N° d'ordre: 3662

# THÈSE

Présentée devant

**devant l'Université de Rennes 1**

pour obtenir

le grade de : DOCTEUR DE L'UNIVERSITÉ DE RENNES 1
Mention INFORMATIQUE

par

Sébastien SAUDRAIS

Équipe d'accueil : Triskell - IRISA
École Doctorale : Matisse
Composante universitaire : IFSIC

Titre de la thèse :

*Qualité de Service Temporelle*
*pour*
*Composants Logiciels*

soutenue le 5 décembre 2007 devant la commission d'examen

| | | | |
|---|---|---|---|
| M. : | Isabelle | PUAUT | Président |
| MM. : | Louis | FÉRAUD | Rapporteurs |
| | Philippe | LAHIRE | |
| MM. : | Noël | PLOUZEAU | Examinateurs |
| | Laurence | DUCHIEN | |
| | Jean-Marc | JÉZÉQUEL | |

*A Sandrine,*

*Après la pluie vient le beau temps*

## Remerciements

Je remercie Isabelle PUAUT, Professeur à l'Université de Rennes 1, qui me fait l'honneur de présider ce jury.

Je remercie Louis FÉRAUD, Professeur à l'Université Paul Sabatier, et Philippe LAHIRE, Professeur à l'Université de Nice Sophia antipolis, d'avoir bien voulu accepter la charge de rapporteur.

Je remercie Noël PLOUZEAU, Maitre de Conférences à l'Université de Rennes 1, et Laurence DUCHIEN, Professeur à l'Université de Lille 1, d'avoir bien voulu juger ce travail.

Je remercie enfin Jean-Marc JÉZÉQUEL, Professeur à l'Université de Rennes 1, qui a dirigé ma thèse.

Je remercie l'équipe Triskell pour m'avoir accueilli pendant ces trois années de thèse et pour l'aide reçu au niveau recherche et technique.

Je tiens également à remercier mes parents et l'ensemble de ma famille pour leur soutien.

Enfin merci à Sandrine pour son soutien de tous les jours.

# Table des matières

# Introduction

Le génie logiciel englobe les « règles de l'art » de l'ingénierie de la réalistion des systèmes manipulant l'information[FEBF06]. Il a pour objectif de rendre les logiciels plus fiables et de respecter le cahier des charges du logiciel dans les délais et les coûts prévus. Le génie logiciel est défini comme *l'application d'une approche systématique, disciplinée et quantifiable au développement, au fonctionnement et à la maintenance du logiciel, c'est-à-dire l'application de l'ingénierie au logiciel* (norme IEEE 610.12). Il a vu son importance grandir lors de la crise de l'industrie du logiciel de la fin des années 70 puis perdurer au vue de la croissance de la taille des logiciels et de leur complexité.

L'approche par composants [HC01] permet de faire face à la taille et la complexité du logiciel en le considérant comme un assemblage d'éléments logiciels de granularité plus ou moins importante. Ce découpage permet de s'intéresser à un sous-ensemble du logiciel, puis d'assembler celui-ci avec d'autres sous-ensembles afin d'obtenir le logiciel complet. Un composant peut être vu comme une boîte noire avec des interfaces définissant les interactions possibles avec son environnement. L'approche par composants permet la réutilisation des briques élémentaires. Lors de l'assemblage de composants, une vérification de compatibilité permet d'assurer que les composants sont assemblables. La vérification porte sur le type de données échangées mais aussi sur d'autres critères comme la synchronisation des différentes entités du système. Chaque composant doit donc fournir un ensemble de propriétés représentant ce qu'il fournit mais aussi ce qu'il requiert pour bien fonctionner, ce qu'on appelera son « contrat ».

La conception par contrats [Mey92a] est une méthode de conception logicielle permettant de définir un contrat gouvernant l'interaction d'entités logicielles. L'idée de la méthode est que les entités logicielles ont des responsabilités envers les entités avec lesquelles elles interagissent. Un contrat, utilisé dans le cadre d'une conception à base de composants, permet de définir lors de l'assemblage de composants ce que chacun des composants doit effectuer. Il est comparable à un contrat entre personnes où chaque personne s'engage à fournir des services et à en recevoir. Un contrat définit les contraintes entre les entités c'est-à-dire les droits et les obligations entre le fournisseur du service et le client le requérant. Le contrat permet d'évaluer le logiciel sur les fonctionnalités présentes, le critère quantitatif, mais pas sur la qualité de fonctionnement du logiciel, sur les propriétés extra-fonctionnelles.

