Using UML sequence diagrams as the basis for a formal test description language *

Simon Pickin[1] and Jean-Marc Jézéquel[2]

[1] Dpto. de Ingeniería Telemática, Universidad Carlos III de Madrid, Spain
Simon.Pickin@it.uc3m.es
[2] IRISA, Campus de Beaulieu, Université de Rennes, France

Abstract. A formal, yet user-friendly, test description language could increase the possibilities for automation in the testing phase while at the same time gaining widespread acceptance. Scenario languages are currently one of the most popular formats for describing interactions between possibly distributed components. The question of giving a solid formal basis to scenario languages such as MSC has also received a lot of attention. In this article, we discuss using one of the most widely-known scenario languages, UML sequence diagrams, as the basis for a formal test description language for use in the distributed system context.

1 Introduction

Testing is crucial to ensuring software quality and the testing phase absorbs a large proportion of development costs. Despite this fact, testing remains more of a craft than a science. As a result, the productivity gains to be obtained from a more systematic treatment of testing, and from the consequent greater level of automation in the testing phase, are potentially very large. The use of a formal test description language is a key part of such a systematic treatment, as explained in [24].

One of the main benfits of such a language is the ability to abstract away from the less important detail. As well as being crucial to managing complexity, abstraction is the means by which platform-independent descriptions are made possible. Graphical analysis and design languages are at the heart of recent moves to raise the level of abstraction in usual software development practice. However, the lack of a formal basis to the most widely-used of these languages, such as UML, limits the extent to which they can be used to facilitate automation. Among such graphical languages, scenario languages are becoming popular for describing interactions between possibly-distributed components, due to the fact that they present the communications and the temporal orderings between them

* This work was initiated in the COTE project of the French RNTL research programme during Simon Pickin's stay at IRISA
© *Springer-Verlag*

in a clear and intuitive fashion. They would therefore seem to be the notation of choice for describing tests in which the communication aspect is predominant.

In this article we discuss defining a formal test description language based on UML sequence diagrams and give an overview of such a language called *TeLa*, originally developed in the COTE project [12] (an early version of TeLa is presented in [20]). In so doing, we deal with the main semantic issues involved in using UML sequence diagrams, and scenario languages in general, as the basis for a formal language, in our case, for a test description language. Despite the importance of these issues, given the rapid uptake of UML, there is a lack of detailed analyses of them in the literature, not least in the official UML documentation. Here we provide such an analysis not only for UML 1.4/1.5 [16] sequence diagrams but also for the sequence diagrams of the upcoming UML 2.0 [17] standard.

We chose to base our language on UML in order to increase the chances of the work having some industrial impact, notably in widely-used development processes and CASE tools. In addition, this choice should make testing more accessible, not only due to the user-friendly syntax but also since: for a System Under Test (SUT) that is an implementation of a model designed in UML, if the test language is also based on UML, the relation between the tests and (part of) the design model is more manifest. In the component testing context, this accessibility also facilitates the use of tests as component documentation. This documentation can be viewed as a type of constructive contract, as an installation aid, as a regression testing aid in case of changes in the implementation of the component or of its environment, etc. TeLa was conceived as a test language for component-based applications. The use of TeLa as the interface to the Umlaut UML simulator and the TGV test synthesis tool in the COTE project is treated in [21].

We currently restrict our interest to black-box testing, though scenario languages could also be used for some types of grey-box testing. We aim our language at a higher level of abstraction than, for example, TTCN [6] for which a scenario-based graphical syntax has also been developed in recent years. The UML Test Profile, see [18], is testimony to the industrial interest in a UML-based test description language. The work on TeLa reported on here begun before that on the Test Profile and though the two approaches have similarities they are not currently compatible, see Section 3.4.

In Section 2 and Section 3 we analyse the suitability of UML 1.4/1.5 sequence diagrams and UML 2.0 sequence diagrams, respectively. In Section 4 we give a flavour of the TeLa language and in Section 5 we present some important aspects of its semantics. In Section 6 we draw conclusions from this work.

2 Suitability of UML 1.4/1.5 Sequence Diagrams

In this section we tackle the issue of basing a formal test description language on UML 1.4/1.5 sequence diagrams, as they are defined in the UML standard. The inconsistencies discussed in Section 2.1 oblige us to modify the semantics

as discussed in Section 2.2. The expressiveness of the language must also be increased as discussed in Section 2.3.

2.1 Semantic inconsistencies of UML 1.4/1.5 Sequence Diagrams

In the UML 1.4/1.5 specifications, the semantics of sequence diagrams is defined in terms of two relations between messages, *predecessor* and *activator*, in a manner similar to [13]. Predecessor relates a message sent by a method to earlier messages sent by the same method while activator relates a message sent by a method to the message that invoked that method. Two messages can then be said to be on the same causal flow if this pair of messages is in the transitive closure of the union of these two relations. Clearly, messages can only be ordered if they are on the same causal flow.

UML 1.4/1.5 sequence diagrams betray their origins, which lie in procedural diagrams—diagrams involving a single causal flow and in which all calls are synchronous—used to describe the behaviour of centralised OO programmes. They represent an attempt to generalise these procedural diagrams to incorporate treatment of concurrency, asynchronous calls, active objects, etc. Unfortunately, the resulting generalisation is not consistent.

In their current state, then, UML 1.4/1.5 sequence diagrams are unsuitable for use as the basis for a formal test description langauge pitched at a relatively high-level of abstraction. In the following sections we present the main problems with the UML 1.4/1.5 semantics.

The sequence numbering notation. This notation is inconsistent in the following sense. Concurrency is supposed to be modelled using the thread names and explicit predecessors part of the notation. However, this part of the notation appears to assume that messages emitted on *different* lifelines can be related via the predecessor relation, e.g. see Fig. 3-71 of [16], contradicting the semantics (in particular, the semantics of the activation part of the sequence numbering notation that uses the dot notation!).

