*, *asynchronous invocation* and *synchronous invocation reply*[4]. When dealing with the projected semantics, that is, the semantics in terms of tester action only, we will often omit the superscript T.

SUT component-internal actions can only occur in the case where SUT internal structure is explicitly represented and can only be creation actions. Tester component-internal actions can be used to model the TeLa internal actions “assign”, “assert”, “create” and “escape”, where the last of these is used to include other-language fragments. They could also be used to model timer actions, though we do not deal with such actions here.

In the non-enumerated data case, each element of Σ has an associated (possibly empty) tuple of types called the *signature of the action*[5]. In the enumerated data case, the signature is not needed since any parameters are considered part of the action name. In the non-enumerated data case, the message parameters of the syntax become the action parameters associated to the semantic actions. In the enumerated data case, on the other hand, the message parameters of the syntax are part of the semantic actions themselves.

1.6.1.1 PROPER AND IMPROPER ACTIONS

Both SUT and tester, emission and reception actions may, or may not, be proper. For the tester, a *proper emission action* resp. *proper reception action* is defined to be an emission, resp. reception, action on a tester component for which the communication partner is an SUT entity, i.e. it does not concern a tester-internal message. For the SUT, a *proper emission action* resp. *proper reception action* is defined to be an emission, resp. reception, action on an SUT component for which the communication partner is a tester entity, i.e. it does not concern an SUT-internal message. Other emissions or receptions are said to be *improper*.

1.6.1.2 ACTIONS AND OWNING PORTS / COMPONENTS

Component-internal actions are assumed to include information about the owning component. Emission and reception actions are assumed to include information about the sending and receiving port of the corresponding message, except for proper tester reception actions. The latter actions may only include information about the receiving tester port, depending on the underlying communication system. We suppose that each port is owned by a component, the connection between the two being specified statically in a component diagram and dynamically by a naming convention.

---

[4] We do not distinguish signals and asynchronous invocations.

[5] Though syntactically the same operation name may be used for several operations with differing signature, in such cases, a different name is used in the semantics.

Thus, an emission action is a triple (*s*, *r*, *m*) where *s* is the sending port, *r* is the receiving port and *m* is the message. In the case of a tester emission action, the emission is proper or improper according to whether *r* is an SUT or tester port. In the case of an SUT emission action, the emission is proper or improper according to whether *r* is a tester or SUT port. We will use the notation !*s*.*r*.*m*

An SUT reception action is also a triple (*s*, *r*, *m*) in which *s*, *r* and *m* are as above. The action is proper or improper according to whether *s* is a tester or SUT port. We will use the notation ?*s*.*r*.*m*

A tester reception action may be a triple of the form (*s*, *r*, *m*) in which *s*, *r* and *m*, are as above. The action is proper or improper according to whether *s* is an SUT or tester port. Depending on the system, a proper tester reception action may instead be a pair of the form (*r*, *m*), in which *r* is the receiving tester port and *m* is a message. We will use the notation ?*s*.*r*.*m* for the first type of tester reception actions and ?*r*.*m* for the second type of tester receptions. Proper tester receptions are ordered pairs in systems whose underlying communication mechanism is such that information about the sending SUT port is not available[6].

Sending ports are always ports of base-level components. If receiving ports are not ports of base-level components, it is supposed that the receiving component has a policy for unambiguously channelling the corresponding message to a base-level component.

A component-internal action is a pair (*c*, *a*), where *c* is a component and *a* is a local action.

1.6.1.3 OBSERVABLE ACTIONS AND CONTROLLABLE ACTIONS

The set of proper tester reception actions, also called *observable actions*, is denoted Σ_{obs}. All other tester actions belong to the set of *controllable actions*, denoted Σ_{con}, including the improper tester receptions, i.e.:

$$\Sigma_{con} = \Sigma_{!} \cup \Sigma_{int} \cup \Sigma_{?} - \Sigma_{obs}.$$

The improper tester receptions and emissions, along with the tester component-internal actions constitute the *improper controllable actions*, while the proper tester emissions constitute the *proper controllable actions*. Improper controllable actions may also be called tester internal actions. A *proper tester action* is either a proper tester emission or proper tester reception, ie. either a proper controllable action or an observable action.

The improper tester reception actions are in some sense special among the controllable actions. For example, due to the nature of choice in TeLa, there can be no minimal conflict involving an event labelled by an improper reception action. We therefore define the *primary controllable actions* to be the tester emissions together with the tester component-internal actions.

1.6.1.4 CREATION AND DESTRUCTION ACTIONS

Component creation can be described in two ways in TeLa test descriptions. The first way of describing creation is via a creation message sent by the creating component and received by the created component, similar to the UML sequence diagram notation. In the same way as for any other send or receive action, the sending port referred to by a send or receive action of a create message is always a base-level port. The receiving port (the port of the created component) is subject to the following restriction: if the snapshot before the creation is *S* and

[6] Note that even in the case where this information is not available in the actual test, it may be available in the specification of the application and therefore useable by test synthesis.

in the creation action, the base-level component $c_{creator}$ creates the component c_{new} of type $\alpha \in$ X, then:

$$\exists\beta \in X(\ \beta <_X \text{type}(c_{new}) \ \wedge\ \exists c \in S\ (\ \text{type}(c) = \beta \wedge\ c <_S c_{creator})\ ) \ \Rightarrow\ c <_S c_{new}.$$

where a snapshot is a legal instance of a component model, $\leq_X$ denotes containment of component types and $\leq_S$ denotes containment of components. That is, if the creator component is properly contained in a component, c, whose type contains the type of the created component, the created component must also be properly contained in c. This means, in particular, that a tester component cannot create SUT components and an SUT component cannot create tester components.

The second way is via a component-internal action. The first two arguments of this internal action are the sending port and the receiving port; these ports are those of subcomponents of the component owning the component-internal creation action and are subject to the same restrictions as are the ports involved in the send or receive actions of creation messages.

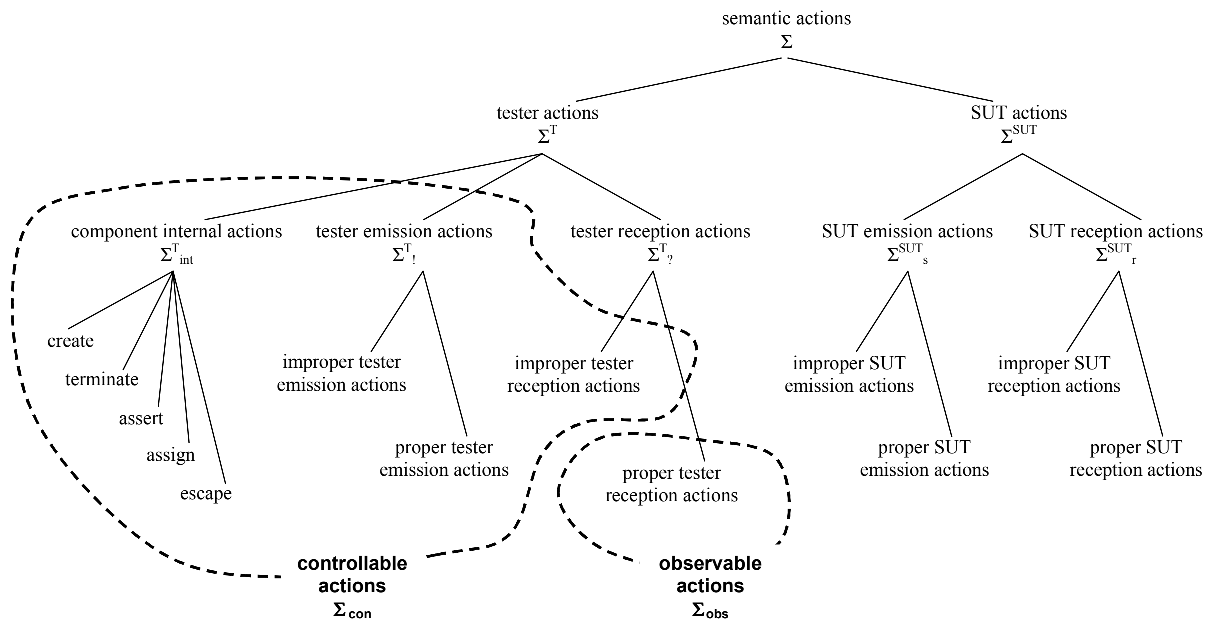


Figure 5-1: The set of actions of TeLa. The semantics after projection contains only the tester actions: those of the left branch.

Figure 5-1 shows the different TeLa actions, as described in this section. As in MSCs and UML 1.4 sequence diagrams, there is no disabling or destruction action apart from the termination action performed by a component itself. As in MSCs and UML 1.4 sequence diagrams, this action is denoted by a cross terminating the lifeline. Note that termination of a component involves the termination of all its subcomponents.

1.6.2 Consistency of actions with an interaction framework

We first assume that the TeLa sequence diagram contains no lifelines representing ports so that the set of lifelines defines an interaction framework. As already stated, the sub-case where the SUT is represented via multiple lifelines presupposes the existence of the SUT component model.

Recall the definition of *Reach*($\aleph$)|C, for a cut $\aleph$ and a set of components $C \subseteq C^{\Delta}$ of the component universe, and of the function $\rho_{\aleph} : C^{\Delta} \mapsto C^{\aleph}$ taking a component onto its representative element in an interaction framework.

An emission action, $!s.r.m$, or a reception action of the form $?s.r.m$, is said to be consistent with an interaction framework $\aleph_1$ if there exists a decomposition $\aleph_2$ of $\aleph_1$ s.t. $c_s, c_r \in C^{\aleph 2}$, where c_s, c_r are the owning components of s and r (i.e. $c_s = comport_S(s)$, $c_r = comport_S(r)$, for some S) respectively, and $C^{\aleph 2}$ is the characteristic set of $\aleph_2$. This is iff:

- $Reach(\aleph_1)|\{c_s, c_r\} \neq \emptyset$
- $\exists\, c_s^{\aleph 1}, c_r^{\aleph 1} \in C^{\aleph 1},\ \rho_{\aleph 1}(c_s) = c_s^{\aleph 1}$ and $\rho_{\aleph 1}(c_r) = c_r^{\aleph 1}$

For a tester emission action, c_s is a tester component, and c_r is a tester or SUT component, according to whether the action is improper or proper (i.e. to whether it is part of an inter-tester coordination message or not). For a tester reception action, c_r is a tester component and c_s is a tester or SUT component, according to whether the action is improper or proper (i.e. to whether it is part of an inter-tester coordination message or not). For an SUT emission action, c_s is an SUT component and c_r is an SUT or tester component, according to whether the action is improper or proper (i.e. to whether it is part of an inter-SUT coordination message or not). For an SUT reception, c_r is an SUT component and c_s is an SUT or tester component according to whether the action is improper or proper (i.e. to whether it is part of an inter-SUT coordination message or not).

A proper tester reception of the form $?m.r$ is said to be consistent with an interaction framework $\aleph_1$ if there exists a decomposition $\aleph_2$ of $\aleph_1$ s.t. $SUT, c_r \in C^{\aleph 2}$, where SUT is the whole SUT component (which may, or may not, be a base-level component), c_r is the owning component of r and $C^{\aleph 2}$ is the characteristic set of $\aleph_2$. This is iff:

- $SUT \in C^{\aleph 1}$ (we assume that $\aleph 1$ is not the trivial partition)
- $\exists c_r^{\aleph 1} \in C^{\aleph 1},\ \rho_{\aleph 1}(c_r) = c_r^{\aleph 1}$ (this implies that $Reach(\aleph_1)|\{c_r\} \neq \emptyset$)

Clearly, c_r is a tester component. Notice that if the system is such that proper tester receptions are of this form, the whole SUT component cannot be decomposed in TeLa sequence diagrams.

A component-internal action (c, la) is said to be consistent with an interaction framework $\aleph_1$ if there exists a decomposition $\aleph_2$ of $\aleph$ s.t. $c \in C^{\aleph 2}$ where $C^{\aleph 2}$ is the characteristic set of $\aleph_2$. This is iff:

- $\exists\, c^{\aleph 1} \in C^{\aleph 1},\ \rho_{\aleph 1}(c) = c^{\aleph 1}$ (this implies that $Reach(\aleph_1)|\{c\} \neq \emptyset$)

If the action is a creation action, c may be an SUT component or a tester component. Otherwise, c is a tester component.

If, syntactically, “assert” and “asign” internal actions are permitted to straddle several lifelines, semantically, the corresponding component-internal actions can be considered to be consistent with any interaction framework.

1.6.3 Consistency of actions with a generalised interaction framework

We now assume that the TeLa sequence diagram contains lifelines representing SUT ports so that the set of lifelines defines a generalised interaction framework.

Recall the definition of $Reach(\chi)|C$, for a generalised partition χ and a set of components $C \subseteq C^{\Delta}$, and of the function ρ taking a component onto its representative element in a generalised interaction framework.

An improper tester emission or reception action, a proper tester reception of the form $?r.m$ and a tester component-internal action, are said to be consistent with a generalised interaction framework if they are consistent with its covering interaction framework.

A proper tester emission action, $!s.r.m$, or proper SUT reception action, $?s.r.m$ is said to be consistent with a generalised interaction framework χ_1 if there exists an owned-ports preserving decomposition χ_2 of χ_1 s.t. $c_s \in C^{\chi 2}$ and $r \in P^{\chi 2}$, where c_s is the owning component of s, and $C^{\chi 2}$, $P^{\chi 2}$ are the characteristic sets of χ_2. This is iff:

- $r \in P^{\chi 1}$, so r is a port of the whole SUT component
- $\exists c_s^{\chi 1} \in C^{\chi 1},\ \rho_{\chi 1}(c_s) = c_s^{\chi 1}$ (this implies that $Reach(\chi_1)|\{c_s\} \neq \emptyset$)

A proper tester reception action of the form $?s.r.m$, or a proper SUT emission action, $!s.r.m$, is said to be consistent with a generalised interaction framework χ_1 if there exists an owned-ports preserving decomposition χ_2 of χ_1 s.t. $c_r \in C^{\chi 2}$ and $s \in P^{\chi 2}$, where c_r is the owning component of r and $C^{\chi 2}$, $P^{\chi 2}$ are the characteristic sets of χ_2. This is iff:

- $s \in P^{\chi 1}$, so s is a port of the whole SUT component
- $\exists\ c_r^{\chi 1} \in C^{\chi 1},\ \rho_{\chi 1}(c_r) = c_r^{\chi 1}$ (this implies that $Reach(\chi_1)|\{c_r\} \neq \emptyset$)

An improper SUT emission action $!s.r.m$ is said to be consistent with a generalised interaction framework χ_1 if:

- $s, r \in P^{\chi 1}$, so r and s are ports of the whole SUT component

1.6.4 Location of events in a (generalised) interaction framework

An alphabet of actions is consistent with a (generalised) interaction framework of a component development if each individual action is consistent with that framework. The fact that the notion of interaction framework takes into account the dynamics (i.e. the component development) means that if an alphabet of actions is consistent with a (generalised) interaction framework, an action cannot occur on a port belonging to a component that has not yet been created on has already terminated.

Let E be the set of events of a non-interleaving semantics test description, Σ be the alphabet of actions and $f : E \rightarrow \Sigma$ be the labelling function. Let Σ be consistent with the interaction framework $\aleph$ of that test description. Let c_p denote the owning component of the port p. We define the *location function* $\vartheta_\aleph$: $E \mapsto C^\aleph$, taking an event to the element of the characteristic set of the interaction framework on which it is located, as follows:

- For an emission event, e, s.t. $f(e) = !s.r.m$: $\vartheta_\aleph(e) = \rho_\aleph(c_s)$
- For a reception, e, s.t. $f(e) = ?s.r.m$ or $a = ?r.m$: $\vartheta_\aleph(e) = \rho_\aleph(c_r)$
- For a tester component-internal event, e, s.t. $f(e) = c.la$: $\vartheta_\aleph(e) = \rho_\aleph(c)$

Now let the alphabet of actions, Σ, be consistent with the generalised interaction framework χ of the test description, where χ is not an interaction framework (i.e. SUT lifelines represent ports). Let c_p denote the owning component of the port p. We define the *location function* ϑ_χ: $E \mapsto C^\chi \cup P^\chi$, taking an event to the element of the characteristic sets of χ on which it is located, as follows:

- For any tester event, e: $\vartheta_\chi(e) = \vartheta_{\aleph(\chi)}(e) \in C^\chi$
- For an SUT emission event, e, s.t. $f(e) = !s.r.m$: $\vartheta_\chi(e) = s \in P^\chi$
- For an SUT reception action, e, s.t. $f(e) = ?s.r.m$: $\vartheta_\chi(e) = r \in P^\chi$

where $\aleph(\chi)$ is the covering interaction framework of χ

We have thus defined the location function for all events and for all types of generalised interaction frameworks.

1.6.5 Structural consistency

Recall that we assume that a TeLa diagram has an underlying component model, defined either by the test architecture on its own, or by the test architecture together with an SUT component model.

A TeLa diagram is said to be structurally consistent if:

- The creation/destruction behaviour of the diagram is consistent with the test description component model (e.g. respects the multiplicity constraints). In this case, the test description component model, the initial snapshot and the creation/destruction behaviour defines the test description component development.
- Any diagram-cut represents a generalised partition of some snapshot of the test description component development. In this case, the set of lifelines of the diagram defines a generalised interaction framework χ.
- The alphabet of actions Σ of the diagram is consistent with the interaction framework defined by the set of lifelines. In this case, the physical location of the actions on the diagram (i.e. which lifeline) conforms to that defined by location function ϑ_χ, see above.
- The location function ϑ_χ is surjective (if it is not, this means that there are components or ports which are represented in the diagram but which have no actions located on them and are therefore redundant).

In fact, consistency of a diagram should also take into account consistency between the alphabet of actions and the connectors of the component model. However, as we have not formalised this aspect of our component models, we do not include this condition in our definition of structural consistency.

A TeLa sequence diagram can be decomposed into another TeLa sequence diagram as long as the generalised interaction framework denoted by the second diagram is a decomposition of that denoted by the first diagram and structural consistency is preserved. Evidently, in the decomposed diagram the location function is redefined. Given a TeLa test description, we denote by φ, the location function of the maximally-decomposed generalised-interaction framework.

2 Elements of behavioural semantics

2.1 Outline of behavioural semantics for TeLa test descriptions

2.1.1 Introduction

In this section, we outline how a formal semantics can be defined for the TeLa language in terms of an alphabet of tester actions. In the general case, symbolic treatment of data is inevitable, firstly since a test description is usually parameterised (which we model via static variables), and secondly, since the data values contained in the parameters of messages sent by the SUT (which we model via anonymous variables and dynamic variables) cannot be known in advance, and some data types permit an unbounded number of possible data values.

In order to introduce the inevitable complexity in a progressive manner, we first outline how to define an interleaving semantics with enumerated data, then a non-interleaving semantics with enumerated data, and finally an interleaving semantics with non-enumerated data. We leave the non-interleaving semantics with non-enumerated data for future work. In all cases, we deal with the semantics after projection onto tester instances, see Chapter 3, Section 1.4. The definitions given for each model are formulated in such a way as to facilitate their generalisation to the more complex and more generic models.

Though we choose to use non-interleaving models that are classified in [SasNieWin96] as behavioural models, namely event structures, we choose to use non-interleaving models that are classified in [SasNieWin96] as system models, namely labelled transition systems. This is since we consider event structures to be the most natural abstract non-interleaving semantics for the TeLa language but we seek to generalise the conformance testing conceptual framework developed in the labelled transition system context [Jér02] [JarJér02].

2.1.2 Interleaving model with enumerated data

2.1.2.1 LABELLED TRANSITION SYSTEM BASIS

We base the following discussions on the representation of test descriptions as labelled transition systems (LTSs), similar to that of [Jér02] [JarJér02].

A labelled transition system is a quadruple M = (S, q_0, Σ, $\rightarrow$) where:

- S is a finite non-empty set of states,
- S_0 is the set of initial states; we will use models with a single initial state s_0,
- Σ is the alphabet of actions, as discussed in Section 1.6.1,
- $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation.

Unlike most LTS models, in the LTSs of [Jér02] [JarJér02], it is not assumed that all internal actions are identical, in order to facilitate the use of LTSs for modelling test objectives that may include SUT internal actions. However, in [Jér02] [JarJér02] the LTS of the synthesized test does not contain any internal actions. The LTSs we use to model test descriptions can also contain internal actions of the tester. Hence, if we were also to perform TGV-style test synthesis, the specification would need to contemplate SUT internal actions, which would be hidden as part of the test synthesis, and SUT-environment internal actions, which would not

be hidden during the test synthesis but would become tester internal actions. This would also enable us to use test synthesis after having chosen an arbitrary component of a specification as the SUT.

We will refer to an LTS derived from a test description via a projection onto tester lifelines as a test description LTS, or simply test description where no confusion can arise. Clearly, we are assuming that such an LTS can always be derived from a test description (though here we do not explore how).

2.1.2.1.1 Definition of concepts required for TeLa semantics

We say that a transition is *enabled* in its start state. By extension, if a transition labelled by an action *a* is enabled in state *s* we say that action *a* is enabled in state *s*. If the test description is deterministic, see Section 2.2.2.1, the enabled actions and enabled transitions are in 1-1 correspondence.

We consider that the tester has the initiative for firing enabled transitions labelled by controllable actions while the tester environment, i.e. the SUT, has the initiative for firing enabled transitions labelled by observable actions.

A *run* through a test description is a sequence of transitions s.t.:

- the start state of the initial element of the sequence is the initial state of the test description,
- the start state of every element of the sequence other than the initial one is the end state of the previous element of the sequence.

A *maximal run* through a test description is a run in which the last element is a terminal transition, that is a transition whose end state is a sink state.

A *trace* is the sequence of actions defined by a run.

An *execution* is a set of runs which is totally ordered by inclusion and where each element extends its predecessor in the ordering by a single transition. A *maximal execution* is an execution whose last element is a maximal run.

2.1.2.2 VERDICT ANNOTATIONS ON TEST DESCRIPTION LTSS

In order to define verdicts explicitly in the semantic domain, even if they are to remain implicit in the syntactic domain, we introduce the possibility of annotating terminal transitions – those leading to a sink state – of our test description LTSs with a verdict. We then call terminal transitions annotated with a verdict pass transitions, fail transitions or inconclusive transitions, according to the type of verdict.

A state *s* is said to be complete w.r.t. the set of observable actions Σ_{obs} if, for all $a \in \Sigma_{obs}$, there is an enabled transition of *s* labelled by *a*. A test description is said to be *test complete* if all terminal transitions are annotated with a verdict, and all states having an enabled observable action are complete w.r.t. to Σ_{obs}.

Given a test-complete test description, the last transition of each maximal run is necessarily annotated with a verdict so we can associate a verdict to each maximal run. We say that the *test verdict* of a maximal execution of a test description is the verdict associated to the last element of this execution.

In order for verdicts to be consistently defined, we impose the condition that any two maximal runs of a test-complete test description which have the same trace must have the same associated verdict. Consistently-defined verdicts can be guaranteed by imposing determinism.

In the enumerated data case, only pass and fail verdicts are implicit so if there are no explicit verdicts, we need only impose minimal determinism along with the assumption that any non-determinism is either resolved on a controllable action or leads to successful termination (implicit pass verdict) or the same explicit verdict on both branches.

A semantically test-complete test description is achieved in TeLa by using syntactically-implicit verdicts. In Section 2.3.2, we show how implicit pass and fail verdicts can be made explicit to give a semantically test-complete test description. Another way to achieve test completeness, representing a more low-level view, is to use explicit defaults as in [ETSI03a] or [UTP03].

2.1.3 Non-interleaving model with enumerated data

2.1.3.1 EVENT-STRUCTURE BASIS

As the notion of choice is important in defining controllability (see later), we base the following discussions on the representation of test descriptions as event structures, such as that of [HélJarCai02]. We do no deal with a parallel operator since, in the presence of loops, this operator introduces considerable complexity.

In our models, events are labelled by atomic actions as discussed in the introduction to this section. Note that the notion of event used here is closer to the UML notion of event occurrence than the UML notion of event. An event structure [Win87] ε is a tuple $(E, \leq, \#, f)$ where:

- E is a set of events,
- $\leq$ is a partial order relation (i.e. a reflexive, transitive and anti-symmetric relation) known as the causality relation s.t. $\{e_1 \mid e_1 \leq e_2\}$ is finite for all $e_2 \in E$,
- $\#$ is an irreflexive, symmetric relation known as the conflict relation s.t. $e_1 \# e_2 \wedge e_2 \leq e_3 \Rightarrow e_1 \# e_3$, $\forall e_1, e_2, e_3, \in E$, that is, conflicts are inherited via the causality relation; two events that are in conflict cannot appear in the same execution,
- $f : E \rightarrow \Sigma$ is a labelling function from E to the alphabet of actions Σ.

We will refer to an event structure derived from a test description via a projection onto tester lifelines as a test description event structure, or simply a test description where no confusion can arise. Clearly, we are assuming that such an event structure can always be derived from a test description (though here we do not explore how).

We denote by $\varepsilon \restriction S$ the restriction of a test description event structure ε to the set of events $S \subseteq E$, that is, the event structure defined as $(S, \leq_S, \#_S, f_S)$ where

- $\leq_S = \leq \cap (S \times S)$,
- $\#_S = \# \cap (S \times S)$
- $f_S = f \restriction S$

2.1.3.1.1 Definition of concepts required for TeLa semantics

A configuration of a test description event structure $\varepsilon = (E, \leq, \#, f)$ is a subset $C \subset E$ s.t. $\varepsilon \restriction C = (C, \leq_C, \#_C, f_C)$ satisfies:

- C is causally closed, i.e. if $e \in C$ and $e' \leq e$ for $e' \in E$, then $e' \in C$
  (if an event is in C, all its predecessors in ε are also in C)
- C is conflict free, i.e. $\#_C = \varnothing$

An event $e \in E - C$ is *enabled* in a configuration C of event structure $\mathcal{E}$ if $C \cup \{e\}$ is also a configuration. By extension, if an event labelled by an action a is enabled in configuration C we will say that action a is enabled in C. If the test description is deterministic, see Section 2.2.3.1, the enabled actions and enabled events are in 1-1 correspondence. An event $e \in C$ is a *leaf* of the configuration C of event structure $\mathcal{E}$ if $C - \{e\}$ is also a configuration.

Two events are in *minimal conflict* if they are in the conflict relation but neither is in conflict with any of the predecessors of the other. Note that two events which are enabled in a configuration are either concurrent or in minimal conflict. A *maximal configuration* is a configuration for which there are no enabled events. Below we define the looser concept of a maximal test configuration.

We consider that the tester has the initiative for executing enabled events labelled by controllable actions while the tester environment, i.e. the SUT, has the initiative for executing enabled events labelled by observable actions.

An *execution* of a test description is a set of configurations which is totally ordered by inclusion, where each element extends its predecessor in the ordering by a single event. A *maximal execution* is defined as an execution whose last element is a maximal configuration. Below we define the looser concept of a maximal test execution.

A *run* through a test description is a total ordering obtained by extending the partial ordering defined by a configuration (c.f. topological sorting of a poset).

A *maximal run* through a test description is a run obtained by extending the partial ordering defined by a maximal configuration. Below we define the looser concept of a maximal test run.

A *trace* or *linearisation* is the sequence of actions defined by a run.

2.1.3.2 VERDICT ANNOTATIONS ON TEST DESCRIPTION EVENT STRUCTURES

In order to define the pass and fail verdicts explicitly in the semantic domain, even if they are to remain implicit in the syntactic domain, we introduce the possibility of annotating terminal events – those with no successors – of our test description event structures with a verdict. Such annotations correspond to local verdicts. We then call terminal events annotated with a verdict, pass events, fail events or inconclusive events, according to the type of verdict.

A configuration C is said to be complete w.r.t. the set of observable actions Σ_{obs} if, for all $a \in \Sigma_{obs}$, there is an enabled event of C labelled by a. A test description is said to be *test complete* if all terminal events are annotated with a verdict, and all configurations having an enabled observable action are complete w.r.t. to Σ_{obs}.

Given a test-complete test description, we associate a (global) fail verdict to a configuration which includes a fail event (fail is an existential notion) and call such a configuration a fail configuration. We associate a (global) pass verdict to a maximal configuration whose terminal events are all pass events (pass is a universal notion) and call such a configuration a pass configuration. We associate a (global) inconclusive verdict to a maximal configuration which includes a inconclusive event but does not include a fail event and call such a configuration an inconclusive configuration. We can also define a provisional inconclusive configuration as a non-maximal configuration which includes a inconclusive event but does not include a fail event (a provisional inconclusive verdict can degrade to a fail verdict). We then define a *final test configuration* as a pass configuration, a fail configuration or an inconclusive configuration.

We define a *full test execution* as an execution whose largest element is a final test configuration. We say that the *test verdict* of a full test execution of a test description is the verdict associated to the last element of this execution.

A *full test run* through a test description is a run obtained by extending the partial ordering defined by a final test configuration in such a way as to ensure that if the configuration is a fail configuration, the last element is a fail event[7]. If we so wish, we could also impose this latter condition for a maximal test configuration that is an inconclusive configuration.

In order for verdicts to be consistently defined, we impose the condition that isomorphic configurations of a test-complete test description must have identical verdict annotations. This then ensures that any two full test runs that have the same trace have the same associated verdict. Consistently-defined verdicts can be guaranteed by imposing determinism. In the enumerated data case, only pass and fail verdicts are implicit so if there are no explicit verdicts, we need only impose minimal determinism along with the assumption that any non-determinism is either resolved on a controllable action or leads to successful termination (implicit pass verdict) or the same explicit verdict on both branches.

A semantically test-complete test description is achieved in TeLa by using syntactically-implicit verdicts. In Section 2.3.3, we show how implicit pass and fail verdicts can be made explicit to give a semantically test-complete test description. This is done to show that the notion of implicit verdict is well-defined. Another way to achieve test completeness, representing a more low-level view, is to use explicit defaults as in [ETSI03a] or [UTP03].

2.1.4 Interleaving model with non-enumerated data

The outline of the model presented here does not yet treat creation actions and escape internal actions.

2.1.4.1 SYMBOLIC LABELLED TRANSITION SYSTEM BASIS

In this section we generalise the simple LTS definitions to the non-enumerated data, interleaving case. A novelty of the non-enumerated case is that, with any non-trivial data language, finding reachable states inevitably involves constraint solving. Thus, whether an execution is possible or not is something that cannot easily be decided statically.

We base the following discussions on the representation of test descriptions as symbolic LTSs, similar to that of [RusBouJér00]. The main differences between the presentation given here and that [RusBouJér00] are the following. Firstly, we do not use message variables in emission actions. In [RusBouJér00], this is done in order to simplify the mirror operation of the test synthesis, by which synchronous inputs are changed into synchronous outputs and vice versa. Here, we are not directly concerned with test synthesis. Secondly, our treatment of tester internal actions is different due to the desire to cover the distributed context. Thirdly, we do not consider the assignments made on a proper reception as atomic with that reception, due to the desire to view the reception itself as an observable action, while the consequent assignments are to be viewed as a tester internal, and therefore controllable, action. One reason for this, is to facilitate the treatment of determinism of Section 2.2.4.1, in particular, the notion of minimal determinism.

---

[7] For a given complete test run obtained by extending a given fail configuration, there exist other complete test runs differing only in the number of events they contain which are concurrent to the fail event and defined as runs through equivalent fail configurations. The smallest such run is called the equivalent *minimal full test run*.

In our models, a symbolic LTS is defined as a tuple (S, s_0, Σ, T, V) where:

- S is a set of states
- s_0 is the initial state
- Σ is an alphabet of actions as described in the introduction to this section
- T is a set of transitions as described below
- V is a set of variables of three types: static variables, dynamic variables and message variables

A transition is a triple (s, l, s'), where s, s' are states called resp. the *start state* and the *end state* and l is a *transition label*. We say that a transition is an *outgoing transition* of its start state. A transition label is a tuple (*guard*, *act*, *params*, *assign*) where:

- *guard* is a boolean expression over static, dynamic and message variables,
- *act* is an element of Σ, with associated signature,
- *params* is a (possibly empty) tuple of expressions over static, dynamic and message variables; there is one expression for each element of the signature of the action and the expression is of the corresponding type,
- *assign* is a (possibly empty) list of assignments, i.e. a list of expressions of the form l.h.s. := r.h.s., where the l.h.s. is a single dynamic variable and the r.h.s. is an expression involving static, dynamic and message variables of the same type as the l.h.s..

All variables have global scope. Static variables parameterise the whole symbolic LTS, that is, their value cannot change from state instance to state instance. The value of a dynamic variable can only change between one state instance and another if the two states are connected by a transition labelled by an "assign" component-internal action, and the variable appears on the l.h.s. of one of the assignments of this action. The value of a message variable can only change between one state instance and another if the two states are connected by a proper reception action of which that variable is a parameter.

Each dynamic variable is owned by a base-level component (= object, in the usual case). Each message variable is owned by the component which owns the target port of the proper reception which is the most recent local predecessor. The ownership of dynamic and message variables has no significance here and is only introduced with a view to generalising to the non-interleaving case.

We impose the following well-formedness conditions on the transition labels used in a test description:

Guards

We will assume that all guards are well-formed in the sense that each individual guard is satisfiable.

- For an emission action or a component-internal action, the guard is an expression over static and dynamic variables only.
- For a proper reception action, we will prefer models in which the guard expressions constrain only the values of the message variables, any constraint on the values of the dynamic variables of the state instance being encoded by the use of a different state. We will assume this to be the case for symbolic LTSs derived from TeLa test descriptions.
- For an improper reception action, the guard is trivial.

Assignment list and coherence with structured action names

- For an emission action or proper reception action, the list is empty.

- For an improper reception action, the dynamic variable on the l.h.s. of each assignment is owned by the component which owns the target port of that action or by one of its subcomponents.
- For a component-internal action, the dynamic variable on the l.h.s. of each assignment is owned by the component which owns the action or by one of its subcomponents and the expression on the r.h.s. is over static and dynamic variables only.

Parameter list and signature:

- For an emission action or improper reception action, each expression is over static and dynamic variables only.
- For a proper reception action, each expression comprises a single message variable; we further assume that the same message variable is always used in the same parameter position
- For a component-internal action, the parameter list is empty.

The set of possible tuples (a, *value_list*), where a is an action and *value_list* is a list of typed values conforming to the signature of a, is called the set of valued actions VA. A *state instance* is a pair where the first element is a state and the second is a valuation of the static, dynamic usual message variables. An *initialised symbolic LTS* is a symbolic LTS together with a valuation of the static variables, called the *instantiation*, and of the dynamic variables, called the *initialisation*. An initial state instance is a pair (initial state, instantiation + initialisation).

Occurrences of the TeLa "assign" internal action are modelled as unguarded transitions labelled by the component-internal action "assign" with non-empty assignment list. Occurrences of the TeLa "assert" internal action are modelled as guarded transitions labelled by the component-internal action "assert" with empty assignment list. An internal communication of a tester component can either be viewed as a pair of actions, comprising a tester send action and a corresponding tester receive action, or as a component-internal action of that component.

2.1.4.1.1 Instances of transitions labelled by controllable actions

A valuation of a set of variables is a mapping associating a value (in the semantic data domain) to each element of V. Let v be a valuation of the static and dynamic variables and w be a valuation of the message variables. We say that a transition $t = (s, l, s')$ labelled l = (*guard*, *act*, *params*, *assign*) where *act* is an emission action, a component-internal action or an improper reception action is *enabled* and *fireable* in a state instance (s,v,w) if $v(guard)$ = *true* (recall that for transitions labelled by improper receptions, the guard is trivial so such transitions are enabled in all instances of states for which they are outgoing transitions).

In the case of an emission action, if t is fireable in (s,v,w), we say that t, together with the valuations v and w, defines a transition instance with *start state instance* (s,v,w) and *end state instance* (s',v,w). We say that (*act*, *values*) $\in VA$ is the action instance of the transition instance, where values is the tuple $v(params)$, and write the transition instance as $(s,v,w) \rightarrow^{act(values)} (s', v,w)$.

In the case of a component-internal action or improper reception action, if t is fireable in (s,v,w), we say that t, together with the valuations v, w and v', defines a transition instance with start state instance (s,v,w) and end state instance (s',v',w), where v' is a valuation of the static and dynamic variables defined as follows:

- v and v' agree on all static variables and on any dynamic variable which does not appear on the l.h.s. of an element of the assignment list,
- for each dynamic variable x for which $x := E^x$ is an assignment of *assign*, $v'(x) = v{\cdot}w\ (E^x)$

Note that the r.h.s. of the assignments can concern message variables. This facility is used to assign received values to dynamic variables. We will only use it for assignments that occur as a direct consequence of an observable action.

In the case of an improper reception action, we say that (*act*, *values*) $\in$ *VA* is the action instance of the transition instance, where values is the tuple v(*params*), and write the transition instance as $(s,v,w) \rightarrow^{act(values)} (s',v',w)$. In the case of a component-internal action, since the signature is the empty list, there are no valued actions and we write $(s,v,w) \rightarrow^{act} (s', v',w)$. If a transition is enabled / fireable in a state instance, by extension we say that the action of that transition is enabled / fireable in that state instance. If the test description is deterministic, see Section 2.2.4.1, the enabled actions and enabled transitions are in 1-1 correspondence.

2.1.4.1.2 Instances of transitions labelled by observable (= proper reception) actions

Let v be a valuation of the static and dynamic variables and w be a valuation of the message variables. We say that a transition $t = (s, l, s')$ labelled l = (*guard*, *act*, *params*, *assign*) where *act* is a proper reception action is *fireable* for valuation w' of the message variables in state instance (s,v,w) if $v{\cdot}w'(guard) = true$. We say that such a transition is *enabled* in a state instance (s,v,w) if $\exists$ a valuation w' of the message variables for which it is fireable. Thus, for observable actions, the guard can concern message variables. This facility is used to place conditions on received values. Notice that w (as opposed to w') plays no role in the transition.

If t is fireable in (s,v,w), we say that t together with the valuations v, w and w' defines a transition instance with start state instance (s,v,w) and end state instance (s',v,w').

We say that (*act*, *values*) $\in$ *VA* is the action instance of the transition instance, where values is the tuple $v{\cdot}w'$ (*params*) and write the transition instance as $(s,v,w) \rightarrow^{act(values)} (s',v,w')$. If a transition is enabled and fireable for valuation w' in a state instance, by extension we say that the action of that transition is enabled and fireable for valuation w' in that state instance. If the test description is deterministic, the enabled actions and enabled transitions are in 1-1 correspondence.

2.1.4.1.3 Definition of concepts required for TeLa semantics

The semantics of the symbolic LTS is the LTS defined as (SI, SI_0, VA, TI) where *SI* is the set of state instances, SI_0 the set of initial state instances, *VA* the set of valued actions and *TI* the smallest transition relation $SI \times VA \times SI$ defined by the definition of transition instance above.

The set of initial state instances is defined by an instantiation (assignment of values to the static variables) and an initialisation (assignment of values to the dynamic variables). We assume that the initialisations are constrained by an initial condition. Message variables do not need initialising as they are only used after being instantiated in an observable transition.

We consider that the tester has the initiative for firing enabled transitions labelled by controllable actions while the tester environment, i.e. the SUT, has the initiative for choosing a w' and firing transitions labelled by observable actions.

We will refer to a symbolic LTS derived from a test description via a projection onto tester lifelines as a test description symbolic LTS or simply a test description where no confusion

can arise. Clearly, we are assuming that such a symbolic LTS can always be derived from a test description (though here we do not explore how).

A *run* through a test description is a sequence of transition instances s.t.

- the start state instance of the first element of the sequence is the initial state instance of the instantiated and initialised test description,
- the start state instance of every element of the sequence other than the initial one is the end state instance of the previous element.

A state instance (s,v) is *reachable* from the initial state instance, i.e. for a given instantiation and initialisation, if there exists a run in which the last element has end state instance (s,v).

A *maximal run* through a test description is a run in which the last element is an instance of a terminal transition, that is, a transition whose end state is a sink state.

A *trace* is the sequence of action instances defined by a run.

An *execution* is a set of runs which is totally ordered by inclusion, where each element extends its predecessor in the ordering by one transition instance. A *maximal execution* is an execution whose last element is a maximal run.

The assumption on the guards of transitions labelled by observable actions says that if an observable action guard is satisfied for the valuations v, w and w', where (s,v,w) is a reachable state instance for some instantiation and initialisation, it is also satisfied for valuations v', w'' and w', for any valuations v' and w'' s.t. (s,v',w'') is also a reachable state instance for some instantiation and initialisation. We could simply the implementation by removing the reachability criterion from this assumption.

We then add the following assumption on the guards of transitions labelled by observable actions. For any allowed instantiation and initialisation:

- for any guard on an outgoing transition of a state s labelled by an observable action, $\exists$ v, w, w' valuations s.t. (s,v,w) is reachable for some instantiation and initialisation and $v.w'(guard) = true$.

The two assumptions on the guards of observable transitions then give that if a state has an outgoing transition labelled by an observable action, this transition is enabled in all reachable instances of that state for any instantiation and initialisation.

2.1.4.2 VERDICT ANNOTATIONS ON TEST DESCRIPTION SYMBOLIC LTSS

In order to define verdicts explicitly in the semantic domain, even if they are to remain implicit in the syntactic domain, we introduce the possibility of annotating terminal transitions – those leading to a sink state – of our test description symbolic LTSs with a verdict. We then call terminal transitions annotated with a verdict pass transitions, fail transitions or inconclusive transitions, according to the type of verdict.

A state instance (s,v,w) is said to be complete w.r.t. the set of observable actions Σ_{obs} if, for all $a \in \Sigma_{obs}$ and for all valuations w' of the message variables there is a transition labelled by a which is fireable for w' in (s,v,w'). A state s is said to be input complete if all instances of s are complete w.r.t. Σ_{obs}.