La qualité de service est *l'ensemble des exigences de qualité pour le comportement observable d'un ou plusieurs objets* [Iec95]. La qualité de service d'un logiciel permet de définir une information qualitative en plus de l'information quantitative sur le logiciel développé. Le critère quantitatif est la conformité du logiciel développé par rapport au cahier des charges fourni par le client. Plusieurs techniques permettent de s'assurer de ce respect comme le *model-*

*checking* lors de la spécification du logiciel ou le test lors de son implantation. Ces techniques permettent de vérifier que les fonctionnalités spécifiées dans le cahier des charges sont bien intégrées dans le logiciel. Le critère qualitatif permet à l'utilisateur de définir des propriétés extra-fonctionnelles. Ces propriétés servent à indiquer une qualité sur les fonctionnalités offertes par le logiciel. On ne souhaite pas seulement savoir si le logiciel fait bien ce que l'on désire mais on veut de plus connaître des informations qualitatives sur ces fonctionnalités. Ces informations peuvent concerner les consommations des ressources du système, la précision des résultats ou le temps d'exécution pris par une fonctionnalité. La qualité de service est un point de vue qui dépend du logiciel développé. Une consommation de ressources peut être vue comme une fonctionnalité pour un système embarqué mais une propriété extra-fonctionnelle pour un autre logiciel. La qualité de service temporelle correspond aux propriétés de temps non vitales pour la réalisation des fonctionnalités du système. Lors de l'exécution d'une fonctionnalité d'un logiciel, l'utilisateur veut que le résultat soit correct par rapport au cahier des charges, c'est-à-dire qu'il fournit la fonctionnalité requise. L'utilisateur peut également souhaiter que le résultat soit disponible dans un temps donné, c'est-à-dire que le service est rendu avec une qualité de service temporelle correcte. Dans cette thèse, nous nous intéressons à la qualité de service temporelle dans des applications à composants.

### Problématiques

La notion de qualité de service temporelle permet à l'utilisateur de raisonner sur des informations sur les qualités temporelles souhaitées et fournies. L'expression de ces propriétés temporelles lors du développement d'un logiciel doit permettre de participer à l'amélioration du processus de développement. Les problèmes majeurs de l'expression de la qualité de service temporelle sont :

- les informations temporelles sont parfois prises en compte uniquement lors de la spécification du logiciel mais ne sont plus utilisées lors de l'exécution. Les estimations obtenues après la spécification ne sont pas vérifiées à l'exécution et l'utilisateur doit faire confiance au logiciel ou estimer par lui-même la qualité de service.
- les informations temporelles peuvent aussi être vérifiées uniquement à l'exécution sans connexion avec celles définies lors de la spécification. Les assemblages de composants risquent alors de bien fournir les fonctionnalités désirées mais pas obligatoirement avec une bonne qualité de service.
- l'expression des propriétés temporelles n'est pas compatible entre différents logiciels. Les exigences de qualité de service temporelle sont souvent syntaxiques et la sémantique associée est différente selon les logiciels développés. La réutilisation de ces composants oblige l'architecte à revoir la sémantique des propriétés temporelles.

### Contributions

La gestion de la qualité de service temporelle au cours du développement d'une appli-

cation à composants est composée de trois phases principales : l'introduction des propriétés temporelles, la vérification de ces propriétés lors de spécification (i.e. vérification vis-à-vis du modèle) puis lors de l'exécution (i.e. vérification vis-à-vis de la réalité).

L'introduction de propriétés temporelles de qualité de service dans l'application est le passage du cahier des charges vers la spécification. Cette introduction doit permettre d'intégrer les propriétés en utilisant des formalismes avec une sémantique bien définie afin de pouvoir être utilisé lors de la vérification. L'introduction est simple si l'architecte est familier avec les formalismes temporels. Sinon, les informations de temps restent sous la forme d'annotations et ne seront pas ou peu prises en compte pendant le développement.