Moreover, the activation part of this notation is unworkable except in complete causal flows. The use of guards, guarded loops and branching would make it even more unworkable.

Focus bars and asynchronous messages / active objects. It is not clear from the UML 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. However, if focus bars are not used, according to the semantics, no ordering relations can be inferred between asynchronous messages emitted on different lifelines.

There is little in the standard to justify other interpretations such as that used, without explanantion, in [5].

Lack of ordering on incomplete causal flows. Message ordering cannot be specified without complete causal flows. Sequence diagrams cannot therefore be used for specification, where abstraction is fundamental, but only for representing complete execution traces.

Lack of ordering between different causal flows. Even when representing a complete execution trace, messages on different causal flows are not ordered w.r.t. each other.

In particular, a signal is a causal sink that is only related to messages that precede it on the same causal flow. Thus, it has no temporal relation to any message emitted or received on the receiving lifeline except causal ancestors.

Similarly, a message emitted spontaneously by an active object is a causal source that is only related to messages that succeed it on the same causal flow (and even this, only if focus bars / nested sequence numbers are used, see above). Thus, it has no temporal relation to any message emitted or received on the emitting lifeline except causal descendants.

The notion of a message being "completed". The predecessor relation is said to relate "completed" messages on a lifeline. In the case of asynchronous messages, how does the sender know when a message has "completed"?

The definition of active / passive object. The vagueness of the definition of control flow scheme in UML, i.e. the concept of active/passive object, means that the exact role of this concept in sequence diagrams is not clear.

2.2 Clarifying the Semantics of Sequence Diagrams

In this section we modify the semantics of sequence diagrams to solve the problems discussed in Section 2.1.

An Interworking-Style or "Message-Based" Semantics. The most obvious solution to the problems discussed in the previous section is to remove the activator relation from the semantics and simply relate messages emitted on the same or on different lifelines via the predecessor relation[3]. The semantics is therefore defined as a partial order of messages.

There are several ways in which predecesor relations between messages could be inferred from a sequence diagram. The most intuitive solution is to use a semantics similar to that of interworkings [14], but which also allows the use of synchronous calls and focus bars. Note that such a semantics cannot be related to the UML 1.4/1.5 metamodel (i.e. the abstract syntax), since the addition of loops and branching (see below) could lead to infinite depth, and even infinite width, metamodel structures.

[3] In the absence of an activator relation, the question arises as to the role of the focus bar, apart from relating synchronous invocation messages to the corresponding reply messages. We return to this point in Section 2.3

An MSC-style or "Event-Based" Semantics. However, an interworking-style semantics is not well-adapted to distributed system description. Neither is it well-adapted to test description since, in the case of messages sent (resp. received) by the tester to (resp. from) the SUT, the tester behaviour to be described encompasses only the emission (resp. reception) of the message by the tester but not its reception (resp. emission) by the SUT.

A semantics which distinguishes emission and reception events is clearly more suitable for describing tester behaviour. We are therefore led to use a partial order of events semantics, similar to that of MSC [10] rather than a partial order of messages semantics, simliar to that of interworkings.

A Message-Based Semantics Inside an Event-Based Semantics. In order to be able to use the simplicity of the interworking-style semantics when appropriate, we define a message-based semantics as a restriction of an event-based semantics, extending the work of [4], see [19] for details. This enables us to specify when required that any diagram, or even part of a diagram, that satisfies the RSC (Realisable with Sychronous Communication) property [2], is to be interpreted using the message-based semantics.

Among the multiple uses of this facility, when modelling centralised implementations it can be used to avoid cluttering up diagrams with synchronization messages. In the COTE project, this facility was used to represent the output of the TGV test synthesis tool in sequence-diagram form, see [21].

2.3 Increasing the Expressiveness of Sequence Diagrams

As well as being ill-defined, UML 1.4/1.5 sequence diagrams are lacking constructs which are essential for a test description language of the type discussed in the introduction. Though most of these constructs could also be seen as crucial to other uses of sequence diagrams, here we do not address this wider issue but concentrate, instead, on test description. In looking for suitable constructs, we seek inspiration from the MSC standard. In the following sections we discuss both the constructs we will need to add and the existing constructs which we will need to modify. However, before doing so, we briefly discuss the significant limitations that we do not address.

Perhaps the most important of these limitations is the absence of a gate construct and of a parallel operator, unlike the situation in MSC. This is due to the complexity of their interaction with the loop operator, which would make it all too easy to produce unimplementable sequence diagrams. On the downside, the lack of a parallel operator may make it difficult to describe certain types of distributed testing scenarios involving concurrent choices. Two important limitations which are shared with MSC, namely the absence of a multi-cast construct and the inability to specify the creation of an arbitrary number of components, also deserve a mention.

Sequential Composition. Representation of large numbers of message exchanges requires a means of composing diagrams sequentially. Sequential composition is also essential for representing different alternative continuations, see the section on branching, below.

Use of the event-based semantics means that *weak sequential composition*—that used in the MSC standard—is essential for compositionality, that is, in order for behaviour to be conserved if a diagram is split into a sequence of smaller diagrams. Though weak sequential composition is the most suitable default composition mechanism, we also require a way of explicitly specifying *strong sequential composition*, that is, composition in which all events of the first diagram precede all events of the second diagram, in order to describe the sequential execution of different test cases. This is needed to model test case termination which involves a global syncronisation in order for a global verdict to be reached. This latter type of sequential composition has no counterpart in MSC.

Internal Action. Modelling component creation/destruction in the presence of lifeline decomposition requires a construct for specifying the creation of a subcomponent on a lifeline. Representing data manipulation requires a construct for specifying assertions and assignments on a lifeline. These three types of action could be placed in an internal action located at a precise position on a lifeline in a similar way to the MSC internal action. Semantically, lifeline termination is also viewed as an internal action. Finally, an "escape" internal action, allowing other language code to be inserted (subject to hypotheses on its effects) is highly desireable in the UML context.