A test description is (*strictly*) *test complete* if all terminal transitions are annotated with a verdict, and for all instantiations and initialisations, all reachable state instances having an enabled observable action are complete w.r.t. Σ_{obs}. With the assumptions on the guards of observable transitions, the second condition reads: for all instantiations and initialisations, all

reachable instances of any state having an outgoing transition labelled by an observable action are complete w.r.t. Σ_{obs}. If we relax the second condition slightly by removing the reachability constraint we obtain the definition of *universally test complete*. We can relax the condition a bit further to say that all states having at least one instance which has an enabled observable action are complete w.r.t. Σ_{obs}. By the assumptions on observable action guards, this is equivalent to saying that all states with an outgoing transition labelled by an observable action are complete w.r.t. Σ_{obs}, giving the definition of *statically test complete*.

Note that:

statically test complete ⇒ universally test complete ⇒ test complete

As stated in the outline of this semantics, we will generally assume that the guard expressions on transitions labelled by observable action constrain only the message variables. Assuming this is the case, if an observable action is enabled in one reachable instance of a state for some instantiation and initialisation, it is enabled in all reachable instances of that state for all possible instantiations and initialisations. If, in addition, an observable action being enabled in one instance of a state implies it is enabled in all instances of that state (i.e. without the reachability condition), then:

statically test complete ⇔ universally test complete

Given a test-complete test description, the last element of each maximal run is necessarily an instance of a transition which is annotated with a verdict so we can associate a verdict to each maximal run. We say that the *test verdict* of a maximal execution of a test description is the verdict associated to the last element of this execution.

In order for verdicts to be consistently defined, any two maximal runs of a test-complete test description that have the same trace must have the same associated verdict. Consistently-defined verdicts can be guaranteed by imposing determinism. Evidently, consistency of fail verdicts is of greatest importance. This can be guaranteed by imposing minimal determinism along with the assumption that any non-determinism is either resolved on a controllable action or leads to successful termination (implicit pass verdict) or the same explicit verdict on both branches.

A semantically test-complete test description is achieved in TeLa by using syntactically-implicit verdicts. In Section 2.3.4, we show how implicit pass, fail and inconclusive verdicts can be made explicit to give a semantically test-complete test description. Another way to achieve test completeness, representing a more low-level view, is to use explicit defaults as in [ETSI03a] or [UTP03].

2.2 Determinism and controllability in TeLa

2.2.1 Introduction

In this section we discuss notions of determinism. The essence of the notion of determinism can be expressed as follows: a deterministic abstract machine is an abstract machine that never has more than one behavioural option so that its behaviour is completely determined by its inputs. That is, for an identical set of inputs, the machine´s behaviour is identical.

We base the discussions on the TeLa semantics outlined in the Section 2.1. As in these sections, in order to introduce the inevitable complexity in a progressive manner, we first outline how to define controllability and determinism in an interleaving semantics with

enumerated data, then in a non-interleaving semantics with enumerated data, and finally in an interleaving semantics with symbolic data. We leave the non-interleaving semantics with symbolic data for future work. In all cases, we deal with the semantics after projection onto tester instances, see Chapter 3, Section 1.4.

According to the classification of [SasNieWin96], we now handle a non-interleaving, system, linear-time model (deterministic labelled transition systems), a non-interleaving, system, branching-time model (labelled transition systems), an interleaving, behavioural, linear-time model (deterministic event structures) and a interleaving, behavioural, branching-time model (event structures).

2.2.1.1 INPUT-OUTPUT INTERLEAVING MODELS

A standard automaton is said to be deterministic if no state has two outgoing transitions with the same label. When the labels are viewed as describing the inputs and the set of states through which the automata passes on receiving these inputs is viewed as describing the behaviour, the automata definition of determinism is seen to correspond closely to the essential definition of determinism described above. If a distinction between observable and internal transitions is introduced, as in process algebra models, this definition of determinism can be refined to exclude automata in which there is a state which has an outgoing transition with a given label but which also has an outgoing internal transition followed by an outgoing transition with that same label. We refer to these definitions as the standard and refined deterministic automata definitions.

As stated in Chapter 2, Section 2.2, input-output models have proved particularly useful in testing, particularly asynchronous testing. In these models the set of transition labels is divided into three distinct subsets: input actions, output actions and internal actions. If we apply the standard definition of automata determinism to these input-output models, ignoring the distinction between the three different kinds of label, the resulting definition no longer corresponds to the essential notion of determinism. For example, a deterministic model may contain a state with two different outgoing transitions labelled by output actions and leading to different states, as long as the two output actions are distinct. However, the intuition here is that, since these actions are outputs, the machine described by this model has more than one behavioural option, independently of its inputs.

In the input-output model context in which the internal actions are hidden (tau-reduction), a definition of determinism which is closer to the essential one is the standard automata definition together with the following extra condition: if a state has an outgoing transition labelled by an output action it has no other outgoing transitions. If internal actions are not hidden, we can either use the same definition, supposing that internal actions have priority over external actions, or substitute “output actions” by “internal actions or output actions” in the extra condition. The extra condition ensures that for an identical input trace (abstraction of a trace to the input actions), the behaviour is identical.

In fact, in the input-output model context, both the simplistic application of the standard automata definition, and the definition with the extra condition, are of interest. This is since models which are deterministic in the former sense but not in the latter are also useful, e.g. the “test graphs” of [Jér02] [JarJér02]. We follow [Jér02] [JarJér02] and use the term determinism for the former sense and the term *controllability* for the latter sense.

Let us look a little more closely at models which are deterministic in the standard automata sense but not in the essential sense; in our terminology, models which are deterministic but not controllable. These models may contain two types of *controllability conflicts*. The first type represents an *auto-controllability conflict*. It occurs when a state has an outgoing

transition labelled by one (internal or) output action and another outgoing transition labelled by another (internal or) output action (e.g. an indefinite choice between controllable actions). The second type represents a *race condition controllability conflict* between the tester and the SUT. The second type occurs when a state has an outgoing transition labelled by an (internal or) output action and another outgoing transition labelled by an input action. It could be resolved by using a priority policy or a queueing policy.

For the interleaving, enumerated data case, we present a slightly different formulation of the notions of determinism and controllability to that of [Jér02] [JarJér02] for several reasons. Firstly, we wish to extend the set of controllable actions of the tester to include internal actions which are not hidden, in order to have models which are more suited to a distributed context. In our models, the internal actions are either component-internal actions or actions which are part of message exchanges between tester components. Furthermore, in the distributed context, the policy of giving priority to internal actions over outputs may not always be the most appropriate, so our definition of controllability applies to both internal actions and outputs. Secondly, we wish to include treatment of data. Thirdly, we wish to present the different concepts in such as way as to facilitate their generalisation to the non-interleaving context.

We note in passing that in the testing field, the term non-determinism is sometimes also used for a test description containing states where the tester is waiting for a message from the SUT and several possible messages are acceptable (e.g. an indefinite choice between tester inputs). This notion of non-determinism, which we call *observable non-determinism,* really concerns the SUT rather than the tester itself; it corresponds to the external behaviour of the SUT being non-deterministic.

2.2.1.2 INPUT-OUTPUT NON-INTERLEAVING MODELS

In non-interleaving models it is possible to distinguish between concurrency and alternative behaviour. This possibility suggests other definitions of determinism and of controllability.

2.2.1.2.1 Several possible notions of determinism

We introduce a weaker notion of determinism than that which we call determinism above for non-interleaving models used as test descriptions, referring to this notion as *minimal determinism*. This notion was introduced in Chapter 3, Section 1.4.4.3. Minimal determinism allows non-deterministic choices, but not non-deterministic concurrency, between identical controllable actions, but not between identical observable actions. Minimal determinism, together with the property that any non-determinism is either resolved on a controllable action or leads to successful termination (implicit pass verdict) or the same explicit verdict on both branches, is sufficient to be able to discuss the property of correctness. Minimal determinism is also the part of the definition of determinism (as defined above) which is not included in the definition of controllability, that is, minimal determinism + controllability $\Rightarrow$ determinism.

2.2.1.2.2 Several possible notions of controllability

We also introduce five new notions of controllability for non-interleaving models used as test descriptions. Rough definitions for these concepts are as follows. Recall that the set of observable actions comprises the set of input actions, and that the set of controllable actions comprises the set of output actions and the set of internal actions. A model is *essentially controllable* if it has no alternatives (minimal conflicts) involving an observable action and a controllable action. A model is *properly concurrently controllable* if it is essentially controllable and has no concurrency between observable actions and output actions. A model

is *concurrently controllable* if it is essentially controllable and it has no concurrency between observable actions and controllable actions. A model is *properly (centrally) controllable* if it is essentially controllable and it has no concurrency between observable actions and controllable actions, nor between output actions. A model is (*centrally*) *controllable* if it is essentially controllable and it has no concurrency between observable actions and controllable actions, nor between controllable actions (i.e. if a controllable action is never jointly enabled with any other action).

2.2.1.2.3 Local controllability

In the non-interleaving semantics case, it would also be of interest to define local notions of controllability, concurrent controllability etc. i.e. notions that are relative to an interaction framework, that is, w.r.t. a set of tester components. For example, a possible definition of locally controllable would be that no two outputs or internal actions *owned by the same component* can be enabled in any configuration. We leave further exploration of this interesting possibility to future work.

2.2.2 Interleaving model with enumerated data

Since in the interleaving case, choice and concurrency are confused so that choice points are not defined, it does not seem appropriate to discuss minimal determinism. However, while it is true that a trace semantics is deterministic by definition, we are using an LTS semantics. Minimal determinism can be discussed in this context by supposing that any non-determinism does not arise due to non-deterministic concurrent events (this being easily detected syntactically if the language does not include a parallel operator).

2.2.2.1 DETERMINISM

With a view to generalisation, we will use a definition of determinism that depends on a notion of local determinism which, in turn, depends on a notion of overlapping transitions.

Derivation of a deterministic automata from a test description can be ensured by imposing syntactic restrictions, in which case we speak of the test descriptions being deterministic, or by stipulating that the semantics is always that of the corresponding deterministic automata, in which case we speak of test descriptions being determinised (c.f. the delayed choice operator of the process algebra MSC semantics).

2.2.2.1.1 Local determinism

Two transitions are said to *overlap* in state *s* if:

- they are labelled by identical actions
- they are both enabled in *s*

We say that a test description is *minimally deterministic at s* if no overlapping transitions have either of the following properties:

- the transitions are concurrent (as choice and concurrency are not distinguished in the semantics, we must assume that such transitions can be eliminated by syntactic restrictions).
- the transitions are non-concurrent and are labelled by an observable action.

So a test description which is minimally deterministic at *s* allows non-concurrent transitions labelled by a controllable action to overlap in *s*.

We say that a test description is *deterministic at s* if there are no overlapping transitions in *s*.

2.2.2.1.2 Global determinism

2.2.2.1.2.1 *Minimal determinism*

As stated in Section 2.2.1.2.1, we allow the possibility of using test descriptions which are not fully deterministic. However, they must at least be minimally deterministic, that is, a test description which is not minimally deterministic is not well-formed. A test description is said to be *minimally deterministic* if it is minimally deterministic at all states.

2.2.2.1.2.2 *(Full) determinism*

A test description is said to be *deterministic* if it is deterministic at all states[8]. A test description is deterministic if and only if no two runs through it have the same trace.

In a well-formed test description we assume that isomorphic transitions – overlapping transitions between the same two states – and isomorphic states – those with identical possible futures (i.e. run suffixes beginning in that state) – are identified (i.e. not distinguished). This being the case, the above definitions are equivalent to ones in terms of the uniqueness of the state reachable from any state via any transition labelled by a given action.

2.2.2.2 CONTROLLABILITY

With a view to generalisation, we use a definition of controllability that depends on a notion of local controllability which, in turn, depends on a notion of controllability conflict.

2.2.2.2.1 Local controllability

A test description is said to have a *controllability conflict* at state s if two actions are enabled in s and one of the two is a controllable action. We say that a test description is *controllable at s* if it has no controllability conflict at s.

2.2.2.2.2 Global controllability

A state of a minimally deterministic test description in which the only enabled action is a single controllable action is called a *controlling state*. A state of a minimally deterministic test description in which the only actions enabled are observable actions is called an *observing state*.

A minimally deterministic test description is said to be *controllable* if it is controllable at all states[9]. The controllability property of minimally-deterministic deterministic test descriptions simply says that all tester states having enabled actions are either controlling or observing. Clearly:

$$\text{minimal determinism} + \text{controllability} \Rightarrow \text{determinism}$$

We say that a controllable test description defines a *test case*.

2.2.3 Non-interleaving model with enumerated data

In this section we generalise the simple automata definitions to the enumerated data, non-interleaving (distributed) case where choice and concurrency are clearly distinguished.

---

[8] To be more exact, the definition could demand determinism at reachable states only, however, this would make the property more difficult to verify. Similarly for minimal determinism.

[9] To be more exact, the definition could demand controllability at reachable states only, however, this would make the property more difficult to verify.

2.2.3.1 DETERMINISM

With a view to generalisation, we will use a definition of determinism that depends on a notion of local determinism which, in turn, depends on a notion of overlapping events. Our definition of determinism is that termed “operational determinism” in [Ren96].

If we wish to impose determinism syntactically in TeLa, we must disallow, firstly, choices involving alternatives with minimal events labelled by identical actions[10] and, secondly, coregions whose scope includes two events labelled by identical actions. Alternatively, we could impose the condition semantically and interpret that the semantics is given by determinising (though this may be complex), using a kind of partial-order version of the delayed choice operator. In the former case, we speak of deterministic test descriptions and in the latter case, determinised test descriptions, though determinisation is not necessarily possible.

2.2.3.1.1 Local determinism

Two events are said to *overlap* in configuration *C* if:

- they are labelled by identical actions,
- they are both enabled in *C*

We say that a test description is *minimally deterministic at C* if no overlapping events have either of the following properties:

- the events are concurrent (if two such events exist then ∃ an auto-concurrent configuration),
- the events are in minimal conflict and labelled by an observable action.

So a test description which is minimally deterministic at *C* allows non-concurrent events labelled by a controllable action to overlap in C.

We say that a test description is *deterministic at C* if there are no overlapping events in *C*.

2.2.3.1.2 Global determinism

2.2.3.1.2.1 Minimal determinism

As stated in Section 2.2.1.2.1, we allow the possibility of using test descriptions which are not fully deterministic. However, they must at least be minimally deterministic, that is, a test description which is not minimally deterministic is not well-formed. A test description is said to be *minimally deterministic* if it is minimally deterministic at all configurations. Minimal determinism allows minimal conflicts between events labelled by the same controllable action.

2.2.3.1.2.2 Full determinism

A test description is said to be *deterministic* if it is deterministic at all configurations. A test description is deterministic if and only if no two runs through it have the same trace.

Also of interest are the following results, see [Ren96]. If two events of a deterministic test description share the same label either they are causally related or they are in a non-minimal conflict. The less restrictive concept of causal determinism can be defined as the absence of causally indistinguishable events, i.e. distinct events that share the same label and have the same immediate predecessors.

---

[10] We assume that there is no parallel operator.

2.2.3.2 CONTROLLABILITY

With a view to generalisation, we use a definition of controllability that depends on a notion of local controllability which, in turn, depends on a notion of controllability conflict.

In the partial order case, controllability has two aspects due to the distinction between choice and concurrency, namely, restrictions on allowable choices and resolution of all tester concurrency except that between observable events. It is of interest to introduce these two aspects separately.

2.2.3.2.1 Local controllability

Given a configuration *C*, a test description is said to have an:

- *essential controllability conflict at C* if two events, one of which is labelled by a controllable action, are enabled and in conflict in *C*,

A test description is said to have a *race-condition controllability conflict* at configuration *C* if two events, one of which is labelled by a controllable action and one by an observable action, are enabled in *C*. A test description is said to have an *auto-controllability conflict* at configuration *C* if two events, both of which are labelled by controllable actions, are enabled in *C*.

Given a configuration *C*, a test description is said to have a:

- *alternation controllability conflict at C* if it has an essential or race-condition controllability conflict at *C*,
- *controllability conflict at C* if it has an essential, race-condition or auto-controllability conflict at *C*.

Note that a test description has a controllability conflict at *C* if two events, one of which is labelled by a controllable action, are enabled in *C*.

Given a configuration *C*, we say that a test description is:

- *essentially controllable at C* if it has no essential controllability conflict at *C*.
- *concurrently controllable at C* is if has no alternation controllability conflict at *C*
- (*centrally*) *controllable at C* if it has no controllability conflict at *C*

In a configuration at which a test description is essentially controllable, two enabled events can be in conflict only if both are labelled by observable actions and concurrent only if both are labelled by observable actions, both by controllable actions, or one by a controllable action and one by an observable action.

In a configuration at which a test description is concurrently controllable, two enabled events can be in conflict only if both are labelled by observable actions and concurrent only if both are labelled by observable actions or both by controllable actions.

In a configuration at which a test description is centrally controllable, two enabled events can be in conflict or concurrent only if both are labelled by observable actions.

2.2.3.2.1.1 *Taking into account internal actions*

By distinguishing between proper and improper controllable actions and assuming that tester internal actions are not hidden, we can distinguish two more degrees of controllability. The first is intermediate between essential controllability and concurrent controllability while the second is intermediate between concurrent controllability and (central) controllability.

A test description is said to have a *proper race-condition controllability conflict* at configuration *C* if two events, one of which is labelled by a proper controllable action and one by an observable action, are enabled in *C*. A test description is said to have a *proper auto-*

controllability conflict at configuration *C* if two events, both of which are labelled by proper controllable actions, are enabled in *C*.

Given a configuration *C*, a test description is said to have a

- *proper alternation controllability conflict* at *C* if it has an essential or proper race-condition controllability conflict at *C*.
- *proper controllability conflict* at *C* if it has an essential, race-condition or proper controllability conflict at *C*.

Given a configuration C, we say that a test description is:

- *properly concurrently controllable* at *C* is if has no proper alternation controllability conflict at *C*.
- *properly* (*centrally*) *controllable* at *C* if it has no proper controllability conflict at *C*.

In a configuration at which a test description is properly concurrently controllable, two enabled events can be in conflict only if both are labelled by observable actions and concurrent only if both are labelled by tester input actions, both by tester output actions, or one by a tester output action and one by a tester internal action.

In a configuration at which a test description is properly (centrally) controllable, two enabled events can be in conflict only if both are labelled by observable actions and concurrent only if both are labelled by tester input actions, both by tester internal actions, or one by a tester output action and one by a tester internal action.

2.2.3.2.2 Global controllability

A configuration of a minimally deterministic test description in which the set of enabled actions contains:

- only concurrent controllable actions is called a concurrent controlling configuration.
- only concurrent controllable actions, but a single proper controllable action, is called a proper controlling configuration.
- only a single controllable action is called a controlling configuration.
- only observable actions and improper controllable actions concurrent with them is called a proper observing configuration.
- only observable actions is called an observing configuration.

A minimally deterministic test description is essentially controllable if it is essentially controllable at all configurations. That is, there are no minimal conflicts in which one of the events is labelled by a controllable action. Clearly:

$$\text{minimal determinism} + \text{essential controllability} \Rightarrow \text{determinism}$$

A minimally deterministic test description is

- properly concurrently controllable, PConC, if it is PConC at all configurations,
- concurrently controllable, ConC, if it is ConC at all configurations,
- properly (centrally) controllable, PCenC. if it is PCenC at all configurations,
- (centrally) controllable, CenC, if it is CenC at all configurations.

If internal actions are hidden, properly concurrently controllable = concurrently controllable and properly (centrally) contollable = (centrally) controllable.

Note that, assuming minimal determinism:

- concurrent controllability ⇔ all tester configurations having enabled transitions are either concurrent controlling or observing
- (central) controllability ⇔ all tester configurations having enabled transitions are either controlling or observing.

Taking tester internal actions into account and, again, assuming minimal determinism:

- proper concurrent controllability ⇔ all tester configurations having enabled transitions are either concurrent controlling or proper observing
- proper (central) controllability ⇔ all tester configurations having enabled transitions are either proper concurrent controlling or observing.

2.2.3.2.2.1 Test cases and degree of parallelism

We say that:

- an essentially-controllable test description defines a *parallel test case*
- a concurrently controllable test description defines a *coherent parallel test case*
- a centrally controllable test description defines a *centralisable test case*.

Taking tester internal actions into account, we say that:

- a properly concurrently controllable test description defines a *externally-coherent parallel test case*
- a properly controllable test description defines an *externally-centralisable test case*

A parallel test case can be interpreted as defining one or several externally-coherent parallel test cases. An externally-coherent parallel test case can be interpreted as defining one or several coherent parallel test cases. A coherent parallel test case can be interpreted as defining one or several externally-centralisable test cases. An externally-centralisable test case can be interpreted as defining one or several centralisable test cases.

In each case, a more restrictive test case is obtained from a less-restrictive test case by adding non-local causality relations to resolve tester concurrency:

- from parallel to externally-coherent parallel by resolving concurrency between events labelled by proper controllable actions and observable actions
- from externally-coherent parallel to coherent parallel by resolving concurrency between events labelled by improper controllable actions and observable actions
- from coherent parallel to externally-centralisable by resolving concurrency between events labelled by proper controllable actions
- from externally centralisable to centralisable by resolving concurrency between events labelled by proper or improper controllable actions

Due to the fact that the tester does not control events labelled by observable actions, any orderings added to resolve concurrency between an event labelled by an observable action and an event labelled by a controllable action should ensure that the former precedes the latter and not vice versa.

Notions of controllability at each component, relative to a partition of the tester into a set of components, could also be defined as refinements of a parallel test case, as mentioned in Section 2.2.1.2.3.

In the same way as for the mechanisms implementing the extra ordering relations between tester actions added by the SUT lifelines, or for resolving non-local choices, or for implementing verdicts in the presence of non-local choices between observable actions, the mechanisms by which concurrency is resolved will depend on whether the actual implementation is distributed or not.

2.2.3.3 RELATED WORK

[Mit01] deals with deterministic MSCs with no alternatives or loops. Verdicts are implicit, though the mechanisms are not explained and there is no discussion of test completeness.

Property 2, resp. property 3, of the "tightly-coupled" partial-orders of [Mit01] prohibits concurrency between controllable events, resp. between controllable and observable events. The derivation of a "concurrent test" from an MSC without choices in [Mit01] corresponds to the derivation of what we define as a centralisable test case from a parallel test case by resolution of unwanted concurrency. A "viable test" derived from an MSC in [Mit01] is simply a test in which the resolution of concurrency between observable and controllable actions obeys the restriction given in the preceding section; the formulation is slightly different due to the fact that [Mit01] does not pre-suppose a semantics by projection onto tester lifelines. [Mit01] does not address the question of non-local choices in an MSC.

[DeuTob02], on the other hand, use a projection semantics in terms of pomsets (equivalence classes of prefixes of maximal partial orders; to each such equivalence class corresponds a configuration of the corresponding event structure). They deal with alternatives, coregions and finite iteration. Verdicts are explicit, though in the absence of TTCN-style defaults, it is difficult to see how test completeness could be obtained. Note that [DeuTob02] use the term "tester observable actions" for the set of tester actions, where these are proper or improper emissions or receptions, and the term "determinism" in its essential sense, that is controllability.

The well-formedness condition WF_1 of [DeuTob02]´s "test purposes" together with the restriction of no auto-concurrency (concurrency of events with the same label) given earlier in the article, corresponds to what we define as determinism of test descriptions, see [Ren96]. How pomsets are derived from MSCs is not fully specified; for this, the authors refer to the semantics of [KatLam98]. However, the latter semantics is a linear-time semantics, i.e. only deterministic pomsets are derived from MSCs so the need for condition WF_1 is not clear.

The well-formedness condition WF_2 of [DeuTob02]´s "test purposes" corresponds to what we define above as essential controllability of test descriptions. Incidentally, the definition of choice point used by [DeuTob02] for this definition is overkill, given the determinism assumption: WF_1 + no auto-concurrency. The derivation of a valid test case from a well-formed test purpose in [DeuTob02] corresponds to the derivation of what we define as a centralisable test case from a parallel test case by resolution of unwanted concurrency. However, in [DeuTob02]´s approach to controllability issues, it is not clear whether proper and improper receptions are distinguished. Furthermore, the validity relation between test cases and "test purposes" does not take into account the direction of added causality relations between observable and controllable actions ([Mit01]´s "viability"). Finally, they do not address the question of non-local choices in a "test purpose".

Clearly, the use of an event-structure semantics makes it easier to discuss these concepts.

2.2.4 Interleaving model with non-enumerated data

In the presence of non-enumerated data, the properties of determinism and controllability, if defined in an analogous way to the enumerated data case, become run-time properties concerning "fireable" transitions, rather than outgoing transitions. Deciding either of these two properties statically implies deciding statically the mutual satisfiability of boolean expressions, for which there can be no general algorithm if the data language is non-trivial.

One approach is to abandon controllability and instead use priority of observable actions over controllable actions to resolve race-condition controllability conflicts, i.e. controllability conflicts between observable and controllable actions. In practice, the tester waits a certain time to see if an observable action occurs before performing any controllable actions which are enabled. However, this does not generalise well to the non-interleaving case. Furthermore,

this approach does not deal with indefinite-choice controllability conflicts, i.e. indefinite choices between controllable actions. We will therefore seek to define weaker notions of determinism and controllability, that can be more easily verified.

2.2.4.1 DETERMINISM

We will use a definition of determinism that depends on a notion of local determinism which, in turn, depends on a notion of overlapping of transitions. In the non-enumerated data case, different definitions of when two transitions are overlapping can be defined. The more easily verifiable, less discriminating notions define more transitions as overlapping and thus define a smaller class of test descriptions as deterministic.

The strongest possible notion of determinism of a test description is a notion that is relative to the instantiation and initialisation and to a valuation of the message variables for each observable action in each state instance. However, as it is the tester environment, i.e. the SUT, which is responsible for the valuation w, this notion of determinism is execution-dependent. We do not consider this notion of execution-dependent determinism useful, so the strongest notion we use is determinism for all valuations of the message variables, that is independently of the values of the parameters of the valued observable actions.

In the non-enumerated data case, the derivation of a deterministic, or even universally deterministic, symbolic LTS from a general symbolic LTS is an undecidable problem for any non-trivial data language.

2.2.4.1.1 Local determinism

Below, we define some examples of overlap relations and consequent local definitions of determinism.

2.2.4.1.1.1 Signature-overlap

Two transitions are said to *signature-overlap* in state s if:

- they are both outgoing transitions of s,
- they are labelled by identical actions (recall that each action has an associated signature)

2.2.4.1.1.2 Overlap for transitions labelled by observable actions

Two transitions that signature-overlap in state s and that are labelled by an observable action (we will use the term observable transitions) are said to *overlap* for the valuation w' of the message variables in a state instance (s,v,w) if:

- both transitions are enabled in (s,v,w) and fireable for valuation w'.

Two observable transitions that signature-overlap in state s are said to *potentially overlap* in a state instance (s,v,w) if they overlap for some valuation w' of the message variables (i.e. their guards are jointly satisfiable in (s,v,w)). With the usual assumption on the guards of observable transitions, joint satisfiability of guards at one reachable instance of $s \Rightarrow$ joint satisfiability of guards at all reachable instances of s.

We could also have defined a notion of overlap of observable transitions in a state instance (s,v,w) if they signature-overlap in s and are both enabled in (s,v,w) (i.e. their guards are satisfiable in (s,v,w) but not necessarily jointly). However, with the usual assumption on the guards of observable transitions and the usual assumptions on guards, observable transitions are always enabled in any reachable instance of a state for which they are outgoing transitions.

2.2.4.1.1.3 Overlap for transitions labelled by controllable actions

We now deal with transitions labelled by controllable actions (we will use the term controllable transitions). We define the following properties of pairs of controllable signature-overlapping transitions p and q in state instance (s,v,w):

C1. *Joint satisfaction of guards under v*: both transitions p and q are enabled in (s,v,w)

C2. *Equality of action parameters under v*: the values of the action parameters of the two transitions p and q under v are pairwise equal

C3. *Compatibility of assignment lists under v*: the variables on the l.h.s. of the assignments are the same pairwise and if the assignment lists are as follows $\{x_p^{\,i} := E^{x\,i}_{\;p}\}_{i\in\{1..j\}}$ and $\{x_q^{\,i} := E^{x\,i}_{\;q}\}_{i\in\{1..j\}}$ then $\{\, v{\cdot}w(\mathrm{E}^{x\,i}_{\;p}) = v{\cdot}w(E^{x\,i}_{\;q})\,\}_{i\in\{1..j\}}$

Two transitions that signature-overlap in state s and that are labelled by an emission action (we will use the term emission transitions) are said to *overlap* in a state instance (s,v,w) if properties C1 and C2 hold. Two emission transitions that signature-overlap in state s are said to *almost overlap* in state s if property C2 holds for all valuations v of the static and dynamic variables.

Two transitions that signature-overlap in state s and that are labelled by an improper reception action (we will use the term improper reception transitions) are said to *overlap* in a state instance (s,v,w) if property C2 and C3 hold (C1 holds trivially). Two improper reception transitions that signature-overlap in state s are said to *almost overlap* in state s if properties C2 and C3 hold for all valuations v of the static and dynamic variables.

Two transitions that signature-overlap in state s and that are labelled by a component-internal action (we will use the term component-internal transitions) are said to *overlap* in a state instance (s,v,w) if property C1 and C3 hold (C2 holds trivially). Two component-internal transitions that signature-overlap in state s are said to *almost overlap* in state s if property C3 holds for all valuations of the static and dynamic variables.

Notice that our definition of overlap for observable transitions does not take into account compatibility of assignment lists whereas our definition of overlap for controllable transitions does take into account this property.

2.2.4.1.1.4 Superdeterminism

We say that a test description is *minimally superdeterministic at s* if:

- there are no concurrent almost-overlapping controllable transitions at state s (as choice and concurrency are not distinguished in the semantics, we must assume that concurrent transitions can be eliminated by syntactic restrictions),
- there are no almost-overlapping observable transitions at any instances of state s

So a test description which is minimally superdeterministic at s allows non-concurrent transitions labelled by a controllable action to almost-overlap at s.

We say that a test description is *superdeterministic at s* if there are no almost-overlapping transitions at s.

2.2.4.1.1.5 (Strict) determinism

We say that a test description is *minimally* (*strictly*) *deterministic at* (s,v,w) if

- there are no concurrent overlapping controllable transitions at state instance (s,v,w) (as choice and concurrency are not distinguished in the semantics, we must assume that concurrent transitions can be eliminated by syntactic restrictions),

- there are no potentially-overlapping observable transitions at state instance (s,v,w)

So a test description which is minimally (strictly) deterministic at (s,v,w) allows non-concurrent transitions labelled by a controllable action to overlap in (s,v,w). We say that a test description is *minimally* (*strictly*) *deterministic at s* if it is minimally deterministic at all instances of state s.

We say that a test description is (*strictly*) *deterministic at* (s,v,w) if there are no overlapping controllable transitions and no potentially-overlapping observable transitions at state instance (s,v,w). We say that a test description is (*strictly*) *deterministic at s* if it is deterministic at all instances of state s.

2.2.4.1.2 Global determinism

As stated in Section 2.2.1.2.1, we allow the possibility of using test descriptions which are not fully deterministic. However, they must at least be minimally deterministic, that is, a test description which is not minimally deterministic is not well-formed.

2.2.4.1.2.1 Minimal determinism

A test description is said to be *minimally deterministic*, if, for all instantiations and initialisations, it is minimally deterministic at all reachable state instances.

A test description is said to be *universally minimally deterministic* if it is minimally deterministic at all states (i.e. at all instances of all states). The important aspect of this definition is that it removes the need to check reachability.

A test description is said to be *minimally superdeterministic* if it is superdeterministic at all states. The important aspect of this definition is that it removes the need to check reachability and joint satisfiability of guards. However, it is not very discriminating.

2.2.4.1.2.2 (Full) determinism

A test description is said to be *deterministic*, if, for all instantiations and initialisations, it is deterministic at all reachable state instances.

A test description is said to be *universally deterministic* if it is deterministic at all states[11] (i.e. at all instances of all states). The important aspect of this definition is that it removes the need to check reachability.

A test description is said to be *superdeterministic* if it is superdeterministic at all states. The important aspect of this definition is that it removes the need to check reachability and joint satisfiability of guards.

Showing determinism involves checking for the absence of statically-overlapping transitions with the following properties:

- In the case of observable transitions, joint satisfiability of guards in any reachable state instance (with the usual assumptions on observable transition guards, we have satisfiability in one reachable state instance$\Rightarrow$satisfiability in any reachable state instance).
- In the case of emission transitions, joint satisfaction of guards and pairwise equality of action parameter values in any reachable states instances.

[11] One could instead use only nominally reachable states, that is, states which are reachable in the underlying LTS obtained by removing all variables, action parameters, pre- and post-conditions and valuations; however, this would make the properties more difficult to verify.

- In the case of component-internal transitions, joint satisfaction of guards and compatibility of assignment lists in any reachable state instances.
- In the case of improper reception transitions, pairwise equality of action parameter values and compatibility of assignment lists in any reachable state instances.

Showing universal determinism involves checking for the absence of statically-overlapping transitions with the following properties:

- In the case of observable transitions, joint satisfiability of guards.
- In the case of emission transitions, joint satisfiability of guards and pairwise equality of action parameter values in any state instance for which the guards are jointly satisfied.
- In the case of component-internal transitions, joint satisfiability of guards and compatibility of assignment lists in any state instance for which the guards are satisfied.
- In the case of improper reception transitions, pairwise equality of action parameter values and compatibility of assignment lists in any state instance.

Showing superdeterminism involves checking for the absence of statically-overlapping transitions with the following additional properties:

- In the case of observable transitions, no additional properties.
- In the case of emission transitions, pairwise equality of action parameter values in any state instance.
- In the case of component-internal transitions, compatibility of assignment lists in any state instance.
- In the case of improper reception transitions, pairwise equality of action parameter values and compatibility of assignment lists in any state instance.

For the definition of determinism (minimal or otherwise), in a well-formed test description we assume that isomorphic transitions – overlapping (controllable) or potentially-overlapping (observable) transitions between the same two state instances – and isomorphic state instances – those with identical possible futures (i.e. run suffixes beginning in an instance of that state) – are identified (i.e. not distinguished). This being the case, the above definitions are equivalent to ones in terms of the uniqueness of the state instance reachable from any state instance via any transition instance labelled by a given action instance.

2.2.4.2 CONTROLLABILITY

We use a definition of controllability that depends on a notion of local controllability which, in turn, depends on a notion of controllability conflict. In the non-enumerated case, different definitions of controllability are possible. The more easily verifiable, less discriminating notions define a smaller class of test descriptions as controllable.

Similarly for determinism, we are not interested in execution-dependent notions of controllability, that is, controllability for certain instantiations and intialisations.

2.2.4.2.1 Local controllability

A state is said to have a *static controllability conflict* if it has two outgoing transitions, one of which is labelled by a controllable action. Here, we wish to distinguish the two types of static controllability conflict. A state *s* is said to have an *indefinite-choice static controllability conflict* if it has two outgoing transitions labelled by controllable actions. A state *s* is said to have a *race-condition static controllability conflict* if it has two outgoing transitions, one of which is labelled by an observable action and one of which is labelled by a controllable action.

2.2.4.2.1.1 *Local quasi-controllability*

A state is said to have a *quasi-controllability conflict* if one of the following is true:

- it has a race-condition static controllability conflict,
- it has an indefinite-choice static controllability conflict in which at least one of the transitions is trivially guarded,

A test description is said to be *quasi-controllable at s* if *s* has no quasi-controllability conflict. Note that improper receptions are unguarded controllable actions so quasi-controllability means that *s* can have no static controllability conflict involving an improper reception[12].

Though not very discriminating, this notion has the advantage of being easily verified statically.

2.2.4.2.1.2 *Local controllability*

A state instance (s,v,w) is said to have an *indefinite choice controllability conflict* if two controllable actions are fireable in (s,v,w). A state instance (s,v,w) is said to have a *race-condition controllability conflict* for valuation w' of the message variables if a controllable action is enabled in (s,v,w) and an observable action is fireable in (s,v,w) for valuation w'.

In a test-complete test description, if a reachable state instance has a race condition controllability conflict for one valuation of the message variables, it has such a conflict for all such valuations. We say that a state instance (s,v,w) of a test-complete test description has a *controllability conflict* if it has two enabled actions and one of them is a controllable action.

A test-complete test description is said to be *controllable at* (s,v,w) if (s,v,w) has no controllability conflicts. A universally test-complete test description is said to be *controllable at s* if it is controllable at all instances of *s*.

2.2.4.2.2 Global controllability

A state instance of a minimally-deterministic test description in which the only actions that are enabled are the complete set of observable actions is called an *observing state instance*. A state instance of a deterministic test description in which the only action that is enabled is a single controllable action is called a *controlling state instance*.

A state of a minimally-deterministic test description in which the only actions labelling outgoing transitions are observable actions is called a *statically-observing state*. A state of a deterministic test description in which the only actions labelling outgoing transitions are controllable actions and, if there is more than one such transition, all such transitions are non-trivially guarded, is called a *quasi-controlling state*.

A minimally deterministic test description is (*strictly*) *controllable*, if, for all instantiations and initialisations, it is test-complete and controllable at all reachable state instances. The controllability property of deterministic test descriptions simply says that for all instantiations and initialisations, any reachable state instance having an enabled action is either controlling or observing. Clearly:

$$\text{minimal determinism} + \text{controllability} \Rightarrow \text{determinism}$$

A universally minimally deterministic test description is *universally controllable* if it is universally test-complete and it is controllable at all states (i.e. at all instances of all states). The universal controllability property of universally deterministic test descriptions simply

---

[12] Note that in the TeLa semantics, improper reception actions cannot be minimal actions for a choice so any indefinite-choice static controllability conflict invovling such an action is due to concurrency.

says that any state instance having an enabled action is either controlling or observing. The important aspect of this definition is that it removes the need to check reachability. Clearly:

$$\text{universal controllability} \Rightarrow \text{controllability}$$

$$\text{universal minimal determinism} + \text{universal controllability} \Rightarrow \text{universal determinism}$$

Under the assumptions discussed in the section where test completeness is defined, we can substitute “statically test-complete” for “universally test-complete” in the definition of universally controllable. Furthermore, we can define an observing state instance of a minimally deterministic test description as an instance of a statically-observing state. The controllability, resp. universal controllability, property of deterministic test descriptions now says that all states with a reachable instance, resp. all states, are either statically observing or are such that any reachable instance, resp. any instance, having an enabled controllable action is a controlling state instance.

A deterministic test-description is *quasi-controllable* if it is quasi-controllable at all states. The quasi-controllability property of deterministic test descriptions simply says that all states with outgoing transitions are either statically observing or quasi-controlling. Note that due to the fact that the guards on the outgoing transitions of a quasi-controlling state are not necessarily mutually exclusive:

$$\text{quasi-controllability} \not\Rightarrow \text{controllability}$$

However, the important aspect of this definition is that it is easily verified statically. Furthermore, a test description which is quasi-controllable can be made universally controllable by:

- assuming that improper reception actions have priority over other controllable actions
- explicitly assigning a different priority to each outgoing transition that is labelled by a primary controllable action, in each state where there are several such outgoing transitions (effectively implementing a “case” statement).

However, this is not always desirable, so in TeLa we provide a mechanism to do this when required.

2.3 Semantics of implicit verdicts

2.3.1 Introduction

In the semantic domain, we must be able to model the default behaviour explicitly. As stated above, the situation we wish to describe with an implicit fail verdict is an unspecified reception from the SUT. In the non-enumerated data case, and for a restricted class of test descriptions (those that are essentially controllable), we also have the implicit inconclusive verdict. The situation we wish to describe with this implicit verdict is the evaluation of a guard or set of guards to false by the tester (other inconclusive verdicts must be specified explicitly).

We base the following discussion on the TeLa semantics outlined in Section 2.1. Recall that, in this section, we have already introduced the notion of annotating transitions or events with verdicts. As in this section, in order to introduce the inevitable complexity in a progressive manner, we first outline how to define implicit verdicts in an interleaving semantics with enumerated data, then in a non-interleaving semantics with enumerated data, and finally in an interleaving semantics with symbolic data. We leave the non-interleaving semantics with

symbolic data for future work. In all cases, we deal with the semantics after projection onto tester instances, see Chapter 3, Section 1.4.

2.3.2 Interleaving model with enumerated data

2.3.2.1 IMPLICITLY-DEFINED VERDICTS

For a minimally deterministic test description the *implicit pass verdict* can be informally defined as:

> The occurrence of an action labelling a terminal transition, that is, one that leads to a sink state of the test description.

For a minimally deterministic test description the *implicit fail verdict* can be informally defined as:

> The occurrence of an unspecified proper message reception at the tester, that is, the occurrence of an observable action which is not the label of any of the enabled transitions in that state and is therefore not allowed.

If we also have controllability, no observable action is allowed in a state that is not an observing state, so fail can only occur in observing states.

We do not discuss here implicit fail verdicts on timing properties such as lack of a timely response etc.

2.3.2.2 MAKING THE IMPLICIT VERDICTS EXPLICIT

In this section we show how to construct an LTS in which the implicit verdicts are explicit. This is done in order to show that the implicit verdicts are well-defined. It is not our intention to imply that any implementation of this semantics would construct an LTS in which these verdicts were represented explicitly.

In order for the implicit pass and fail verdicts to be made explicit in a coherent manner, we need only minimal determinism and the condition that all non-determinism is either resolved on a controllable action or leads to successful termination (implicit pass verdict) or the same explicit verdict on both branches.

We annotate all existing terminal transitions of the test description with pass verdicts before extending the test description with new terminal transitions annotated with verdicts as explained below. By construction, the last transition of each maximal run is then annotated with a verdict as required for a test-complete test description.

In the interleaving case with enumerated data we reify the implicit fail verdicts by completing w.r.t. observable actions, extending the test description with new terminal transitions annotated with a fail verdict as follows:

Let $Obs(s)$ denote the set of enabled observable actions in state s. In each state s:

- for each element $a \in \Sigma_{\text{obs}} - Obs(s)$ (the complement in the set of observable actions), add a terminal transition annotated with a fail verdict and labelled by action a.

We assume that observable actions have the structure $?x.m$ where x is a tester port (in some systems $?y.x.m$ where y is an SUT port) and m is a message. We can then structure the set of enabled observable actions in the following way:

$$Obs(s) = \{ \{ ?x.m \mid m \in M_s^x \} \mid x \in P \}$$

where M is the set of messages, $M_s^x \subseteq M$ is the set of messages allowed on tester port x in state s and P is the set of tester ports. Note that for some s and x, M_s^x may be empty.