La vérification des propriétés temporelles lors de la spécification permet d'avoir une estimation de la future qualité de service de l'application.

La vérification de la qualité de service à l'exécution permet de s'assurer que le code de l'application a une qualité de service conforme à celle qui a été prévue lors de la spécification du logiciel. Cette vérification est effectuée en observant le comportement de l'application et en le comparant avec celui de la spécification. L'observation peut se faire à l'aide d'un moniteur de qualité de service. Celui-ci est produit à partir de la spécification de manière manuelle ou automatique. Un moniteur manuel nécessite un temps supplémentaire de développement et est une source d'erreur supplémentaire dans le développement de l'application. Un moniteur généré automatiquement offre l'avantage de provenir d'un processus de transformation réutilisable et de ne pas augmenter le temps de développement.

Cette thèse étudie la gestion de la qualité de service de temps tout au long du processus de développement d'un logiciel. Dans un premier temps, nous nous intéressons à la phase de spécification du logiciel. Nous définissons une méthodologie permettant d'ajouter des propriétés de qualité de service de temps pour des architectes ne souhaitant pas connaître les formalismes utilisés pour la vérification du temps. Cette méthodologie utilise une approche à base de motifs. Chaque motif permet de représenter une propriété temporelle comme la durée ou la période. Les informations de temps sont ajoutées dans le comportement de composants et dans les contrats entre les composants. L'architecte peut sélectionner les motifs qu'il souhaite intégrer à son logiciel. Nous utilisons les formalismes temporels des automates temporisés pour le comportement des composants et d'une logique temporelle temporisée pour les contrats.

Dans un deuxième temps, nous proposons une génération automatique du moniteur de qualité de service. L'automatisation permet de disposer d'un cadre générique pour le moniteur et de pouvoir rejouer la génération si les spécifications sont modifiées en réduisant les erreurs potentielles issues d'une implantation manuelle. Le moniteur généré est écrit en Giotto, langage proposant une abstraction de temps par rapport aux fonctionnalités.

La gestion de la qualité de service de temps est illustrée par la Figure 1. La partie gauche correspond à la spécification de l'application. L'automate temporisé d'un composant est le comportement attendu d'un composant respectant la qualité de service. La composition des composants permet d'obtenir le comportement complet de l'application. Les composants sont ensuite développés de façon traditionnelle pendant que le moniteur de qualité de service est généré à partir du comportement global. La partie droite montre la surveillance de l'exécution de l'application par le moniteur.