Unlike in MSCs, for more flexibility when used in conjuntion with lifeline decomposition, we allow assignment and assertion internal actions to straddle several lifelines, thus involving implicit synchronisations.

Branching. We require a construct to specify the situation in which several alternatives are possible.

If the tester has several possible alternative emission actions, we would normally expect the conditions determining which action is chosen to be fully specified in the test description. However, in the case where the SUT has several possible alternative emission actions, in black-box testing, the conditions determining which action is chosen may depend on details of the internal state of the SUT that are unknown to the tester, particularly if the SUT is a concurrent and/or distributed system. This latter phenomenon is sometimes referred to as *observable non-determinism*. We require a choice construct that is sufficiently general to be able to describe *indefinite choices*, i.e choices involving observable non-determinism.

The "presentation option" branching construct of UML 1.4/1.5 sequence diagrams is unsuitable for the following reasons:

– indefinite choices cannot be specified,

- since the different alternatives appear on the same diagram, diagrams involving branching can only be guaranteed to be unambiguous if causal flows are completely specified,
- if guards are not mutually exclusive, the same behaviour may describe choice or concurrency depending on data values; with any non-trivial data language, therefore, the question of whether a given construct always denotes a choice will often be undecidable.

These properties make this construct of little use for specification. A construct similar to the alternatives of MSCs would answer our requirements.

Loops. We require a construct for specifying general iterative behaviour.

The recurrence construct of UML 1.4/1.5 sequence diagrams is unsuitable since it was conceived for use with lifelines representing multi-objects, without the meaning of emissions and receptions of other messages on such lifelines being clarified. The "presentation option" iteration construct of UML 1.4/1.5 sequence diagrams is unsuitable since it can only be used to specify a finite number of iterations. A construct similar to the MSC loop would answer our requirements.

Explicit Concurrency (Coregion). We require a construct to explicitly break the default ordering on lifelines, e.g. to specify a multi-cast performed by an SUT component, where the tester neither knows, nor cares, about the order in which the messages are emitted.

The explicit predecessor part of the UML sequence numbering notation is unsuitable due to its ambiguity in the presence of loops and its user-unfriendliness. A construct similar to the MSC coregion would answer our requirements. However, we will be led to use a coregion construct that is richer than the MSC coregion. In MSC, focus bars have no semantics. An important part of the semantics we give to focus bars below concerns their interaction with coregions.

Synchronisation Messages / Local Orderings. We require a means of explicitly ordering two events which would not otherwise be ordered. Where the two events are on the same lifeline, this can be used in conjunction with the coregion to specify more complex concurrent behaviour.

In MSC, the general ordering construct is used for this purpose. However, we would prefer to syntactically distinguish general orderings between events on the same lifeline and those between events on different lifelines. While the MSC syntax is adequate for the former, we prefer to use a "virtual synchronisation" message for the latter[4]

Semantics of Focus Bars. In UML 1.4/1.5 sequence diagrams, focus bars are used to represent method executions. This concept is a valuable one in the context of object- and component-based applications. We would therefore like

[4] It is "virtual" since it denotes new relations but no new events.

to give focus bars a semantics consistent with this interpretation in the absence of an activator relation. Focus bars have been introduced in MSCs but have no formal semantics. We consider this situation rather unsatisfactory.

As stated above, focus bars relate request to reply in synchronous invocations. Since we will not use the UML 1.4/1.5 sequence-diagram semantics, we can define the use of focus bars to be optional with asynchronous invocations without losing ordering between events on different lifelines.

We propose to formalise the semantics of the interaction of focus bars with coregions as follows. If a focus bar falls within the scope of a coregion, the ordering relations between the events in the scope of the focus bar are unaffected by that coregion. Thus, a focus bar falling inside a coregion is a shorthand for a set of local orderings, see Fig. 1 for an example.

In addition, however, we propose that passiveness of the lifeline in question imposes a constraint on the ordering relations between the events in the focus bar scope, on the one hand, and the other events of the coregion that are not in the focus bar scope, on the other. For example, in Fig. 1, Tester 2 being passive corresponds to the constraint that neither of the events $!m_1$ and $!m_8$ can occur between $?m_2$ and $!r_2$ or between $?m_5$ and $!m_7$, where ? (resp. !) signifies reception (resp. emission). This dependence of the ordering properties on the control flow scheme models the implementation of concurrency on passive components as being via task scheduling rather than via execution threads.

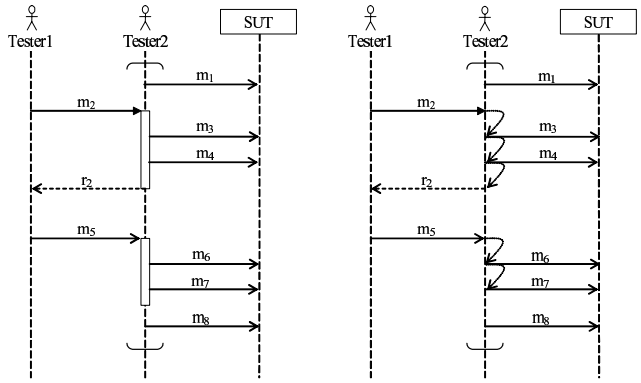

Fig. 1. Focus bars falling inside the scope of a coregion (l.h.s.); equivalent orderings denoted using the local ordering construct (r.h.s.).

Care must be taken as regards the meaning of focus bars in the presence of lifeline decomposition, particularly if the lifeline is specified to be passive. In this latter case, it turns out that we will need to oblige the conservation of focus bars on lifeline composition by using auto-invocations.