In terms of this structure, the complement is as follows:

$$\Sigma_{\text{obs}} - Obs(s) \;=\; \{\; \{\; ?x.m \;\mid\; m \in M - M_s^x \;\} \mid x \in P \;\}$$

Note that, in the interleaving case, the structure of the actions labelling the added fail transitions plays no role in placing them in the automata. However, as we see below, it plays an important role in the non-interleaving case.

In the enumerated-data, interleaving case, the meaning of the "otherwise fail" notion is well-defined, and can thus be left implicit, for minimally deterministic test descriptions.

2.3.3 Non-interleaving model with enumerated data

2.3.3.1 IMPLICITLY-DEFINED VERDICTS

For a minimally deterministic test description the *implicit pass verdict* can be informally defined as:

> The occurrence of each and every one of the actions labelling terminal events, that is, events with no successors.

For a minimally deterministic test description the *implicit fail verdict* can be informally defined as:

> The occurrence of an unspecified proper message reception at the tester, that is, the occurrence of an observable action which is not the label of any of the enabled events in that configuration and is therefore not allowed.

If, in addition, the test description is essentially controllable, no event labelled by a controllable action is allowed to be in minimal conflict with any other event. If, in addition, the test description is concurrently semi-controllable, no observable action is allowed in a configuration that is not semi-observing so a fail verdict can only be derived in a semi-observing configuration. If, in addition, the test description is concurrently controllable or centrally controllable, no observable action is allowed in a configuration that is not an observing configuration so a fail verdict can only be derived in an observing configuration.

We do not discuss here implicit fail verdicts on timing properties such as lack of a timely response etc.

2.3.3.2 MAKING THE IMPLICIT VERDICTS EXPLICIT

In this section we show how to construct an event structure in which the implicit verdicts are explicit. This is done in order to show that the implicit verdicts are well-defined. It is not our intention to imply that any implementation of this semantics would construct an event structure in which these verdicts were represented explicitly.

In order for the implicit pass and fail verdicts to be made explicit in a coherent manner, we need only minimal determinism and the condition that all non-determinism is either resolved on a controllable action or leads to successful termination (implicit pass verdict) or the same explicit verdict on both branches.

We annotate all existing terminal events of the test description with pass verdicts, before extending it with new terminal events annotated with verdicts as explained below.

In the non-interleaving case with enumerated data we reify the implicit fail verdicts by completing w.r.t. observable actions, extending the test description with new terminal events annotated with a fail verdict as follows:

Let $Obs(C)$ denote the set of enabled observable actions in configuration C. As for the interleaving case, each configuration C which is not a pass configuration must be extended with a terminal event annotated with a fail verdict, for each action of $\Sigma_{obs} - Obs(C)$, the complement in the set of observable actions. However, in the non-interleaving case, placing these new events in the event structure is not as simple as placing the new transitions in the automata in the interleaving case.

As mentioned in the introduction to this section, we assume that observable actions have the structure $?x.m$ where x is a tester port (in some systems $?y.x.m$ where y is an SUT port). We can then structure the set of enabled observable actions in the following way:

$$Obs(C) = \{ \{ ?x.m \mid m \in M_C^x \} \mid x \in P \}$$

where M is the set of messages, $M_C^x \subseteq M$ is the set of messages allowed on tester port x in configuration C and P is the set of tester ports. Note that for some C and x, M_C^x may be empty.

In terms of this structure, the complement is as follows:

$$\Sigma_{obs} - Obs(C) = \{ \{ ?x.m \mid m \in M - M_C^x \} \mid x \in P \}$$

The new event labelled by a given element $?x.m \in \Sigma_{obs} - Obs(C)$ and annotated with a fail verdict must be added to the test description event structure so as to satisfy the following conditions, unless doing so would introduce non-determinism (the same fail event may be shared by different configurations):

3. *Local conflict*: for each enabled observable event e' of C on the same port x as the new fail event e add the conflict relation $e' \# e$.
4. *Local causality*: for each event e' of C on the same port x as the new fail event e, add the predecessor relation: $e' \leq e$.
5. *Non-local causality due to test non-local choices*: if $\exists e'' \in E$ s.t. e'' is in non-local (otherwise already dealt with in rule 2) minimal conflict with a leaf event e' of C, e'' is enabled in $C - \{e'\}$ and $f(e'') = a$ for some $a \in \Sigma_{obs} - Obs(C)$, add an immediate predecessor relation between e' and the new fail event e labelled by a: $e' \leq e$ (other causality relations given by transitivity). In a distributed implementation, this non-local causality may be implemented by adding tester coordination messages.

Note that, by the existential nature of the fail verdict, we do not need to add conflict nor causality relations between fail events.

Thus, in the enumerated-data, non-interleaving case, the meaning of the "otherwise fail" notion is well-defined, and can thus be left implicit, for minimally deterministic test descriptions.

2.3.4 Interleaving model with non-enumerated data

2.3.4.1 Implicitly-defined verdicts

For minimally deterministic test descriptions, the *implicit pass verdict* can be defined as any occurrence of a terminal transition, that is, one that leads to a sink state of the test description.

Similar to [RusBouJér00], a test fails if a certain reception occurs and the disjunction of the guards on outgoing transitions labelled by the corresponding action evaluates to false; if there are no outgoing transitions labelled by the corresponding action, an outgoing transition labelled by this action whose guard is *false* is considered to be present. For a minimally deterministic, test description, the *implicit fail verdict* can be informally defined as:

> the occurrence of an unspecified message reception at the tester, that is, the occurrence of a observable action instance which is not the label of any of the fireable transition instances in that state instance and is therefore not allowed.

If we also have controllability, no observable action instance is allowed in a state instance that is not an observing state instance so fail can only occur in an observing state instance.

We also want to derive a verdict if the evaluation of the guards on outgoing transitions labelled by controllable actions leaves the tester with no enabled transitions. We would like to derive a verdict in this case since it is not always practical to exactly specify all the conditions required of a value received from the SUT at the time of reception, in order to derive a fail verdict if these conditions are not satisfied. These conditions on acceptable values are, in practice, often defined by subsequent use of these values e.g. in the guard of a subsequent component-internal action. However, an unsuitable response from the SUT is not the only possible cause of the tester being left with no enabled transitions, so we cannot derive a fail verdict in this case. In such cases, therefore, we derive an inconclusive verdict, moreover, we only derive it in quasi-controlling states of quasi-controllable test descriptions.

For a quasi-controllable test description, the *implicit inconclusive verdict* can be informally defined as:

> the condition that the tester has no fireable transitions in a quasi-controlling state.

Note that the evaluation of a TeLa “assert” internal action may give rise to such an inconclusive verdict[13].

We do not discuss here implicit fail verdicts on timing properties such as lack of a timely response etc.

2.3.4.2 MAKING THE IMPLICIT VERDICTS EXPLICIT

In this section we show how to construct a symbolic labelled transition system in which the implicit verdicts are explicit. This is done in order to show that the implicit verdicts are well-defined. It is not our intention to imply that any implementation of this semantics would construct a symbolic labelled transition system in which these verdicts were represented explicitly.

In order for the implicit pass and fail verdicts to be made explicit in a coherent manner, we need only minimal determinism and the condition that all non-determinism is either resolved on a controllable action or leads to successful termination (implicit pass verdict) or the same explicit verdict on both branches. However, in the non-enumerated data case, we also have implicit inconclusive verdicts. To make these verdicts explicit in a coherent manner, we need either determinism, or controllability; the latter, together with minimal determinism, implies the former.

We annotate all existing terminal transitions of the minimally deterministic test description with pass verdicts before extending the test description with new terminal transitions

---

[13] In the non-interleaving case, we would want to derive this verdict if none of a set of alternative controllable transitions are fireable.

annotated with verdicts as explained below. By construction, the last transition of each maximal run is then annotated with a verdict as required for a test-complete test description.

Let *Obs* denote the set of observable actions, *Con* denote the set of controllable actions. Let $trans(s, a)$ denote the set of outgoing transitions of state *s* labelled by action *a* and $trans(s, A)$ denote $\cup_{\{a\in A\}}\ trans(s, a)$ for any $A \subseteq \Sigma$. Let $guard_t$ denote the guard of a transition *t*.

In the interleaving case with enumerated data we reify the implicit fail verdicts by completing w.r.t. observable actions, extending the test description with new terminal transitions annotated with a fail verdict as follows. In each state *s*:

- For each $a \in Obs$, add a terminal transition annotated with a fail verdict and labelled by action *a*. The guard of this transition is $\wedge_{\{t\in Obs(s,\ a)\}} \neg\ guard_t$, if $trans(s, a)$ is non-empty, or is trivial otherwise. As for other transitions labelled by observable actions, each expression of the parameter list of these new transitions comprises a single message variable. The assignment list of the new fail transition is empty.

Recall that quasi-controllability means that in each state s:

$$trans(s,Con) \cap trans(s,Obs) = \emptyset$$
$$\text{and if } \#(trans(s, Con)) > 1 \text{ then } \forall t \in trans(s, Con),\ guard_t \neq true$$

For a quasi-controllable test description, we reify the implicit inconclusive verdicts by extending the test description with new terminal transitions annotated with an inconclusive verdict as follows. In each state *s*:

- If $\#(trans(s, Con)) > 1$, add a terminal transition annotated with an inconclusive verdict and labelled by the assert action. The guard of this transition is $\wedge_{\{t\in trans(s,Con)\}} \neg\ guard_t$. The assert action parameter list and assignment list are empty.

In the non-enumerated data, interleaving case, the implicit verdicts are well-defined. We will say that a controllable test description defines a *test case*. We will also say that a quasi-controllable test description defines a quasi- test case. However, this latter notion of test case cannot be guaranteed to be fully executable without assigning priorities, due to the undecidability of the problem of joint satisfiability of guards. In TeLa, improper receptions are assumed to have priority over other controllable transitions and priorities can be explicitly assigned between these other controllable transitions.

3 Incorporating a more restrictive behavioural semantics

3.1 Partially-ordered messages from partially-ordered events

For a restricted class of diagrams, we can obtain a partially-ordered message style semantics (conforming to the maximalist interpretation) while remaining in the context of a partially-ordered event semantics (conforming to a maximalist interpretation) by adding extra orderings between the events. A reasonable physical interpretation of the addition of these orderings is that they correspond to the assumption that the time of transit of messages is negligible, see Chapter III, Section 2.5 for more details.

We say that a message emission event and its corresponding message reception event define a communication pair and we refer to each as being the paired event of the other. Let $\mapsto$ denote the restriction of the transitive reduction of $\leq$ to non-local orderings (i.e. for which $\varphi(e_1) \neq \varphi(e_2)$) where $\leq$ is the partial ordering obtained from a TeLa expression. We write $e_1 \mapsto e_2$ for $(e_1, e_2) \in \mapsto$ and $e_1 \leq e_2$ for $(e_1, e_2) \in \leq$.

To simplify the treatment, we first suppose that $\leq$ is s.t. all non-local causality is via messages, that is, there is no MSC-style general ordering between non-local pairs of events. We then suppose that the TeLa diagrams contain no crowns, in the sense of [ChaMatTel96], that is, no causal overtaking and no causal crossing (and, of course, no causal loops). Fig. 5-2 shows three examples of crowns.

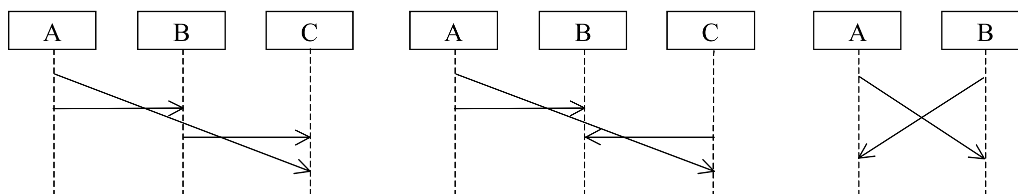


Figure 5-2: Examples of so-called “crowns”

The resulting class of sequence diagrams is that defined by [ChaMatTel96] as RSC (“Realizable with Synchronous Communication”) and corresponds to the *non-buf weakly implementable* MSCs of [EngMauRen02] characterised by the existence of a trace which is implementable on an architecture with no communication buffers. As stated by the authors of the latter article, this is a necessary condition for MSCs to be interpreted as synchronous interworkings [MauWijWin93]. In fact, the semantics we require for our partially-ordered messages is exactly that of synchronous interworkings.

Notice that, the condition that if $e_1, e_2 \in E$ are ordered, communication events, their paired events $e_1', e_2' \in E$ cannot be inversely ordered, that is:

$$e_1 \leq e_2 \ \wedge \ (e_1 \mapsto e_1' \ \vee \ e_1' \mapsto e_1) \ \wedge \ (e_2 \mapsto e_2' \ \vee \ e_2' \mapsto e_2) \ \Rightarrow \ \neg \ (e_2' \leq e_1')$$

is a strictly weaker condition than RSC, as can be deduced from a cursory glance at the middle diagram of Fig. 5-2. This latter condition defines the class of sequence diagrams which are denoted MO (“Message Ordered”) or CO (“Causally Ordered”) in [ChaMatTel96]. It is easily seen that both MO/CO and RSC are, in turn, stronger conditions than simply assuming that communication channels between the components represented by lifelines are FIFO.

On a TeLa description which is RSC, let $\downarrow$ denote the transitive reduction of the restriction of the partial ordering $\leq$ to local orderings (i.e. for which $\varphi(e_1) = \varphi(e_2)$). In the absence of decomposition of diagrams and of coregions, $\downarrow$ relates adjacent events on the same lifeline. Let $\rightarrow\, = \,\downarrow \cup \mapsto$, so $\rightarrow$ is the immediate predecessor relation. We write $e_1 \downarrow e_2$ for $(e_1, e_2) \in \downarrow$ and $e_1 \rightarrow e_2$ for $(e_1, e_2) \in \rightarrow$.

For $\leq$ the partial order obtained from a TeLa scenario structure, define the relation $<<$ as follows (we write $e_1 << e_2$ for $(e_1, e_2) \in <<$). Events e and e' satisfy $e << e'$ iff one of the following is true:

1. $e << e'$ or $e = e'$

2. $e << e_1 \wedge e_1 << e'$

3. $\neg\exists e_2$ s.t. $e \mapsto e_2$ (i.e. e is not a send event)
$\wedge\ \exists e_1$ s.t. $e' \mapsto e_1$ (i.e. e' is a send event and e_1 is the corresponding receive event)
$\wedge\ e \downarrow e_1$

4. $\neg\exists e_2$ s.t. $e_2 \mapsto e'$ (i.e. e' is not a receive event)
$\wedge\ \exists e_1$ s.t. $e_1 \mapsto e$ (i.e. e is a receive event and e_1 is the corresponding send event)
$\wedge\ e_1 \downarrow e'$

5. $\exists e_1, e_2$ s.t. $e_1 \mapsto e$ and $e' \mapsto e_2$ (i.e. e is a receive event and e' is a send event)
$\wedge\ e_1 \downarrow e_2$

Clearly, $<< \supseteq \leq$. The relation $<<$ also defines a partial-order on the set of events and effectively implements the semantics of synchronous interworkings while maintaining an event-based, rather than a message-based approach. The semantics obtained by inferring the relation $<<$ instead of the relation $\leq$ from test descriptions (which necessarily must satisfy the RSC constraint) will be referred to as the partially-ordered messages interpretation.

The cases of points 3, 4 and 5 of the definition of the relation $<<$ are illustrated in Fig 5-3, where the added immediate causality relations are represented by discontinuous-line arrows.

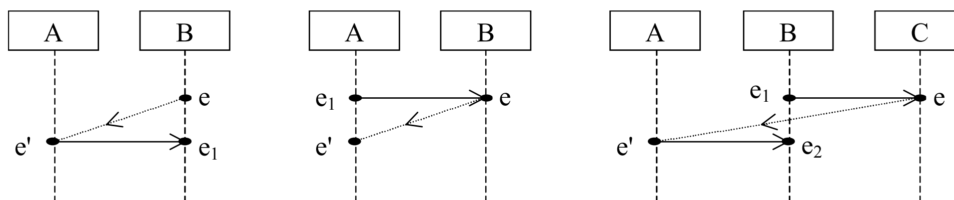


Figure 5-3: Illustration of the extra immediate-causality relations added by the relation $<<$ w.r.t. the relation $\leq$

The definition of $<<$ ensures that if e is a communication event with paired event e':

$$e_1 << e << e_2 \;\Rightarrow\; e_1 << e' << e_2$$

If we denote by 4a the condition of point 4 of the above definition extended to include cases when e' is a receive event, then points 1, 2 and 4a define a relation which is equivalent to supposing that every message is acknowledged. However, the required semantics is a single-flow-of-control type semantics equivalent to supposing that every message is acknowledged *and* that MSC race conditions are resolved as follows: the sender of the message which is to arrive after a certain event waiting to receive notification that this event has indeed occurred before sending. The intermediate “synchronous realisation semantics” defined by points 1, 2 and 4a of the above definition could also be useful but we do not explore this point here. Note that conditions 3 and the part of condition 5 that is not contained in condition 4a are

concerned with resolving race conditions. We suppose that if this semantics is required, it can be specified with the means at the disposal of the specifier by explicitly using synchronous invocations.

3.2 Partially-ordered messages and explicit concurrency

For messages in the scope of a coregion, the extra orderings of the << relation w.r.t. the ≤ relation are illustrated in Figure 5-4, where here we use the MSC notation for coregions.

The case where two messages are in the scope of a coregion at emission and not in the scope of a coregion at reception or vice versa was mentioned in Chapter 3, Section 1.4.3.2 as being a problem for a partially-ordered messages semantics (see also Fig. 3-1). With the partially-ordered message style semantics defined above via the << relation, in such cases, it is the ordering between messages defined by events not in a coregion, rather than the lack of it defined by the events in a coregion, which predominates. Thus, the addition of the extra orderings results in the cancellation of the effect of any such problematic uses of the explicit concurrency operator. This is illustrated by the messages m_1 and m_2 and by the messages m_5 and m_6 of Figure 5-4.

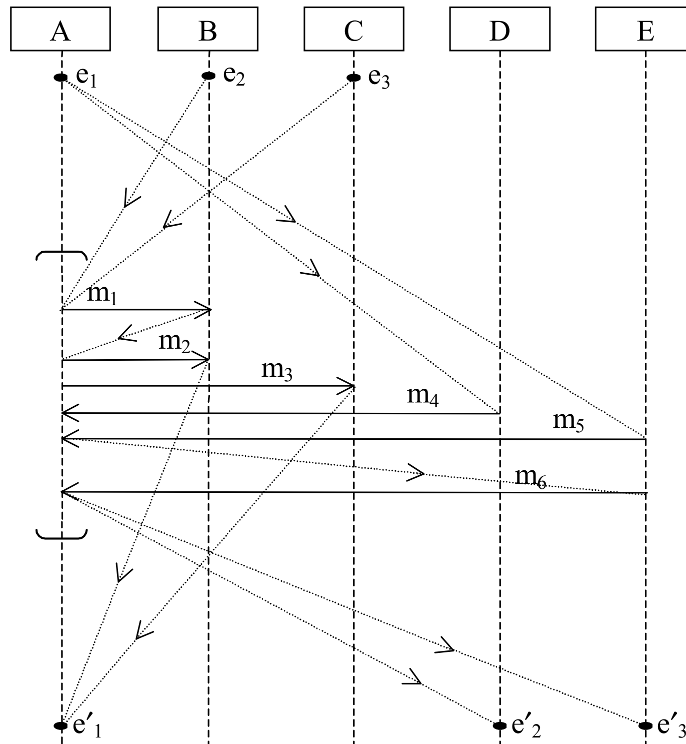


Figure 5-4: Illustration of the extra immediate-causality relations added by the relation << w.r.t. the relation ≤ in the presence of explicit concurrency on lifelines

3.3 Implementation of partially-ordered message semantics

The way we have illustrated the extra causality relations, makes it clear how these can be implemented in an asynchronous context using coordination messages, see Fig 5-5 and Fig. 5-6. It is clear from the second of these figures that in the presence of explicit concurrency on

lifelines, coordination messages will not be enough and extra local ordering relations will also be needed.

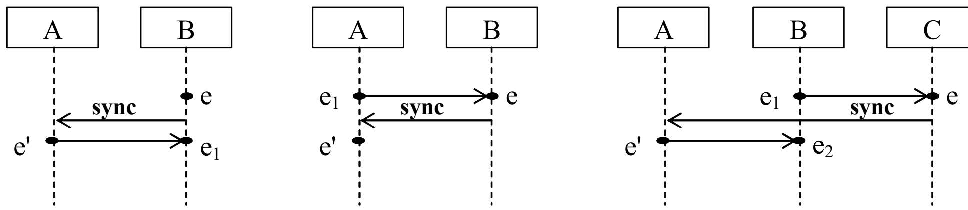


Figure 5-5: The extra causality relations of Fig 5-3 implemented via synchronization messages

Let $!m$, resp. $?m$, denote the event corresponding to the emission, resp. reception, of the message m. In Fig 5-6, the synchronisation message labelled u_1, resp. u_3, represents the extra immediate causality relations between event e_2, resp. e_3, and the events $!m_1$, and $!m_3$. The synchronisation message labelled u_4, resp. u_5, represents the extra immediate causality relations between event e_1 and the event $!m_4$, resp $!m_5$. The synchronisation message labelled u_2, together with the local ordering, represents the extra immediate causality relation between the events $?m_1$ and $!m_2$. The synchronisation message labelled v_5, together with the local ordering, represents the extra immediate causality relation between the events $?m_5$ and $!m_6$. The synchronisation message labelled v_1, resp. v_3, represents the extra immediate causality relation between the events $?m_2$, resp. $?m_3$, and event e'_1. Finally, the synchronisation messages labelled v_4, resp. v_6, represents the extra immediate causality relations between the events $?m_4$ and $?m_6$, and event e'_2, resp. e'_3.

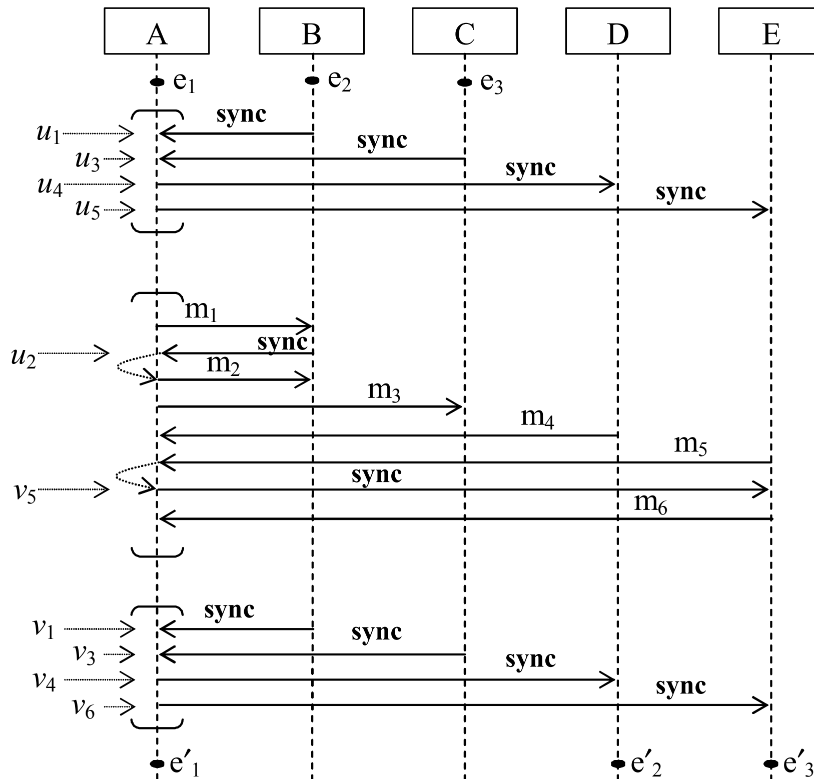


Figure 5-6: The extra causality relations of Fig. 5-4 implemented via synchronization messages

As stated in Chapter 4, the synchronization messages of TeLa do not denote any events; they correspond to MSC general orderings between existing events located on different lifelines. If we suppose, on the contrary, that synchronization messages do denote new events, due to transitivity, a simpler way of specifying the synchronizations necessary for the partially-

ordered messages semantics exists, albeit using a larger number of synchronization messages. This is simply accomplished by adding a synchronisation message before and after every non-synchronization message, making use of extra coregions where appropriate in the case where the non-synchronization messages are in the scope of a coregion.

3.4 Trace formulation of partially-ordered message semantics

Clearly, the set of linearisations conforming to the partially-ordered messages interpretation is a subset of the set of linearisations according to the partially-ordered events interpretation. An equivalent way of deriving this subset would be to place constraints on the set of possible traces conforming to the partially-ordered events interpretation. It is of interest to know what these constraints are.

[EngMauRen02] suggest that "an interworking can be considered as the restriction of the semantics of an MSC to only the nobuf-implementable traces". It would seem then that this is the constraint we require on traces. However, the authors' definition of nobuf-implementable traces does not take into account concurrency. This fact does not affect the authors' definition of the weakly nobuf-implementable requirement for a diagram to be interpreted as a synchronous interworking, since this is an existential property. However, it does affect the above definition of the interworking semantics. With the authors' definition of non-buf implementable, the above definition of the interworking semantics is incorrect. The required definition, not on individual traces but on the set of traces defining the semantics, is as follows

A set of traces, Λ, is said to be nobuf-implementable if and only if for all pairs of labels ($!a$, $?a$) labelling the paired events of a communication,

$$\sigma_1 = \sigma^{pre} \cdot !a \cdot x \cdot ?a \cdot \sigma^{pos} \in \Lambda \quad \Rightarrow \quad \sigma_2, \sigma_3 \in \Lambda$$

$$\text{where } \sigma_2 = \sigma^{pre} \cdot x \cdot !a \cdot ?a \cdot \sigma^{pos} \quad \text{and} \quad \sigma_3 = \sigma^{pre} \cdot !a \cdot ?a \cdot x \cdot \sigma^{pos}$$

That is, the traces of the partially-ordered messages interpretation satisfy the property that the only actions that can occur between those of a communication pair, are actions labelling events which are concurrent with the events of the communication pair.

3.5 Partially-ordered message semantics for a component

We have presented the partially-ordered messages interpretation of TeLa test descriptions. However, we could also apply the partially-ordered messages interpretation to parts of TeLa test descriptions. The existence of a hierarchical component model, and the notion of cut/partition of a snapshot, enables us to specify a level of components for which intra-component communication conforms to the partially-ordered messages interpretation, while inter-component communication conforms to the partially-ordered events interpretation. This possibility may be applied to the whole test system, to the whole tester component, or to individual tester components.

The definition of the partially-ordered messages semantics is in terms of the transitive reduction of the restriction of the partial ordering $\leq$ to local orderings. The notion of "local" in this definition is crucial to the compatibility of diagram decomposition and the partially-ordered messages interpretation. We have defined "local orderings" as those between events for which the location function for the structurally-consistent maximal decomposition returns the same value. The limitation on whether this maximal decomposition is one involving only

base-level components is imposed principally by the ports referenced in the message arrow labels of the diagram.

The definition of the relation << on a component *c* is as follows. We suppose that the RSC property holds for inter-component messages and that the MO property holds for messages sent from the component to its environment (but not for messages in the other direction). The former assumption is essential and the latter enables what appears to be a more useful semantics.

The definition proceeds as for the definition of << on the whole test description except for the stipulation that rules 3-5 only apply to events e, e' such that $\varphi(e)$, $\varphi(e')$ are both subcomponents of *c*, and the addition of rule 3a as follows:

3a. $\exists e_2$ s.t. $\varphi(e_2)$ is a subcomponent of c and $e_2 \mapsto e$ (e_2 is a send event)
$\wedge$ $\varphi(e')$ is a subcomponent of $c \wedge \exists e_1$ s.t. $e' \mapsto e_1$ (e' is a send event)
$\wedge$ $e \downarrow e_1$

The three cases of Fig. 5-3 now apply only to the situation where both e and e' are events on subcomponents of *c*. Rule 3a is a special case of rule 2 of the general definition but with the messages being sent from inside the component in question and received outside of it. This rule is illustrated in Fig. 5-7, where we suppose that c_1 and c_2 are subcomponents of *c* and *d* is a peer component of *c* (that is another member of the generalised interaction framework used for the diagram).

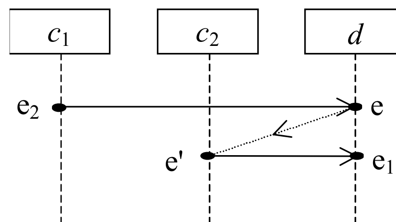


Figure 5-7: Illustration of the extra immediate-causality relations described in rule3a

Chapter VI : Test Synthesis from UML Models of Distributed Software

The work reported on in this chapter was carried out in the context of the COTE project [JarPic01] of the French national research programme RNTL (Réseau National des Technologies Logicielles). A short version was presented in [PicJarTra02].

The initial work on developing the test synthesis method with Umlaut/TGV is reported on in [JézGuePen98], [JérJézGue98] and [Gue01]. The first article proposes a validation framework for integrating formal validation and verification technology in the OO lifecycle. The second presents this framework in more detail, describing the overall scheme for generating a simulation API from a UML model in order to be able to use verification and validation tools, in particular, the TGV test synthesis tool. The third describes how this scheme was implemented in the Umlaut simulator.

Here, we complete the original method and suggest how it could be generalised. In the case of the Umlaut simulator, we also refine the tool support and suggest improvements to it, while at the same time clarifying both the semantics and the constraints the derivation of this semantics imposes on the UML model. To do so, we have benefited from the evaluation and testing of the method & tool that we have undertaken jointly with the partners playing the role of users in the COTE project, one of the main aims of this project being to investigate applicability of the approach to test synthesis in the UML domain.

We also extend the method in order to achieve full integration in UML, firstly, through the use of UML scenario-based test objectives and UML scenario-based test cases and, secondly, through the XMI model exchange with commercial UML CASE tools. We provide all the mechanisms and underlying concepts needed for this integration and, again in the context of the COTE project, have participated in the partial implementation of these extensions and of the connection with the Objecteering tool.

We recapitulate the main proposals for improving the current method at the end of each of the main sections. In doing so, we also briefly discuss the impact any proposed changes would have on the current semantic basis and current tool support, where we include in the latter not only the Umlaut UML simulator but also the TGV test synthesis tool. One of our main concerns in this regard is in extending the method from system testing to general component testing.

The present article also aims to document both the original method & tool and the extensions to it, laying out clearly the different phases and the different semantic and implementation choices in each phase. Throughout the document, there are implementation notes concerning the current prototype tool. These notes pinpoint the places where the current implementation is incomplete or where there are difficulties in implementing the method as presented. In this way, the present article can also serve as the technical manual for the tool.

In the following section we provide the definitions of the additional terms that will be be needed in this chapter and an overview of the method, dividing it into four main parts. The first of these parts, concerning the derivation of a formal model (a labelled transition system) from the UML model of the application is presented in Section 2. The next part, concerning the derivation of a formal model (a labelled transition system) from a UML representation of a test objective - in the O-TeLa language - is presented in Section 3. The kernel of the method and tool, the test synthesis on the formal models, is presented in Section 4. Section 5 presents the last part, the mapping from the labelled transition system representation of the resulting test case to a UML representation of that test case - in the TeLa language.

We illustrate the method described in this document by referring throughout to an example involving a simplified Air Traffic Control system.

1 Introduction

In this section we provide the background and motivation for the work on test synthesis from UML models and define the terms we will use in the rest of the document. We finish the section with an overview of the test synthesis method which, at the same time, provides us with the structure of the rest of the document.

1.1 Background

The need for the automatic synthesis of functional test cases from UML specifications is increasingly being felt in industry. Furthermore, UML is being used in an ever wider range of contexts, in particular, that of distributed system development. The testing of distributed applications has to take into account their use of asynchronous communication and their inherent concurrency.

When dealing with many real-world applications, in particular those involving a significant degree of concurrency, testing of all possible invocation orderings is unrealistic due to a combinatorial explosion in the number of such orderings permitted by the specification. Thus, in applications involving concurrency, user-defined test objectives constitute a way of limiting the number of test cases to be produced by test synthesis from a specification. These test objectives serve to guide the test synthesis process. They can be described in the form of high-level test scenarios which are then easily understood as *behavioural test patterns* by developers.

Other advantages of using test case synthesis according to test objectives for both centralised and distributed applications are the following:

- *Ease of use*: test objectives are independent of low-level design and implementation choices. While defining a high-level test scenario is not difficult when the main classes are identified, refining and adapting it to the final software product is an arduous process. However, the details of the low-level design are contained in the UML specification; completing the test objective with these details is therefore a task which can be left to the automatic synthesis. Allowing test designers to work at the test objective level rather than the test case level thus frees them from the need to specify the low-level detail, enabling them to concentrate on the essential aspects of the test.
- *Coherent development process*: the main expected behaviours can easily be represented as test objectives. Such test objectives can be derived from the use-case scenarios, thus contributing to the overall coherence of the development process.
- *Version/product independence*: test objectives can be chosen to be independent of software versions and variants. This is particularly important in a product-line context [AtkBayBun02], since generic test objectives may be defined for an entire product line [NebPicTra02].

In this document we present a method for the automated synthesis of test cases (with built-in oracle in the form of test verdicts) from test objectives described as high-level scenarios. The method is supported by a prototype tool. The inputs to the method are:

- a set of test objectives, in a UML sequence diagram based notation,

- a UML model of the application, comprising at least a class diagram and a state diagram for each of the classes and actors (in fact, the method could also be used with fewer state machines, see later),
- a description of the initial state of the application in the form of a UML object or deployment diagram.

It should be noted that test objectives may be defined before the UML model of the application is completed: only the main interfaces of classes and actors are needed. A set of test cases – to be represented in a UML sequence diagram based notation [PicJarHeu01] – exactly defining the ordering of call sequences and associated test verdicts, as well as any required object creation, is then automatically synthesised. In defining our method, we have addressed the following issues concerning conformance testing in a UML framework:

- the definition of a complete process with a formal basis to synthesize test cases from UML specifications according to test objectives,
- the definition of a formal operational semantics for UML specifications,
- the definition of a scenario-based language within the UML framework to express test objectives and test cases.

Though the user only deals with UML, our approach to testing is a formal one and we therefore have a precise notion of what is being tested and what is the meaning of a verdict. The underlying formal basis of the method is the synchronous product of Input-Output Labelled Transition Systems (IOLTS). The tool implementing this method results from the incorporation of the TGV tool into the Umlaut UML environment. Umlaut [HoJézGue99] is a CASE tool that manipulates the UML meta-model, enabling automatic model transformation. TGV [JarJér02] is a test synthesis tool based on an on-the-fly and partial traversal of the enumerated labelled transition system of the specification. It was chosen here for its formal basis, for the desirable properties which its test synthesis algorithms can be proven to possess and for the ability to treat systems of significant size which its on-the-fly approach confers. It has already demonstrated its capabilities on large SDL [ITU-T02] and Lotos [ISO89] specifications.

In this document we concentrate on black-box system testing; we derive test cases which concern only the interface between the system under test (SUT) and the external actors. As a consequence, our method does not rely on the implementation under test being derived from instrumented code, unlike other approaches to system testing from UML descriptions. Our synthesis method could also be usefully applied in testing individual components or in testing the integration of different components. However, in the absence of adequate support for components in UML 1.4 and with the current implementation of Umlaut, it is not easily applied to a model which may contain communications between the different entities of the SUT environment. If these entities are actors, as in system testing, such communication cannot appear in the model since it is not allowed by UML, see Section 2.4.1.

1.2 Definition of behavioural concepts

In this section, we extend the definitions of Chapter 2, Section 1, adding those needed for discussing test synthesis. To the general definitions of this chapter we add those of test objective and SUT boundary. We then give additional versions of the definitions of Chapter 2, Section 1 and of the new definitions; for each concept we give one definition adapted to an LTS-based semantics and another definition adapted to a scenario-based semantics. These separate definitions help us to define the relationship between the concepts at the different

levels of our test synthesis method, which requires translating from scenarios to LTSs and back.

1.2.1 General formulation

To the concepts defined in Chapter 2, Section 1, we add two more of importance for test synthesis.

1.2.1.1 TEST OBJECTIVE

A *test objective* is a generic behaviour specification representing an abstract view of some behaviour of the SUT that we wish to test. In test synthesis, the intention is to derive a test case by using the test objective as a criterion for selecting the behaviour to be tested from among the behaviours permitted by the specification (c.f. a property). The test objective will usually include communications between the system and its environment or an abstract view of such communications.

We suppose that in the derived test cases, the SUT environment role will be played by the tester. A test case derived from a test objective specifies how to stimulate any SUT implementation via the SUT component interface and in function of its responses at this interface in such a way as to cause it, if it conforms to its specification, to execute a scenario which fulfils that objective.

1.2.1.1.1 Preamble, test body and postamble

A test case is sometimes divided into three stages with respect to a test purpose:

- The *test case preamble* is the part of the test case taking the SUT from its initial state to a state in which the test purpose can be achieved.
- The *test case body* is the part of a test case containing all the events essential to achieving the test purpose.
- The *test case postamble* is the part of a test case taking the SUT from the state in which it was left by the test body to a state in which a global verdict can be assigned. The first part of the postamble may consist of invocations whose aim is to determine observable aspects of the state in which the CUT was left by the test body, information which may be needed in order to reach a verdict.

We distinguish the term “test objective” from the term “test purpose” since, in order to reduce the calculations carried out by the test synthesis tool, our test objectives may be also select part or all of the preamble and postamble (though similar entities are still called test purposes in [SchEbnGrab00]).

1.2.1.2 THE SUT BOUNDARY IN TEST SYNTHESIS

Though we require the tests derived from a test objective to be of a black-box nature, that is, their description cannot involve events labelled by SUT actions, in the general case, there is no fundamental reason to similarly limit the test objectives to behaviours which do not involve arbitrary SUT actions, including SUT-internal actions, that is, SUT actions which are not shared actions between the SUT and its environment in the synchronous communication sense (see Section 1.2.2.6). In test synthesis, this can be accomplished by parameterising the test synthesis method by the SUT boundary.

The need to represent SUT actions, even SUT-internal actions, in test objectives may arise from a desire to test certain behaviour of a component which is embedded inside the SUT. A

prime example is the case where we are in fact testing a CUT properly contained in the SUT. The meaning of SUT actions in a test objective is that if the SUT implementation conforms to its specification, the execution by the tester of the controllable actions of the derived test will cause the SUT implementation to execute the actions of the test objective and produce the observable actions of the derived test.

1.2.2 LTS-based formulation

1.2.2.1 LABELLED TRANSITION SYSTEM (LTS)

An LTS is a quadruple $M = (Q^M, q_0^M, \Sigma^M, \rightarrow_M)$ where:

- Q^M is a finite non-empty set of states,
- q_0^M is the initial state,
- Σ^M is the alphabet of actions (or Σ-actions[1]),
- $\rightarrow_M \subseteq Q^M \times \Sigma^M \times Q^M$ is the transition relation.

Note that, in contrary to usual practice, our labelled transitions systems do not have internal actions; we will only have need for such actions in input-output labelled transition systems.

1.2.2.2 INPUT OUTPUT LABELLED TRANSITION SYSTEM (IOLTS)

An IOLTS is a quintuple $M = (Q^M, q_0^M, \Sigma^M, \rightarrow_M, \aleph_M)$ where:

- Q^M $(Q^M, q_0^M, \Sigma^M, \rightarrow_M)$ is an LTS,
- $\aleph_M$ is a function from Σ^M to the set $\{i, o, \tau\}$, partitioning the set of actions into input actions, $\Sigma_I^M = \aleph_M^{-1}(i)$, output actions, $\Sigma_O^M = \aleph_M^{-1}(o)$, and internal actions $\Sigma_{int}^M = \aleph_M^{-1}(\tau)$.

Thus, an IOLTS is a labelled transition system in which a notion of *system boundary* is defined by distinguishing the *external actions* – those on this boundary – from the *internal actions*, and the external actions are partitioned into inputs at the system boundary and outputs at the system boundary. In testing, the inputs of the SUT will be the outputs (controllable actions) of the tester and the outputs of the system will be the inputs (observable actions) of the tester.

It is important to note that the concept of IOLTS defined here (that used by the test synthesis tool) does not allow for external actions which are neither inputs nor outputs. That is, the system boundary contains all the external actions; external actions which are internal to the system environment are not contemplated[2]. It is also significant that there is not one unique internal action. This allows us to use test objectives which include internal actions of the specification (the actions are hidden after, rather than before, the synchronous product operation in test synthesis).

---

[1] We will prefix the term action with Σ when referring to the actions in the LTS case whenever confusion may arise with the actions of the "action language" making up UML "action expressions", which we refer to as UML-actions. As we will see later, in the LTS representing the semantics of a UML specification, a single Σ-action may be composed of several UML action expressions, each composed of UML-actions.

[2] The possibility of including such actions would be very useful (see the guidelines for the specification of actor state machines in Section 2.1.2) so that an extension of the basis of the TGV tool in this sense is desirable. In an extended test synthesis mirror operation, the SUT-env internal actions would become tester-internal actions.

1.2.2.3 TEST RESULT (LTS VERSION)

The *test result* is the sequence of external actions actually performed in a test execution together with the verdict for that test execution. The actions of this sequence are either executed by the tester (controllable actions) or executed by the SUT and observed by the tester (observable actions), during the execution of a test.

1.2.2.4 (CONFORMANCE) TEST CASE (LTS VERSION)

A *test case* is a deterministic, controllable IOLTS specifying the stimulation of the SUT by the tester via the SUT component interface, the observation of its responses at this interface, and the assignment of a verdict by the tester. The verdict is assigned in function of whether the test result is consistent with a trace of communications between the SUT and its environment which is permitted by the SUT specification. The consistency is defined via a conformance relation; in test synthesis using the TGV tool, the *ioco* relation is used[3]. The IOLTS has three distinct sets of sink states corresponding to the three verdicts pass, fail and inconclusive, a verdict being accessible from every state. It also satisfies the condition that each state either has one outgoing transition which is a single tester output or has an outgoing transition for every possible tester input (see Section 4.1.3 and Chapter 5, Section 2.1.2.2).

1.2.2.5 TEST OBJECTIVE (LTS VERSION)

A test objective is a generic LTS representing an abstract view of traces the SUT may execute, according to its specification, and that we wish to test. Test synthesis then works by using this LTS as a criterion (c.f. property) for selecting the behaviour to be tested from among the traces permitted by the specification. All the transitions of the test objective LTS are labelled by actions of the specification LTS; in the usual case, some or all of these transitions are labelled by actions describing the emission or reception of invocations between the system and its environment. The genericity enabling the test objective LTS to be used as a selection criterion on the specification LTS is obtained in three ways:

- by abstraction on the actions labelling transitions (using regular expressions),
- by providing a facility for specifying the completion w.r.t. the alphabet Σ of the specification LTS (the asterisk, see Appendix A)
- by implicitly completing all states w.r.t. the alphabet Σ of the specification LTS via a loop transition; in the synchronous communication context, this allows the specification to perform actions which are not specified in the test objective when matching the two.

The test objective contains two types of special states, *accept states* and error states which are here known as *reject states*. Both must be sink states. Since the states of an LTS contain no information and are therefore all identical, these two types of sink state are actually modelled as states having a single appropriately-labelled outgoing transition.

The transitions leading to a reject state in the test objective LTS specify the parts of the specification LTS which do not need to be explored in order to synthesize a test case; they can greatly improve the efficiency of test synthesis. They may be used to exclude messages which are known to actively interfere with the purpose of the test. Perhaps the most common use, however, is to help minimise the synthesized test case by excluding messages which are

---

[3] This states that ∀ traces of visible actions σ in the IOLTS of the specification,

$$\text{outputs_after_}\sigma\text{ (Impl)} \subseteq \text{outputs_after_}\sigma\text{ (Spec)},$$

that is, the outputs of the implementation after σ are included in the outputs of the specification after σ (locks being considered as a particular type of output, detectable via the use of timers).

known to be superfluous for the purposes of the test. This reduction of "noise" is particularly useful for synthesizing tests of concurrent applications in the interleaving-model context.

1.2.2.6 THE SUT BOUNDARY IN TEST SYNTHESIS (LTS VERSION)

As in the general formulation, we wish to allow internal actions to be represented in test objectives and parameterise the test synthesis with the SUT boundary.

In the IOLTS framework, communication is modelled through shared actions i.e. synchronous communication in the sense of CCS [Mil89] or CSP [Hoa85]. In the shared-action communications of the IOLTS framework, one of the two parties performs an input while the other performs an output. In accord with this notion of communication, the actions on the SUT boundary are those which are shared between the SUT and its environment, where, in the test case, the tester takes the place of this environment. Recall that with the definition of IOLTS currently used, the set of actions on the SUT boundary is assumed to be the same set as the set of SUT-external actions (i.e. there are assumed to be no actions internal to the SUT environment).

In IOLTS-based test synthesis, see Section 4, a test case for the SUT, represented as an IOLTS, is synthesized from the following three elements:

- a closed-system specification of the SUT and its environment, represented as the LTS, S, with alphabet Σ^S
- a function $\aleph^S$, as defined above, that makes an IOLTS of the specification LTS by partitioning the actions of its alphabet Σ^S according to the SUT boundary: $\Sigma^S = \Sigma_I^S \cup \Sigma_O^S \cup \Sigma_{int}^S$,
- a test objective, represented as an LTS with alphabet Σ^{TO}, where $\exists$ a name mapping:

$$is_abstraction_of : \Sigma^{TO} \rightarrow \wp(\Sigma^S)$$

taking actions of the test objective LTS to non-empty sets of actions of the specification LTS, defined by the regular expressions embedded in the names of the actions of Σ^{TO}.

In the simplest approach to test synthesis, in the calculation of the synchronous product, the range of the *is_abstraction_of* function is restricted to $\wp(\Sigma_I^S \cup \Sigma_O^S)$ so that the elements of Σ^{TO} are matched with elements of $\Sigma_I^S \cup \Sigma_O^S$. That is, the elements of Σ^{TO} can only represent SUT-external actions of the specification; with the definition of IOLTS currently used, these are necessarily communications between the SUT and its environment. In the approach used in the TGV tool, however, in the calculation of the synchronous product, the range of the *is_abstraction_of* function is all of $\wp(\Sigma^S)$, i.e. $\wp(\Sigma_I^S \cup \Sigma_O^S \cup \Sigma_{int}^S)$, so that the elements of Σ^{TO} are matched with elements of $\Sigma_I^S \cup \Sigma_O^S \cup \Sigma_{int}^S$. That is, elements of Σ^{TO} can also represent SUT-internal actions of the specification

The use of (abstract versions of) SUT-internal actions in test objectives is accomplished by parameterising the synthesis method by the function $\aleph$ of the IOLTS definition above: SUT-internal actions are not hidden, using the information provided by the function $\aleph$, until *after* the calculation of the synchronous product of the test objective LTS and the system specification LTS (see Section 4.1.2). This is the reason why in the IOLTS definition, contrary to usual practice, it is not assumed that all internal actions are identical.

The issue of defining the SUT boundary in system test synthesis using the LTS generated from a UML specification by the Umlaut simulator is discussed in Section 2.4.

1.2.3 Scenario-based formulation

1.2.3.1 EVENTS, ACTIONS AND MESSAGES

We define the *event*[4] as the basic behavioural unit, each event being labelled by an *action* (or Π-action[5]). These actions are one of the following:

- the emission or the reception of a synchronous or asynchronous invocation between two components
- the emission or the reception of a synchronous invocation reply between two components,
- a component-internal action (we include timer manipulations in the component-internal actions).

We define a *message*[6] as an ordered pair comprising a send and a receive event whose labels concern the same invocation and the same two components.

1.2.3.2 SCENARIOS AND SCENARIO-STRUCTURES

A *scenario* is a partially-ordered set of events involving messages being sent between different components, each event being labelled by an action. A *scenario structure* is a (possibly infinite) number of (possibly infinite-length) alternative scenarios.

In the approach described here, both test cases and test objectives, will be described as scenario structures (though the use of alternatives may be restricted in test objectives); in both cases, the notation used will be referred to as a scenario language. The syntax used, both for the test objective scenario structures and the test case scenario structures, extends the UML 1.4 sequence diagram syntax in order to be able to describe a full range of behaviours[7].

1.2.3.3 TEST RESULT (SCENARIO VERSION)

The *test result* is the scenario of external events actually actually performed in a test execution together with the verdict for that test execution. The events of this scenario are either executed by the tester (in which case they are labelled by controllable actions) or executed by the SUT and observed by the tester (in which case they are labelled by observable actions), during the execution of a test.

1.2.3.4 (CONFORMANCE) TEST CASE (SCENARIO VERSION)

A test case is a scenario structure specifying the stimulation of the SUT by the tester via the SUT component interface, the observation of its responses at this interface, and the assignment of a verdict by the tester. The verdict is assigned in function of whether the observed test result is consistent with a scenario involving the SUT and its environment which

---

[4] The notion of event used here corresponds to the UML 2.0 notion of "event occurrence".

[5] We will prefix the term action with Π when referring to the actions in the partial order case whenever confusion may arise with the actions of the "action language" making up UML "action expressions", which we will refer to as UML-actions. The relation between the Π-actions and UML-actions may not be one-to-one, notably concerning the processing in auto-invocations or concerning other actions which are internal to a component.

[6] The term message was used ambiguously in UML 1.4; in some parts of the document a synchronous invocation is treated as being composed of two messages and in other parts of only one message. For clarity, we state that in our use of the term message, a synchronous invocation is composed of two messages.

[7] Our syntax and semantics are broadly compatible with the upcoming UML 2.0 sequence diagrams, both our diagrams and theirs being largely inspired by the ITU-T MSC standard.

is permitted by the SUT specification. The consistency is defined via a conformance relation. In the approach presented in this document, test synthesis is performed by translating to LTSs; the conformance relation is therefore given by this translation and the conformance relation on LTSs.

1.2.3.5 TEST OBJECTIVE (SCENARIO VERSION)

A test objective is a set of generic scenario structures (though the use of alternatives may be more limited than in a test case) representing an abstract view of scenarios that the SUT may execute, according to its specification, and that we wish to test. Test synthesis then works by using these scenario structures as a criterion (c.f. property) for selecting the behaviour to be tested from among the scenarios permitted by the specification. The test objective will usually include messages exchanged between the SUT and its environment or an abstract view of such messages. The genericity enabling the set of scenario structures defining the test objective to be used as a selection criterion on the specification scenario structure is obtained in two ways:

- By abstraction on component names, method names or on method parameters using wildcards: in the approach presented in this document, test synthesis is performed by translating from scenario structures to LTSs; wildcards are simply translated to regular expressions.
- By allowing the specification to perform actions which are not specified in the test objective when matching the two: in the approach presented in this document, test synthesis is performed by translating from scenario structures to LTSs so that the genericity is defined by this translation and the automata completion described in the LTS case.

Concerning the first point, the abstraction discussed above is to be resolved by matching the actions labelling the events of the test objective with those labelling the events of the specification *during* the test synthesis; the test objective thus selects a single test case. In [BonMauBou01] and [BonPot03], the same abstractions are used but here the intention is to resolve them by matching the actions labelling the events of the test objective with those labelling the events of the specification *prior* to test synthesis, in a kind of pre-compilation phase. This pre-compilation generates a set of more concrete test objectives so the end result is that the original abstract test objective selects a set of test cases, each selected by a more concrete test objective. [BonPot03] uses the term "test strategy" for an test objective specified with this intention. Note that in [BonPot03], the matching is performed using only the external actions, i.e. it is supposed that test objectives cannot contain internal actions, though this restriction does not seem to be fundamental. Though much genericity can be resolved in either of these two ways, genericity that is exclusive to the tester must be resolved in pre-compilation.

Concerning the second point, it may be desirable to use a more distinguishing definition in the partial-order case than in the LTS case, even if this distinction is lost in the translation to LTSs. Thus, the abstraction relation between a test-objective scenario structure and a corresponding test-case scenario structure could be defined to conserve the distinction between choice and concurrency as in [Jar02].

In the most general formulation of the method we have developed, a test objective comprises two parts, each part specified as a set of scenario structures (restricted, as mentioned above):

- the specification of the (external or internal) SUT behaviour that the test designer wants to test; the *accept* scenarios are used as positive criteria for selecting scenario structures of the specification which are relevant for the test case;

- the specification of the (external or internal) SUT behaviour that the test designer wants to avoid in the test; the *reject* scenarios are used as negative criteria to avoid selecting scenario structures of the specification which are irrelevant for the test case;

Regarding the role of the reject part of the test objective, since in the approach presented in this document, test synthesis is performed by translating to LTSs, this role is given by this translation and the role of the reject element in the LTS case. However, a major use of the reject element in the LTS case is simply to deal with unwanted interleavings, so that the role of the reject in a fully partial-order test synthesis method such as that defined in [Jar02] is likely to be reduced.

1.2.3.6 THE SUT BOUNDARY IN TEST SYNTHESIS (SCENARIO VERSION)

As in the general formulation, we wish to parameterise the test synthesis with the SUT boundary. In the approach presented in this document, test synthesis is performed by translating to LTSs; the SUT boundary for the partial-order representation is therefore given by this translation and the SUT boundary in the LTS case.

How the SUT boundary used for system test synthesis with TGV/Umlaut is reflected in the scenario-based test-objectives used for this test synthesis is discussed in Section 3.2.3. It will depend firstly on the relation between the Π-actions of the scenario-based representation of the test objective and the Σ-actions of the LTS generated by Umlaut. We try to ensure there is a one-to-one correspondence between actions at the two levels, at least for those Π-actions which are to be mapped onto external actions, though even in the latter case, it is not always possible. The influence of the atomicity of the LTS generated by Umlaut on the test objectives is discussed in Section 3.2.2.

1.2.3.7 CURRENT PROTOTYPE: IMPLEMENTATION NOTES

The test objectives which can be specified in sequence diagram form in the prototype developed in the COTE project [JarPic01] are more limited. Firstly, they use only a single accept behaviour scenario structure. The expressiveness of the *reject* scenario structures is partially recovered via messages annotated reject that can be used inside the accept scenario, leading to more compact representations in some cases. Secondly, the scenario structures are supposed totally ordered.

1.3 Overview of the approach

Based on a UML specification, a real implementation of the system can be derived. However we can never be 100% sure that the real system works as expected. One way to improve confidence in the system is to check its conformance with respect to the specification - which we assume to be correct - using an appropriate suite of test cases. A test driver is then used to execute the test on the implemented system by simulating its environment.

1.3.1 The method

Fig. 6-1 gives an overview of the method for synthesising test cases from a UML specification, according to test objectives. The inputs of the method are a UML specification (class diagram, state diagrams and object/deployment diagram) and a test objective in the form of scenario structures described in the O-TeLa language. The output is an abstract test case in the form of a scenario structure described in the TeLa language from which an

executable test case can be generated for a given platform (Corba, Java/RMI, .Net …). Both the O-TeLa and TeLa language are based on UML sequence diagrams but may also make use of UML activity diagrams and UML class diagrams, and may also refer to the class diagrams of the specification. Both input and output can be provided as XMI files enabling the initial modelling, as well as the visualisation of the derived test case, to be done with any UML case tool. In the COTE project, the Objecteering tool is used.

The method itself is divided into four main parts:

1. *Formal specification derivation*, in which a simulation API is derived from the UML specification, thereby enabling the LTS defining its semantics to be built incrementally on demand (termed “on the fly”), see Section 2. The atomicity of the transitions of this LTS define the atomicity of the simulation steps. In system test synthesis, we also derive the specification of the SUT boundary (which Σ-actions are internal and which are external to the SUT) and the direction of the external Σ-actions (which are inputs and which are outputs). The partition of the Σ-actions of the specification LTS into input Σ-actions, output Σ-actions and internal Σ-actions, which enables IOLTSs to be defined from the LTSs of the method is also discussed in this section.
2. *Formal objective derivation*, in which an LTS is derived from the test objective, see Section 3. Though currently the whole LTS is built, this part of the method could conceivably also be done via an API enabling “on the fly” treatment of the test objectives as well as of the UML specification.
3. *Test synthesi*s on the formal models, in which an IOLTS representing the test case is derived on-the-fly, see Section 4.
4. *UML test case derivation*, in which a UML representation is derived from the resulting IOLTS test cases, see Section 5.

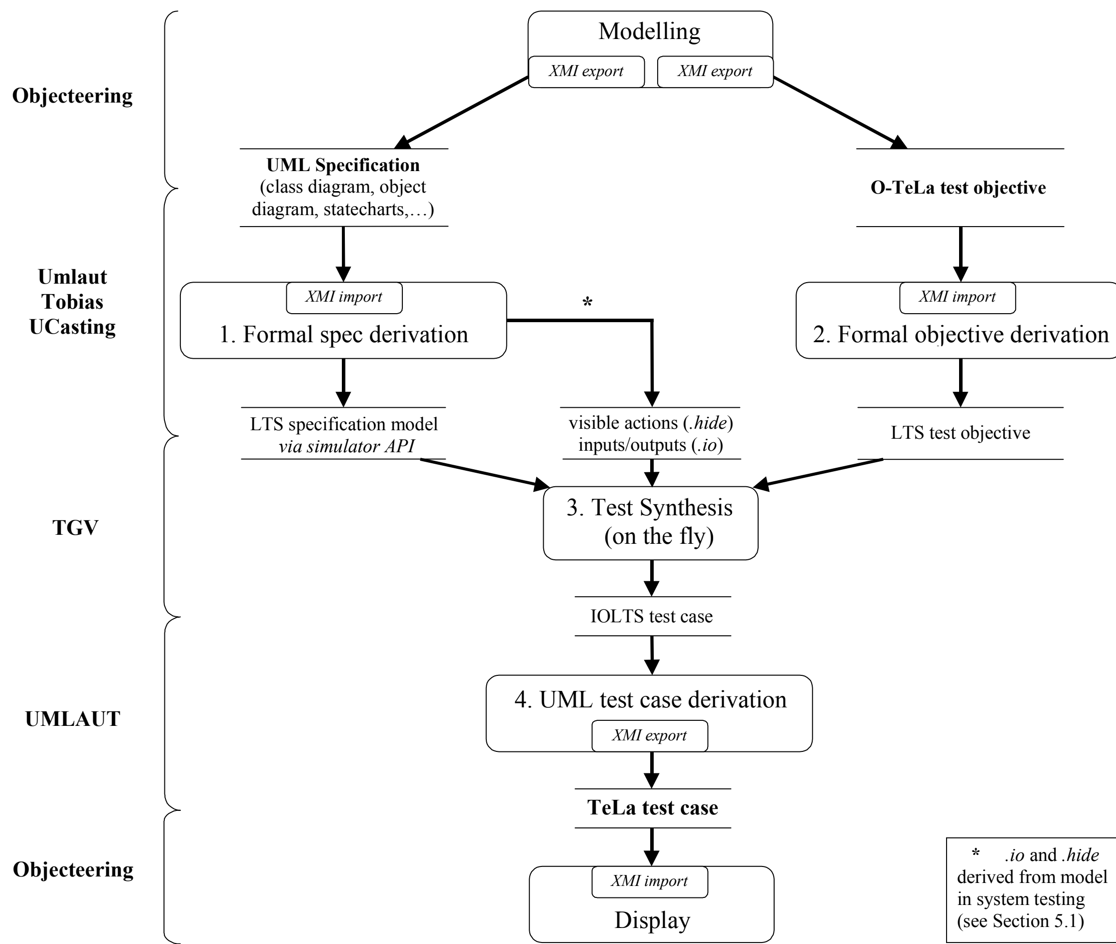


Figure 6-1: The synthesis method.

1.3.1.1 CURRENT PROTOTYPE: IMPLEMENTATION NOTES

Two-tier TeLa diagrams, i.e. those composed of sequence diagrams linked by activity diagrams, are not used so that there is no exchange of activity diagrams between Objecteering and Umlaut. In addition, the passage from an IOLTS test case to TeLa test case has not been fully automated.

1.3.2 The tools implementing the method

1.3.2.1 OBJECTEERING

Objecteering is a commercial UML case tool which can generate an XMI representation of a UML model from its internal representation of that model and, conversely, can read an XMI representation of a UML model, generating an internal representation of that model.

1.3.2.2 UMLAUT

Umlaut [HoJézGue99] is a framework dedicated to the manipulation of UML models. The Umlaut module of interest here is the simulation module, which generates a simulation API from the UML specification. The semantics of the UML specification is the LTS which would be created by using this API to explore all possible states. However, this LTS is not created explicitly, instead, the API is used by the test synthesis tool (and other tools which we do not

discuss in this paper) to explore the part of the LTS of interest (that is, the state graph is created on-the-fly).

1.3.2.3 TGV

TGV [JarJér02] is a test synthesis tool which has been integrated into the Umlaut environment. We leave any further description to Section 4.

1.3.2.4 U-CASTING AND TOBIAS

U-Casting [AerJen03] and Tobias [BonMauBou01] are tools for generating test objectives from OCL constraints (U-Casting) or from more abstract test objectives (Tobias), termed test strategies.

1.3.3 Using the prototype integrated tool

For the procedure to be followed in order to use:

- Objecteering to create a UML model and generate its XMI representation
- U-Casting and Tobias to derive test objectives in the form of LTSs either from a UML model or from an O-TeLa expression specified using Objecteering,
- the Umlaut simulator to import an XMI representation of a model created using Objecteering and to derive a simulation API from this model,
- TGV and a simulation API derived by the Umlaut simulator in order to synthesize a test case, in the form of an IOLTS, from a test objective in the form of an LTS
- Objecteering to display TeLa test cases

see the documents of the COTE project [JarPic01] (in particular Livrable 4.6).

As already stated, the derivation of the formal test case, that is, the translation of the IOLTS resulting from test synthesis into a TeLa diagram, has not been automated. Also as already stated, the principles involved in this translation are treated in Section 5 of the present document.

2 Derivation of LTS from UML model

The semantics of a UML specification is defined as its accessibility graph in the form of a labelled transition system (LTS). The division of the transitions of this LTS into external and internal, and the division of the external transitions into inputs and outputs then defines the IOLTS required for the test synthesis algorithms. In this section, we first discuss the production of a “simulatable” UML specification and its importation in Umlaut in XMI form. We then show how this model is transformed and how a simulation API is generated from the transformed model. Next we show how the SUT boundary which parameterises test synthesis is defined for this API; in the system testing case, the SUT boundary is automatically derived from the UML specification. Figure 6-2 shows the part of the method that is presented in this section.

Finally, we show how the formal specification derivation proceeds for the ATC application example and we give some possible enhancements to this part of the method.

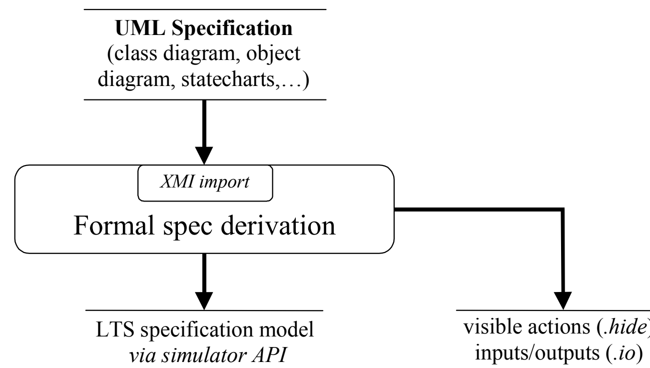


Figure 6-2: Formal specification derivation part of the method.

2.1 Prerequisites for the derivation of an LTS

The minimum requirements placed on the application model by the test synthesis method are that it contain a class diagram, an object or deployment diagram and a state machine for each of the classes (with a particular syntax for the action expressions of the state-machine transitions)[8]. Inevitably, however, there are also some restrictions on the use of these diagrams in order for a model to be “simulatable”, that is, in order for the Umlaut simulator to be able to derive a simulation API from it. This is since, as a general-purpose language, UML is too permissive in some respects, in which cases it must be restricted, and is incomplete in other respects, in which cases information must be added. It also contains some ambiguities which must be resolved. The “simulatable” model edited in an external case tool and exported in XMI form must then be imported into Umlaut.

2.1.1 Specifying static aspects through UML class diagrams

Static aspects of Umlaut-simulatable UML models are described using class diagrams. The derivation of a simulation API from a UML model imposes some restrictions on the class diagrams defining the static aspects of that model. These are mainly due to the semantic

[8] Any OCL present in these diagrams is not taken into account when generating the LTS.

choices which the simulator inevitably makes, in particular those concerning inter-object communication. See Section 2.3 for more details.

2.1.1.1 SYNCHRONOUS AND ASYNCHRONOUS OPERATIONS

In Umlaut-simulatable models, invocations must be synchronous or asynchronous according to whether the corresponding operation is declared with a return type or not. Thus, we say that the corresponding operation itself is synchronous or asynchronous according to whether or not it is declared with a return type. Signals must be modelled as asynchronous invocations.

2.1.1.2 ACTIVE AND PASSIVE OBJECTS

Since an invocation to an active object passes via a FIFO queue, see Section 2.3.2, objects which are declared to be active are not allowed to have synchronous operations. This is to ensure that all objects are in a well-defined state of their associated state machines at the end of a simulation step, i.e. that the set of action expressions belonging to the state-machine transitions that have fired have all been completely executed. The situation in which, at the end of a simulation step, a caller is blocked between two states of its state machine waiting for a reply, cannot then arise.

2.1.1.3 USE OF ASSOCIATIONS

Another restriction concerns whether invocations can be made to objects of classes which are not connected via an association. This point it not clear in UML and different policies are sometimes taken on it. For clarity, we state here that the policy of the Umlaut simulator is that an invocation cannot be made unless the corresponding association has been explicitly specified between the two classes. Similarly, in order for an object of a class to create objects of another class, the two classes must also be linked via an association.

2.1.1.4 ALLOWING FOR DYNAMIC CREATION OF OBJECTS

A further restriction concerns dynamic creation of objects during simulation. As explained in Section 2.1.2.4, there may be a need to declare additional attributes in the class diagram in order to store dynamically-created objects.

2.1.1.5 CURRENT PROTOTYPE: IMPLEMENTATION NOTES

The implementation of inheritance is incomplete so that, currently, problems may arise if models involving inheritance are used.

2.1.2 Specifying dynamic aspects through UML state diagrams

The dynamic aspect of UML is a key part to giving a UML specification an operational semantics. A state machine attached to a class specifies the life cycle of objects of that class, that is, it describes how objects respond when another object sends them a message or when a given event occurs. The derivation of a simulation API from a UML model imposes some restrictions on the state machines defining the dynamic aspects of that model. These are mainly due to the semantic choices which the simulator inevitably makes, in particular those concerning inter-object communication, see Section 2.3 for more details.

2.1.2.1 STATE MACHINES FOR THE EXTERNAL ACTORS

The synthesis method requires a closed system, see Section 2.3.3, and therefore also requires that state machines be associated to the external actors. The behaviour of the system as a whole is given by the combined execution of the state machines contained in the system and

the state machines that model its environment (modulo certain assumptions about inter-object communication).

2.1.2.2 UNSPECIFIED RECEPTION

The behaviour of UML state machines on unspecified reception is described in the UML 1.4 standard in the following terms:

> "If no transition is enabled and the event is not in the deferred event list of the current state configuration, the event is discarded and the run-to-completion step is completed"
>
> *UML 1.4, §2.12.4.*

and again:

> "If an event does not trigger any transition, it is discarded",
>
> *UML 1.4 §3.78.1*

From these quotes it is clear that, semantically speaking, each state contains a default loop transition for each operation or signal which does not trigger any of the other outgoing transitions in that state; these implicit transitions have no effect apart from simply removing the corresponding event from the input queue. We consider that an invocation to an operation which is not declared in the class diagram (or sending a signal not so declared) is an error that can be detected statically, so that our UML state machines need only be semantically completed with respect to the set of operations/signals defined in the corresponding class diagram.

2.1.2.3 THE ACTION LANGUAGE USED IN UMLAUT STATE MACHINES

UML state machines are parameterised by action expressions, which can be placed on transitions or in states. Thus to make a UML state machine executable, an operational semantics is needed for the language used to describe the UML-actions. The action expression of a UML state machine transition may be quite complex since the granularity of transitions in UML state machines is often quite coarse. Until quite recently [OMG03] no language was prescribed for these expressions, so that in UML 1.4 any text can be used. The current version of the Umlaut simulator therefore uses an ad-hoc action language to describe the creation/destruction of objects and links, the assignment of values to attributes, the invocation of methods, etc.

In fact, a sublanguage of the Eiffel programming language is chosen in order to avoid parsing action expressions, since this is the language used to implement the simulation API which is derived from the executable UML specification[9]. The syntax of this language must therefore be used in the action expressions of the model defined in Objecteering, where it is treated simply as text, see Fig. 6-3 for examples. The identifiers for the elements of the model (objects, roles, attributes etc.) are used directly in this action language.

The fact that action expressions are not compiled but simply copied whole to the appropriate place in the simulation code restricts the action language to a sublanguage of Eiffel, due to the Eiffel scoping rules. For example, local variables cannot be declared in action expressions since to do so would lead to the Eiffel syntax not being respected in the resulting simulation code (local variables cannot be declared at any point in Eiffel code). Therefore, the only variables that can be used in an action expression are either formal parameters of the

[9] In a similar way, in the Agedis project [1], where the UML specification is compiled to the IF language, IF is also used as the action language. Similarly, in the Rhapsody tool, the action language is C++.

operation whose invocation is the trigger of the transition or attributes of the object itself (however, concerning the attributes, see Section 2.3.7). Use of loop indexes, for example, requires that they be declared as attributes. This problem could perhaps be solved using procedure declarations.

In order to use the action language, we need to know how an object is to refer to other objects of the model. In the Umlaut simulator, each object has an implicit attribute for each association in which it is involved, and the identifier of this attribute is the role name at the other end of the association. In the case of an association with multiplicity greater than one, all objects of the same class linked via an instance of the same association play the same role. In such cases, rather than identifying a simple attribute, the role name identifies an array in the Umlaut simulator action language (Eiffel). Thus an object at one end of a set of links distinguishes between multiple objects playing the same role at the other end of the links by using the features of the Eiffel array class, in particular the `item` feature.

2.1.2.4 MODELLING OBJECT CREATION

A predefined creation procedure (or, in C++ terms, constructor) called `make` exists for creating objects of any given UML class in the Umlaut action language. If a user-defined creation procedure is to be used, this procedure should call the pre-defined one. However, there are some limitations in the treatment of user-defined creation procedures. To which transition of the state-machine of the class of the created object is the creation procedure to be associated? The UML stereotype «create» is described in the following terms in the standard:

> Create is a stereotyped call event denoting that the instance receiving the event has just been created. For state machines, it triggers the initial transition at the topmost level of the state machine (and is the only kind of trigger that may be applied to an initial transition).
> *UML 1.4, §2.12.2.1*

Apparently, then, the invocation of the creation procedure is the trigger for the transition from the initial pseudo-state to the first proper state and the action expression of this transition expresses the content of the creation procedure. However, UML state machines are limited to a single initial pseudo-state from which there can be only one outgoing transition with only one trigger. It would seem, therefore, that UML classes cannot have multiple creation procedures! If a user-defined creation procedure is to be used in a model to be simulated with the Umlaut simulator, it must be stereotyped «create».

Another problem with dynamic creation concerns the fact that the syntax of the action language used (Eiffel) demands that such objects be assigned to a variable on creation. However, as stated above, local variables cannot be declared in action expressions. In consequence, when object creation is to take place via an association of multiplicity greater than one, a temporary variable to which the created objects can be assigned must already exist since, in the multiplicity-greater-than-one case, the role name identifies an Eiffel array.

One solution to this problem is to declare, in the creating class, an attribute whose type is the class of the object(s) to be created. This attribute can then be used to temporarily store the references to newly-created objects. After creation, the new object reference can then be added to the array identified by the role in the corresponding association. This addition can be done either by using the `add` feature of Eiffel arrays, or by using a special-purpose feature of

the arrays used by the Umlaut action language: `add_<rolename>`[10]. An alternative solution is to declare a separate attribute, whose type is the class of the created object, for each object which is to be created dynamically in the course of simulation.

2.1.2.5 GUIDELINES FOR THE SPECIFICATION OF PASSIVE-OBJECT STATE MACHINES

Passive object state machines are currently not allowed to contain triggerless transitions. The use of such transitions in passive-object state machines would imply that the state from which this transition is an outgoing transition is locally unstable in the sense that the object could never remain in such a state awaiting some external event. We require all state machine states to be locally stable.

If it is required that a decision be reflected in the structure of the state machine rather than being contained in an action expression of a single transition, it can be modelled using a conditional branch via a choice pseudo-state. This possibility then removes any need for guarded triggerless transitions in passive objects.

2.1.2.6 GUIDELINES FOR THE SPECIFICATION OF ACTIVE-OBJECT STATE MACHINES

The atomicity of the transitions of the LTS generated by the Umlaut simulator means that several UML-actions, such as the sending and receiving of synchronous or asynchronous invocations or of synchronous invocation replies, may be subsumed in a single LTS transition. Furthermore, the Σ-action used to label such a transition will only contain information about the first of these UML-actions to occur, The other UML-actions are not explicitly represented in the LTS semantics so that all information concerning them will be inaccessible. See Section 2.3.5 for details of this labelling function. Therefore, the atomicity of the transitions of the LTS dictates the need for guidelines for the specification of active-object state machines.

Information concerning the value returned as the result of a synchronous invocation of an operation by an active object will be inaccessible in the LTS generated by Umlaut. Therefore, if a value sent to an active object is to figure explicitly in the generated transition system, it must be sent via an asynchronous invocation. This visibility problem sometimes obliges what is more naturally modelled as a synchronous invocation to be specified as two asynchronous invocations.

To be fully conformant to the object paradigm, all processing should be by method invocation. This implies that the action expression of a triggerless transition of an active-object state machine should contain only invocations. In fact, in order to generate LTSs which are well-adapted to simulation, ideally, the action expression of a triggerless active-object transition should contain exactly one invocation and nothing else. This stronger restriction also ensures that the labels on the transitions of the LTS generated by Umlaut all have a very simple form. A triggerless transition of any other form can be converted into two transitions, one of which is an auto-invocation – a triggerless transition of the required form – and the other of which contains the required processing in its action expression and is triggered by this auto-invocation. Clearly, an operation whose body is represented by the action expression containing the required processing must be declared, in order for its auto-invocation to be used as the trigger of this second transition.

---

[10] Previously, the array itself was not created until the first use of the `add_<rolename>` feature, so the first use of this feature was not equivalent to a use of the array feature `add`. However, the array is now created in the predefined creation procedure of the owning object.

Due to the granularity of the LTS generated from the specification by the Umlaut simulator, see Section 2.3.2, and the treatment of auto-invocations in active-objects, see Section 2.3.7, in an auto-invocation, the state-machine transition invoking the operation and the state-machine transition executing its body are part of the same LTS transition of the global system. This LTS transition will be labelled by the action expression of the triggerless transition, i.e. by the auto-invocation, see Section 2.3.5.

Note that these guidelines concerning auto-invocation ensure that no internal processing is visible. The test objectives used in test synthesis cannot therefore refer to any such internal processing and it will not be visible in any synthesized test case. With the current granularity of the LTS generated by Umlaut, this is not in any way a drawback, but it might be considered as such in a simulator with a finer semantics.

2.1.2.7 GUIDELINES FOR THE SPECIFICATION OF ACTOR STATE MACHINES

The atomicity of the transitions of the LTS generated by Umlaut, see Section 2.3.2, and the need to relate the synchronous-communication architecture required by TGV to the communication architecture assumed by Umlaut, dictate the need for guidelines for the specification of actor state machines.

Concerning the visibility problem for values sent to active objects mentioned above, in the case where the active objects are actors and the values in question are sent by the system, the guidelines are of great importance for system test synthesis. As regards the need to use two asynchronous invocations to model the synchronous invocation of a passive system object operation by an external actor, this is currently obligatory, see the implementation notes at the end of this section. If the reception by an external actor of such an asynchronous invocation is not to trigger the execution of an action expression, there is no need to explicitly specify a transition triggered by this reception. This is because, due to the UML state machine behaviour on unspecified reception described above, if no such transition is specified, a transition with an empty action expression triggered by the invocation is implicit. However, leaving such transitions implicit is not necessarily good practice.

It is stated above that the action expression of a triggerless active-object transition should contain exactly one invocation. In the specific case of the external actors, in order to generate transition systems which are optimal for test synthesis (and, in particular, for test objective specification), the action expression of *any* transition of their state machines should contain no more than one invocation. In general, an invocation of an actor to the system will be contained in a triggerless (and usually spontaneous, i.e. unguarded) transition. In some cases, however, such an invocation may be contained in the action expression of a transition which is invoked by the actor itself (but never by the system), see the active-object guidelines concerning auto-invocations above.

Currently, actor auto-invocations are not properly taken into account by the test synthesis algorithms[11]; they lead to Σ-actions which are internal to the system environment but which, in test synthesis, will be treated as SUT inputs, see Section 2.4.2. However, in most cases, if the present guidelines are followed, the action expression of the transition triggered by the auto-invocation of the actor operation, <*op_actor*>, will indeed contain a single invocation to an SUT operation, <*op_SUT*>, i.e. an SUT input. To facilitate the use of test objectives, we here propose the naming convention <*op_actor*> = <*op_SUT*> , i.e. we use the same name for

[11] As already stated, the tool assumes that all transitions labelled by external actions describe communications between the SUT and its environment so that all external transitions are either system inputs or system outputs.

the actor operation invoked by the auto-invocation as for the system operation whose invocation is contained in the action expression constituting the body of that actor operation.

It is very ill-advised to specify a transition in an actor state-machine whose action expression contains an actor-to-system invocation but which is triggered by a system-to-actor invocation. If such transitions are used, the atomicity of the LTS generated by Umlaut will make it impossible to partition the external Σ-actions into system inputs and system outputs, this being crucial for meaningful test synthesis.

When specifying actor state machines, care should be taken with loops which involve object creation. This is since such loops may currently lead to infinite branching, see also Section 2.3.1.

2.1.2.8 DEFAULT "DAISY" STATE MACHINES

For actors where a state machine is not provided, a sender "daisy" state machine – a machine having one state and a set of spontaneous loop transitions, each invoking one of the operations to which the actor is associated – could be assumed. Due to the behaviour of UML state machines on unspecified reception, see above, transitions with empty action expressions and triggered by invocations to any of the actors' operations do not need to be specified. Of course, use of "daisy" machines for the external actors instead of explicitly-specified state machines would increase the size of the LTS generated by Umlaut.

For system classes where a state machine is not provided, a receiver "daisy" state machine – a machine having one state and a set of loop transitions with an empty action expression, each triggered by one of its operations – could be assumed. In fact, due to the behaviour of UML state machines on unspecified reception, see above, these transitions exist by default even if not specified explicitly. Instead of a "daisy" state machine, therefore, a state machine with one state (apart from the initial pseudo-state) and no outgoing transitions could be used. The transitions of the LTS generated from the specification will be the same whether these transitions are left implicit or not.

For applicable test synthesis, strictly speaking, such system-object "daisy" machines can only be used for objects which do not contribute to the observable behaviour which is to be tested. However, this restriction does not apply to interactive simulation.

2.1.2.9 SOME SIGNIFICANT RESTRICTIONS

Hierarchical state machines, that is, ones with composite states, are not yet dealt with so that all state machines must be flat. However, dealing with hierarchical state machines is not trivial, and in extending the current implementation to deal with them, restrictions will need to be imposed in order to guarantee that they can always be flattened (i.e. that an equivalent non-hierarchical state machine can always be derived).

Other significant restrictions are that action expressions in states (including entry and exit actions) are not treated; nor are deferred events or history pseudo-states.

2.1.2.10 CURRENT PROTOTYPE: IMPLEMENTATION NOTES

In the current implementation, an attribute to temporarily store the references to newly-created objects must also be used, not only when object creation is to take place via an association of multiplicity greater than one, but also when object creation is to take place via an association of multiplicity equal to one. This is since the Eiffel class corresponding to a UML class accessed via an association for which no object exists in the initial state is

currently a deferred (abstract) class whereas the Eiffel class corresponding to a UML class which is the type of an attribute is always an effective (concrete) class.

In the current prototype, the implementation of synchronous operations of passive system objects by external actors is incomplete. Currently, external actors can only invoke operations of system objects asynchronously.

Whether or not conditional branches are fully taken into account in the current prototype remains to be verified. Conditional branches are an important feature, particularly in passive object state machines, helping to avoid overly large action expressions, as mentioned in the guidelines for the specification of passive-object state machines.

The special-purpose `add_<rolename>` feature is of limited use without similar features to delete an object from the set of objects linked to a given object `O` and perhaps also to navigate from `O` to one of these objects (instead of using the `item` feature of Eiffel arrays). A more satisfactory solution would be for the simulation API generator to define a specific subclass of collections with all the desired features (add, remove, etc.) and to use this instead of Eiffel arrays.

In the current prototype, if a class or actor of the model does not have an associated state machine, a "daisy" state machines is not generated in any circumstances so that, with the current implementation, a state machine must be given for each of the classes and actors of the model.

The simple use of copy-paste of action expressions in order to avoid parsing them leads to other problems besides those mentioned above concerning the use of temporary variables. In particular, the use of carriage return characters or double quote characters in the actions expressions of triggerless transitions leads to the generation of incorrect Eiffel code and Eiffel compilation problems. However, if the guidelines for the specification of active objects given in this section are followed, this problem should never arise. A problem which is more difficult to avoid is that, even in the parameters of an operation call of an action expression (more precisely, in all but the last parameter of such an operation call), the use of double quote characters can cause problems, though in this case at simulation time rather than compilation time.

In the current prototype, guards on triggerless transitions of active objects have no effect, that is, they are assumed to be always true, see Section 2.3.4.

In the state diagrams of Objecteering, the action expression of a transition appears explicitly in the transition label. As a consequence, when the action expression becomes large, the diagrams become unreadable. In the context of the COTE project, FT R&D have developed an Objecteering module to specify action expressions in a UML note rather than on the transition itself. This is a very practical way to avoid having transitions with extremely long labels.

2.1.3 Specifying the initial configuration through a UML object diagram

Class diagrams describe the possible configurations of objects in the system. However, calculation of the possible ways in which the configuration can evolve, i.e. simulation, requires the specification of an initial configuration. The derivation of a simulation API from a UML model imposes some restrictions on the object diagram defining the initial state of that model. These are mainly due to the semantic choices which the simulator inevitably makes, in particular those concerning inter-object communication. See Section 2.3 for more details.

In a simulator which cannot treat data symbolically, all attributes and links which are to be used before being assigned to must either be instantiated in the initial state or be defined to have a default value. In the Umlaut simulator, an object diagram or, if we also wish to show the localisation of each object in a distributed application, a deployment diagram, is used for specifying initially-existing objects and initial values of their attributes.

Clearly, it is not possible to specify an arbitrary global state (see Section 2.3.2) of the system using a UML object or deployment diagram, since the original UML model does not provide a means to specify the current control state of the state machines, the current state of the input queues, etc. In consequence, the test preamble can rarely be obviated by a judicious choice of the initial state of a model to be used in test synthesis.

2.1.3.1 DISTINGUISHING OBJECTS OF THE SAME CLASS

Though an array-type solution is clearly necessary for dynamically-created objects, it would be convenient to be able to use "role-instance names" to distinguish links to objects of the same class that exist in the initial state. However, there is no means to do so in UML object diagrams.

If an object linked to different initially-present objects of the same class (with different initial values) needs to distinguish between them *before* receiving any invocation from them, it must first navigate to them to read a distinguishing attribute, or invoke a method which returns the value of a distinguishing attribute, and then store the value of `<role_name>.item(<number>)` in a way that associates it to the distinguishing value (N.B. `item` is the feature of the Eiffel array class that is used to access individual elements of an array).