Semantics of the Control Flow Scheme Notion. The concept of activeness or passiveness would appear to be a valuable one in object- and component-based applications, albeit a difficult one to pin down. In MSC, this notion does not exist. Again, we consider this situation rather unsatisfactory.

We propose to formalise the notion of passiveness in the sequence-diagram context as a set of constraints on the allowed traces, see [19] for details. If a non-interleaving semantics is used, the notion of passiveness can be defined as a restriction on the allowed linearisations, thus as an (optional) additional semantic layer. These constraints will affect the implementation of concurrency, as described above, and the blocking, or otherwise, of the sender in synchronous calls. Such a definition also helps to ensures that the semantics of the focus bar is consistent with its interpretation as a method execution.

Suspension Region. The suspension region of MSC was introduced to model the blocking of the client during a synchronous invocation. However, we require such blocking to depend on whether or not the lifeline in question represents a passive entity. We have deliberately defined a notion of passiveness that is independent of the basic semantics in order to be able to consider the behaviour of a diagram with, or without, control flow scheme attributes. For this reason, we do not want it to be reflected in the syntax and do not therefore want to use a suspension region construct. Though we do not use such a construct, the vertical region between a synchronous invocation emission event and the reception event of the corresponding synchronous reply has similar behaviour to the focus bar, e.g. concerning interaction with coregions (recall that there may be callbacks) and is similarly affected by the control flow scheme of the lifeline. Below, we refer to this vertical region as the "potential suspension region".

Symbolic Treatment of Data. The importance of allowing parameters of whole test cases, together with the fact that the values returned by the SUT are unknown, makes a semantics involving symbolic treatment of data inevitable. However, due to the complexity of the task, though the main aspects of the non-interleaving, enumerated-data case and the interleaving, symbolic-data case were formalised in [19], the formalisation of the non-interleaving, symbolic-data case was left for future work.

3 Suitability of UML 2.0 Sequence Diagrams

In this section we discuss the sequence diagrams of the upcoming UML 2.0 standard w.r.t. the requirements presented in the previous section.

UML 2.0 sequence diagrams are heavily inspired by the MSC standard. Their semantics is an MSC-style event-based semantics, thus avoiding many of the problems discussed in Section 2.1. Moreover, most of the additional constructs described in Section 2.3 are present, in particular, weak sequential composition, branching, loops, coregions (a derived operator defined in terms of a parallel

operator) and general orderings. Strong sequential composition can be modelled, albeit in a rather cumbersome manner, using the *strict* operator[5].

3.1 Some Particular Points of Interest

Some of the main points of interest in attempting to use UML 2.0 sequence diagrams as a basis for a test description language are as follows:

Execution Occurrence and Internal Actions. The execution occurrence construct of UML 2.0 sequence diagrams, denoting a pair of "event occurrences": the start and finish of an "execution", aims at generalising the UML 1.4/1.5 notion of focus bars.

From the definition of the coregion operator of UML 2.0 sequence diagrams, if an execution occurrence (or, in fact, any other "interaction fragment") falls in the scope of a coregion, the ordering relations between the events in the scope of the execution occurrence are unaffected by that coregion. Thus the situation is similar to that which we propose for focus bars above. However, no interaction with the control flow scheme notion is discussed. In fact, the control flow scheme is not discussed at all in the context of UML 2.0 interactions[6].

MSC-style internal actions are not present in UML 2.0 sequence diagrams and, since "event occurrences" are either message emissions or receptions, an internal action cannot be modelled by an execution occurrence whose start and finish events are simultaneous, even if such execution occurrences were allowed.

Suspension Region. In [17], the notion of suspension region that was present in earlier drafts has been removed, though there is no indication in the document that this has been done for the same reasons as in our work. Moreover, it is not stated that the "potential suspension region" as defined above consitutes an implicit execution occurrence. Thus, such a region falling inside a coregion may give rise to ambiguities.

Sequence Expression Notation. The sequence numbering scheme remains, in spite of its user-unfriendliness and the fact that it is unworkable both for incomplete causal flows and in the presence of loops, branching and guards. However, in [17], unlike in [16], (thankfully!) it's use would seem to be to be restricted to the so-called communication diagrams, which are therefore likely to be of little use outside of the procedural case.

---

[5] In the testing context, if SUT entities are represented explicitly using lifelines, the operands of this operator must not include these SUT lifelines so as not to contradict the black-box hypothesis.

[6] UML 2.0 sequence diagrams are a concrete syntax for UML 2.0 interactions

Scope of "Interaction Fragments". The scope of the "interaction fragment" is not constrained at all in [17]. It is therefore easy to define hard-to-interpret diagrams such as one in which a loop contains the reception of a message but not its emission. The question of defining a set of syntactic restrictions to avoid such problems is currently not tackled.

3.2 Problems With UML 2.0 Constructs Not Present in MSC

The biggest problem with UML 2.0 sequence diagrams concerns the constructs which are new w.r.t MSC, namely the *strict*, *critical region* and *state invariant* constructs and the *neg*, *ignore*, *consider* and *assert* operators.

Though not stated, 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.

If a valid trace can contain 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 the *consider* or an *assert* operator? If a valid trace cannot contain other such actions, is there a need for an *assert* operator? From Fig. 345 of [17], it would seem that message types not appearing in the interaction can be "considered" using the *consider* operator. If a trace involves the sending and receiving of such a message 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. 345? If not, in what way is such a message type to be "considered"?

The new constructs open up a veritable pandora's box of expressions whose meaning is obscure. For example, what is the meaning of an expression "ignoring" a message type a in parallel with an expression involving an occurrence of 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? What about an *ignore* or *neg* expression in the scope of a *strict* operator or inside a *critical region*? What is the meaning of a *strict* expression or a *critical region* in parallel with an expression containing some of the same messages?