In the case where an object linked to different initially-present objects of the same class needs to distinguish them *after* receiving an asynchronous invocation from them, a predefined constant, "self", containing the value of the identifier of the particular object, together with variables ranging over object identifiers in order to send this value as that of such a variable, would be useful. However, such a self-referencing constant does not exist in UML, only in OCL (in spite of this fact, the value "self" is used as a parameter of an invocation in the diagrams shown in Figs. 3-24 and 3-59 of the UML 1.4 standard!). Since the action language of the Umlaut simulator is a sublanguage of the Eiffel language, the Eiffel keyword `Current` can be used for this purpose. Such a predefined constant will also need to be part of the test objective language.

2.1.3.2 CURRENT PROTOTYPE: IMPLEMENTATION NOTES

The current implementation is incomplete in that there cannot be more than one link between any two given objects in the object diagram describing the initial state. If there are several associations between the corresponding classes, the implementation associates this unique link to one of them according to criteria which are not semantically meaningful. As a consequence, other links must be instantiated explicitly in action expressions, assigning the appropriate object to the action language variable with the appropriate role name in the other object (in both directions if the association is navigable in both directions).

2.1.4 Importing the XMI representation of the "simulatable" model

The only part of the model exchange via XMI that is fully implemented in the prototype is the passage of the UML specification from Objecteering to Umlaut.

A UML model is represented in XMI as an instance of the UML meta-model and the XMI file therefore refers to objects which are instances of UML meta-classes. The parsing of the XMI

representation of the UML model can only be done in one pass by explicitly storing an accumulating a list of pending additions to the model being constructed. The need for this list arises since model additions involving a given UML meta-model object may refer to a super-class of this object rather than its leaf class. Thus, though the first time a UML meta-model object appears in the XMI file it is given an identifier, its leaf class may not actually be known until later in the XMI file. For example, the trigger of a transition of a UML state-machine may be given as an object of type Event with identifier `id`, but only later in the file is the information provided as to what is the leaf type of `id`, i.e. what type of Event is being referred to, CallEvent, SendEvent, etc.

Another difficulty with the use of XMI is the variety of possible representations due to the evolution of the UML standard: UML 1.3, UML 1.4 etc. and of the XMI standard: XMI 1.0, XMI 1.1, XMI 1.2 etc.

2.1.4.1 CURRENT PROTOTYPE: IMPLEMENTATION NOTES

We have used the UML1.3 and XMI 1.0 output of Objecteering, in spite of the fact that XMI 1.0 leads to particularly verbose descriptions. The parser also serves as an adapter between UML 1.3 and the version of UML used by Umlaut. This latter version of UML could mostly be classified as a UML1.1++ or UML1.3–– except for the treatment of expressions and multiplicity which conforms neither to UML 1.1 nor to UML 1.3.

The XMI parser currently also contains an ad-hoc solution to a problem arising with the Umlaut metamodel representation. Though the links in the UML metamodel are navigable in both directions, in the XMI files, normally one direction is given, in the interests of space economy. In the Umlaut interface, the links between objects of a metamodel are reified so consistency is easily managed; it is easy to ensure that if a link is instantiated in one direction, the link in the other direction is also instantiated. However, in the Umlaut core used by the simulator, links between objects are not reified and the links in the two directions are not easily related. The reason for this lack of reification appears to be in order to permit an object to reference another object in the simulator action language by directly using the role name from the model.

A model produced via the Umlaut interface is consistent in this regard by construction. However, the same is not true of a model imported from an XMI file. Currently, then, this is resolved rather unsatisfactorily on a case-by-case basis in the XMI parser.

2.2 Transforming the UML specification

To ease the task of giving a formal semantics to a UML specification, it is automatically transformed into an equivalent one using a much simpler subset of UML, consisting mainly of classes and operations. Most of this transformation deals with UML state machines, merging them into UML's type system, using dynamic multiple classification (i.e. where the type of a given object can change dynamically).

A state machine comprises a set of states, an event queue, and a thread that dispatches events taken from the queue. The event queue and the thread are reified (i.e. represented explicitly as classes) in the transformed model. The emission of an asynchronous invocation by an active or passive object to an active object is transformed into the placing of an event corresponding to this invocation in the event queue of the receiving object. The reception of an asynchronous invocation by an active object from an active or a passive object is transformed into the removal of an event corresponding to this invocation from the event queue of the receiving

object and the execution of the action expression of the transition triggered by this invocation (i.e. the body of the corresponding method).

The states of the state machines are also reduced to the simpler concept of class: each state can indeed be seen as a specific subtype of the class to which the state machine is attached (the state machine's context). This subtype has the same interface (signature) as the context: this interface comprises one operation for each event that the state machine can react to. If a given state has an outgoing transition triggered by a given event, then the subtype corresponding to this (sub-)state implements an operation corresponding to that event with a method whose body contains the effect of the transition (that is, the set of UML actions to be executed when the transition is fired). Note that in UML state machines, inner transitions have priority over outer ones, which fits perfectly with this scheme: the priority policy directly maps to the classical dynamic binding of object-oriented languages. Fig. 6-3 illustrates the transformation, which is similar in nature to the state design pattern [GamHelJoh94]. Under this scheme, a transition between states of an objects state machine is transformed into a change from one type to another (UML supports dynamic and multiple classification so the transformed model is still a UML model).

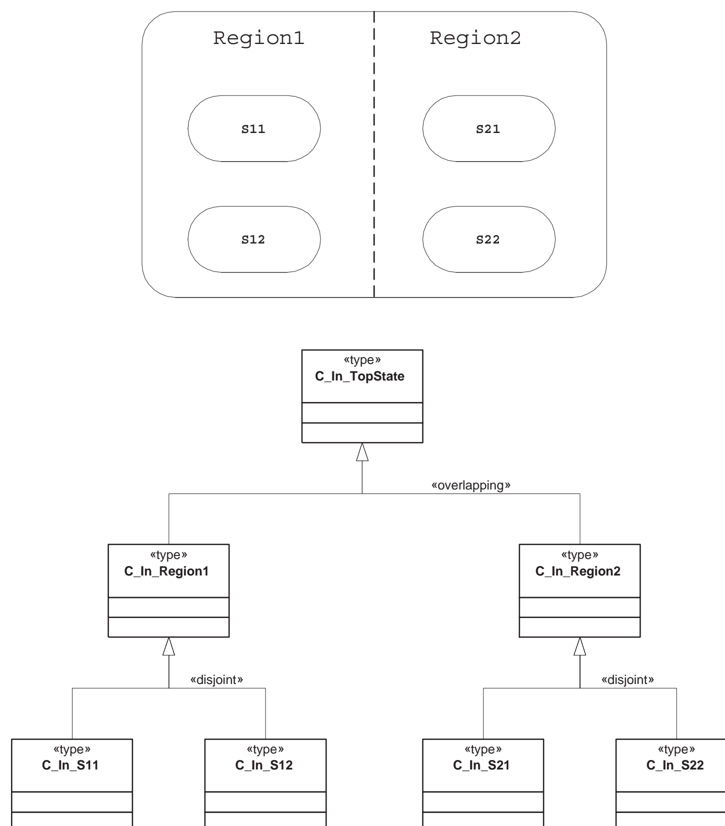


Figure 6-3: Illustration of the transformation of UML state machines.

Of course, the initial configuration described in the object or deployment diagram must also be transformed so as to include the objects representing communication queues and event dispatchers.

2.2.1.1 TRANSITIONS WITH NO TRIGGER

The state-pattern transformation procedure mentioned above does not deal with state-machine transitions having no trigger. There are only two cases to consider, since triggerless transitions

in passive objects are not allowed as explained in Section 2.1.2.5: guarded or unguarded transitions of active-object state machines. Spontaneous transitions (i.e. those with neither trigger nor guard) of active-object state machines are transformed into default operations of the appropriate class (there may be several). Such transitions are commonly found in the state machines of the external actors. Guarded transitions are similarly transformed but the guard must be evaluated each time the operation is chosen and the body only executed in the case where the guard evaluates to true.

2.2.1.2 NON-DETERMINISM DUE TO OVERLAPPING GUARDS

Another feature of this transformation is that it assumes that the state machines are deterministic in the sense that a set of outgoing transitions of a state sharing the same trigger (or a set of guarded outgoing transitions of a state without a trigger) must have mutually-exclusive guards. Based on this assumption, the guards of such a set of transitions are transformed into an if-then-else ladder. Of course, with any reasonably-expressive data language, mutual exclusivity of guards is undecidable, so this simplification is significant.

2.2.1.3 CURRENT PROTOTYPE: IMPLEMENTATION NOTES

The current implementation is incomplete in that guards in triggerless active-object transitions are not taken into account. They are thus always treated as spontaneous active-object transitions. See Section 2.3.4.1 for a workaround. In addition, hierarchical state machines are not yet dealt with so that all state machines must be flat.

2.3 Generating the simulation API

From the transformed specification, Umlaut generates a simulation API with which to construct a labelled transition system (LTS) defining the semantics of the UML specification. This API can then be used by TGV (or other tools) to construct all or part of this LTS as required, enabling on-the-fly treatment.

2.3.1 Structure of the simulation API

The simulation API provides functions for:

- the calculation of the initial global state,
- the calculation of the fireable transitions in a given global state,
- the calculation of the successor state, given a global state and a transition which is fireable in that state,
- the comparison of global states.

In a given global state, the set of fireable transitions are calculated by enumerating the control states of the active objects in the system (recall that after the transformations described in the previous section, a control state is represented as a type), and checking which state-machine transitions are enabled in these local control states, see Section 2.3.2 for more details. The global states are stored using a deep copy of the whole structure of objects.

The comparison function between global states enables cycles and confluences in the LTS to be detected. This comparison function relies on local state comparison functions for each class of objects.

Concerning the implementation language of the simulator, the simulation API is generated in an object-oriented programming language (Eiffel), firstly, to facilitate the handling of dynamic creation and destruction of objects and, secondly, to facilitate the implementation of dynamic reclassification.

2.3.1.1 CURRENT PROTOTYPE: IMPLEMENTATION NOTES

To reduce the size of the states of the state space, the local comparison functions could be redefined so as to abstract away from those properties of the system that we are not interested in. However, currently, no interface is provided for redefining them.

The current implementation of the state comparison function uses a notion of state which is too fine, distinguishing between global states which only differ in their object identifiers. As this distinction can easily lead to infinite branching in the part of the LTS generated from a loop involving object creation, a more sophisticated implementation is needed in which such isomorphic global states are not distinguished. What is required for this implementation is a notion of equality between object graphs.

2.3.2 Atomicity of derived LTS transitions

The execution of an LTS transition defines a simulation step. As we see below, in the current implementation of Umlaut, the granularity of the transitions in the generated LTS (and thus of the Σ-actions which can be used in test synthesis) is rather coarse: global states correspond to "stable" configurations of the UML system, that is, ones in which each object of the system is in a well-defined state of its associated UML state machine.

More precisely, the following gives all the configuration information used to define our notion of global state:

- the control state of the state machine of each of the objects
- the values of the attributes of each of the objects,
- the existing navigable links between objects,
- the state of the communication queues of the active objects.

Recall that after the transformations described in the previous section, the control state of the state machine of an object is represented by the dynamic type of that object. This definition of global state avoids the structure of the derived LTS (but not necessarily the labels on the transitions) being dependent on the action language which parameterises UML state machines.

It is further assumed that:

- calls to active objects are asynchronous and pass via FIFO queues (one queue per object)
- calls to passive objects can be synchronous or asynchronous but do not pass via queues

These suppositions are consistent with the style of distributed system modelling in which all non-local calls are implemented as asynchronous calls to active objects. Recall that we do not distinguish between asynchronous invocations and signals so there is a single event queue. This means that synchronous calls to active objects are not treated.

Recall also that the emission of an asynchronous invocation to an active object corresponds to the placing of an event in the event queue of that object and the reception of an asynchronous invocation by an active object corresponds to the removal of an event from the event queue of that object together with the execution of the corresponding action expression. With this in mind, we see that the above restrictions ensure that each LTS transition is associated to a run-

to-completion step of an active object state machine. As our state machines have no hierarchy, each LTS transition corresponds to one of the following:

- the execution of the action expression of an active-object state-machine transition triggered by the reception of an asynchronous invocation (from an active or a passive object); this action expression may contain assignments, creation of objects, invocations to other objects etc.,
- the execution of the action expression of an active-object state-machine transition with no trigger; in the general case, this action expression may also contain assignments, creation of objects, invocations to other objects etc.; if it contains the latter, it is thus the start of a causal chain or a set of causal chains,

In both cases, if the action expression contains invocations to passive objects, these may in turn contain nested invocations, so a single LTS transition may involve the execution of action expressions from many different objects. Any other UML-action must already be part of an LTS transition which necessarily commences with one of the above two types of events. Recall that an external actor is treated by Umlaut as an active object.

Recall that UML state machines are complete in the sense that they have a transition triggered by each of the operations of their class diagram. This completeness is due to the presence of implicit transitions realising the behaviour of UML state machines on unspecified reception, see Section 2.1.2.2. Implicit state machine transitions are treated in the same way as explicit state machine transitions in the generation of the simulation API by the Umlaut simulator.

The granularity of transitions of the LTS which can be built using the simulation API generated by the current Umlaut simulator, together with the suppositions concerning passive and active objects, have significant consequences on the style of UML model which can be used for test synthesis. For example, if the returned value of a call from an external actor to the model is needed for the test, this call cannot be modelled as a synchronous call and must be modelled as two asynchronous calls, as mentioned in Section 2.1.2.7.

2.3.3 The need for a closed specification

Recall that the system can receive stimulation from its environment and the external actors are modelled as active objects that can send messages to objects (either passive or active) residing in the system, see Section 2.1.2.1. A purely reactive system will not have any spontaneous behaviour if it is not sent any stimulus by the environment. To simulate such a system exhaustively, one must also simulate an environment able to send the system any acceptable input. Therefore, the system must be closed. In the UML models accepted by the Umlaut simulator, the system is closed by providing the external actors with active-object state machines that send input to the system and receive output from it.

As stated in Section 2.1.2.2, the UML state machine behaviour on unspecified reception means that transitions with an empty action expression need not be explicitly specified. This may often be the case for external actor state machines. However, for reasons of clarity, it is not necessarily a recommendable practice to omit these transitions.

A common way to reduce the state-explosion problem arising due to modelling concurrency through interleaving, reducing unwanted interleavings when performing exhaustive simulation or model-checking with a simulator such as the Umlaut simulator, is to assume that the environment is “reasonable”. A reasonable environment sends stimuli to the system only when it is unable to continue execution without further input. Ideally, the tool would allow a choice of whether or not the “reasonable” environment assumption is to be used, when generating the simulation API.

2.3.3.1 CURRENT PROTOTYPE: IMPLEMENTATION NOTES

The reasonable environment is not yet implemented in the prototype. However, the size of the queues used in the communication between the system and its environment (the actors) is parameterisable; in fact, the parameterisation applies to all active object queues, see Section 2.3.6. The implementation is such that an LTS transition in which an actor sends an invocation to a queue which is full will not be enabled. With this implementation, restricting the size of the communication queues between the system and the external actors limits the possible interleavings to some extent and, at least as far as actor-to-system invocations are concerned, cannot lead to deadlock if the guidelines for specifying actor state machines given in Section 2.1.2.7 are followed. However, it is not as useful as the reasonable environment[12].

2.3.4 Transitions with no trigger

Many transformations based on the state pattern do not take into account transitions which have no trigger. As already mentioned, there are only two cases to consider since passive-object triggerless transitions are prohibited (and if used, will cause the state-machine to block in the current implementation).

The simplest case is the unguarded, or spontaneous, transitions of active objects. If the current state of an active-object state machine has such an outgoing transition, a corresponding LTS transition is always generated. Such transitions commonly occur in the state machines of the external actors. In the case of guarded transitions, if the current state of an active-object state machine has such an outgoing transition, an LTS transition is generated if the guard evaluates to true.

2.3.4.1 CURRENT PROTOTYPE: IMPLEMENTATION NOTES

In the current prototype, guards on triggerless active-object transitions are not taken into account (they are always considered true). They are treated as spontaneous transitions and are thus enabled and can be executed even when their guard is not verified. This problem can sometime be avoided without the actor state machine becoming too cumbersome by using extra control states to represent the different values of the guards which can affect transitions.

2.3.5 Labels of the derived LTS transitions

The label of an LTS transition contains a representation of the UML actions which are executed on moving from the source “stable” configuration to the target “stable” configuration. The labelling function used by the Umlaut simulator is chosen to be as meaningful as possible since it not only dictates the labels on the transitions of the test cases which are the output of the test synthesis method, but also dictates the labels that must be used in the test objectives. The test synthesis needs to be able to match the labels on the transitions of the test objective LTS (which may include regular expressions) with those on the transitions of the system LTS.

The format of the labels generated by the Umlaut simulator is as follows:

1. *Firing of an active-object state-machine transition triggered by reception of an asynchronous invocation*:

   The LTS transition generated for

---

[12] In test synthesis, the TGV option `-outprior` has an effect which is similar to that of a reasonable environment

- the reception at the active object <receiver> of an asynchronous invocation, with parameter values <val$_1$>,…, <val$_n$>, of the operation <opname>,
- the execution of the action expression of the corresponding state machine transition,
- the consequent execution of the action expression of any other state-machine transitions contained in the same simulation step,

is labelled:

<receiver>?<opname>(<val$_1$>,… ,<val$_n$>)

Notice that, as is the usual case in OO systems, on reception of an asynchronous invocation, there is no knowledge of the identity of the sender. This point is of some importance, see Sections 2.4.1, 3.4.2, 4.3.1 and 5.4.2.

2. *Firing of an active-object state machine transition with no trigger*

The LTS transition generated for

- the execution of the action expression having the Umlaut action language textual description <action_expression_text> of the state-machine transition of the active object <sender>
- the consequent execution of the action expression of any other state-machine transitions contained in the same simulation step

is labelled:

<sender>!<action_expression_text>

The text of the action expression is copied as-is, so that any variables involved in it are not instantiated with their actual values. Whether or not the transition is guarded, the guard does not figure in the transition label (furthermore, as already stated in Section 2.3.4, in the current implementation, the guard is assumed to be true).

In particular, if the action expression comprises a single asynchronous invocation of the operation <opname> belonging to the object playing the role <receiver_role> (at the other end of a link), with parameters <param1>,…, <paramn>, the transition label will have the following simple syntax:

<sender>!<receiver_role>.<opname>(<param1>,… ,<paramn>)

If the guidelines concerning active-object state machines given in Section 2.1.2.6 are followed, all LTS transitions corresponding to triggerless transitions of active objects will be labelled by the second type of label. Furthermore, due to the fact that synchronous invocations from actors to the system are not implemented, the UML-action described by the label is the only one executed on that LTS transition. In test synthesis, this property of the labels of LTS transitions generated from triggerless transitions in actor state machines greatly facilitates the specification of test objectives, see Sections 3.2.2 and 3.2.3.

The grammar for the Aldebaran-format representation [FerGarKer96] of the test objective LTS, and of the test case IOLTS, used in system test synthesis with Umlaut/TGV – including the grammar for the transition labels – is given in Appendix A.

2.3.5.1 TREATMENT OF DYNAMICALLY CREATED OBJECTS

An internal representation is used for the names of dynamically-created objects (that is, those not present in the initial state) appearing in the labels of the derived LTS. This internal representation is based on a root name which is the name of the class of the object.

Currently, objects passed as parameters in invocations are not flattened, in the sense that the value of their attributes does not appear on the transition label. Consequently, in test

synthesis, the value of attributes of objects passed as parameters cannot be used in test objectives and will not appear in the labels of a synthesized test case.

2.3.5.2 CURRENT PROTOTYPE: IMPLEMENTATION NOTES

Thorough testing of the prototype on models involving dynamic creation of objects has revealed weaknesses which are of some interest, since the use of simulation techniques in the context of dynamic creation is a field which has not been explored to any great extent. One of these weaknesses, namely that concerning the implementation of the state comparison function, has already been mentioned in Section 2.3.1.1.

The main weakness of the current implementation in this regard concerns the naming of dynamically-created objects. The algorithm currently used leads to the name of one and the same dynamically-created object changing from global state to global state. In the case where none of the LTS transitions differ only in that the actions used to label them refer to truly different dynamically-created objects of the same class, this unintended feature simply creates unwanted non-determinism. In the case where two or more LTS transitions differ only in that the actions used to label them refer to truly different dynamically-created objects of the same class, the possibility of distinguishing the labels on these transitions is lost and the generated LTS is therefore incorrect.

When the simulator is used with the TGV tool, the unwanted non-determinism of the former case can be removed by using the renaming option of the tool, `-rename`, where the renaming options are usually specified in a file with suffix *.rename*. Each label which involves a dynamically-created object and which appears on a external transition must be renamed to a similar label by substituting a given string for the name of all dynamically-created objects of the same class.

2.3.6 Parameterisation of active-object queue sizes

In the Umlaut simulator, the size of the queues of the active objects can be parameterised. This is done by changing the value returned by the *message_queue_capacity* feature of the *active_state_machine* class[13]. The queue size affects the simulation in the following manner: if the execution of a transition is found to increase the size of any of the active-object queues beyond the specified limit, this transition is not enabled.

While this policy is clearly useful to restrict the interleaving and the size of the corresponding LTS, it should be realised that it may cause deadlocks. However, such deadlocks are not the result of a poorly-implemented queue-size restriction but only occur when the need for a larger queue size is an essential part of the specification. For example, if the queue size is set to one and the only transition that could be enabled contains two asynchronous invocations to the same active object, this transition will not be enabled (unless the maximum queue size is increased to two) and a deadlock will result.

2.3.7 Auto-invocation and attribute access

Auto-invocations by active objects are not treated like other invocations in the sense that they do not lead to an event being put in the event queue of the active object. Even though this means that the auto-invocation is dealt with as if it were synchronous, like other operations in active objects the auto-invoked operation must still be asynchronous (i.e. no return value) since it could also be invoked by other objects. Auto-invocations by passive objects are not

[13] Contained in the file $UMLAUT_ROOT/products/Simulator/rts/active_state_machine.e.

considered to generate state-machine events (otherwise, for synchronous invocations, run-to-completion semantics means that such a situation causes a deadlock).

If the guidelines for specifying actor state machines of Section 2.1.2.7 are followed, most actor auto-invocations trigger a transition whose action expression contains a single invocation of a system operation. Furthermore, the atomicity of the LTS generated by Umlaut means that, since there is no intermediate queue, a single LTS transition is generated for the auto-invocation together with the consequent system invocation. The naming convention given in the guidelines then ensures that the label of this LTS transition, generated according to the rules of Section 2.3.5, adequately describes both the auto-invocation and the consequent system operation invocation (since they share the same name).

Attribute access is rewritten using set and get features. An attribute set is an asynchronous operation while an attribute get is synchronous. Therefore, a policy which is coherent with the treatment of active-object auto-invocations is to only allow access to active object attributes by the objects themselves. It may be possible to extend this implementation allowing associated passive objects to access an active object's attributes.

2.3.7.1 CURRENT PROTOTYPE: IMPLEMENTATION NOTES

In the current implementation an attribute read, i.e. an attribute get, of an active object, even by the object itself, causes a simulation error or a deadlock in simulation. The explanation would seem to be that attribute gets, even those invoked by the object itself, are not treated synchronously and are simply put in the objects event queue.

A work-around for this problem is to move such attributes into associated passive objects of the system.

2.4 From LTS to IOLTS

The division of the actions of the alphabet of an LTS into external and internal, and the division of the external actions into inputs and outputs is what distinguishes an IOLTS from an LTS (the role of the function $\aleph$ in the definition of IOLTS of Section 1.2.2.2). When used in testing, tester outputs will correspond to the SUT inputs and tester inputs will correspond to the SUT outputs. The information concerning which actions are internal, and that concerning which of the external actions are inputs and which are outputs, is provided to the TGV tool via the command line options `-hide` and `-io`, respectively, conventionally in files with suffixes *.hide* and *.io* respectively

Defining the function $\aleph$ on the set of Σ-actions of the LTS generated by the Umlaut simulator is not completely obvious, firstly, due to the atomicity of the transitions of the LTS generated by Umlaut, see Section 2.3.2, and secondly, due to the need to relate the synchronous communication architecture required by TGV to the communication architecture assumed by Umlaut.

In the system testing case, these problems are dealt with jointly as follows:

- by imposing guidelines on the specification of the external actor state machines, see Section 2.1.2.
- by considering the communication queues of the external actors and of the system active objects that communicate with them to be inside the SUT boundary, see below.

With these assumptions, the information defining the function $\aleph$ can be derived automatically from the system specification.

2.4.1 The SUT boundary and the restriction to system testing

In component testing, apart from the Component Under Test (CUT), the SUT may include other components that are considered to be correct; if there are no such components, the CUT and the SUT coincide. Though the test synthesis algorithms only use the notion of SUT, this does not mean that the test synthesis assumes the SUT and CUT coincide. The CUT will play a role in the choice of test objectives and in the interpretation of test verdicts, and must be the only component of the SUT which is not presupposed to be correct.

Given a specification of a system, ideally, we would like to be able to choose the boundaries of the SUT to be those of a given component of any size within the specified system. However, the labelling of the LTS would need to be based on a semantics which is more discriminating than that of a usual OO programming language, in order for an active object receive event to contain information about the sending instance.

2.4.1.1 RESTRICTION TO SYSTEM TESTING DUE TO CURRENT UMLAUT IMPLEMENTATION

With regard to the labelling function in the Umlaut simulator (see Section 2.3.5), the semantics used is no finer than that of the language used to implement it, i.e. Eiffel. Thus, using the LTS generated by the Umlaut simulator, it is not always possible to know whether a receive event on a component outside the black box is that of a message sent from inside the black box or from outside it.

In the case where the SUT is chosen to be the whole specified system, any reception by external actors of an invocation sent by another object must have been sent from inside the system, since UML does not allow any communication between external actors. Thus, with the current labelling function (and in absence of a component model allowing the description of environmental dependencies and corresponding connections between ports in UML 1.4), Umlaut/TGV is only suitable for system testing. We will therefore use the terms system and SUT interchangeably in the rest of this section.

Currently, then, synthesis of test cases for an SUT which is only a part of the specified system must be done by creating a special-purpose test specification in which this SUT is the entire system specified, that is, a specification composed of the part of the original system one wishes to test and external actors (including their state machines) interacting with it, so as to ensure that communication between entities external to the SUT is not modelled.

2.4.1.2 RESTRICTION TO SYSTEM TESTING DUE TO CURRENT TGV IMPLEMENTATION

The above paragraphs deal with the problems with trying to use an LTS generated by the current implementation of the Umlaut simulator for test synthesis of general component test cases rather than system test cases. However, even if the required information was passed on by the Umlaut simulator to the test synthesis tool, this tool would also need to be modified in order to deal with this information. The definition of IOLTS used does not contemplate actions which are internal to the SUT environment.

As stated above, restricting to system testing ensures that there are no communications between components of this environment, i.e. the actors. In a test case, these communications would be test coordination messages between the components of the tester. This restriction, imposed due to the current Umlaut simulator implementation is then almost enough to ensure also that test synthesis can be performed satisfactorily with TGV using the simulation API generated by Umlaut. However, auto-invocations on SUT environment components are still possible, even in the system testing case. Indeed, the guidelines of Section 2.1.2.7 practically

ensure that they will be used. Inevitably, then, the current treatment of auto-invocations by TGV is not satisfactory; see the next section for details.

2.4.2 Definition of the SUT boundary in test synthesis with Umlaut/TGV

In a test case, the tester plays the role of the SUT environment. In system testing, the SUT is the entire system and the SUT environment is constituted by the external actors.

In the semantic model underlying the test synthesis algorithms, communication between the tester and the system is synchronous in the sense of CCS [Mil89], CSP [Hoa85], etc., i.e. communication actions are shared by the two communicating parties; in fact, an input of one of the parties is executed jointly with an output of the other party. The communications between the SUT and its environment are synchronous actions in this sense, and define the boundary of the black-box for the system test synthesis. In fact, for the TGV tool, the black-box used for the testing is defined by simply listing either the external actions or the internal actions. As already stated, TGV assumes that all external actions are on the system boundary, i.e. that they are communication actions which are shared between the SUT and its environment[14].

Due to the atomicity of the LTS transitions generated by Umlaut, ensuring that the labels on the transitions of this LTS are of the required form for test synthesis is not immediately obvious, even for the drastically reduced set of admissible test objectives discussed in Section 3.2.2. In the rest of this section, we investigate the compatibility of the Umlaut-generated labels with the test synthesis method.

In the semantics used by the Umlaut simulator, the invocation of an operation of a system environment object (an actor) by an object of the system is performed asynchronously via a FIFO queue (one queue per object). Similarly for the invocation of an operation of an active object of the system by the system environment. See Section 2.3 for more details. These queues can be thought of as modelling the communication medium. A notion *effective SUT* for IOLTS-based test synthesis can be defined as being composed of the SUT together with this communication medium.

2.4.2.1 Σ-ACTIONS ON THE SUT BOUNDARY: SUT OUTPUTS

In a usual synchronous-communication model (in the CCS sense), an asynchronous invocation passing via a queue such as that from an active system object O to an external actor A would be composed of two atomic actions, one labelling the emission event (O placing the event in A's event queue) and one labelling the reception event (A removing an event from its event queue). However, in the Umlaut simulator semantics, though one LTS transition is generated for the reception event, the emission event may be part of an LTS transition involving many other events, occurring on many different objects.

This does not create any difficulties in defining the SUT boundary, however, if we treat the external actor queue as being inside this boundary. That is, suppose an event in the input queue of an external actor corresponds to an asynchronous invocation of an operation of that actor by a system object. We interpret the removal of this event from the input queue of that actor as an action shared by the SUT and its environment that the SUT views as an output.

---

[14] With the rules defined here, this leads to auto-invocations of actor state-machines being treated as system inputs!

2.4.2.2 Σ-ACTIONS ON THE SUT BOUNDARY: SUT INPUTS

In a usual synchronous-communication model (in the CCS sense), an asynchronous invocation passing via a queue such as that from an external actor A to an active system object O is modelled as two atomic actions: one labelling the emission event (A placing the event in O's event queue) and one labelling the reception event (O removing the event from its event queue). In the Umlaut simulator semantics, one LTS transition is generated for each of these two events.

To define the SUT boundary for the test synthesis, we simply treat the active system object queue as being inside this boundary. That is, suppose an event in the input queue of an active object of the system corresponds to an asynchronous invocation of an operation of that object by an external actor. We interpret the placing of this event in the input queue of that active object of the system as an action shared by the SUT and its environment that the SUT views as an input.

In the Umlaut simulator semantics, a synchronous (in the OO sense) or asynchronous invocation from an external actor to a passive system object gives rise to a single LTS transition, since there is no intermediate queue. The label of this LTS transition can be directly considered to be the action shared by the SUT and its environment that the SUT views as an input. As regards communications in the reverse direction (i.e. from the system to an external actor) synchronous (in the OO sense) invocations to external actors are not allowed since actors are active objects and, due to the atomicity of transitions, synchronous (in the OO sense) return messages are never visible. Such communications do not therefore need to be considered here.

2.4.2.3 IMPORTANCE OF THE MODELLING GUIDELINES FOR ACTOR STATE-MACHINES

Regarding actor-to-system invocations, we have assumed here that each such invocation is contained in an action expression of a separate triggerless transition or a transition invoked by an auto-invocation (thus also assuming that sytem-to-actor invocations never trigger actor-to-system invocations). This will be the case if the guidelines for specifying actor state machines of Section 2.1.2.7 are followed. These guidelines are given in order to generate transition systems which are amenable to analysis and optimal for test synthesis.

If these guidelines are not followed and multiple actor-to-system invocations are performed in the action expression of a triggerless transition, the corresponding SUT-input action comprises the placing of events in the input queues of *several* active objects of the system, and possibly also the invocation of operations of *several* passive objects of the system. The situation is even worse if an actor-to-system invocation is performed in the action expression of a transition triggered by a system-to-actor invocation. In this case, a single LTS transition labelled as an SUT-output action will include events which we would like to label as SUT-input actions and the usefulness of the test synthesis becomes questionable.

With the atomicity of the LTS generated by Umlaut and the rules given below for defining the external actions and for dividing them into inputs and outputs, LTS transitions labelled by actor auto-invocations are considered to be system inputs. Since the body of the auto-invoked operation will usually contain an invocation to the system, this is as required. Occasionally, however, the body of the operation will not contain an invocation to the system (e.g. object creation and initialisation). Such cases are not contemplated by the current test synthesis tool and according to the rules defined below will incorrectly be considered system inputs.

2.4.2.4 SYNTAX OF THE *.HIDE* FILE FOR SYSTEM TEST SYNTHESIS USING UMLAUT/TGV

The syntax for the *.hide* file required by TGV for an LTS derived from a UML specification with external actors `<actor1>`,… , `<actorn>` by the Umlaut simulator has the following form:

```
hide all but
<actor1>?.*
<actor1>!.*
…
<actorn>?.*
<actorn>!.*
```

A file called `<spec_name>.hide` with this syntax is generated from the UML specification contained in the file `<spec_name>.uml` by the Umlaut simulator.

2.4.2.5 CURRENT PROTOTYPE: IMPLEMENTATION NOTES

As stated in Section 2.1.2.7, the possibility of external actors synchronously (in the OO sense) invoking operations of the SUT is not implemented in the prototype so all communications between external actors and the SUT (that is, in both directions) are asynchronous via FIFO queues, eliminating one of the types of SUT input discussed above

2.4.3 Partitioning the external actions in test synthesis with Umlaut/TGV

The external actions (= those on the SUT boundary, with the current implementation) must now be divided into inputs and outputs.

The transitions defined as SUT-input actions for TGV (to be viewed as tester-output actions) are those derived from the *emissions* of the external actors. In the case where the target system object is an active object, from the point of view of the effective SUT – which includes the event queue of the target active object – an actor emission is a reception from the external actor by the target object's event queue. If the guidelines of Section 2.1.2.7 concerning actor state-machines are followed, for each actor emission, there will be a corresponding LTS transition labelled by this emission, since such an LTS transition is derived from each external actor state-machine transition without a trigger.

The transitions defined as SUT-output actions for TGV (to be viewed as tester-input actions) are those derived from the *receptions* of the external actors. From the point of view of the effective SUT – which includes the event queues of the actors – such an actor reception is an emission by the external actor event queue to the external actor. An LTS transition labelled by such a reception action is derived from each external actor state-machine transition with a trigger (discounting auto-invoked transitions).

2.4.3.1 SYNTAX OF THE *.IO* FILE FOR SYSTEM TEST SYNTHESIS USING UMLAUT/TGV

Therefore, the following generic syntax can be used for the *.io* file required by TGV for any LTS derived from a UML specification by Umlaut:

```
input
.*!.*
```

The syntax of the `<spec_name>.io` file generated by the UML simulator from the UML specification contained in file `<spec_name>.uml` is more verbose but equivalent:

```
input
<actor1>!.*
…
```

```
<actorn>!.*
```

2.5 "Umlaut-simulatable" model for the ATC application

The class diagram of the ATC application is shown in Fig. 6-4. The system consists of four classes: *Flight* and *FlightPlan*, used to store flight data, *FlightPlanManager*, and *ControllerWorkingPosition*, which control the system and interact with the environment (the thick borders on the class diagram indicate that the latter two classes are active, that is, objects of these classes have their own flow of control).

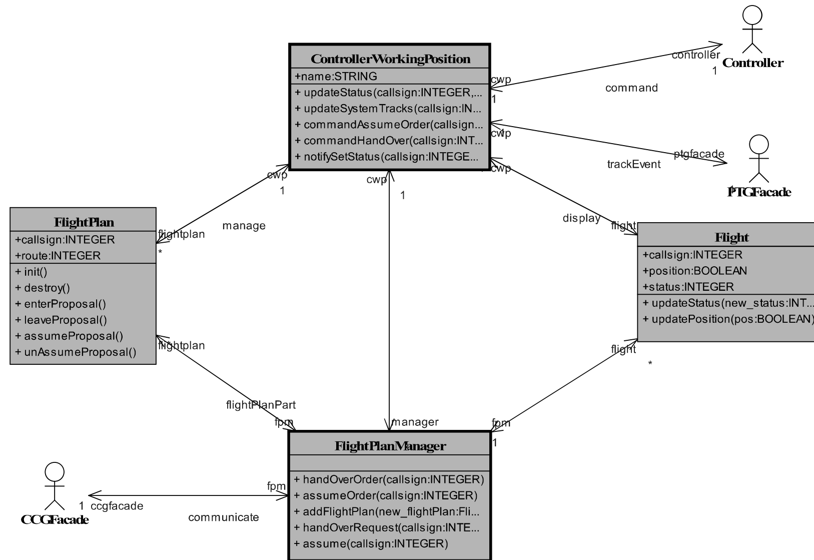


Figure 6-4: The class diagram of the ATC application example.

In the ATC application, the initial configuration shows that the deployed system knows of two flights (each with an associated flight plan), initially located out of the area managed by the ATC, see Fig. 6-5. An example of a state machine from the ATC application – that of the (passive) class *FlightPlan* – is shown in Fig. 6-6.

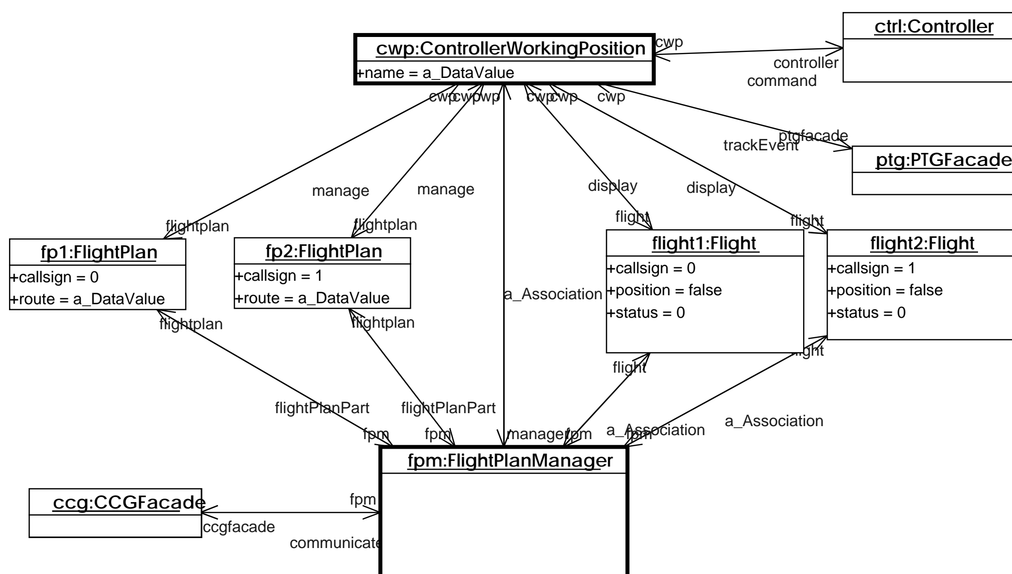


Figure 6-5: The object diagram of the ATC application example.

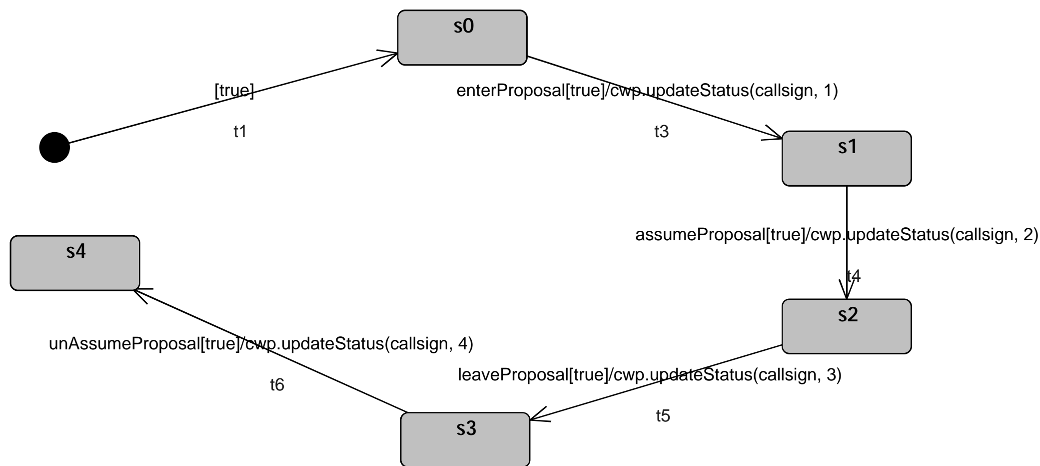


Figure 6-6: The state machine of the *FlightPlan* class of the ATC application example.

2.6 Formal specification derivation: possible enhancements

2.6.1 Atomicity of derived LTS transitions

The most obvious possible enhancement to the simulation API concerns the atomicity of the transitions. By taking into account the stack of nested calls in the notion of global state, it would be possible to define a finer granularity of transitions, in which an LTS transition is generated for each inter-object communication event (perhaps a send and receive event to a passive object could be combined in the same transition). This would greatly improve all uses of the Umlaut simulator (interactive simulation, test synthesis, etc.). With the current implementation, it would imply making the Eiffel call stack explicit.

However, a finer granularity of transitions would seem to oblige the use of a notion of global state which is "unstable", i.e. in which some objects of the system are between control states of their associated state machines. One way of avoiding this would be to generate a transition system of macro-transitions, like those of the Umlaut simulator, which are composed of micro-transitions, one for each inter-object communication event. The atomicity of executed transitions can be defined in terms of the macro-transition notion (so the system is never in an "unstable" state), while the visibility (and therefore the use in test objectives and the synthesised test cases) can be that of the micro-transition. An approach similar to this is used in the Agedis project [Agedis].

2.6.2 Component testing and SUT-environment internal actions

The labels on an LTS transitions corresponding to the firing of an active-object transition triggered by the reception of an asynchronous invocation do not contain information about the identity of the sender. If a finer semantics was used in the Umlaut simulator than that of the language used to implement it, i.e. Eiffel, the labels could contain such information. If coupled with an extension of TGV allowing it to treat external Σ-actions which are neither SUT inputs nor SUT outputs, it would enable the test synthesis method used here to be used for general component testing and not just system testing, as mentioned in Section 2.4.1.

Test synthesis for general component testing would also need to take into account synchronous invocations from the SUT environment to the SUT and from the SUT to the

SUT environment. As a first step, the current implementation could be completed to allow actors to send synchronous invocations to the system and then extended to allow passive-object actors that can receive synchronous invocations from it.

The definition of SUT boundary would be derived in function of the chosen component. Furthermore, the enhancement to the atomicity of the transitions proposed in Section 2.6.1 would make the present enhancement considerably more workable.

2.6.3 Storage of global states

The storage of global states, currently done using deep copies, could also be improved. The UML-actions executed when a transition is fired usually have a very localised impact on the system (e.g. changing the local state of a single object). While local states and queue contents are quite volatile, in general the structure of object inter-connections evolves very slowly. In many cases then, a significantly reduced footprint could be achieved by factoring out the "stable" information from successive configurations. However, the calculation of the "stable" information can be difficult without restrictions on included code, such as the Eiffel code in the action expressions in the current Umlaut implementation.

2.6.4 Dynamically-created objects

The flattening, in transition labels, of dynamically-created objects passed as parameters would enhance the usability of the generated LTS as mentioned in Section 2.3.5.

3 Derivation of LTS from scenario-based test objective

In this section we briefly mention some important points concerning the O-TeLa language and then show how an LTS representation of a test objective, as used by the IOLTS-based test synthesis, is derived from a scenario-based representation in the form of O-TeLa scenario structures. Fig. 6-7 illustrates the part of the method that is presented in this section. We then show how the formal objective derivation proceeds for the ATC application example before giving some possible enhancements to this part of the method.

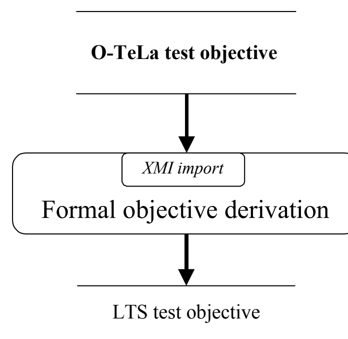


Figure 6-7: Formal objective derivation part of the method.

Our work has mainly been concerned with the test description language TeLa. The work on a more general approach to O-TeLa test objectives than that implemented in the prototype of the COTE project [JarPic01] is therefore less advanced than the equivalent work on TeLa. For this reason also, in this section, when dealing with aspects which are common to TeLa and O-TeLa, we refer to Section 5.

3.1 The O-TeLa language

As stated in Section 1.2.3.5, in the general case, scenario-based test objectives are described by two sets of scenario structures: *accept* scenario structures and *reject* scenario structures. The scenarios making up the test case scenario structure are those that are in correspondence with each of the *accept* scenario structures and none of the *reject* scenario structures of the test objective. Both types of scenario structures can be specified using a notation similar to TeLa [PicJarHeu01], denoted O-TeLa. In the general case, like TeLa, O-TeLa is based on UML sequence diagrams but its semantics is given in terms of partial orders of events rather than of messages, and constructs are added to give a greater expressive power.

In test synthesis, the labels on the arrows in O-TeLa sequence diagrams are to be matched with the labels on the transitions of the LTS generated from the UML specification. This is somewhat delicate, particularly in the case of dynamically-created objects, i.e. those that do not exist in the initial state. The use of the O-TeLa language is therefore heavily influenced by the structure of the LTS generated by the Umlaut simulator.

3.1.1 Constructs of O-TeLa used to represent LTS test objectives

Our work has mainly been concerned with the test description language TeLa. For this reason, in describing the use of the O-TeLa constructs, here do so by analogy with TeLa.

We can allow different O-TeLa representations of the same test objective. The allowable representations are similar to those used for the TeLa representation of IOLTS discussed in Section 5, substituting "SUT environment" for "tester", except that in the general case, O-TeLa diagrams show not only SUT-environment structure but also SUT structure. Unlike for the TeLa representation of IOLTSs shown in Fig. 6-18, O-TeLa diagrams involving multiple SUT lifelines and/or multiple SUT environment lifelines can use the usual partial-order of events semantics. However, in the translation of the scenario-based test objectives to LTS, the distinction between choice and concurrency is lost.

3.1.2 Current prototype: implementation notes

As stated in the definition of scenario-based test objectives of Section 2.2.4, the O-TeLa language of the prototype uses only a single *accept* scenario which may be annotated with *reject* messages. This simplified syntax and semantics of the O-TeLa language is that which was implemented in the COTE project [JarPic01] along with the translation to LTSs of scenario structures described in it.

3.2 From O-TeLa expression to LTS

3.2.1 O-TeLa sequence diagram lifelines

As in TeLa (see Section 5.2.1), lifelines represent components in a hierarchical component model. The hierarchy always contains at least two levels, namely, that defined by the objects (the lowest level components) and that defined by the two components: SUT and SUT environment.

3.2.2 Atomicity in O-TeLa scenarios

Calculation of the synchronous product in the test synthesis proceeds by matching the labels of the transitions of the LTS representing the test objective with those of the LTS representing the UML specification. The atomicity of the transitions in the LTS generated by Umlaut, see Section 2.3.2, and the labels used for these transitions, see Section 2.3.5, are thus of crucial importance for the test objectives. The LTS which is the target of the translation from the scenario-based test objectives must have the same atomicity and labelling scheme as the LTS generated by the Umlaut simulator.

Recall that in the general presentation of the method, we wish to parameterise the test synthesis with the SUT boundary in order to be able to represent SUT-internal actions in test objectives.

In the general case, if we are to take advantage of the full flexibility of TGV and parameterise the test synthesis with the SUT boundary, SUT-internal actions must also have the same atomicity in the two LTSs. However, as explained in Section 2.3.2, in the general case, one transition of the UML specification LTS subsumes many UML-actions, whereas the atomicity of the Π-actions of the test objective scenario structures is close to that of the UML-actions. Therefore, the need to have the same atomicity between the two LTSs necessarily implies many-to-one relations between the Π-actions of the scenario-based test objective and the Σ-actions of the corresponding LTS test objective.

3.2.2.1 SIMPLIFIED TEST OBJECTIVE LTSS

Though it would be possible to use scenario-based test objectives with such a complex translation scheme, it would certainly not be practical since such use would require a thorough knowledge of the UML specification, a very good understanding of the atomicity assumptions of the Umlaut simulator, and special syntactic constructions in the O-TeLa sequence diagrams (perhaps based on focus bars). However, if we restrict to test objective LTSs which contain only transitions labelled by external actor emissions and receptions, the guidelines concerning external-actor state machines of Section 2.1.2.7 (and the fact that synchronous invocations by external actors are not implemented) can, in most cases (but see Section 3.2.4), ensure a one-to-one relationship between Π-actions of the test objective scenario structures, Σ-actions of the corresponding test objective LTS and Σ-actions of the UML specification LTS.

Consequently, despite the flexibility which the test synthesis method allows with regard to representing SUT-internal actions in test objectives, we drastically simplify the test synthesis by assuming that test objective LTSs only contain transitions labelled by external actor emissions and receptions, these being the external Σ-actions defined for system test synthesis in Section 2.4. We will then generate one LTS transition for each Π-action describing an emission from an external actor of an invocation to the SUT and one LTS transition for each Π-action describing the reception by an external actor of an invocation from the SUT. That is to say that only the events on SUT-environment lifelines will give rise to transitions in the test objective LTS.

3.2.2.2 CONFIRMING THE EXACT SYNTAX OF THE UML SPECIFICATION Σ-ACTIONS

When defining test objectives, it may be helpful to use the other CADP tools [FerGarKer96] to interactively explore the LTS of the UML specification (using the *xsimulator* or *ocis* tools), or to exhaustively construct this LTS (using the *generator* tool). In particular, this technique can be used to confirm the exact syntax of the transition labels.

3.2.3 The SUT boundary in O-TeLa scenarios

After the simplifications of Section 3.2.2, LTS transitions need only be derived from the events which are to be translated to transitions labelled by labelled by Σ-actions which are visible in the synthesized tests, i.e. external Σ-actions.

The SUT boundary for use with TGV/Umlaut is defined in Section 2.4.1. Concerning asynchronous invocations between the SUT and its environment, clearly, the events that should be translated to transitions labelled by SUT boundary Σ-actions involving such invocations are those on SUT environment lifelines that are receptions from, or emissions to, the SUT. In the current implementation, either there is a single SUT environment lifeline or there is a lifeline for each external actor.

If we also wish to complete the current implementation and take into account synchronous invocations from the SUT environment to the SUT, even if we suppose that the corresponding send and receive event are indivisible in the sense that no other event can occur between them (c.f. the semantics of "interworkings"), we would inevitably lose the one-to-one correspondence between scenario Π-actions and LTS Σ-actions. However, in the interests of compatibility with the asynchronous invocation case, we could choose to view the event on the SUT environment lifeline as being the SUT boundary event. The label on the corresponding LTS transition would then derived from this Π-action.

In summary, the external Π-actions are those that label events occurring on SUT-environment lifelines. The Π-action SUT boundary is the set of Π-actions labelling events (send and receive) which occur on SUT environment lifelines and which are part of messages exchanged with the SUT. In fact, lifting the assumptions concerning external Σ-actions from the LTS level to the scenario level, we have that the current test synthesis method supposes that all Π-actions on the Π-action SUT boundary are communications with the SUT, see Section 3.2.4.

3.2.4 SUT environment auto-invocations in O-TeLa scenarios

Even in system testing, as has already been made clear, the assumption at the end of the last section does not hold. Furthermore, as stated in Section 2.4.1, the guidelines for specifying actor state machines of Section 2.1.2.7 practically guarantee that we will have to deal with such actions, since SUT environment (in system testing, actor) auto-invocations are external Π-actions which are not on the SUT-boundary.

According to the definition of the SUT boundary and the division of the Σ-actions on this boundary into inputs and outputs, given in Sections 2.4.2 and 2.4.3, actor auto-invocations will be classified as SUT inputs. As mentioned in Section 2.4.2, if the guidelines for specifying actor state machines of Section 2.1.2.7 are followed, most auto-invocations trigger a transition whose action expression indeed contains a single invocation of an SUT operation, i.e. an SUT input. Furthermore, the atomicity of the LTS generated by Umlaut means that the auto-invocation and this system input will give rise to a single LTS transition. The naming convention given in the guidelines then ensures that the label on this LTS transition describes both the auto-invocation and the consequent system invocation since they share the same name.

Describing the auto-invocation itself in test objective scenarios, or describing any processing the auto-invoked operation performs apart from the invocation of the SUT operation, should be avoided where possible. However, this may be impossible to avoid in the case where the action expression of the transition triggered by the auto-invocation contains no invocations itself, but is, nevertheless, important to the test objective. The label of the LTS transition derived from such an explicit auto-invocation using the name of the actor and the name of the actor operation auto-invoked must be that derived by the Umlaut simulator for this LTS transition, as defined in Section 2.3.5.

In the usual case, the only aspect of the auto-invoked operation of interest to the test objective is the SUT invocation contained in its body. We would therefore like to represent only this SUT invocation rather than the auto-invocation which triggers it. As usual, the SUT invocation is represented in the test objective scenario as a message from the SUT environment to the SUT. The naming convention of the guidelines means that we can easily derive the label of the LTS transition from the name of the actor and the name of the system operation invoked (i.e. from the name of the send event) so that it coincides with the one derived by the Umlaut simulator for this LTS transition, as defined in Section 2.3.5.

3.2.5 Deriving an LTS from a set of scenario structures

3.2.5.1 General scheme for the scenario structures to LTS translation

First we derive an LTS from each scenario structure making up the test objective. From the:

accept scenario structures $\{seq_i^+\}_{i \in I}$ we derive LTSs $\{S_i^+\}_{i \in I}$
reject scenario structures $\{seq_j^-\}_{j \in J}$ we derive LTSs $\{S_j^-\}_{j \in J}$

where these LTSs are completed w.r.t. Σ, the alphabet of the application model, by loops in each state.

We then derive a single LTS, *TO*, which is the LTS of the test objective. *TO* is defined as follows:

$$TO = \cap_{i \in I} S_i^+ \cap \neg \cup_{j \in J} S_j^-$$

where negation denotes complement. The accepting states of *TO* are the so-called *accept* states and it is again completed w.r.t. Σ, but this time by transitions to the so-called *reject* states, rather than by loops.

3.2.5.2 DERIVING AN LTS FROM A SINGLE SCENARIO STRUCTURE

Deriving an LTS requires restrictions on the use of the scenario language. How much these restrictions can be loosened is still the subject of theoretical investigation. The compilation towards finite state automata raises several questions. In the intimately-related MSC case, from a theoretical point of view, it is known that we must restrict to bounded HMSCs [AluYan99], those in which loops without synchronisation are forbidden. To describe our test objectives, then, we need to define a sub-class of the O-TeLa language for which an efficient translation to finite state automata can be defined but which is, at the same time, sufficiently expressive.

As mentioned in Section 3.2.2, we only wish to derive LTS transitions from the events on SUT environment lifelines. However, in the case where several lifelines are used for the SUT environment, ordering relations may be imposed on the events on these lifelines by message exchanges with the SUT. That is, the required LTS is obtained (at least in principle) in two stages: derivation of an LTS taking into account SUT lifelines followed by projection of the resulting LTS onto the SUT environment lifelines. This is analogous to the way the TeLa semantics is obtained by projection onto the tester lifelines.

3.2.6 Component names in arrow labels

The Σ-actions labelling the test objective LTS transitions must conform to those of the specification LTS generated by Umlaut, see Section 2.3.5. After dealing with the atomicity considerations and those concerning the SUT boundary, the remaining issue to be dealt with in the translation of Π-actions labelling test objective scenario events to Σ-actions labelling test objective LTS transitions concerns obtaining component names (in the current implementation, these are object names) as part of the Π-action names. In the current implementation, there are only two levels of components: that defined by the objects and that defined by the two components: SUT and the SUT environment. Here we define the rules for this two-level case.

First suppose that only two lifelines are used in the test objective scenarios, one representing the SUT and the other representing the SUT environment. The label on an arrow representing an invocation emitted by the SUT environment and received by the SUT is prefixed by the emitting actor name and by the receiving SUT object name (separated by the "!" character). The label on an arrow representing an invocation emitted by the SUT and received by the SUT environment is prefixed by the receiving actor name.

Next suppose that only one lifeline is used for the SUT but multiple lifelines are used for the SUT environment, one for each actor. The label on an arrow representing an invocation emitted by the SUT environment and received by the SUT is prefixed by the emitting actor name. The label on an arrow representing an invocation emitted by the SUT and received by the SUT environment is not prefixed by a component name.

Now suppose that only one lifeline is used for the SUT environment but multiple lifelines are used for the SUT, one for each SUT object. The label on an arrow representing an invocation emitted by the SUT environment and received by the SUT is prefixed by the emitting actor name. The label on an arrow representing an invocation emitted by the SUT and received by the SUT environment is prefixed by the receiving actor name. However, with the current implementation, the emitting SUT object name will not appear in the corresponding Σ-action, see Section 2.4.1.

Finally suppose that multiple lifelines are used for the SUT and for the SUT environment, one for each object and one for each actor. In this case, none of the arrow labels is prefixed by a component name. However, in the case of an invocation emitted by the SUT and received by the SUT environment, with the current implementation, the emitting SUT object name will not appear in the corresponding Σ-action, see Section 2.4.1.

3.3 A test objective for the ATC application

In the ATC example, suppose we want to generate a test checking that flight number 0 will be correctly assumed (i.e. that the controller *ctrl* will be notified that the flight status is updated with the value 4) when the flight enters the ATC zone (i.e. when the radar *ptg* updates the flight position to `true` meaning "in zone"). This is described by the *accept* scenario structure, consisting of a single scenario, on the l.h.s. of Fig. 6-6. To help to produce a test case, we also specify that we are not interested in any of the events concerning flight number 1. This is described by the *reject* scenario structure, consisting of a single scenario, on the r.h.s. of Fig. 6-8.

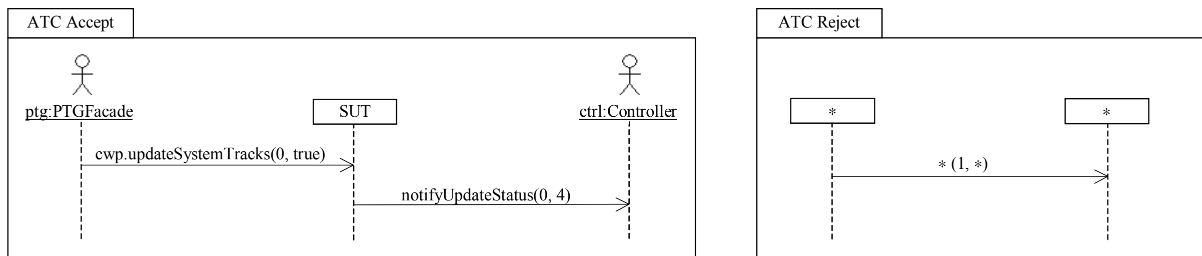


Figure 6-8: A test objective for the ATC example (general case).

From these scenarios we then derive the two LTSs shown in Fig. 6-9, where the loop transitions completing the LTSs w.r.t. the rest of the alphabet of the model in states other than the accept state are left implicit.

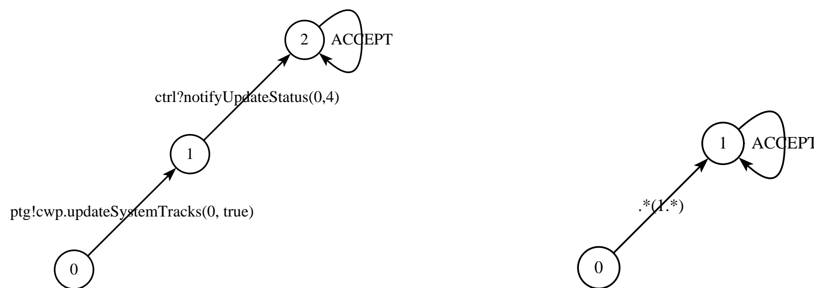


Figure 6-9: Accept and reject LTSs derived from the test objective scenarios of Fig. 6-8.

The derivation of the reject LTS merits some clarification. As for the accept LTS, we only derive one transition, rather than two, from a message sent to the SUT from its environment (resp. received from the SUT by its environment), this transition being labelled by the sending action (resp. receiving action). However, we need only derive one transition in the case of any other message for which the tester can be the sender, such as the message sent to "any" object shown here. This is since the sending action of such a message is sufficient for use in pruning the graph.

By taking the intersection of the accept LTS with the complement of the reject LTS and completing w.r.t. transitions to a reject state, we derive the following LTS for the test objective:

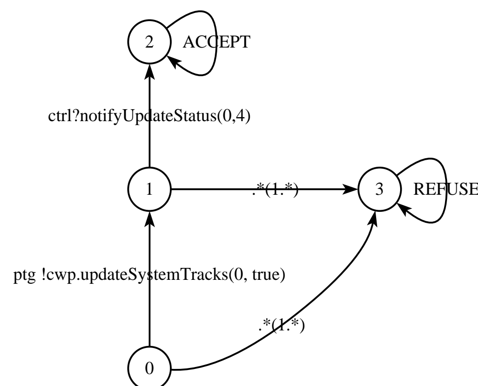


Figure 6-10: LTS representing the test objective for the ATC example shown in Fig. 6-8.

The loop transitions for the rest of the alphabet of the model in states other than the accept or reject state are again left implicit. Note that, while the LTSs accepted by TGV use regular expressions, the scenario structures are restricted to the more user-friendly wildcards in object names, operation names, and operation parameter names (though this may require more sequence diagrams than the use of regular expressions).

Recall that since states contain no information in an LTS, a transition to an accept or reject state is actually modelled as a transition to a state in which the only possible transition is accept or reject, respectively. Note also that the current syntax for the reject states uses a transition named "REFUSE", rather than a transition named "REJECT".

The Aldebaran format [FerGarKer96] for this test objective is as follows:

```
des(0, 6, 4)
(0,"ptg!cwp.updateSystemTracks(0, true)",1)
(0,".*(1.*)", 3)
(1, "ctrl?notifyUpdateStatus(0,4)", 2)
(1, ".*(1.*)", 3)
(2, "ACCEPT", 2)
(3, "REFUSE", 3)
```

The sending of the `updateSystemTracks` invocation by the `ptg` actor will be considered an input of the system and the reception of the `notifyUpdateStatus` by the `ctrl` actor will be considered an output of the system, see Section 2.4.2. We do not derive transitions for the consumption of the `updateSystemTracks` invocation by the SUT, nor for the emission of the `notifyUpdateStatus` invocation by the SUT (in fact, in the latter case, the current atomicity of the Umlaut simulator means that it would be problematic to do so, see Section 3.2.2).

3.4 Formal objective derivation: possible enhancements

3.4.1 Atomicity in O-TeLa test objectives

The enhancements proposed in Section 2.6.1 would allow greater flexibility in the test objectives. A change in the atomicity of the manipulated LTSs via the notion of macro and micro transition would enable test objective LTSs to use micro-transitions. Consequently, the method could then take full advantage of the flexibility offered by TGV by adding the possibility of including SUT-internal events in the test objective scenarios, since the relation between the Π-actions of the scenario representation and the Σ-actions of the LTS representation could be chosen to be one-to-one (or almost) even for SUT-internal actions.

In addition to allowing SUT internal actions, a finer granularity in the LTS would also allow test objectives to contain nested invocations and synchronous invocation replies.

3.4.2 Component testing and SUT-environment internal actions

Component testing via the enhancements proposed in Sections 2.6.2 and 4.3.1 entails a more flexible notion of SUT boundary which would need to be reflected in the test objectives. Clearly, the atomicity enhancements of Section 2.6.1 and Section 3.4.1 would make the present enhancement more workable.

3.4.3 Compositional approach to test objectives

A compositional notion of test objective would be a valuable enhancement to the method. A promising approach which is under investigation [Tsc02] involves defining a boolean algebra of test objectives, in which each objective is composed of a positive and negative part. However, this necessarily involves a one-to-many correspondence between test objectives and test cases and the nature of a suitable such correspondence is not yet clear.

3.4.4 On-the-fly treatment of test objectives

The synchronous product of the specification with the overall LTS (or rather with the positive and negative parts of the individual LTSs) could be performed lazily, as could the derivation of an LTS from each of the scenario structures. This on-the-fly approach would also permit a greater expressiveness in the test objective scenario structures, particularly with respect to the use of loops.

4 Test synthesis from model LTS guided by objective LTS

In this section, we give a brief overview of the TGV tool, its underlying theory and algorithms. TGV was developed jointly by IRISA and Verimag; more details can be found in [JarJér02]. Fig. 6-11 illustrates that part of the method that is presented in this section. We then show how the test synthesis proceeds for the ATC application example before giving some possible enhancements to this part of the method.

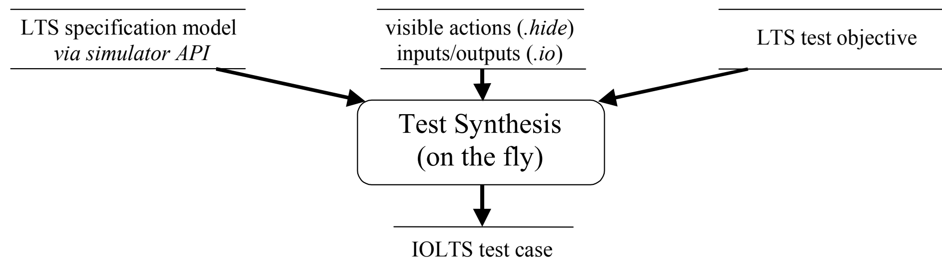


Figure 6-11: Test synthesis part of the method.

4.1 The test synthesis engine

The test synthesis uses the TGV tool, see [JarJér02] or for a more detailed presentation [Jér02], based on the theory developed in [Tre96] and [Pha94] and using algorithmic techniques from [Tar72], but could also use other tools with similar properties.

4.1.1 Inputs and outputs

The inputs to the IOLTS-based test synthesis are, firstly, the following two elements:

- A simulation API, which can be used to lazily construct the LTS representing the operational semantics of the specification of the entire system (including actors).
- An LTS representing the test objective, which partially describes sequences of the specification. The LTS is completed w.r.t. Σ, the alphabet of the specification, by (implicit) loops in each state and the labels on its transitions may contain regular expressions.

and, secondly, in order to define IOLTSs from the LTSs, the following two elements:

- The specification of which Σ-actions are internal (conventionally in a file with the suffix *.hide*), thus defining the SUT boundary and possibly resulting in non-determinism.
- The specification of which Σ-actions on the SUT boundary are inputs and which are outputs from the SUT point of view (conventionally in a file with suffix *.io*).

The output is an IOLTS representing a test case, that is, a predefined set of interactions between the tester – assuming the role of the system environment – and the implementation. It is annotated with the test verdicts:

- *pass*, on termination of executions which fulfil the test objective (i.e. those that lead to an *accept* state of the test objective LTS),
- *inconclusive*, on termination of executions which do not fulfil the test objective but on which the behaviour is consistent with the specification (this verdict is derived as soon as possible);

The *fail* verdict is usually left implicit (on reception of any input not explicitly specified). Note that it is not related to the *reject* states.

4.1.2 Method

The test case is derived by exploring that part of the LTS of the specification which is selected by the test objective (the *reject* part of the objective plays a crucial role in this selection). This involves calculating the synchronous product of the two IOLTSs, calculating an equivalent deterministic IOLTS (after taking into account quiescence, see below) and extracting a test case as the mirror image of a particular type of controllable sub-graph of this "determinised" product and, finally, completing the input states with transitions to a fail state.

The mirror operation simply interchanges inputs and outputs in order to move from a specification viewpoint to a tester viewpoint. A synthesised IOLTS is said to be controllable if none of its states has a controllability conflict; a state has a controllability conflict if it has more than one outgoing transition and one of these transitions is an emission (recall that all data is enumerated so we do not need to concern ourselves with whether guards are mutually exclusive or not).

4.1.3 Theoretical basis

This test case derivation is based on the following formal notion of conformance (known as *ioco,* see [Tre96]): an implementation conforms to a specification if it cannot produce outputs which are unexpected w.r.t. the specification, after executing a trace of observable actions which is allowed by the specification. In the theory, the absence of visible activity (quiescence) – resulting from deadlocks, livelocks or waiting for input – must be observable in order for it to be preserved in the determinisation and is therefore considered to be a particular type of output. In practice, such "outputs" are detected by timers. Infinite loops of observable actions can be dealt with using a timer which is global to the test case.

The theory also assumes that an implementation (conformant or not) can never refuse an input and that in the general case, the set of possible inputs resp. outputs of an implementation is a superset of the set of inputs resp. outputs of the specification. All but the controllability part of the test synthesis algorithm is performed on-the-fly, that is, lazily with respect to the construction of the state graph. This enables the test synthesis algorithms to handle specifications of arbitrary size.

The synthesis ensures essential properties on the synthesised test cases with respect to *ioco*. In particular test cases are sound or correct, that is, they reject only non-conformant implementations. The converse property, exhaustiveness, is unreachable in practice due to loops and to the fact that the tester does not control the SUT. Nevertheless, the test synthesis method itself can be guaranteed to be exhaustive, in the sense that for any non-conformant implementation, it is possible to synthesise a test case that may reject it (only "may" due to possible inconclusive verdicts arising from the fact that the tester does not control the SUT).

4.2 Test case synthesis for the ATC application

Feeding TGV with the test objective visualised in Fig. 6-8 and the LTS derived from the ATC UML specification using the following command (see the TGV user manual for an explanation of the TGV options):

```
uml.open atc.uml tgv –io atc.io –hide atc.hide –unlock –tpprior \
```

```
-outprior -output TO.aut TC.aut
```

we obtain the following IOLTS (here, in Aldebaran format [FerGarKer96]), describing the test case:

```
des (0, 22, 21)
(0, "ptg!cwp.updateSystemTracks(0, true); OUTPUT", 1)
(1, "ccg!fpm.assume(0); OUTPUT", 2)
(2, "ccg!fpm.handOverRequest(0); OUTPUT", 3)
(3, "ctrl?notifyUpdateStatus(0,1); INPUT", 4)
(4, "ccg!fpm.assume(0); OUTPUT", 5)
(5, "ccg!fpm.handOverRequest(0); OUTPUT", 6)
(6, "ctrl!cwp.commandAssumeOrder(0); OUTPUT", 7)
(7, "ccg?notifyAssume(0); INPUT", 8)
(7, "ctrl?notifyUpdateStatus(0,2); INPUT", 9)
(8, "ctrl?notifyUpdateStatus(0,2); INPUT", 10)
(9, "ccg?notifyAssume(0); INPUT", 10)
(10, "ccg!fpm.assume(0); OUTPUT", 11)
(11, "ccg!fpm.handOverRequest(0); OUTPUT", 12)
(12, "ctrl!cwp.commandAssumeOrder(0); OUTPUT", 13)
(13, "ccg!fpm.assume(0); OUTPUT", 14)
(14, "ctrl!cwp.commandHandOver(0); OUTPUT", 15)
(15, "ccg?notifyHandOver(0); INPUT", 16)
(15, "ctrl?notifyUpdateStatus(0,3); INPUT", 17)
(16, "ctrl?notifyUpdateStatus(0,3); INPUT", 18)
(17, "ccg?notifyHandOver(0); INPUT", 18)
(18, "ccg!fpm.assume(0); OUTPUT", 19)
(19, "ctrl?notifyUpdateStatus(0,4); INPUT (PASS)", 20)
```

This IOLTS is visualised in Fig. 6-12. Notice that the first and last transitions coincide with those of the test objective LTS.

Such a test case could be executed by a tester on an implementation of the ATC model, as outlined in Section 5. For example, in the transition from state 2 to state 3, the tester sends a request for the flight 0 to be handed over (via the *OUTPUT* message *handOverRequest*). In reply, it waits for a notification of the update of the flight status to 1 (via the *INPUT* message *notifyUpdateStatus*). TGV reconstructs observable action sequences that lead the implementation to satisfy the test objective. However, it does not necessarily find the shortest sequences, since minimisation is impossible to achieve using on-the-fly algorithms. For instance, the transition from state 1 to state 2 leaves the system unchanged, and thus is not useful (but not harmful either). The concurrency in the system leads to the diamonds between states 7 and 10 and between states 15 and 18. To make the test case more readable, the transitions leading to a *Fail* verdict on unexpected input (of the tester) are left implicit.

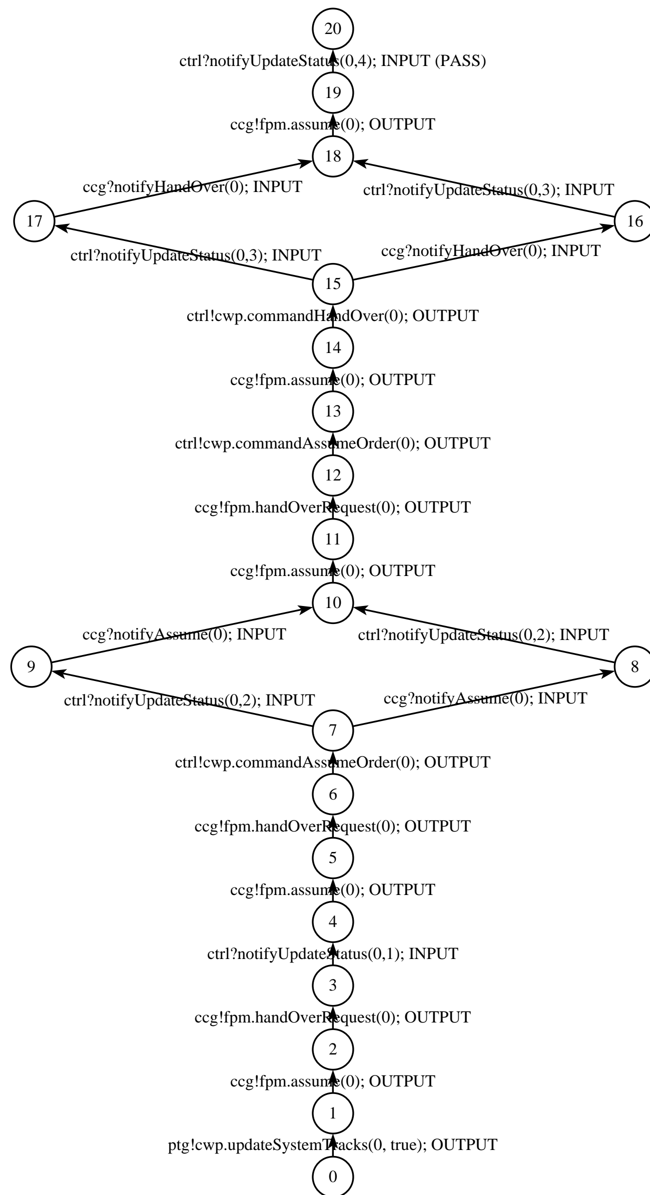


Figure 6-12: IOLTS describing a test case synthesised according to the test objective IOLTS of Fig. 6-10.

4.3 Test synthesis: possible enhancements

4.3.1 Component testing and SUT-environment / tester internal actions

As already stated, the definition of IOLTS on which the current test synthesis is based does not include the possibility of external actions which are neither inputs nor outputs. We require two such types of external actions: those internal to a component and those between components. Under an extension of the mirror operation of the test synthesis algorithms, internal actions of SUT environment components would become internal actions of tester components and communications between SUT environment components would become communications between tester components.

If coupled with a finer semantics in the Umlaut simulator, see Section 2.6.1, these actions would enable us to use the test synthesis algorithms for general component testing of a given specification, rather than simply system testing.

4.3.2 More sophisticated treatment of data

The enumerated character of the IOLTS basis (that is, all data must be instantiated, data cannot be treated symbolically) is undoubtedly a weakness of the method. However, this deficiency can be palliated somewhat by the use of test data generation techniques. In the COTE project [JarPic01], this approach has been explored via in the U-Casting [AerJen03] tool. A more ambitious proposal involves extending the synthesis techniques to include symbolic treatment of data However, state-space exploration with such treatment is a hard problem. An extension of the TGV tool which deals with symbolic data is under development [ClaJérRus01] but is not yet incorporated into the Umlaut environment.

4.3.3 "True concurrency" approach

TGV has been primarily designed to generate sequential test cases. To take advantage of the TeLa test language, information about concurrency must be extracted. A more ambitious approach is to keep explicit initial concurrency of the UML model of the specification, replacing the IOLTS representation by a true-concurrency model, presented in [Jar02].

The use of such an approach would entail the revision of the other three main parts of the method since in the current method they are all to some extent tailored to use in IOLTS-based synthesis.

5 Derivation of scenario-based test case from IOLTS

In this section, we show how to derive a scenario structure in the TeLa language from an IOLTS representing a test case produced by TGV. We first mention some important points concerning the TeLa language before discussing how to represent the output of TGV in TeLa. Fig 6-13 illustrates the part of the method that is presented in this section. We then show how the UML test case derivation part of the method proceeds for the ATC application example. We also use this example to illustrate some possible enhancements to the method. The one concerning the use of the actors to define a default test architecture could be automated relatively easily.

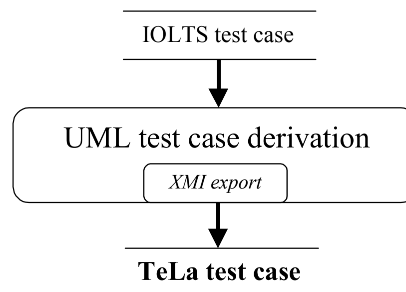


Figure 6-13: UML test case derivation part of the method.

5.1 The TeLa language

The TeLa language [PicJarHeu01] extends the UML 1.4 sequence diagram notation. The extensions are introduced in order to define a language suitable for specifying test cases, in the form of scenario structures, for applications which may contain concurrency and asynchronous communication; the language is not aimed exclusively at the translation of IOLTSs produced by TGV. The extensions to UML 1.4[15] are, in part, inspired by the MSC notation [ITU-T99], as well as by the graphical TTCN-3 notation [ETSI03b] that is largely derived from it. As the MSC standard is apparently also the main inspiration for the UML 2.0 sequence diagrams, our approach is well-positioned with respect to upcoming developments.

5.1.1.1 CURRENT PROTOTYPE: IMPLEMENTATION NOTES

Activity diagrams are not implemented in the prototype of TeLa in the Objecteering tool. The only type of choice implemented in this prototype is the one in which the alternative behaviours differ only in the initial message (which is unsuitable for representing the ATC test case). The coregion operator is implemented in the prototype of TeLa.

5.1.2 Constructs of TeLa used to represent IOLTS test cases

Rather than talk in the abstract, we illustrate the main aspects of using TeLa to represent the output of the TGV tool by representing the test case synthesized from the ATC application

[15] Including weak sequential composition of sequence diagrams, a choice operator (enabling the specification of alternative behaviour continuations on different diagrams), a coregion operator (to introduce explicit concurrency on a lifeline), and an internal action construct (to specify actions such as assignment to tester variables or tester assertions).

example test objective, as shown in Section 4.2. TeLa offers alternative presentations for many constructs as follows.

The most general choice (permitting, for example, a non-local choice, or a local choice involving events which are internal to the SUT, followed by arbitrary alternative behaviour) must be modelled as a 2-tier TeLa diagram, that is, using activity diagrams and activity-diagram references. Fig. 6-15 illustrates how the ATC application example test case is represented using this type of diagram. Many types of choice can be modelled using several TeLa sequence diagrams connected via TeLa sequence diagram references (e.g. local choices between messages followed by arbitrary alternative behaviour). Fig. 6-14 illustrates how the ATC application example test case is represented using this type of diagram.

Choices in which the alternative behaviours only differ in the initial part, that is, choices in which the alternative behaviours are joined shortly afterwards (such as the diamond structures of the ATC application example test case), can be modelled using a single TeLa sequence diagram and the block construct. Fig. 6-16 illustrates how the ATC application example test case is represented using this type of diagram. Finally, choices in which the alternative behaviours only differ in the initial message (the "unique continuation choice" of [Livrable 2.3]) can be modelled using a single TeLa sequence diagram without the need for the block construct. The ATC application example test case cannot be represented using this type of diagram.

Concurrent messages can be represented using the coregion construct. Fig. 6-17 illustrates how the ATC application example test case is represented using this type of diagram (N.B. the use of coregions means that this diagram cannot easily be derived automatically from the IOLTS, see Section 5.2.5.1). Finally, we can also represent the actors of the test on different lifelines, i.e. use the actors as a default test architecture. To illustrate this, we could adapt any one of the representations of Figs. 6-14 – 6-17. to include the explicit test architecture. We choose to do so with Fig. 6-17 since this gives the smallest diagram (however, since it uses coregions, like Fig. 6-17 it cannot easily be derived automatically from the IOLTS). Thus, Fig. 6-18 illustrates how the ATC application example test case is represented using this type of diagram, by extending that of Fig. 6-17. It is important to note that this diagram uses a restriction of the standard partial-order semantics of TeLa, see Section 5.2.5.2.

5.2 From IOLTS to TeLa expression

5.2.1 TeLa sequence diagram lifelines

In TeLa, lifelines represent components in a hierarchical component model. The hierarchy always contains at least two levels, namely, that defined by the objects (the lowest level components) and that defined by two components: SUT and tester. The test case produced by TGV can then always be correctly represented using only two lifelines, the SUT and the tester.

The lifelines of UML 1.4 sequence diagrams represent objects or collaboration roles. As defined in UML 1.4, collaboration roles must have a base classifier (this restriction could definitely be criticised). In TeLa, we wish to use lifelines to represent (roles played by) components of any size, from objects to systems. In particular, we need at least two hierarchical levels of components to represent the tests synthesized by TGV: objects and the two testing-oriented components, the SUT and the tester. Strict compatibility with UML 1.4 could be achieved by specifying the components SUT and tester explicitly in a component

model in order for these components to constitute the obligatory base classifiers of the corresponding collaboration roles.

5.2.2 Atomicity in TeLa scenarios

By construction, all the transitions of the IOLTS test cases produced by TGV are labelled by actions on the SUT boundary. If the guidelines for specifying actor state machines of Section 2.1.2.7 are followed, there is a one-to-one relation between these Σ-actions and the corresponding UML-actions, except for the case of actor auto-invocations (and actor-to-system synchronous invocations, if the Umlaut implementation is completed to allow the latter). Thus, it is relatively easy to construct TeLa scenarios such that the Σ-actions to Π-action relation is one-to-one in all cases but these. These two exceptional cases are dealt with separately in the following sections.

5.2.3 The SUT boundary in TeLa scenarios

As in the O-TeLa case, see Section 3.2.3, the Π-action SUT boundary is defined by the events on SUT lifelines. In the TeLa case, the translation is from the transitions labelled by external Σ-actions to the events labelled by the corresponding Π-actions, rather than the other way round. The assumption concerning external Σ-actions (that they are either inputs or outputs) at the LTS level means that currently, all Π-actions in TeLa diagrams are supposed to be on the Π-action SUT boundary, that is, they are either send or receive events of communications with the SUT (but see Section 5.2.4).

As in the O-TeLa case, if we also wish to complete the current implementation and take into account synchronous invocations from the tester to the SUT, even if we suppose that the corresponding send and receive event are indivisible in the sense that no other event can occur between them (c.f. the semantics of "interworkings"), we would inevitably lose the one-to-one correspondence between and LTS Σ-actions and scenario Π-actions. However, in the interests of compatibility with the asynchronous invocation case, we could choose to view the event on the SUT environment lifeline as being the SUT boundary event. The Π-action labelling this event is then derived from the label on the corresponding LTS transition.

5.2.4 Tester auto-invocations in TeLa scenarios

Even in system testing, as has already been made clear, the assumption mentioned in the last section concerning the Π-actions on the Π-action SUT-boundary does not hold. Furthermore, as stated in Section 2.4.1.2, the guidelines for specifying active-object state machines of Section 2.1.2.7 practically guarantee that we will have to deal with such actions, in the form of tester auto-invocations.

Due to the fact that emission of an auto-invocation will be treated as a system input by TGV, in a simple automatic derivation procedure, it will be shown as a message from the tester to the SUT. Usually, if the guidelines of Section 2.1.2.7 for specifying actor state machines are followed, the body of the invoked operation will indeed contain an invocation from the tester to the SUT with the same name, so this is not so much of a problem. However, occasionally, it will not contain such an invocation and the simplistic TeLa representation of the synthesized test case will be incorrect. To automatically produce a correct representation, it would be necessary to check the UML specification for each tester output Σ-action of the synthesized test case to see whether or not the series of UML-actions to which it corresponds

begins with an auto-invocation and contains no invocation from the SUT environment to the SUT.