3.3 UML 2.0 Sequence Diagram Semantics

The semantics of a single UML 2.0 interaction (not that of a pair of such interactions!) is stated in §14.3.7 of [17] to be a set of valid traces and a set of invalid traces. It is also stated that the union of valid traces and invalid traces does not necessarily constitute the "trace universe", though the exact role of this trace universe in he semantics remains somewhat obscure. 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 this trace universe.

The description of the different constructs seems to betray a confusion between, on the one hand, an interaction as a denotation of a set of traces constructed from the (semantic counterparts of the) actions of that interaction—a

construction which does not require invoking some mysterious trace universe—and, on the other hand, an interaction as a denotation of a set of traces in some larger trace universe, in the manner of a property.

The interaction-as-property interpretation would work in a manner similar to that in which the test objectives of the TestComposer and Autolink tools [23] are used to select traces from among the set of traces of an SDL specification. It is the description of the *state invariant* construct and the *neg*, *ignore*, *consider* and *assert* operators which reflect the interaction-as-property view. The *ignore*, *consider* and *assert* operators affect how the selection of the traces from the nebulous trace universe is performed

Perhaps both interpretations are intended, that is, an interaction is supposed to denote a set of explicitly-constructed traces which can then be used as a property or selection criteria on any trace universe whose atomic actions contain those of the interaction. Notice that the property operates as both a positive and a negative selection criterion since it selects both a set of valid traces and a set of invalid traces, in a similar way to the accept and reject scenarios of [21].

Even without taking into account the problems with the constructs which are new w.r.t. MSC, 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 (c,d) a set of invalid traces, $\cdot$ denotes sequential composition and . trace concatenation, is it the case that $(a, b) \cdot (c, d) = (a.c, b \cup a.d)$?

In summary, the semantics of UML 2.0 sequence diagrams sketched in [17] is in need of clarification. Moreover, it is far from clear that it could be fleshed out into a consistent semantics for the whole language, i.e. one that includes the constructs that are new w.r.t. MSC. Thus, though our language, TeLa, draws heavily on MSC, it is not completely based on UML 2.0 sequence diagrams.

3.4 The UML Test Profile

The aim of the UML Test Profile (UTP) [18] is similar to that of the language TeLa, but this language is directly based on UML 2.0. Aside from the problems with UML 2.0 sequence diagrams discussed above, it is also the case that the test profile addresses a wider range of issues than our work and has taken a less formal approach than ours. The desire to define a mapping to TTCN-3 and JUnit was of a higher priority than defining a more formal semantics. Furthermore, UTP is less based on sequence diagrams than our approach, often requiring a mix of sequence diagrams and state diagrams.

4 Test Description Language: TeLa

The UML sequence-diagram based language TeLa incorporates the corrections of Section 2.2 and the extra constructs of Section 2.3.

In MSCs, the same behaviour can be modelled either using MSCs with inline expressions or using HMSCs. Similarly, in TeLa, tests can be described using

TeLa one-tier scenario diagrams or TeLa two-tier scenario diagrams. The former comprise TeLa sequence diagrams linked by TeLa sequence diagram references, while the latter comprise TeLa sequence diagrams linked using a TeLa activity diagram. However, in constrast to the situation for MSCs, TeLa one-tier scenario diagrams are less expressive than TeLa two-tier scenario diagrams. This is done with the idea of making them simpler to use and closer to UML 1.4/1.5 syntax as well as with the idea of guaranteeing properties of importance in testing such as that of *concurrent controllability*, see Section 5.3. Examples of one-tier scenario diagrams are given in Figure 2 and Figure 4. The concrete syntax for the choice and loop operator using auto-invocations was chosen in the COTE project to be easy to implement in the Objecteering UML tool; clearly, a better concrete syntax could be devised. The equivalent two-tier scenario diagrams are shown in Figure 3 and Figure 5. Since TeLa two-tier scenario diagrams were developed in the COTE project, similar structures, "interaction overview" diagrams, have been introduced in UML 2.0.

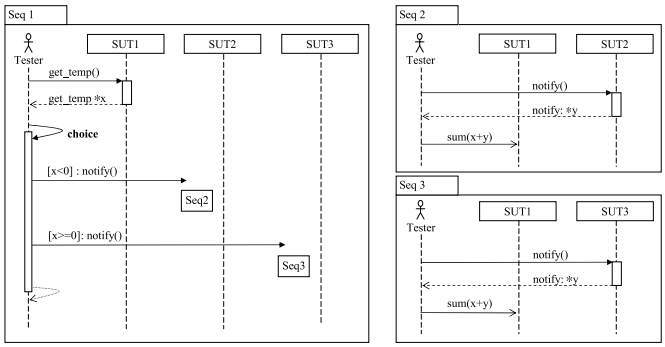

Fig. 2. A TeLa one-tier scenario diagram showing a TeLa sequence-diagram choice and describing the same behaviour as the diagram of Fig. 3.

It is worth mentioning that TeLa sequence diagram loops must be restricted to diagrams with the RSC property [2] in order to be able to define their scope via two valid cuts, see [9]. These are the two valid cuts that contain all events occurring in in the loop scope on the lifeline on which the loop is defined, together with the minimum number of events located on other lifelines. Moreover, we do not allow TeLa sequence diagram loops to occur inside coregions, in order for every sequence-diagram loop to be equivalent to an activity-diagram loop, without the need for a parallel operator at the activity-diagram level.