5.2.5 Deriving a scenario structure from an IOLTS

5.2.5.1 RECOVERING CONCURRENCY FROM AN INTERLEAVING MODEL

Even though the test objectives may be specified in terms of partial orders, the basis of the test synthesis is currently IOLTSs in which the partial order is lost; concurrency is modelled as interleaving so that choice and concurrency are confused. One could hope to reconstruct the concurrency from diamond structures in the IOLTS output by TGV. However, apart from the difficulty in recognising diamond structures involving the interleaving of an arbitrary number of actions, such structures cannot be interpreted as representing concurrency without making assumptions about the application model. This cannot currently be done automatically.

5.2.5.2 TESTER STRUCTURE: DEFINING THE TEST ARCHITECTURE

For a test case synthesized by TGV, the external actors and their interaction with the SUT can be considered to define a default tester structure or test architecture (see Chapter 2, Section 1.3.2.3 for our definition of test architecture). We may wish to use this architecture explicitly in the TeLa scenario structure derived from the TGV output, that is, we may wish to use a distinct lifeline to represent each external actor. The Umlaut-generated labels used in the output of TGV contain enough information to identify the external actor involved in each tester event so this is no impediment.

However, with a general partial order interpretation, representing each actor with a separate lifeline introduces further concurrency. As this concurrency is not present in the IOLTS descriptions, where it is not representable, we cannot introduce it into the TeLa representation of the synthesized tests. Therefore, either we add a large number of coordination messages between the different tester entities or we use a more restricted interpretation of the TeLa diagrams, in which the partial order is of messages rather then of events. This can be obtained as a restriction of the partially-ordered event semantics as shown in Chapter 5, Section 3.

TeLa provides this option, giving a semantics similar to that of UML 1.4 sequence diagrams or to “synchronous interworkings”. In this restricted “interworking” semantics, if two communication events are in the scope of a coregion on one instance but their twin events are ordered on another instance, the coregion has no effect.

In the context of a hierarchical component model, a general test architecture could be given. The test architecture diagram should tell us which objects belong to which test architecture components. Knowledge of which invocation is sent or received by which component is then sufficient to present the TeLa diagrams in which any such test architecture appears explicitly (in the sense that there is one lifeline for each component of it). Note, however, that with the current implementation, there would be no messages between elements of the test architecture, see Section 5.4.2.

5.2.5.3 SUT STRUCTURE

Though we are concerned with black-box testing, this does not mean that test cases described in a sequence diagram style notation such as TeLa cannot show part of the internal structure of the SUT in terms of objects/components. This is since, in the object paradigm, the tester needs the identifiers of the objects/components it is to communicate with, either having this knowledge in the initial state, or acquiring it in the course of the test.

However, aside from the difficulties concerning the recovery of concurrency information discussed in the previous section, in general, representing the "visible" SUT internal structure explicitly in the test cases produced by TGV is not possible, since the Umlaut-generated labels do not contain enough information. The label on the transition corresponding to the reception of an asynchronous invocation by the tester does not contain information as to which SUT object is the emitter. This problem was already discussed in Section 2.4.1.

In some cases, a detailed knowledge of the specification may be sufficient to represent SUT structure in TeLa diagrams, in a manual conversion of the TGV output. This will be the case if the UML specification is such that for each SUT-to-SUTenv invocation appearing in the test case, there is only one candidate for its emitter. That is, detailed knowledge of the action expressions of the specification tells us that only one object makes each type of invocation appearing in the test. The use of a suitable component model, such as that of UML 2.0, would in many cases also allow the missing information to be extracted from the specification. However, a general solution would require the enhancements proposed in Section 2.6.2, allowing true component testing, see below.

5.2.6 Component names in arrow labels

The syntax for the labels on the arrows of TeLa sequence diagrams extends UML syntax by allowing the name of a lower-level component owning the event which is inside the component represented by the lifeline to appear explicitly, prefixing the operation name and parameters. In the current implementation, with only two levels of hierarchy, in the case of an invocation by the tester, respectively, by the SUT, the name of the owning SUT object, respectively, of an external actor, followed by a dot appears before the name of the method. See Figs. 6-14 – 6-17 for examples of this syntax. We now give the rules for the TeLa syntax use of these prefixes, supposing only two levels of components: object level and SUT & tester level. These are similar to those given for O-TeLa in Section 3.2.6.

First suppose that only two lifelines are used in the test case scenarios, one representing the SUT and the other representing the tester. The label on an arrow representing an invocation emitted by the tester and received by the SUT is prefixed by the emitting actor name and by the receiving SUT object name (separated by the "!" character). The label on an arrow representing an invocation emitted by the SUT and received by the tester is prefixed by the receiving actor name.

Next suppose that we use the default test architecture defined by the actors, see Section 5.3.5. Thus, only one lifeline is used for the SUT but multiple lifelines are used for the tester, one for each actor. The label on an arrow representing an invocation emitted by the tester and received by the SUT is prefixed by the emitting actor name. The label on an arrow representing an invocation emitted by the SUT and received by the SUT environment is not prefixed by a component name.

There are no other possibilities for representing the IOLTS synthesized by TGV since neither SUT structure nor other test architectures, see Section 5.2.5, can be used in the current implementation.

5.3 Synthesized test case for the ATC application

If we do not make any assumptions on the UML specification, the diamond structures involving states 7, 8, 9 & 10 and involving states 15, 16, 17 & 18 in Fig. 6-9 must be

modelled using the choice operator. If we can recover the concurrency information, we can model these structures using the coregion operator.

5.3.1 One-tier representation with TeLa sequence-diagram references

Fig. 6-14 shows a TeLa scenario structure representing the test case of Fig. 6-12, which is synthesized from the ATC UML specification according to the test objective of Fig. 6-10, which is in turn represented by the scenario structure of Fig. 6-8. This scenario structure is one-tier, that is, it uses TeLa sequence diagram references to link TeLa sequence diagrams.

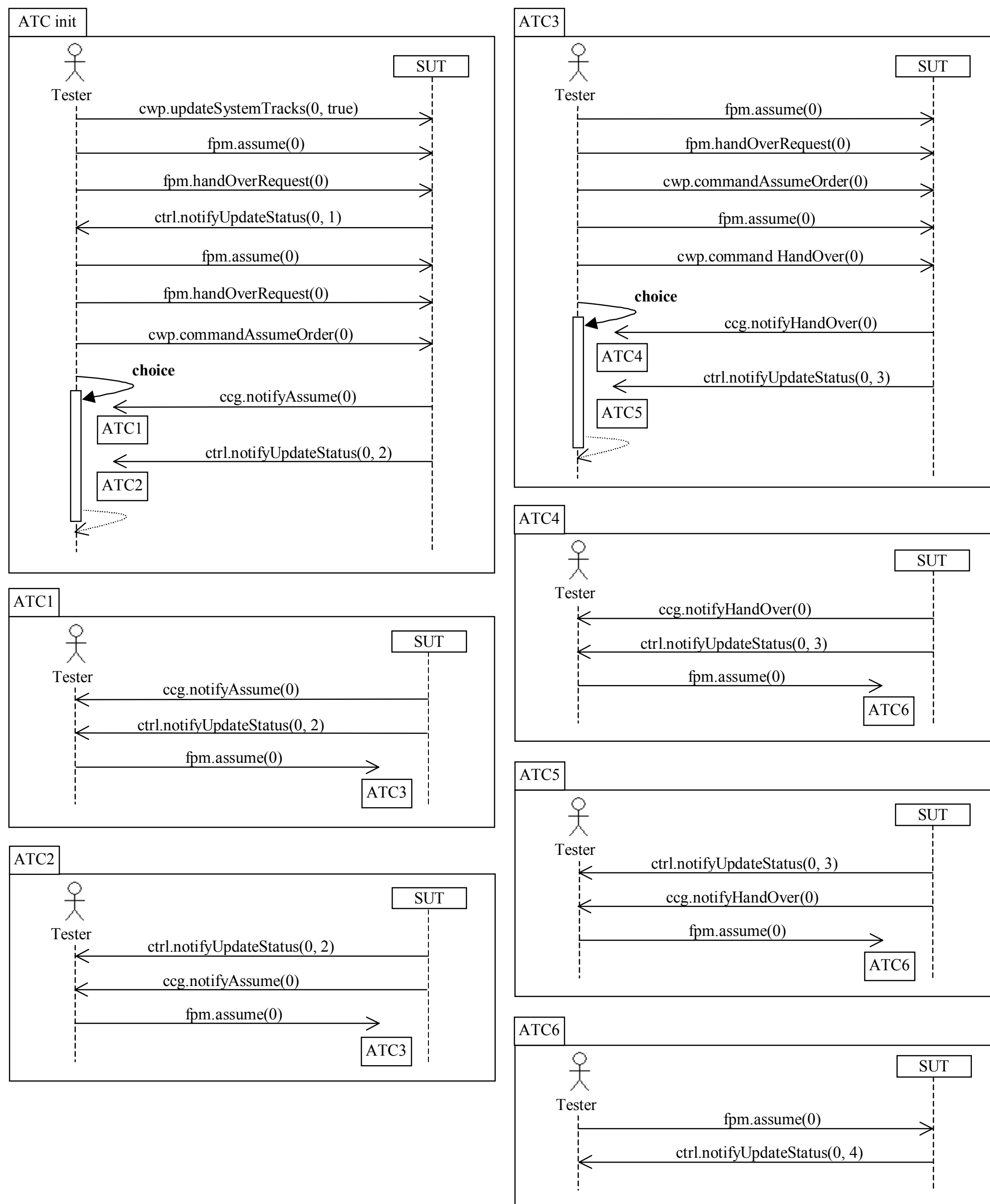


Figure 6-14: 1-tier TeLa representation of the test case described in IOLTS form in Fig. 6-12 (using the TeLa sequence diagram reference construct).

5.3.2 Two-tier representation with TeLa activity-diagram references

Fig. 6-15 shows the two-tier version of Fig. 6-14. This time the TeLa sequence diagrams are connected by being referenced from a TeLa activity diagram. Note that the sequence diagrams themselves involve fewer messages than those of Fig. 6-10, due to the fact that connecting diagrams in a one-tier structure involves duplicating messages.

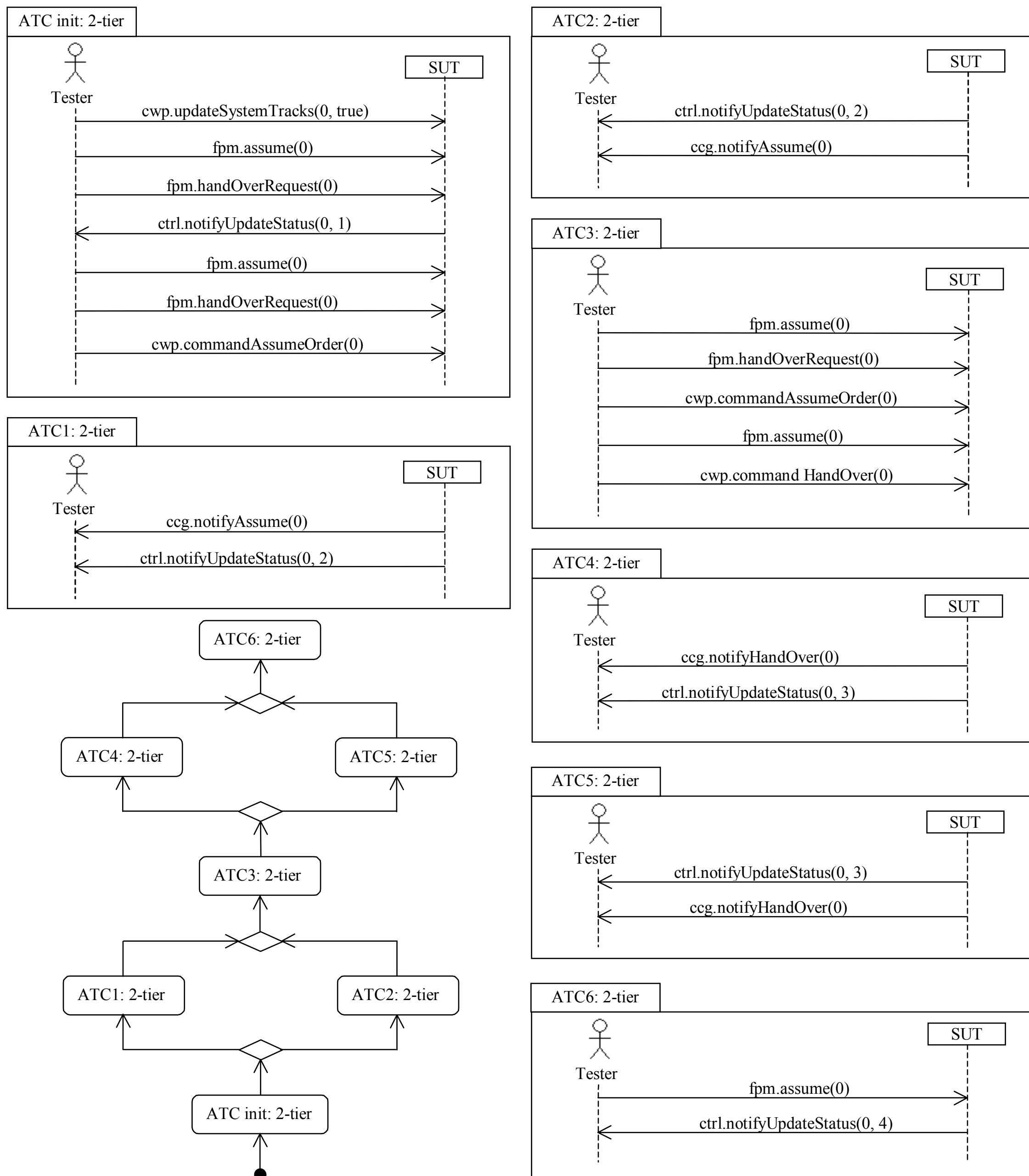


Figure 6-15: 2-tier TeLa representation of the test case described in IOLTS form in Fig. 6-12.

5.3.3 Block structure representation

The representation of Fig. 6-14 uses TeLa sequence-diagram references and that of Fig. 6-15 uses activity-diagram decisions. In this particular case (choices between alternative behaviours, followed by a join), a simpler representation using the block construct, can also be used, see Fig. 6-16. However, as stated in the introduction to Section 5.1, none of these three representations is implemented in the prototype.

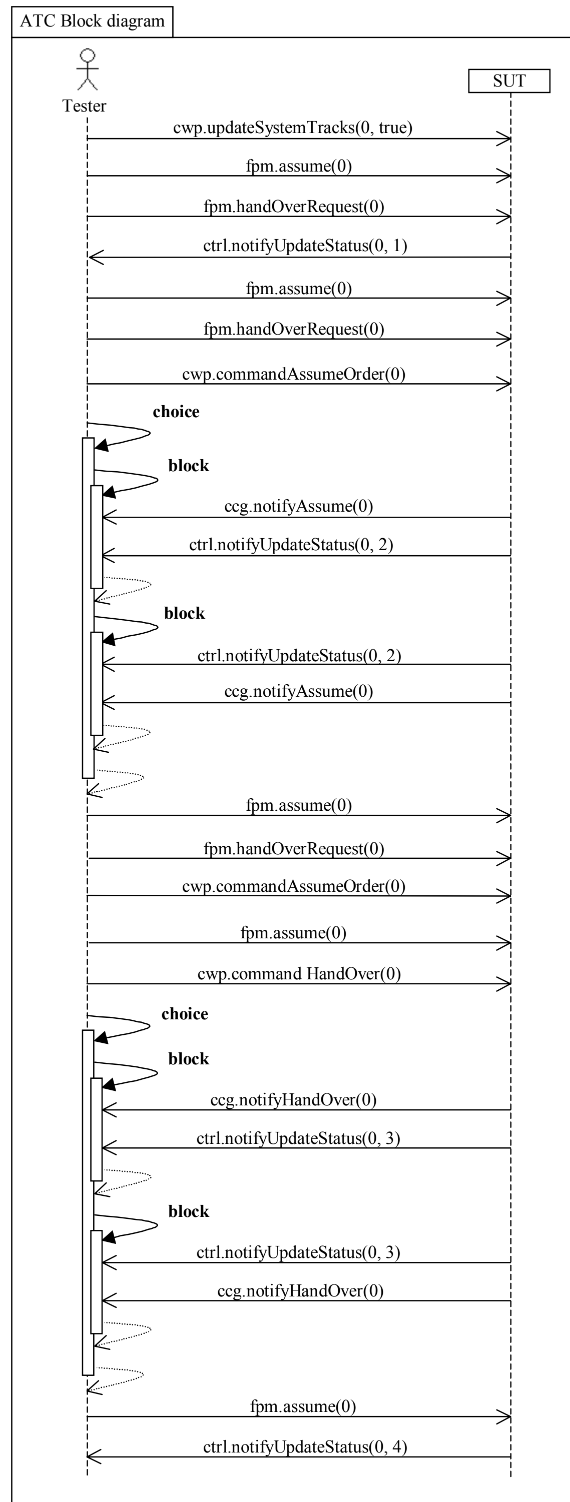


Figure 6-16: TeLa representation of the test case described in IOLTS form in Fig. 6-12 using the block construct

5.3.4 Representation using coregions

Fig 6-17 shows a TeLa scenario structure representing the same test case as that represented in the previous figures. The part of the TeLa language used in this diagram has been implemented in the prototype. The diamonds formed by states 7, 8, 9, 10 and by states 15, 16, 17, 18 have now been interpreted as representing concurrency (by making certain assumptions about the UML model). The coregions are thus used to indicate that the tester can expect to receive the corresponding messages in any order. Note that the coregions are placed on the Tester: in the partial-order semantics of TeLa, without further assumptions, placing the coregions on the SUT would not break the ordering on the tester and it is the events on the tester instance which are significant for the test; the events on the SUT instance only serve to add orderings between the events on the tester instance(s).

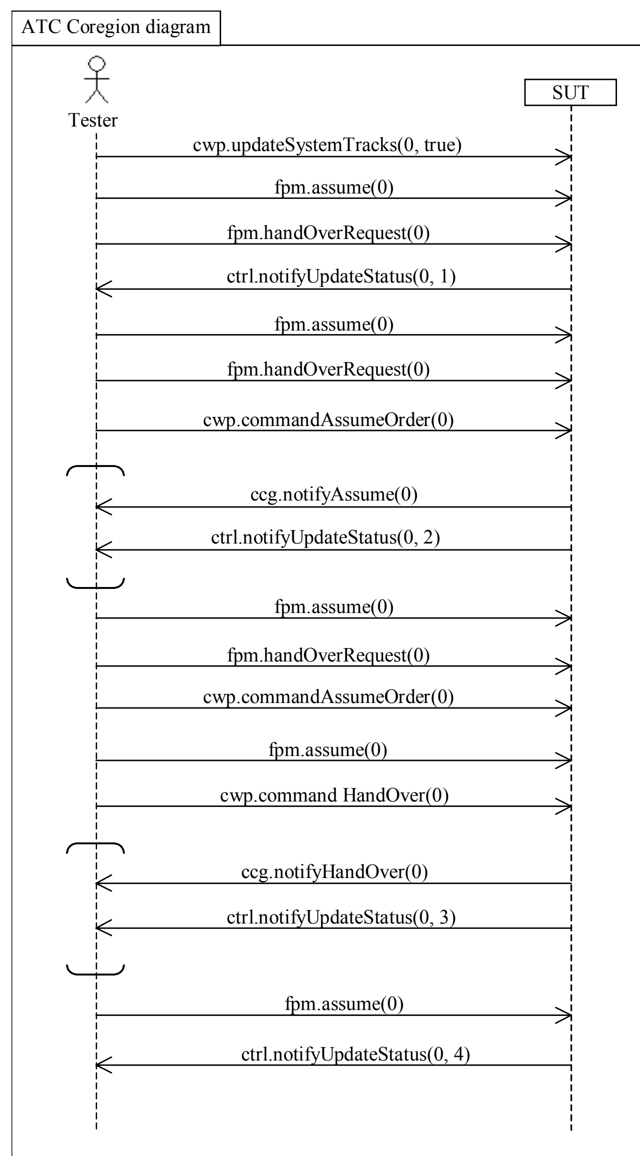


Figure 6-17: TeLa representation of the test case described in IOLTS form in Fig. 6-12 using coregions.

5.3.5 Representation using default test architecture (and coregions)

In TeLa, the tester structure can be shown by using different lifelines to represent different tester components. More generally, a test architecture would be described in a component diagram. In the case where these lifelines represent tester objects, the syntax allowing the name of the tester object to appear in the label of an arrow representing an invocation to that tester object (see Section 5.2.6) is no longer needed.

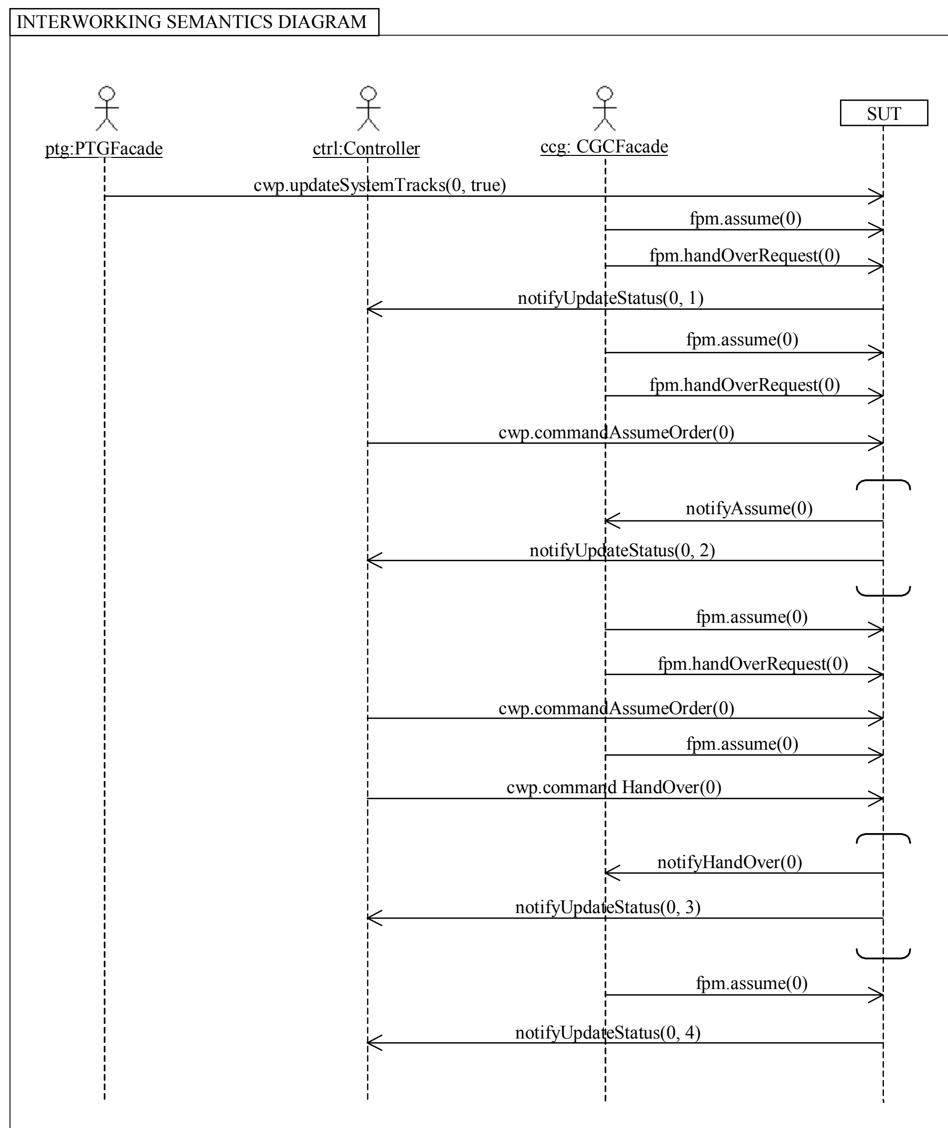


Figure 6-18: TeLa representation of the test case described in IOLTS form in Fig. 6-12 using coregions and showing tester structure.

Fig. 6-18 shows a TeLa scenario structure representing the same test as that represented in the two previous figures but with the tester structure represented explicitly (and, in consequence, with the "interworking" semantics). Note that the coregion operator is now on the SUT. The part of the TeLa language used in this diagram has also been implemented in the prototype.

More generally, in the context of a component approach, the synthesized TeLa test case could be represented according to any given test architecture described as a component model. The notation for prefixing component names on arrow labels gives the flexibility to display all the

information required (in a hierarchical component model, if the base-level name is used, other levels can be deduced).

5.4 UML test case derivation: possible enhancements

5.4.1 Atomicity in TeLa test cases

A change in the atomicity of the manipulated LTSs via the notion of macro and micro transition, as proposed in Section 2.6.1, would allow TeLa test cases containing SUT callbacks and synchronous invocation replies.

5.4.2 Component testing and tester internal actions

In sections 2.6.2, 3.4.2 and 4.3.3, we discussed how more information an the labels of the LTS generated by Umlaut, together with test synthesis based on an extension of the IOLTS to include tester internal actions, would make it possible to use the method for general component testing and not just system testing. The principle effect on the TeLa test cases would be to allow test synthesis to also derive coordination messages between different tester components and to allow SUT structure to be represented in TeLa sequence diagrams. As already stated, the atomicity enhancements of Sections 2.6.1, 3.4.1 and 5.4.1 would make the present enhancement more workable.

However, in such a component testing approach, the translation from IOLTS to TeLa could be parameterised by the test architecture, given as a component diagram, as outlined in Section 5.2.5.2 above. In the synthesis method, then, the test designer would:

- designate any component of the application component diagram as the SUT
- give a component diagram to define the test architecture;
- synthesize test cases for the given architecture; the SUT environment structure may be shown in the O-TeLa test objectives (if there is no component diagram for it, then merely the default one defined by the actors); the test architecture which may be shown in the TeLa synthesized test cases

Chapter VII : Conclusions

1 Original contributions

In this section we present a summary of the original contributions of this thesis.

1.1 Structural semantics of scenario-based test descriptions

Component-based software is increasingly widespread and testing languages adapted to this emerging software paradigm are of increasing importance. However, such languages are currently very immature. The notion of component is at the heart of a component testing language and we therefore need to base our language on some component model. The component model of UML 1.4 is clearly not suitable for our purposes. As regards the component model of UML 2.0, though covering most of the aspects we need, it is not sufficiently precise. We therefore define a more formal notion of component model largely inspired by UML 2.0 ideas, which have evolved in parallel to our formalisation.

1.1.1 A component model

In Chapter 4, we formally define the basis of a component model underlying our scenario-based language in terms of components with ports which are joined via connectors. Though not the aim of this work, this model can be seen as a formalisation of part of the UML 2.0 component model.

1.1.1.1 WHY AN UNDERLYING COMPONENT MODEL?

The underlying component model is crucial to making TeLa a component testing language.

We make precise the idea of the structural basis (set of lifelines) of test description scenarios being views on a component model snapshot, in such as way as to ensure that this basis reflects the *test architecture* and the *test configuration*. The component model also provides a suitable framework in which to interpret multiple SUT (system under test) lifelines in a language describing functional tests, that is, tests in which the SUT is treated as a black-box.

For an application designed according to the component paradigm, the test architecture can be derived from component architecture of the application. Hence, if the test language used has an underlying component model, the test descriptions will then reflect or incorporate aspects of the component architecture of the application.

Therefore, given an application designed according to the component paradigm, we can designate any given component of the application model as the SUT and derive, or partially derive, the structure of the test descriptions from the application component model. The scenario-based tests can then be better related to the scenario-based development assets of the application, such as use-case scenarios. Along with a treatment of tester internal actions, this relation between the application component architecture and the test architecture is the key to generalising established test synthesis techniques from system testing to component testing.

1.1.2 Component hierarchy

The component model we define is hierarchical.

1.1.2.1 WHY A HIERARCHICAL COMPONENT MODEL?

The fact that the component model underlying a test description is hierarchical provides us with a framework in which to define the following concepts:

Composition/decomposition of lifelines. A lifeline representing a component can be decomposed into lifelines representing its subcomponents. This facility enables test descriptions to be viewed at different levels of structural abstraction according to need. A component hierarchy enables composition/decomposition of lifelines to be given meaning for a wider range of descriptions than in MSCs, e.g. those involving guards that contain dynamic variables.

Component-based properties, in particular, those which affect the behaviour of the test being described. We introduce two such properties (see below for more details):

- *Communication semantics*: a component can have event-based or message-based semantics. This property concerns the communications between the subcomponents of a component. If a component has message-based semantics, all its subcomponents must have message-based semantics.
- *Control flow scheme*: a component can be active or passive. This property concerns the ordering of events which are located on the component. A component is passive if all its subcomponents are passive.

Local versions of global properties. A local notion of controllability, for example, would be very useful in distributed testing. As an example of a question of interest for distributed testing, we could ask: is the test locally controllable for some lifeline decomposition?

Nodes. Assuming the units actually deployed (nodes) are to be related to the components, the existence of hierarchy provides flexibility in choosing these nodes. As an example of a question of interest for distributed testing, we could ask: what level of decomposition defines the most suitable physical distribution of the test system? The choice of nodes may be influenced by the component-based properties, e.g. if a component has message-based semantics, it is likely to be implemented in a centralised manner.

1.2 Behavioural semantics of scenario-based test descriptions

Partial-order test descriptions are of great importance for distributed testing. However, currently, even some basic testing concepts are not defined in the partial-order setting. We attempt to rectify this situation by defining the key concepts needed to give a partial-order semantics to our test description language TeLa.

We also wish to ensure that our semantics leads to a user-friendly language. In this respect we address the semantic issues involved in:

- using explicit SUT lifelines,
- using a simpler, more restricted communication semantics that is closer to that of UML 1.x sequence diagrams, when appropriate,
- making the notion of active or passive lifelines / components more well-defined.

1.2.1 Generalisation of formal testing concepts

We generalise testing concepts familiar in interleaving models to the as-yet largely uncharted partial-order case.

1.2.1.1 GENERALISATION OF THE FORMAL DEFINITION OF VERDICT

In Chapter 4, we give a formal definition of the notion of verdict, in a way that generalises smoothly from the relatively well-established interleaving semantics case to the relatively uncharted partial-order semantics case. We define a verdict as an annotation on a transition or event.

In the partial-order case, verdict annotations are local and translate to local verdicts in distributed testing. We therefore also define formally how global verdicts are obtained from local verdicts in the partial-order case. The most problematic is the local inconclusive verdict. Our definition takes into account that a local inconclusive verdict can degrade into a global fail verdict.

The assignment of local verdicts and the derivation of global verdicts from them is of prime importance in the distributed testing context. For test descriptions involving non-enumerated data, we describe the notion of verdict for the partial-order case in Chapter 3, but in Chapter 4, we only outline the formalisation for the interleaving case.

1.2.1.2 GENERALISATION OF THE CONCEPT OF IMPLICIT VERDICT/INPUT COMLPETENESS

In Chapter 4, based on the above-mentioned formal definition of verdict, we outline a formal definition of implicit verdict, in a way that generalises smoothly from the relatively well-established interleaving semantics case to the relatively uncharted partial-order semantics case. Note that for the non-enumerated data case, we not only define an implicit fail verdict, but also an implicit inconclusive verdict.

In the partial-order case, implicit verdicts are local and translate to local verdicts in distributed testing. For test descriptions involving non-enumerated data, we describe the notion of implicit verdict for the partial-order case in Chapter 3, but in Chapter 4, we only outline the formalisation for the interleaving case.

The implicit verdicts we define ensure *test-completeness*, where this is the well-known *input-completeness* of the interleaving case, together with conditions ensuring that any execution which does not deadlock results in a global verdict. The property of test completeness is essential in a testing language. A well-defined notion of implicit verdicts ensures that test descriptions have a suitable level of abstraction while at the same time being test-complete.

We define the properties of a partial-order test description which are necessary for implicit verdicts to be well-defined. For test descriptions involving non-enumerated data, we describe implicit verdicts and the properties necessary for them to be well-defined in Chapter 3, but in Chapter 4, we only outline the formalisation of the interleaving case.

1.2.1.3 GENERALISATION OF THE CONCEPTS OF DETERMINISM / CONTROLLABILITY

In Chapter 4, we give formal definitions of determinism and controllability in a way that generalises smoothly from the interleaving semantics case to the partial-order semantics case. In fact, in the partial-order case, we can weaken the standard notion of determinism to obtain several useful definitions. Similarly, in the partial-order case, we can weaken the standard notion of controllability to obtain several useful definitions.

The notion of minimal determinism is key to implicit fail verdicts being well-defined. It is therefore of prime importance for a test description language.

The different notions of controllability are the key to defining the following hierarchy of test cases: *parallel test cases*, *externally-coherent parallel test cases*, *coherent parallel test cases*, *externally-centralisable parallel test cases* and *centralisable test cases*. A test case can be

moved from left to right in this list by resolution of tester concurrency. The generalisation of the controllability notion to the partial-order setting is also of some importance in distributed testing.

For test descriptions involving non-enumerated data, we describe controllability notions for the partial-order case in Chapter 3, but in Chapter 4, we only outline the formalisation of the interleaving case.

1.2.2 Definition of other crucial scenario-based testing concepts

We generalise explicit verdicts and explicit (global) default alternatives to the as yet largely uncharted partial-order case. We generalise the notion of local choice to the semantics-by-projection case.

1.2.2.1 EXPLICIT VERDICTS

In the partial-order case, explicit verdicts are not necessarily implementable in any meaningful way. We define a coherent notion of explicit verdict for partial-order test descriptions, which, when combined with implicit verdicts, conserves test completeness.

In Chapter 3, we define the properties of a partial-order test description which are necessary for explicit verdicts to be well-defined. Though we do not formalise the semantics of the explicit verdict, doing so would not be difficult since the mechanisms of the explicit verdict are similar to those of the implicit verdicts and the default alternative.

1.2.2.2 EXPLICIT DEFAULT ALTERNATIVE

In the general case, it is difficult to attach meaning to a (global) default alternative in choices between different scenarios in a language such as TeLa, MSC or UML 2.0 sequence diagrams.

In Chapter 3, we define the properties of a partial-order test description which are necessary for (global) default alternatives to be well-defined. Though we do not formalise the semantics of the default alternative, doing so would not be difficult since the mechanisms of the default alternative are similar to those of the implicit verdicts.

1.2.2.3 NOTION OF LOCALITY APPROPRIATE TO SCENARIOS USED AS TEST DESCRIPTIONS

We define a notion of locality that is appropriate for discussing choices between alternative scenarios in a scenario-based test description language whose semantics is given by projection onto tester lifelines. This gives rise to the definitions of *test local choice* and *test non-local choice*.

1.2.3 Definition of a practical semantics

We define the semantics of our test description language with usability in mind.

1.2.3.1 SEMANTICS BY PROJECTION

In Chapter 2, we informally define the semantics of our test description language involving SUT lifelines by projection onto the tester lifelines. This leads to a simpler more user-friendly language than the use of TTCN-3 ports or MSC gates.

1.2.3.2 CHOICE OF TWO COMMUNICATION SEMANTICS

In Chapter 4, we formally define how a message-based semantics can be defined as a restriction on an event-based semantics for a certain class of test descriptions (those that are RSC: Realizable with Synchronous Communication)[1].

The definition of the former as a restriction of the latter enables us to mix the two interpretations in the same diagram, defining the communication semantics as a property of a component that is denoted via an annotation on the component model. One use of this facility is to simplify the description of the parts of the system which are to be implemented in a centralised manner.

The more restricted semantics is more suitable for modelling centralised implementations or procedural diagrams. The flexibility given by having a less restricting and a more restricting semantics enables us to use the simpler, more restricting semantics when the complexity of the full semantics is not needed. The fact that the restricted semantics is closer to the intended semantics of UML sequence diagrams prior to UML 2.0 may be valuable to test designers familiar with these notations. In the context of the restricted semantics, for example, the return message of synchronous invocations can be safely omitted, as was the standard practice in UML 1.4 sequence diagrams.

In circumstances where use of the restricted message-based semantics is sufficient for the whole test description, use of the full event-based semantics would simply oblige the test designer to clutter the test description with synchronization messages, seriously hampering its readability. An example of such a circumstance is the scenario-based representation of the output of an interleaving-semantics based test-synthesis tool such as TGV. The message-based semantics facility was therefore highly relevant for the COTE project as explained in Section 5.

1.2.3.3 CHOICE OF CONTROL FLOW SCHEME

The UML concept of active or passive is ambiguous. In recognition of the difficulty of defining these concepts, the control flow scheme notations introduced in MSC 2000 simply have no semantics. However, the concept behind these definitions is clearly a useful one, particularly in the object- or component-based system context.

In Chapter 3, we interpret the notion of passiveness as a set of restrictions on the allowed linearisations. When a partial-order semantics is used, this constitutes another semantic layer to be added (optionally) to the basic semantics. As stated below, the control flow scheme is a property of a component, denoted via an annotation on the component model describing the test architecture.

We describe the way this extra semantic layer affects the constructs of a scenario-based test description language, notably the focus bar and the coregion. For example, the effect of a focus bar inside a coregion is to re-impose an ordering, while the effect of a focus bar inside a coregion on a passive lifeline is similar to that of the UML 2.0 construct “critical region”.

We claim that our definitions adequately model the notion of passiveness as regards the interpretation of:

- concurrency as being implemented via scheduling rather than execution threads,
- synchronous calls as blocking the sender,
- focus bars as representing method executions.

---

[1] In fact, we only need the RSC property on the part of the diagram that is to use the message based semantics.

1.3 Definition of a scenario-based test description language

In Chapters 2 and 3, we define a scenario-based test description language TeLa based as closely as possible on UML 1.4 sequence diagrams (though, the inadequacies of UML 1.4 meant that this was not as closely as we originally thought possible!) and inspired by MSC 2000. It is suitable for describing centralised and distributed black-box tests of object- and component-based software. It is a language in which the system under test (SUT) is represented by one or more lifelines and, though the emphasis is on control aspects, it includes treatment of data.

1.3.1 Motivation for the definition of TeLa

1.3.1.1 EVALUATION OF EXISTING SCENARIO LANGUAGES AND THEIR SEMANTICS

We evaluate the suitability of UML 1.4, UML 2.0 and MSC 2000, with the emphasis on UML 1.4, as the basis for a test description language.

We evaluate the suitability of the semantics proposed for other scenario-languages, notably the different semantics proposed for MSCs, as the basis for the semantics of a test description language.

1.3.1.2 CHOICE OF SUITABLE LEVEL OF ABSTRACTION

We study the most suitable level of abstraction for scenario-based test descriptions. Our choice for the language TeLa differs from that taken in the UTP proposal in that we allow constructs which cannot be realised directly but require some synchronization mechanism (e.g. synchronization messages, centralised controller, shared variables, etc), or some resolution of indefiniteness, which need not be specified explicitly. Prominent examples of such constructs are:

- test non-local choices (see above),
- internal actions or guards involving variables from multiple base-level components,
- indefinite choices, that is, choices for which it cannot be guaranteed that only one of the alternatives is possible in any execution,
- implicit verdicts, that is, we only describe the correct behaviour
- implicit derivation of a global verdict from local verdicts

The decision to model the choice of control flow scheme as an extra layer of semantics also stems from the desire for the test descriptions of our language to remain at a slightly higher-level of abstraction.

It is worth noting that it is a greater challenge to ensure that implicit verdicts, explicit verdicts and default alternatives are well-defined for more abstract test descriptions.

1.3.2 Principle original features of TeLa

1.3.2.1 USER-FRIENDLY SUBLANGUAGE: ONE-TIER SCENARIO STRUCTURES

We define a user-friendly sublanguage of the full test description language that has less expressive power but is simpler to use than the full language. This sublanguage is sufficient for most typical test cases. We ensure that it automatically has the properties required for the test description to specify a parallel test case, with well-defined implicit verdicts, explicit

verdicts and default alternatives. However, we do not deal with the computational complexity issues involved in ensuring the well-formedness of this sublanguage (nor of the full language!).

1.3.2.1.1 Locally-defined choice operator in one-tier scenario structures

Much of the complexity of scenario languages (assuming no explicit parallel operator) arises from the choice operator. This is since it describes a global choice between elements which are naturally only partially-ordered. The choice operator of the sublanguage mentioned in the previous section, the TeLa sequence-diagram choice operator, only allows a limited range of choices in order to ensure desirable properties.

With user-friendliness in mind, the TeLa sequence-diagram choice operator is locally defined, that is, annotated on a single lifeline. Where appropriate, we describe the properties that a test description must have in order for such an operator to be well-defined.

1.3.2.1.2 Locally-defined loop operator in one-tier scenario structures

With user-friendliness in mind, the TeLa sequence-diagram loop operator is locally defined, that is, annotated on a single lifeline. We describe the properties that a test description must have in order for such an operator to be well-defined.

1.3.2.2 TESTING-SPECIFIC LANGUAGE CONSTRUCTS

We define the necessary test-specific constructs, such as a construct to denote an unknown value in the parameter of a message sent from the SUT and the assignment of this value to a dynamic variable.

1.4 Application of TeLa: test synthesis using Umlaut/TGV

We extend the test synthesis method using Umlaut/TGV to achieve full integration in UML, firstly, through the use of UML scenario-based test objectives and UML scenario-based test cases and, secondly, through the XMI model exchange with commercial UML CASE tools. We also suggest how it could be generalised from system testing to component testing.

In the case of the Umlaut simulator, we refine the tool support and suggest improvements to it, while at the same time clarifying, firstly, the semantics it implements and, secondly, the constraints the derivation of this semantics from a UML model imposes on that model.

1.5 Conceptual framework

We define the conceptual framework in which our scenario-based component test description language is to be inserted, leading to a glossary of component-testing terms. As yet, there is no commonly-accepted conceptual basis and terminology in this field. Though the definitions we provide are inspired by telecom testing terminology and standard software testing terminology, they are original. An earlier version of our glossary of terms served as input to the UTP work.