5 TeLa semantics

In this section we tackle the main issues involved in giving a semantics to our test description language. This includes dealing with the question of verdicts,

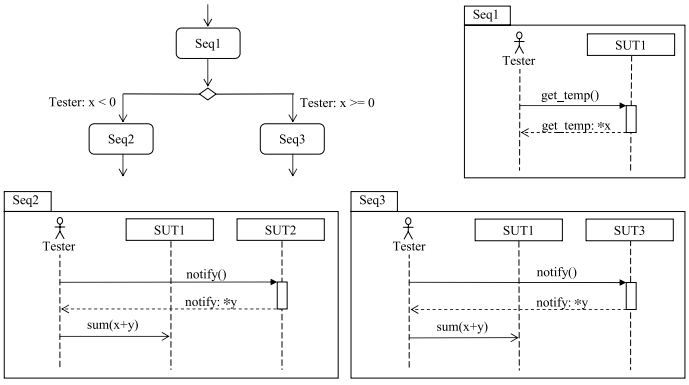

Fig. 3. A TeLa two-tier scenario diagram showing a TeLa activity-diagram choice and describing the same behaviour as the diagram of Fig. 2.

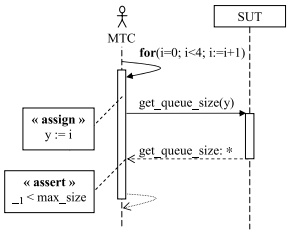

Fig. 4. A TeLa one-tier scenario diagram showing a TeLa sequence-diagram loop and describing the same behaviour as the diagram of Fig. 5.

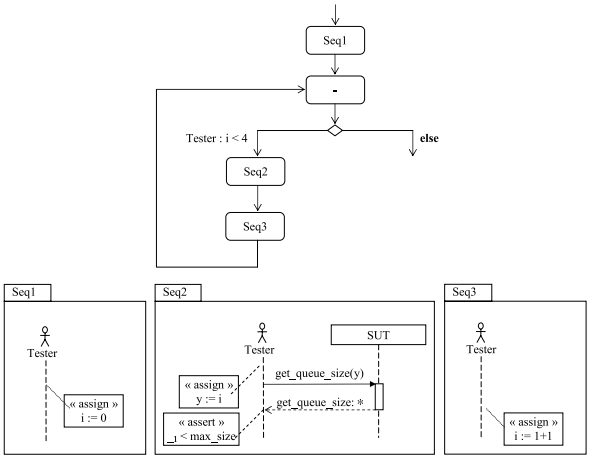

Fig. 5. A TeLa two-tier scenario diagram showing a TeLa activity-diagram loop and describing the same behaviour as the diagram of Fig. 4.

and that of determinism, in order to answer the question of when a test description defines a test case. To date these issues have only been addressed very superficially in the non-interleaving context, e.g. see [15] and [3].

5.1 Structural Semantics

In order to define a framework for lifeline decomposition we define a structural semantics in terms of an underlying hierarchical component model, see [19] for details. By comparison, lifeline decomposition is not addressed in TTCN-3 GFT or UTP. A valid cut of a sequence diagram, see [9], is mapped to a snapshot of this underlying component model (snapshots are necessary due to component creation and destruction). This gives us a means to define the internal structure of components, from the two default components, the tester and the SUT, down to the base-level components. The base-level components are the owners of the dynamic variables used in the specification. We also introduce the use of a dot notation on arrow labels to identify the target port and (base-level) originating port of a message.

This underlying component model is defined by the test architecture specification and possibly also by the specification of the assumed SUT structure. We propose to use UML 2.0 component diagrams as the concrete syntax for our component specifications, augmenting these diagrams with annotations concerning component properties (see below) and the implemented communication architecture. The underlying component model can be used to give a clear meaning to the use of constructs such as focus bars and internal actions under lifeline decomposition. As already stated, in MSC, focus bars have no semantics and the meaning of internal actions or guards in the presence of lifeline decomposition is not addressed. One assumes they are only allowed on lifelines representing base-level entities.

Finally, the structural semantics also provides a framework for deployment and for defining component properties which affect the interpretation of the diagrams. The property of being message-based or event-based and that of being active or passive are examples of such properties. Note that if a component is msg-based, all its subcomponents are msg-based and if a component is active, at least one of its subcomponents must be active.

5.2 Test specific considerations: representation of SUT

As stated in Section 1, a test description is only required to specify the events on the tester. However, unlike TTCN GFT, we choose to represent the SUT explicitly using one or several lifelines. Recall that we cannot use gates since we have exluded these from the language in order to avoid semantic complexity. However, we, in any case, consider the use of explicit SUT lifelines the most user-friendly representation for the following reasons:

- it is closer to the current standard usage of UML sequence diagrams,

- the relation between a UML model and the tests of an implementation of (part of) that model is clearer,
- the situation in which tester events are ordered via SUT events is better communicated by representing the message exchanges involved explicitly,
- it does not give the impression that a specific communication architecture (e.g. FIFO queues per channel) is being denoted; we prefer this to be specified as part of a separate test architecture diagram.

The semantics is then given in two stages: first, derive a partial order of events including SUT events; second, project this partial order onto the tester events, c.f projection of MSCs in [7]. A choice that is local in the second stage semantics, a *test local choice*, is not necessarily a local choice, i.e. a choice that is local in the first stage semantics. See Fig. 6 for an example of a local, but test non-local, choice. Semantics via projection can also give rise to additional non-determinism.

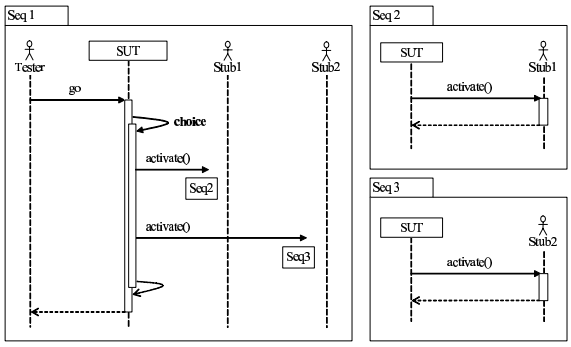

Fig. 6. A TeLa one-tier scenario diagram showing a test non-local choice

Internal Structure. We are now in a position to answer the question of what lifelines represent in TeLa. The tester lifelines represent subcomponents of the single tester component[7]. As we are in a black-box testing context, the SUT lifelines usually represent ports of the SUT component. However, we also allow them to represent the assumed SUT internal structure in terms of subcomponents, if an SUT model is available.

5.3 Dynamic Semantics

We base our semantics on the event-structure semantics defined for MSCs in [8]. According to the classification of [22], this is a non-interleaving, branching-time,

[7] Unlike in [17], here, the terms "component" and "port" denote instances of types rather than types

behavioural semantics. We choose a non-interleaving semantics to clearly distinguish between choice and concurrency and since it is more appropriate for the distributed system context, in particular, for discussing local and global verdicts. We choose a branching-time semantics since this gives us a framework in which to discuss ideas of determinism and controllability. The set of linearisations of the event structures of [8] defines an interleaving, linear-time, behavioural semantics.

Non-interleaving Input / Output models. Input-output models (I/O models), in which the externally visible actions of a specification are divided into inputs and outputs, have proved to be the most applicable in the testing context, see [1] for a survey. To the inputs, resp. outputs of the tester correspond outputs, resp. inputs of the SUT. Our aim is to generalise the use of interleaving input-output models in describing centralised test cases, see [24], to the use of non-interleaving input-output models in describing distributed test cases. We see this as the first step towards generalising the formal approach to conformance testing as a whole.

To deal with internal tester structure and distributed testers, we add the notion of internal action. The tester inputs (emitted by the SUT)—the *observable actions*—are actions for which the test description environment, i.e. the SUT, has the initiative, while the internal actions and tester outputs—the *controllable actions*—are actions for which the tester has the initiative.

Determinism and Controllability in the Non-interleaving Context. In this section, due to the complexity involved, we do not deal with symbolic data and assume that all data is enumerated.

In the input-output model context the automata-theory definition of determinism (no state has multiple identically-labelled output transitions) does not coincide with the "intuitive" notion of determinism (any state with an outgoing transition labelled by an output action has no other outgoing transitions). In this context, the "intuitive" notion of determinism is often termed *controllability*. Both types of determinism are of importance in testing theory. For example, the *test graphs* of [11] are deterministic automata while *test cases* are controllable test graphs. In extending the established input-output testing models to the interleaving case we further refine the two above types of determinism.

In [19], we define the notion of *minimally deterministic test description* as one in which no two concurrent events (resp. events in minimal conflict) are labelled by the same action (resp. observable action). Thus, w.r.t. the usual event-structure definition of determinism, minimal determinism allows minimal conflicts involving events labelled by the same controllable action. However, we also add the condition that any such conflicts must either be resolved on the occurrence of a controllable action or must result in identical verdicts. In terms of the official MSC Semantics, minimal determinism prevents a delayed choice occurring on a fail verdict and therefore ensures that fail verdicts are well-defined. We put forward minimally-deterministic, test descriptions as the non-interleaving analogues of the test graphs of [11].

In [19], we define five notions of controllability ranging from *essential controllability* (EC) through *concurrent controllable* (CC) to *full controllability* (FC). The other two notions are obtained by using the distinction between tester internal actions and tester output actions. A test description is said to be EC if it is minimally deterministic and no event labelled by a controllable action is in minimal conflict with any other event. It is said to be CC if it is EC and no event labelled by a controllable action is concurrent with one labelled by an observable action. It is said to be FC if it is EC and no event labelled by a controllable action is concurrent with any other event.

Defining different types of test case according to whether they have the appropriate controllability property gives us five types of parallel test case. We use the terms *parallel test case*, *coherent parallel test case* and *centralisable test case* for the types corresponding to the above three properties.

Verdicts. Another crucial aspect of test description is modelling verdicts. In TeLa, implicit local verdicts are used as follows:

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

Fail is an existential notion so a single local fail implies a global fail. Pass is a universal notion, so a local pass on all concurrent branches implies a global pass. The meaning of a local inconclusive verdict is that the component which derives the verdict performs no more actions while the other components continue until termination or deadlock to see if they can derive a fail verdict. Communication of local verdicts to the entity responsible for emitting the global verdict is not explicitly modelled in TeLa.

Implicit verdicts represent a higher level of abstraction than the TTCN-3 defaults. The latter have also been taken up by the UTP proposal. We contend that implicit verdicts are also easier to use. However, in TeLa, we also allow certain types of fail and inconclusive verdicts to be explicitly specified for the situation in which this is more convenient. Use of explicit verdicts requires certain restrictions on the test desciption in order for them to be well defined, see [19].

The above notion of verdicts is formalised in [19] for the non-interleaving case but without symbolic treatment of data[8]. Here, we briefly sketch the basis of this formalisation.

We first define a verdict as an annotation on terminal events, that is, events with no successors. We say that a configuration is *complete* w.r.t. a set of observable actions if for each action of the set, there is an enabled event of the configuration labelled by that action. We say that an event structure is *test complete* if all configurations having an enabled action are complete and all terminal events are annotated with a verdict.

[8] And also for the non-interleaving, symbolic data case

We define a fail configuration as a configuration that includes a fail event, a pass configuration as a maximal configuration whose terminal events are all pass events and an inconclusive configuration as a configuration that includes an inconclusive event but does not include a fail event. A *maximal test configuration* is a configuration with an associated verdict. An *execution* is a set of configurations that is totally ordered by inclusion, where each element extends its predecessor by a single event. A *maximal test execution* is an execution whose largest element is a maximal test configuration. The *test verdict* of a maximal test execution is the verdict associated to the last element of the execution.

In order for verdicts to be consistently defined, we must impose the condition that isomorphic configurations of a test-complete event structure have identical verdict annotations. This ensures that any two maximal test runs having the same trace have the same associated verdict. If there are no inconclusive verdicts, this can be guaranteed by demanding minimal determinism, and if there are, by demanding determinism.