2 Future work

2.1 Semantics of TeLa

It would be of interest to develop the work on a formal semantics for TeLa of Chapter 4, in particular, to define the mapping from syntax to semantics passing via the UML 2.0 metamodel and the constructs of the recently-defined UML Testing Profile, where possible. Evidently, defining a non-interleaving semantics with symbolic data is a daunting task and the first step would be to complete the non-interleaving semantics for the enumerated data case.

The development of the full formal semantics should include the incorporation of the explicit verdicts and the explicit default alternative. It could also include the formalisation of the local controllability notions mentioned in Section 2.2.1.2.3 of Chapter 4.

The definition of structural consistency could be extended to allow lifeline decompositions which result assert and assign internal actions straddling multiple lifelines, or in which an assign internal action can be decomposed into a message exchange and subsequent assignment.

2.2 Syntax of TeLa

The concrete syntax of TeLa was heavily influenced by the contemporaneous needs of the COTE project. As a consequence, the auto-invocations syntax for the TeLa sequence-diagram loop and choice operators is certainly not ideal and could definitely be improved.

The actual concrete syntax used is largely irrelevant to the work reported on here but any serious use of TeLa would probably require its modification. This is particularly true now that the exact syntax on which the language was based has become obsolete due to UML 1.4/UML 1.5 being superseded by UML 2.0.

2.3 Tool support

There are many possible extensions of this work concerning the use of Tela and tool support for this use. For tool support, the study of the computational complexity of the conditions used to define test descriptions and test cases is crucial. Below we indicate the most significant of these conditions.

Firstly, it is important to study the possibilities for automatic detection of when a test description is well-defined. That is, for a test description with no explicit default alternatives and no explicit verdicts, or with only explicit inconclusive verdicts on test reception, minimal determinism together with the extra condition on resolution of determinism by controllable actions, as described in Section 3.4.4.3 of Chapter 2. For a test description with explicit default alternatives and/or other types of explicit verdicts, the test description must define a parallel test case.

Secondly, it would be of interest to study the possibilities for automatic detection of when a test description defines each of the types of test case defined in Section 2.2.3.2.2 of Chapter 4: parallel test cases, externally-coherent parallel test cases, coherent parallel test cases,

externally-centralisable test cases and centralisable test cases. It would also be of interest to study the possibilities for CASE tool support in reducing tester concurrency. That is, in converting a parallel test case into an externally-coherent parallel test case, an externally coherent parallel test case into a coherent parallel test case, a coherent parallel test case into an externally-centralisable parallel test case and externally-centralisable parallel test case into a centralisable test case.

Thirdly, it also important to study the automatic detection of when a two-tier scenario structure is in semi-normal or normal form, as defined in Section 3.4.1 of Chapter 3.

Some tool guidance could also be provided for choosing the components whose internal communications are to conform to the message-based semantics and the components which are to be annotated as active.

2.4 Extensions to TeLa

One of the most important general extensions to TeLa would be to incorporate some treatment of time, preferably in the form of time constraints, as in the UML Testing Profile proposal, rather than in the form of explicit timers as in MSCs.

Bibliography

[Abr87] S. Abramsky. “Observational Equivalence as a Testing Equivalence”. *Theoretical Computer Science*, 53 (3), Elsevier 1987.

[AerJen03] L. Van Aertryck and T. Jensen. “UML-CASTING: Test synthesis from UML models using constraint resolution”. Proc. Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL’2003). Rennes, France, Jan. 2003. See http://www.irisa.fr/manifestations/2003/AFADL/

[Agedis] Agedis project. http://www.agedis.de/

[AlgLejHug93] Y. Algayres, F. Lejeune and R. Nahm . “The AVALON project: A Validation Environment for SDL/MSC Descriptions”. In *SDL´93: Using Objects*. Proc. 6th SDL Forum, Darmstadt, Germany, Oct. 1993. Elsevier Science Publishers/North Holland, 1993.

[AluHolPel96] R. Alur, G. Holzmann and D. Peled . “An analyser for Message Sequence Charts”. *Software Concepts and Tools* 17 (2), 1996.

[AluYan99] R. Alur and M. Yannakakis. “Model checking of message sequence charts”. In *CONCUR’99 – Concurrency Theory*. Proc 10th International Conference on Concurrency Theory (CONCUR), Eindhoven, The Netherlands, August 1999. LNCS 1664. Springer-Verlag 1999.

[AtkBayBun02] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua D. Muthig, B. Peach, J. Wust and J. Zettel. *Component-Based Product Line Engineering with UML*. Addison-Wesley 2002

[BaeMau94] J.C.M. Baeten and S. Mauw. “Delayed choice: an operator for joining Message Sequence Charts”. In D. Hogrefe and S. Leue (eds.), *Formal Description Techniques VII*. Proc. 7th International Conference on Formal Description Techniques (FORTE). Chapman-Hall, 1994.

[BakBriJer02] P. Baker, P. Bristow, C. Jervis, D. King, B. Mitchell. “Automatic generation of conformance tests from Message Sequence Charts”. In E. Sherratt (ed.) *Telecommunication and Beyond: the Broader Applicability of SDL and MSC*. Proc. 3rd SAM (SDL and MSC) Workshop (SAM2002), Aberystwyth, UK, Jun. 2002. LNCS 2599. Springer-Verlag 2002.

[BakRudSch01] P. Baker, E. Rudolph, I. Schieferdecker. “Graphical test specification – the graphical format of TTCN-3” In R. Reed and J. Reed (eds.), *SDL 2001 : Meeting UML*. Proc. 10th International SDL Forum, Copenhagen, Denmark 2001. LNCS 2078. Springer-Verlag 2001.

[BecGam98] K. Beck, E. Gamma. “Test infected: programmers love writing tests”. *Java Report* 3(7) 1998.

[Bei90] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Co., 1990.

[Bin00] R.V. Binder. *Testing Object-Oriented Systems : Models, Patterns and Tools*. Addison-Wesley, 2000.

[Boe75] B.W. Boehm. “The high cost of software”. In E. Horowitz (ed.), *Practical Strategies for Developing Large Software Systems*. Addison-Wesley, 1975.

[BonMauBou01] P. Bontron, O. Maury, L. du Bousquet, Y. Ledru, C. Oriat and M.L. Potet. “TOBIAS: un environnement pour la création d’objectifs de test à partir de schémas de test”. Proc. Software and Systems Engineering and their Applications (ICSSEA 2001). 2001

[BonPot03] P. Bontron and M.L Potet. "Stratégie de couverture de test à un haut niveau d'abstraction". Proc. Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2003). Rennes, France, Jan. 2003.
See http://www.irisa.fr/manifestations/2003/AFADL/

[BriTre00] E. Brinksma, J. Tretmans. "Testing Transition Systems: An Annotated Bibliography" In F. Cassez, C. Jard, B. Rozoy, M. Dermot (eds.), *MOdelling and VErification of Parallel Processes*. Proc. MOVEP 2000 Nantes, France. LNCS 2067. Springer-Verlag 2001.

[Bro02]. M. Broy. "Message Sequence Charts in the Development Process – Roles and Limitations". *Electronic Notes in Theoretical Computer Science* 65, 2002.

[BSI98] British Standards Institute (BSI). *Glossary of Software Testing Terms*. BS 7925-1. BSI 1998. Prepared by the British Computer Society. See also http://www.testingstandards.co.uk/bs_7925-1_online.htm and http://www.testingstandards.co.uk/living_glossary.htm.

[CavLeeMac97] A. Cavalli, B. Lee and T. Macavei.. "Test generation for the SSCOP-ATM networks protocol". In A. Cavalli and A. Sarma (eds), *SDL'97: Time for Testing – SDL, MSC and Trends.* Proc. 8th SDL Forum, Evry, Amsterdam, North-Holland, 1997.

[ChaMatTel96] B. Charron-Bost, F. Mattern and G. Tel. "Synchronous, asynchronous and causally ordered communication". *Distributed Computing*, 9(4). Springer-Verlag 1996.

[ClaJérRus01] D. Clarke, T. Jéron, V. Rusu and E. Zinovieva. "STG: A tool for generating symbolic test programs and oracles from operational specifications". In A. Min Tjoa and V. Gruhn (eds.). *Proc Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9).* ACM Press 2001.

[ComPic94] P. Combes, S. Pickin. "Formalisation of a user view of network and services for feature interaction detection". In L.G. Bouma and H. Velthuijsen (eds.). *Feature Interactions in Telecommunications Systems*. Proc. of the 1st Feature Interaction Workshop, Amsterdam, Holland, 1994. IOS Press 1994.

[ComPicRen95] P. Combes, S. Pickin, B. Renard and F. Olsen. "MSCs to express service requirements as properties on an SDL model: application to service interaction detection". In R. Braek and A. Sarma (eds.), *SDL´95 with MSCs in CASE*. Proc. 7th SDL Forum, Oslo, Norway, Sep. 1995. Elsevier, 1995.

[DaiGraNeu02] Z. R. Dai, J. Grabowski and H. Neukirchen. "Timed TTCN-3. A Real-Time Extension for TTCN-3". In *Testing Internet Technologies and Services*. In Proc. the IFIP 14th International Conference on Testing of Communicating Systems (TestCom 2002). Berlin, Germany, Mar. 2002. Kluwer Academic Publishers, 2002.

[DaiGraNeu03] Z.R. Dai, J. Grabowski and H. Neukirchen. "Timed TTCN-3 Based Graphical Real-Time Test Specification". In D. Hogrefe, A. Wiles (eds.). *Testing of Communicating Systems*. Proc. 15th IFIP International Conference on Testing of Communicating Systems, Testcom 2003, Sophia Antipolis, France, May, 2003. LNCS 2644. Springer-Verlag 2003.

[DamHar01] W. Damm and D. Harel. "LSCs: Breathing Life into Message Sequence Charts". *Formal Methods in System Design* 19, 2001. Kluwer Academic Publishers 2001.

[deNicHen84] E. De Nicola and M. Henessy. "Testing Equivalences for Processes". *Theoretical Computer Science*, 34, Elsevier 1984.

[DeuTob02] P.H. Deussen and S. Tobies. "Formal Test Purposes and The Validity of Test Cases". In D. Peled, M. Vardi (eds). *Formal Techniques for Networked and Distributed Systems*. Proc. Formal Techniques for Networked and Distributed Systems (FORTE 2002). LNCS 2529. Springer-Verlag 2002.

[Ek93] A. Ek. "Verifying Message Sequence Charts with the SDT validator". In In O. Faergemand and A. Sarma (eds.). *SDL´93: Using Objects*. Proc. 6th SDL Forum, Darmstadt, Germany, Oct. 1993. Elsevier Science Publishers/North Holland, 1993.

[EkkSchGra00] R. Ekkart, I. Schieferdecker and J. Grabowski. "Development of an MSC/UML Test Format". In J. Grabowski and S. Heymer (eds.). *Formale Beschreibungstechniken für verteilte Systeme*. Proc. FBT'2000. Verlag Shaker 2000. (See http://citeseer.nj.nec.com/rudolph00development.html).

[EkkSchGra00b] R. Ekkart, I. Schieferdecker and J. Grabowski. "Hyper-MSC – a graphical representation of TTCN". In Proc. 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM´2000), Grenoble, France, Jun. 2000.
See http://www.irisa.fr/manifestations/2000/sam2000/papers.html

[Eng01] A. Engels. *Languages for Analysis and Testing of Event Sequences*. Ph.D. thesis, Eindhoven University of Technology, Holland, 2001.

[EngMauRen02] A. Engels, S. Mauw and M.A. Reniers. "A Hierarchy of Communication Models for Message Sequence Charts". *Science of Computer Programming* 44(3). Elsevier North-Holland 2002.

[ETSI01] European Telecommunications Standards Institute (ETSI). *Methods for Testing and Specification (MTS); Methodological approach to the use of object-orientation in the standards making process.* ETSI Guide EG 201 872, V1.2.1 (2001-08). ETSI 2001.

[ETSI03a] European Telecommunications Standards Institute (ETSI). *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3*. ES 201 873, Parts 1 to 6, V2.2.1 (2003-2). ETSI Feb. 2003. Also ITU-T Recommendation Z.140-142. See also: http://www.etsi.org/frameset/home.htm?/ptcc/ptccttcn3.htm

[ETSI03b] European Telecommunications Standards Institute (ETSI). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 3: TTCN-3 Graphical Presentation Format (GFT), ETSI ES 201 873-3, V2.2.1 (2003-2). ETSI Feb. 2003. Also ITU-T Recommendation Z.142.

[FerGarKer96] J.C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier and M. Sighireanu. "CADP: a protocol validation and verification toolbox". In A. Alur and T. Henzinger (eds). *Computer Aided Verification*. Proc. 8th International Conference CAV'96, New Brunswick, New Jersey, USA, Jul. 1996. LNCS 1102. Springer-Verlag, 1996.

[FerJarJér96] J.C. Fernandez, C. Jard, T. Jéron and G. Viho. "Using on-the-fly vérification techniques for the generation of test suites". In A. Alur and T. Henzinger (eds). *Computer Aided Verification*. Proc. 8th International Conference CAV'96, New Brunswick, New Jersey, USA, Jul. 1996. LNCS 1102. Springer-Verlag 1996.

[GamHelJoh94] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series 1994.

[Gau95] M.C. Gaudel. “Testing can be formal too”. In P.D. Mosses, M. Nielsen and M.I. Schwartzbach (eds.) *TAPSOFT’95: Theory and Practice of Software Development*. LNCS 915. Springer-Verlag 1995.

[Goe01] S. Goeschl. “JUnit++ Testing Tool”. *Dr. Dobb’s Journal*, Feb. 2001. (See also http://www.junit.org/).

[Gog01] N. Goga. “Comparing TorX, Autolink, TGV and UIO test algorithms”. In R. Reed and J. Reed (eds.), *SDL 2001 : Meeting UML*. Proc. 10th International SDL Forum, Copenhagen, Denmark, 2001. LNCS 2078. Springer-Verlag 2001.

[Gra94] J. Grabowski. *Test Case Generation and Test Case Specification with Message Sequence Charts*. PhD thesis, Universität Bern, 1994.

[GraGraRud01] J. Grabowski, P. Graubmann and E. Rudolph. “HyperMSCs with Connectors for Advanced Visual System Modelling and Testing”. In R. Reed and J. Reed (eds.), *SDL 2001 : Meeting UML*. Proc. 10th International SDL Forum, Copenhagen, Denmark, 2001. LNCS 2078. Springer-Verlag 2001.

[GraHogNah99] J. Grabowski, D. Hogrefe and R. Nahm. “Test case generation with test purpose specification by MSCs”. In O. Faergemand and A. Sarma (eds), *SDL’93 – Using Objects*. Proc. Sixth SDL Forum, Darmstadt, 1993. North-Holland 1993.

[GraHogNus95] J. Grabowski, D. Hogrefe, I. Nussbaumer and A. Spichiger. “Test case specification based on MSCs and ASN.1”. In R. Braek and A. Sarma (eds), *SDL’95 – with MSC in CASE*. Proc. Seventh SDL Forum, Oslo, 1995. North-Holland 1995.

[GraKocSch99] J. Grabowski, B. Koch, M. Schitt and D. Hogrefe. “SDL and MSC Based Test Generation for Distributed Test Architectures”. In R. Dssouli, G.V. Bochmann and Y. Lahav (eds.). *SDL’99, The Next Millenium*. Proc. 9th SDL Forum, Montréal, Quebec, Jun. 1999. Elsevier, 1999.

[GraSchHog97] J. Grabowski, R. Scheurer, D. Hogrefe, Z. Dai. “Applying SAMSTAG to the B-ISDN protocol SSCOP”. *Testing of Communicating Systems,* Vol. 10. Proc. 10th International Workshop on Testing Communicating Systems (IWTCS´98), Cheju Island, Korea 1997. Kluwer Academic Press 1998.

[GraWal96] J. Grabowski and T. Walter. “Quality-of-Service Testing - Specifying Functional QoS Testing Requirements by using Message Sequence Charts and TTCN”. In Arbeitsberichte des Instituts für mathematische Maschinen- und Datenverarbeitung (Mathematik), Proc. 6th GI/ITG Technical Meeting on Formal Description Techniques for Distributed Systems, Jun. 1996, volume 20, No. 9.

[GraWal98] J. Grabowski, T. Walter. “Visualisation of TTCN test cases by MSCs”. In: Lahav Y., Wolisz A., Fischer L., Holz E. (eds). Proc. 1st Workshop of the SDL Forum Society on SDL and MSC (SAM’98). Informatik Berichte, Vol. 104, Humboldt-Universität zu Berlin, Germany, Jun, 1998.

[GroJerKer99] R. Groz, T. Jéron and A. Kerbrat. “Automated test generation from SDL specifications” In R. Dssouli, G. von Bochmann and G. Lahav (eds.) *SDL’99, The Next Millenium*. Proc. 9th SDL Forum, Montréal, Quebec, Jun. 1999. Elsevier, 1999.

[GroRis97] R. Groz and N. Risser. “Eight years of experience in test generation from FDTs using TVEDA”. In A. Togashi, T. Mizuno, N. Shiratori and T. Higashino. In *Formal Description Techniques and Protocol Specification, Testing and Verification*. Proc. Joint International Conference on Formal Description Techniques (FORTE X) and

Protocol Specification, Testing and Verification (PSTV XVII), Nov. 1997, Osaka, Japan. Chapman and Hall 1997.

[Gue01] A. Le Guennec. *Génie Logiciel et Méthodes Formelles avec UML. Spécification, Validation et Génération de Tests*. Ph.D. Thesis, Université de Rennes 1, 2001.

[Hau01] O. Haugen. "From MSC-2000 to UML 2.0 – the future of sequence diagrams". In R. Reed and J. Reed (eds.), *SDL 2001 : Meeting UML*. Proc. 10th International SDL Forum, Copenhagen, Denmark 2001. LNCS 2078. Springer-Verlag 2001.

[Hél01] L. Hélouët. "Some pathological Message Sequence Charts, and how to detect them". In R. Reed and J. Reed (eds.), *SDL 2001 : Meeting UML*. Proc. 10th International SDL Forum, Copenhagen, Denmark, 2001. LNCS 2078. Springer-Verlag 2001.

[Hél03] L. Hélouët. "Projections et comparaisons de scénarios". Proc. Approaches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2003), Rennes, France, Jan. 2003. See http://www.irisa.fr/manifestations/2003/AFADL/

[HélJar01] L. Hélouet and C. Jard. "Etat de l'art sur les langages de scénarios". In G. Juanole and R. Valette. *Modélisation des systèmes réactifs (Actes de MSR 2001)*. Proc. Colloque francophone sur la modélisation des systèmes réactifs (MSR'2001), Oct. 2001. Hermès Science Publications, 2001.

[HélJarCai02] L. Hélouët, C. Jard and B. Caillaud. "An event structure based semantics for high-level message sequence charts". *Mathematical Structures In Computer Science* 2002, vol. 12. Cambridge University Press 2002.

[HélLeM01] L. Hélouët, P. Le Maigat. "Decomposition of message sequence charts". In Proc. 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM´2000), Grenoble, France, Jun. 2000. See http://www.irisa.fr/manifestations/2000/sam2000/papers.html

[HenBar99] B. Henderson-Sellers, F. Barbier. "Black and White Diamonds". In R.B. France, B. Rumpe (eds*.), UML'99 : The Unified Modeling Language – Beyond the Standard*. Proc. 2nd International Conference, Fort Collins, CO, USA, Oct. 1999. LNCS 1723. Springer-Verlag 1999.

[Hey98] S. Heymer. "A non-interleaving semantics for MSCs". In Y. Lahav, A. Wolisz, L. Fischer and E. Holz (eds.), Proc. 1st Workshop of the SDL Forum Society on SDL and MSC (SAM'98). Informatik Berichte, Vol. 104, Humboldt-Universität zu Berlin, 1998.

[HoJézGue99] W.M. Ho, J.M. Jézéquel, A. Le Guennec and F. Pennaneac'h. "Umlaut: an extendible UML transformation framework". Proc. Automated Software Engineering Conference (ASE'99). IEEE Computer Society 1999.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[IEEE90] *IEEE Standard Glossary of Software Engineering Technology*. IEEE Std. 610 12-1990.

[ISO89] Information Processing Systems (ISO). - Open System Interconnection - *LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Vol. DIS* 8807, ISO/EC 1989.

[ISO/IEC92a] ISO/IEC: *Information Technology - Open Systems Interconnection - Conformance Testing Methodology and Framework - Part 1: General Concepts*, ISO/IEC International Standard 9646. 1992.

[ISO/IEC92b] ISO/IEC: *Information Technology - Open Systems Interconnection - Conformance Testing Methodology and Framework - Part 3: The Tree and Tabular Combined Notation (TTCN)*. ISO/IEC International Standard 9646. 1992.

[ISO95] ISO/IEC JTC1/SC21/WG7. *Basic Reference Model of Open Distributed Processing*. ISO/IEC 10746. ISO 1995. Also ITU-T Recommendation X.901.

[ITU-T97] International Telecommunications Union – Telecommunication Standardization Sector (ITU-T). *Framework on Formal Methods in Conformance Testing*. Series Z: Languages and general software aspects for telecommunication systems, Formal Description Techniques (FDT), Recommendation Z.500. ITU-T, May 1997.

[ITU-T98] International Telecommunications Union – Telecommunication Standardization Sector (ITU-T). *Formal Semantics of Message Sequence Charts*. Series Z: Languages and general software aspects for telecommunication systems, Formal Description Techniques (FDT), Recommendation Z.120, Annexe B. ITU-T, Apr. 1998.

[ITU-T99] International Telecommunications Union – Telecommunication Standardization Sector (ITU-T). *Message Sequence Chart*. Series Z: : Languages and general software aspects for telecommunication systems, Formal Description Techniques (FDT), Recommendation Z.120 (MSC2000). ITU-T, Nov. 1999. See also *Message Sequence Chart. Corrigendum 1*. ITU-T, Dec. 2001

[ITU-T02] International Telecommunications Union – Telecommunication Standardization Sector (ITU-T). *Specification and Description Language*. Series Z: Languages and General Software Aspects for Telecommunications Systems. Formal Description Techniques (FDT) Recommendation Z.100 (SDL2000). ITU-T, Aug. 2002.

[Jar02] C.Jard. “Synthesis of distributed testers from true-concurrency models of reactive systems”. *Journal of Information and Software Technology*. Elsevier 2003.

[JarJér02] C. Jard and T. Jéron. “TGV: theory, principles and algorithms”. Proc. 6th world conference on Integrated Design and Process Technology (IDPT 2002), Pasadena (CA), USA. Society for Design and Process Science 2002.

[JarPic01] C. Jard and S. Pickin. “COTE – Component Testing using the Unified Modelling Language” *ERCIM News*, No. 48. ERCIM EEIG 2001.
See also http://www.irisa.fr/cote/.

[Jér02] T. Jéron. “TGV : théorie, principes et algorithms; un outil de synthèse automatique de tests de conformité pour les systèmes réactifs”. *Téchnique et Science Informatique*, Vol. 21, 2002.

[JérJézGue98] T. Jéron, J.M. Jézéquel and A. Le Guennec. “Validation and test generation for object-oriented distributed software”. In Bernd Krämer, Naoshi Uchihira, Peter Croll and Stefano Russo. *Proc. International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE’98)*. IEEE Computer Society 1998.

[KatLam98] J.P. Katoen, L. Lambert. “Pomsets for Message Sequence Charts”. In Y. Lahav, A. Wolisz, L. Fischer and E. Holz (eds), Proc. First Workshop of the SDL Forum Society on SDL and MSC (SAM’98). Informatik Berichte, Vol. 104, Humboldt-Universität zu Berlin, 1998.

[Kru00] I.H. Kruger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technical University of Munich, Germany. 2000.

[LadLeu95] P.B. Ladkin and S. Leue. “Interpreting Message Flow Graphs”. *Formal Aspects of Computing*, 7 (5), 1995.

[Lam78] L. Lamport. “Time, clocks and the ordering of events in distributed systems”. *Communications of the ACM* 21(7). Jul. 1978

[Lam86] L. Lamport. “On Interprocess Communication”. *Distributed Computing* 1,2 1986. Springer-Verlag 1986.

[LeeYan96] D. Lee and M. Yannakakis. “Principles and Methods of Testing Finite State Machines – A Survey”. In *Proceedings of the IEEE* 84(8). IEEE Computer Society, 1996.

[LeGArn96] P. Le Gall and A. Arnould. “Formal specifications and test: correctness and oracle” In O.J. Dahl, O. Owe and M. Haveraaen (eds.), *Recent Trends in Data Type Specification*. Proc. 11th Workshop on Algebraic Data Type Specification (WADT’95), Oslo, Norway, 1995. LNCS 1130. Springer-Verlag 1996.

[LeTJerJez00] Y. Le Traon, T. Jéron, J.M. Jézéquel and P. Morel. “Efficient OO integration and regression testing”. *IEEE Trans. on Reliability*, 49(1), Mar. 2000.

[Lyn88] N.A. Lynch. “I/O automata: A model for discrete event systems”. In Proc. 22nd Conference On Information Sciences and Systems, Princeton, NJ, USA, March 1988. Also available as Tech. Rep. MIT/LCS/TM-351, Lab. for Computer Science, MIT, Cambridge, Mass.

[MauRenWil01] S. Mauw, M.A. Reniers and T.A.C. Willemse. “Message Sequence Charts in the Software Engineering Process”. In S.K. Chang (ed.). *Handbook of Software Engineering and Knowledge Engineering.* World Scientific, 2001.

[MauWijWin93] S. Mauw, van M. Wijk and T. Winter. “A formal semantics of synchronous Interworkings”. In O. Faergemand and A. Sarma (eds.). *SDL’93 – Using Objects*. Proc. 6th SDL Forum, Darmstadt, Germany, Oct. 1993. Elsevier Science Publishers/North Holland, 1993.

[Mey 88] B. Meyer. *Object-Oriented Software Construction*. 1st Edition. Prentice-Hall 1988.

[Mey 97] B. Meyer. *Object-Oriented Software Construction*. 2nd Edition. Prentice-Hall 1997.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science, 1989

[Mit01] B. Mitchell. “Characterising Concurrent Tests Based On Message Sequence Chart Requirements”. Proc. Applied Telecommunication Symposium, part of Advanced Simulation Technologies Conference (ASTC 2001). The Society for Computer Simulation International. Seattle, Washington, Apr. 2001.

[MusPel99] A. Muscholl and D. Peled. “Message Sequence Graphs and Decision Problems on Mazurkiewicz Traces”. In M. Kutylowski, L. Pacholski, T. Wierzbicki. (eds.): *Mathematical Foundations of Computer Science 1999* Proc. MFCS’99. LNCS 1672. Springer-Verlag 1999.

[Nah94] R. Nahm. *Conformance Testing Based on Formal description Techniques and Message Sequence Charts*. PhD thesis, Universität Bern, 1994.

[NebPicTra02] C. Nebut, S. Pickin, Y. Le Traon and J.M. Jézéquel. “Re-usable test requirements for UML-modelled product lines” In B. Geppert and K. Schmid (eds.). Proc. *International Workshop on Requirements Engineering for Product Lines (REPL’02)*. Avaya Labs Report ALR-2002-033.
See also http://www.research.avayalabs.com/techreport/ALR-2002-033-paper.pdf

[NIS96] National Institute of Standards and Technology (NIST). *Reference Information for the Software Verification and Validation Process*. NIST special publication 500-234. 1996.
See also http://hissa.ncsl.nist.gov/HHRFdata/Artifacts/ITLdoc/234/val-proc.html

[NIS03] National Institute of Standards and Technology (NIST).
See http://www.expect.nist.gov/

[ObeKer99] I. Ober and A. Kerbrat. "Specification and execution of tests using MSC". In J. Wu, S. Chanso and Q. Gao (eds), *Formal Methods for Protocol Engineering and Distributed Systems*. Proc. FORTE/PSTV´99, Beijing, China. Kluwer Academic Publishers, 1999.

[OG03] The Open Group. See http://tetworks.opengroup.org/

[OMG01] Object Management Group (OMG): *Unified Modelling Language Specification, version 1.4*. OMG, Needham, MA, USA. Sep. 2001.

[OMG02a] Object Management Group (OMG): *CORBA Components, version 3*. OMG, Needham, MA, USA. Jun. 2002.

[OMG02b] Object Management Group (OMG): *Common Object Request Broker Architecture: Core Specification, version 3*. OMG, Needham, MA, USA. Dec. 2002.

[OMG03] Object Management Group (OMG): *Unified Modelling Language Specification, version 1.5*. OMG, Needham, MA, USA. Mar. 2003.

[Pel02] D. Peled. "Specification and verification using Message Sequence Charts" *Electronic Notes in Theoretical Computer Science* 65 No. 7, 2002.

[Pha94] M. Phalippou. *Relations d'Implantations et Hypothèses de Test sur les Automates*. Ph.D. thesis, Université de Bordeaux, France. 1994.

[PicJarHeu01] S. Pickin, C. Jard, T. Heuillard, J.M. Jézéquel and P. Desfray. "A UML-integrated test description language for component testing". In A. Evans, R. France, A. Moreira and B. Rumpe (eds.), *Practical UML-Based Rigorous Development Methods*. Lecture Notes in Informatics (GI series), Vol. P7. Kollen-Druck + Verlag, 2001.

[PicJarTra02] S. Pickin, C. Jard, Y. Le Traon, T. Jéron, J.M. Jézéquel and A. Le Guennec. "System test synthesis from UML models of distributed software". In D. Peled, M. Vardi (eds). *Formal Techniques for Networked and Distributed Systems*. Proc. Formal Techniques for Networked and Distributed Systems (FORTE 2002). LNCS 2529. Springer-Verlag 2002.

[PicSanYel96] S. Pickin, C. Sanchez, J.C. Yelmo, J.J. Gil and E. Rodríguez. "Introducing formal notations in the development of object-based distributed applications". In E. Najm and J.B. Stefani (eds.), *Formal Methods in Open Object-based Distributed Systems*. Proc. FMOODS'96. Chapman and Hall, 1996.

[PicSanSan96] S. Pickin, C. Sanchez, J. Sanchez, J.C. Yelmo, J.J. Gil and E. Rodríguez. "An approach to the validation of open object-based distributed applications". In B. Baumgarten, H.J. Burkhardt and A. Giessler (eds.), *Testing of Communicating Systems*. Proc. IWTCS'96. Chapman and Hall, 1996.

[Pra86] V. Pratt. "Modelling concurrency with partial orders". *International Journal of Parallel Programming*, 15:33-71, 1986.

[Rat01] Rational Software. *Component Testing with Rational Quality Architect*. White paper, Rational Software 2001.
(See http://www.rational.com/products/whitepapers/rose_rqa.jsp)

[Ren97] A. Rendón. *Prueba Incremental de Modelos de Sistemas de Tiempo Real*. PhD thesis, Escuela Técnica Superior de Ingenieros de Telecomunicación, Universidad Politécnica de Madrid, España, 1997.

[Ren99] M.A. Reniers. *Message Sequence Chart. Syntax and Semantics*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1999.

[Ren96] A. Rensink. "Denotational, Causal, and Operational Determinism in Event Structures". In H. Kirchner (ed.), *Trees in Algebra and Programming* (CAAP '96), LNCS 1059. Springer-Verlag, 1996.

[RobKheGro97] G. Robert, F. Khendek and P. Grogono. "Deriving an SDL specification with a given architecture from a set of MSCs". In A. Cavalli and A. Sarma (eds), *SDL'97 : Time for Testing - SDL, MSC and Trends*. Proc. 8th SDL Forum, Evry, France, Sep. 1997. Elsevier, 1997.

[RudGraGra99] E. Rudolph, J. Grabowski, P. Graubmann. "Towards a harmonisation of UML sequence diagrams and MSC". Dssouli R., Bochmann G.V., Lahav Y. (eds.): *SDL'99, The Next Millenium*. Proc. 9th SDL Forum, Montréal, Quebec, Jun. 1999. Elsevier 1999.

[RumBlaPre91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall 1991.

[RusBouJér00] V. Rusu, L. du Bousquet and T. Jéron. "An Approach to Symbolic Test Generation". In W. Grieskamp, T. Santen, B. Stoddart (eds.) *Integrated Formal Methods*. Proc. IFM'00. LNCS 1945. Springer-Verlag 2000.

[SanHey94] "ADL – an interface definition language for specifying and testing software". *ACM SIGPLAN notices*, 29(8), Jan. 1994.

[SasNieWin96] A. Sassone, M. Nielsen and G. Winskel. "Models for concurrency: towards a classification". *Theoretical Computer Science* (*TCS*) 170(1–2). Elsevier 1996.

[SchDaiGra03] I. Schieferdecker, Z. R. Dai, J. Grabowski, A. Rennoch. "The UML 2.0 Testing Profile and its Relation to TTCN-3". In D. Hogrefe, A. Wiles (eds.). *Testing of Communicating Systems*. Proc. 15th IFIP International Conference on Testing of Communicating Systems, Testcom 2003, Sophia Antipolis, France, May 2003. LNCS 2644. Springer-Verlag 2003.

[SchRudGra00] I. Schieferdecker, E. Rudolph and J. Grabowski. "Test Development with MSC-2000". Tutorial notes of the 13th IFIP International Workshop on Testing Communicating Systems (TestCom 2000), Ottawa, Sep. 2000.

[SchEbnGrab00] M. Schmitt, M. Ebner, J. Grabowski "Test generation with Autolink and Test Composer". Proc. 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM´2000), Grenoble, France, Jun. 2000.
See http://www.irisa.fr/manifestations/2000/sam2000/papers.html

[SchEkGra98] M. Schmitt, A. Ek, J. Grabowski, D. Hogrefe and B. Koch. "Autolink – putting SDL-based test generation into practice". In A. Petrenko (ed.) *Testing Communicating Systems, Vol. 11*. Proc. 11th International Workshop on Testing Communicating Systems (IWTCS´98), Tomsk, Russia 1998. Kluwer Academic Press 1998.

[Tar72] R. Tarjan. “Depth-first search and linear graph algorithms”. *SIAM Journal of Computing*, 1, 1972.

[Tre96] J. Tretmans. “Test generation with inputs, outputs and repetitive quiescence”. *Software – Concepts and Tools*, 17 (3), 1996.

[Tsc02] V. Tschaen. “Compositionality issues in test Synthesis”. Proc. Modelling and Verifying Parallel Processes Summer School (MOVEP'02), 2002.

[U2P03] U2 Partners: *Unified Modelling Language*: *Superstructure*, *version 2.0*. 2nd revised submission to Object Management Group (OMG) RFP ad/00-09-02. OMG doc #ad/03-01-02, Jan. 2003.
Now available from OMG, as final adopted specification ptc/03-08-02
See http://www.omg.org/cgi-bin/doc?ptc/2003-08-02

[UTP03] U2TP (UML Testing Profile) consortium: *UML Testing Profile (Final Submission)*. Final submission to Object Management Group (OMG). Mar. 2003.
See http://www.fokus.gmd.de/u2tp/Related_Documents/related_documents.htm.
Now available from OMG, as draft adopted specification ptc/03-07-01
See http://www.omg.org/cgi-bin/doc?ptc/2003-07-01

[W3C02] W3C: *Web Services Architecture*. Working Draft. WC3 Aug. 2003.
See http://www.w3.org/2002/ws/.

[Win87] G. Winskel. “Event structures”. In G. Rozenberg, W. Brauer, and W. Reissing (eds), *Petri nets: Applications and relationships to other models of concurrency*, LNCS 255. Springer-Verlag, 1987.

Appendix A : Grammar of .aut files used by TGV/Umlaut

A1 Grammar of .aut input in system test synthesis

```
<axiom> ::= des (<initial_state>, <num_transitions>, <num_states>)
<transition>*
<initial_state> ::= 0
<num_transitions> ::= <integer>
<num_states> ::= <integer>
<transition> ::= (<from_state>, "<label>", <to_state>)
<from_state> ::= <state>
<to_state> ::= <state>
<state> ::= <integer>
<label> ::=   <reg_exp>
            | *
            | ACCEPT
            | REFUSE
```

A1.1 Constraints

- `<num_transitions>` is positive or zero.
- `<num_states>` is strictly positive.
- `<state>` takes a value between `<initial_state>` and `<num_transitions>-1`.
- The labels `ACCEPT` and `REFUSE` can only be used on transitions which are the only outgoing transitions of a state and which furthermore are loop transitions; the accept and reject states must be modelled in this way since no state information is allowed in an LTS.

A1.2 Observations

- `<reg_exp>` is a regular expression using the Unix *regexp* syntax; we suppose that the labels are to match those generated from the UML specification (see grammar on following page and also Chapter 6, Sections 2.3.5 and 3.2.2).
- the `*` is used to denote the complement of the existing outgoing transitions in the `<from_state>` of the `<transition>` in which it occurs; it is commonly used in a transition to a reject state in order to enforce a strict ordering.
- We suppose that we are performing system testing, the *.hide* and *.io* files are as given in Chapter 6, Section 2.4 and the guidelines for the specification of actor state machines of Section 2.1.2.7 of this same chapter have been followed. In this case the `<action expression>` of the `<actor_triggerless_trans_label>` will always be of the form `<partner>.<op_call>` rather than of the form `<more_complex_expression>`
- The `REFUSE` transition is usually referred to via the term “reject”

A2 Grammar of .aut output in system test synthesis

```
<axiom> ::= des (<initial_state>, <num_transitions>, <num_states>) <transition>*
<initial_state> ::= 0
<num_transitions> ::= <integer>
<num_states> ::= <integer>
<transition> ::= (<from_state>, "<label>", <to_state>)
<from_state> ::= <state>
<to_state> ::= <state>
<state> ::= <integer>
<label> ::=   <actor_triggerless_trans_label>
            | <actor_triggered_trans_label>
            | <timer_event>
<actor_triggerless_trans_label>::= <owner>!<action_expression>; OUTPUT [<verdict>]
<actor_triggered_trans_label>::= <actor_name>?<op_call>; INPUT [<verdict>]
<action_expression> ::=   <object_role>.<op_call>
                        | <more_complex_expression>
<timer_event> ::=   timeout <timer>; INPUT [<verdict>]
                  | start <timer>
                  | cancel <timer>
<op_call> ::= <op_name>([<param_list]>)
<param_list> ::= <param>|<param>,<param_list>
<actor_name> ::= <identifier>
<object_role> ::= <identifier>
<op_name> ::= <identifier>
<param> ::= <identifier>
<more_complex_expression> ::= <char_string>
<timer> ::= TAC | TNOAC
<verdict> ::= PASS | (PASS) | FAIL | INCONCLUSIVE | (INCONCLUSIVE)
```

A2.1 Constraints

- <num_transitions> is positive or zero.
- <num_states> is strictly positive.
- <state> takes a value between <initial_state> and <num_transitions>-1.
- <param> defines a unique value (a constant).
- <verdict>, in the case PASS, INCONCLUSIVE and FAIL, can only be given for a transition to a sink state. This is also the case for the non-definitive verdicts (PASS) and (INCONCLUSIVE), if no postamble has been computed. A verdict after an OUTPUT can only be PASS or (PASS).

A2.2 Observations

- We suppose that we are performing system testing, the *.hide* and *.io* files are as given in Chapter 6, Section 2.4 and the guidelines for the specification of actor state machines of Section 2.1.2.7 of this same chapter have been followed. In this case the <action expression> of the <actor_triggerless_trans_label> will always be of the form <partner>.<op_call> rather than of the form <more_complex_expression>
- The tag INPUT or OUTPUT is obtained by performing the mirror operation on the direction (as specified in the *.io* file) of the external (as specified in the *.hide* file) Σ-actions; it refers to the point of view of the tester.

- TAC is used to check that the implementation does not remain silent when it should respond.
- TNOAC is used to check that the implementation does not respond when it may or should remain silent.
- For a test case, the transitions from a state are either all inputs or all outputs: a test case is a controllable test graph.
- A default transition to fail exists in states awaiting inputs.
- There are no guards in the output produced by TGV since all data is enumerated.

Resumé : Nous affirmons le besoin d'un langage intégré dans UML pour décrire des test fonctionnels de composants, et nous proposons un tel langage, TeLa, fondé sur les diagrammes de séquence UML, où l'architecture de test se décrit avec des diagrammes de composant UML. Nous formalisons la base de la sémantique non-entrelacée de TeLa, et les notions de déterminisme / contrôlabilité, complétude en entrée, verdicts implicite / explicite et alternative par défaut. La sémantique par projection sur les événements du testeur permet l'usage d'une syntaxe avec lignes de vie pour le SUT. L´architecture de test fournit un cadre pour définir la décomposition des lignes de vie et des propriétés de composant telles que la sémantique de communication et le schéma de flôt de contrôle. Nous définissons les concepts d'une description de test bien fondée, d'un cas de test centralisable et de quatre types de cas de test parallèle. Nous expérimentons l'utilisation de TeLa dans la synthèse de test avec Umlaut/TGV.

Mots clés: testing, test de logiciel, test de composants, test reparti, test télécom, sémantique non-entrelacée, déterminisme, controllabilité, architecture de test, langages graphiques, langages de description de test, UML, diagrammes de séquence, MSC, scénarios, modèles formels, synthèse de test, automatisation de test.

Abstract: We argue the merits of a UML-integrated language for describing black-box tests of possibly-distributed components and propose such a language, TeLa, based on UML sequence diagrams, with UML component diagrams to describe the test architecture. We formalise the basis of a non-interleaving semantics for TeLa, including notions of determinism / controllability, input completeness, implicit / explicit verdicts, and default alternatives. The semantics by projection onto tester events allows the syntax to include SUT lifelines. The hierarchical component model of the test architecture provides a framework for defining lifeline decomposition and component-based properties such as communication semantics and control flow scheme, the latter being viewed as a restriction on allowed linearisations. We define the concepts of well-defined test description, centralisable test case and four types of parallel test case. We also experiment with the use of the TeLa in test synthesis using Umlaut/TGV.

Keywords: software testing, component testing, distributed testing, telecom testing, non-interleaving semantics, determinism, controllability, test architecture, graphical languages, test description languages, UML, sequence diagrams, MSC, scenarios, formal models, test synthesis, test automation.