Concerning the symbolic data case, we briefly mention a point of interest concerning guards and assertions. Normally, if a guard is not satisfied, the execution path is not feasible whereas if an assertion is not satisfied, an exception is raised. In the presence of implicit verdicts however, this distinction is blurred since if a guard is not satisfied, a fail verdict, which can be viewed as a kind of exception, may result.

6 Conclusion

We have clarified the problems in using UML sequence diagrams as the basis for a formal test description language and have sketched the solution to these problems and the other main semantic issues, as implemented in TeLa, see [19] for more details. The use of this language in test synthesis is described in [21].

References

[1] Brinksma, E., Tretmans, J.: Testing transition systems: An annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., and Ryan, M. (Eds.): Modelling and Verification of Parallel Processes. Proc. of Summer School MOVEP'00. (2000).

[2] Charron-Bost B., Mattern, F., Tel, G.: Synchronous, Asynchronous and Ordered Communication. Distributed Computing 9(4). Springer-Verlag (1996).

[3] Deussen, P.H., Tobies, S.: Formal Test Purposes and the Validity of Test Cases. In: Peled, D. Vardi, M. (Eds.): Formal Techniques for Networked and Distributed Systems (Proc. FORTE 2002). Lecture Notes in Computer Science Vol. 2529. Springer-Verlag (2002).

[4] Engels, A., Mauw, S. Reniers, M.A.: A Hierarchy of Communication Models for Message Sequence Charts. Science of Computer Programming 44(3). Elsevier North-Holland (2002).

[5] European Telecommunications Standards Institute (ETSI): Method for Testing and Specification (MTS); Methdological Approach to the Use of Object-Orientation in the Standards Making Process. ETSI Guide EG 201 872, V1.2.1. ETSI (2001).

[6] European Telecommunications Standards Institute (ETSI): Method for Testing and Specification (MTS); The Testing and Test Control Notation version 3. ETSI Standard ES 201 873 Parts 1 to 6, V2.2.1. ETSI (2003).
[7] Genest, B., Hélouët, L., Muscholl, A,: High-Level Message Sequence Charts and Projections. In: Goos, G., Hartmanis, J., van Leeuwen J. (Eds.): CONCUR 2003 - Concurrency Theory (Proc. CONCUR 2003). Lecture Notes in Computer Science Vol. 2761. Springer Verlag (2003).
[8] Hélouët, L., Jard, C., Caillaud, B.,: An Event Structure Based Semantics for Message Sequence Charts. Mathematical Structures in Computer Science Vol. 12. Cambridge University Press (2002).
[9] Hélouët, L., Le Maigat, P.: Decomposition of Message Sequence Charts. Proc. 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM 2000). Grenoble, France (2000). See: http://www.irisa.fr/manifestations/2000/sam2000/papers.html.
[10] International Telecommunications Union—Telecommunication Standardization Sector (ITU-T): Message Sequence Chart. Recommendation Z.120. ITU-T (1999).
[11] Jard, C., Jéron, T.: TGV: Theory, Principles and Algorithms. In: Proc. 6th world conference on Integrated Design and Process Technology (IDPT02). (2002)
[12] Jard, C., Pickin, S.: COTE—Component Testing Using the Unified Modelling Language. ERCIM News Issue 48. ERCIM EEIG (2001).
[13] Lamport, L.: On Interprocess Communication. Distributed Computing 1(2). Springer Verlag (1986).
[14] Mauw, S., Wijk van M., Winter, T.: A Formal Semantics of Sychronous Interworkings. In: Faergemand, Sarma, A. (Eds.): SDL'93—Using Objects (Proc. SDL Forum 93). Elsevier North-Holland (1993).
[15] Mitchell, B.: Characterising Concurrent Tests Based on Message Sequence Chart Requirements. In: Proc. Applied Telecommunication Symposium. (2001)
[16] Object Management Group (OMG): Unified Modelling Language Specification version 1.5. OMG, Needham, MA, USA (Mar. 2003).
[17] Object Management Group (OMG): UML 2.0 Superstructure Specification. OMG, Needham, MA, USA (Aug. 2003).
[18] Object Management Group (OMG): UML Testing Profile, version 2.0. OMG, Needham, MA, USA (Aug. 2003).
[19] Pickin, S.: Test des Composants Logiciels pour les Télécommunications. Ph.D. Thesis. Université de Rennes, France (2003).
[20] Pickin, S., Jard, C., Heuillard, T., Jézéquel, J.M., Defray, P.: A UML-integrated Test Desciption Language for Component Testing. In: Evans, A., France, R., Moreira, A., Rumpe, B. (Eds.): Practical UML-Based Rigorous Development Methods. Lecture Notes in Informatics (GI Series), Vol. P7. Kollen-Druck + Verlag (2001).
[21] Pickin, S., Jard, C., Le Traon, Y., Jézéquel, J.M., Le Guennec, A.: System Test Synthesis from UML Models of Distributed Software. In: Peled, D. Vardi, M. (Eds.): Formal Techniques for Networked and Distributed Systems (Proc. FORTE 2002). Lecture Notes in Computer Science Vol. 2529. Springer-Verlag (2002).
[22] Sassone, A., Nielsen, M., Winskel, G.: Models for Concurrency: Towards a Classification. Theoretical Computer Science 170(1–2) Elsevier (1996).
[23] Schmitt, M., Ebner, M., Grabowski, J.: Test Generation with Autolink and Test Composer. In: Proc. 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM 2000). Grenoble, France (2000). See: http://www.irisa.fr/manifestations/2000/sam2000/papers.html.
[24] Tretmans, J.: Specification Based Testing with Formal Methods: From Theory via Tools to Applications. In: A. Fantechi, A. (Ed.): FORTE / PSTV 2000 Tutorial Notes. (2000).