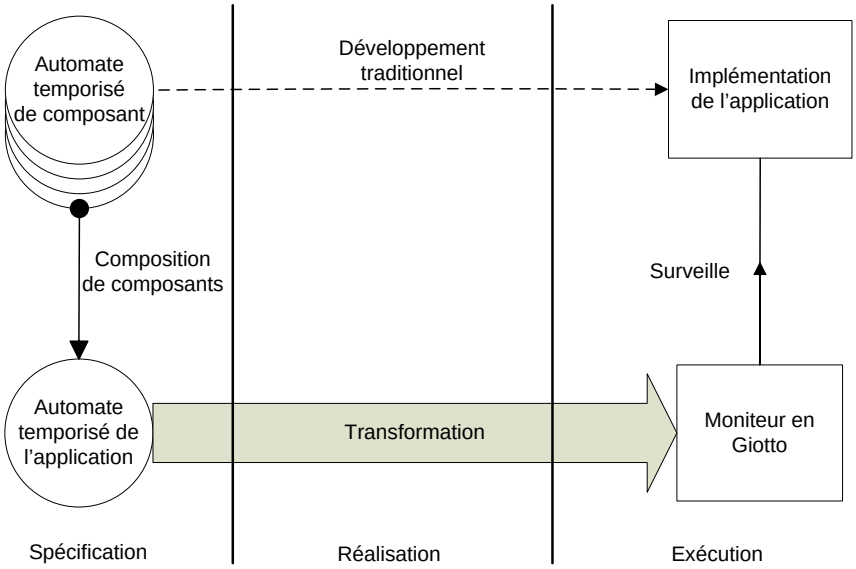



FIG. 1 – Approche globale pour la gestion de la qualité de service temporelle

**Organisation du document**

Ce document est organisé en cinq chapitres. Le chapitre 1 présente le contexte de la thèse et les formalismes utilisés. Le chapitre 2 présente l'ajout des propriétés temporelles dans les composants du système. Le chapitre 3 détaille la réalisation d'un moniteur de qualité de service à partir de la spécification d'un système. Le chapitre 4 présente l'implantation des deux méthodes dans *Thot*, un outil développé dans l'environnement Kermeta. Le chapitre 5 décrit l'application de la méthodologie de cette thèse sur le développement d'un robot. Finalement, le chapitre 6 conclut ce document et ouvre des perspectives.

# Chapitre 1

# Méta-modèles, formalismes et travaux du domaine

Ce chapitre présente le contexte de la thèse : les applications à composants, les contrats et l'ingénierie dirigée par les modèles. Nous présentons ensuite les formalismes utilisés lors de la thèse : le méta-modèle de composant, les automates temporisés et TCTL. Nous présentons enfin les travaux traitant du temps ou de la qualité de service dans les logiciels.

## 1.1 Application à composants, Contrats et Ingénierie dirigée par les modèles

Le contexte de la thèse est la gestion de la qualité de service temporelle pour les applications à composants. Dans un premier, nous présenterons les applications à composants et le cycle de vie des composants. Nous présenterons ensuite la conception par contrat permettant de définir une relation entre les composants. Enfin, nous présenterons l'ingénierie dirigée par les modèles, cadre dans lequel nous avons développé notre approche.

### 1.1.1 Application à composants

#### 1.1.1.1 Présentation

La notion de composant logiciel [HC01, Szy98] est un paradigme introduit comme une suite à la notion d'objet pour pallier les défauts des objets. Le composant offre une meilleure structuration de l'application et permet de construire un système par assemblage de briques élémentaires en favorisant la réutilisation de ces briques.

Le terme *composant* est utilisé avec une signification différente selon le contexte d'utilisation ou le domaine scientifique l'utilisant. Cette variation entraîne une difficulté de compréhension lors d'un changement de contexte. Un composant peut représenter une instance ou un type, des binaires qui se déploient, qui se composent ou s'assemblent.

En outre, le terme de composant logiciel a été utilisé avec des sens parfois très différents. Par exemple, Shaw et Garlan ont défini la notion de composant comme une unité de traite-

ment [GS93], ce qui implique que tout élément logiciel est un composant logiciel. Brown et Short définissent le composant comme une unité indépendante qui fournit un ensemble de services réutilisables [BW96], ce qui par contre, n'implique pas la propriété de composition.

Cette absence de précision a entraîné un établissement difficile d'un discours unifié sur les composants, les détracteurs jugeant que l'approche ne faisait que réinventer la roue.

Cependant, différentes propriétés permettent de caractériser le composant. La définition de Szypersky, globalement acceptée, illustre bien ces propriétés : « *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.* » [Szy98].

Cette définition met en lumière trois propriétés importantes du composant :

- Un composant est sujet à composition. Ce point met en avant la volonté de transformer l'ingénierie du logiciel en un modèle de Lego™ [BO92]. Dans ce modèle, une application est construite à partir d'un assemblage de composants de manière similaire à un assemblage de briques Lego.
- Un composant peut être déployé indépendamment. Cette affirmation définit en fait deux propriétés du composant. La première réside dans la prise en compte dans les modèles de composant de la tâche de déploiement. La deuxième rappelle le nécessaire découplage entre les composants. Contrairement à l'objet, le composant n'introduit pas de mécanisme d'héritage et cherche à minimiser les dépendances entre composants.
- Un composant explicite de manière contractuelle les services qu'il peut rendre et ses dépendances vis-à-vis de l'environnement. Il propose dans un langage défini les services et leurs règles de fonctionnement de façon à aider leur réutilisation au sein de différents assemblages.

Enfin, un dernier point, peu pris en compte dans cette définition et propice à engendrer des problèmes de compréhension, concerne le cycle de vie d'une application. En effet, le terme « composant logiciel » est utilisé depuis la phase de conception jusqu'à la phase d'exécution d'une application, il prend donc plusieurs formes. La section suivante décrit le cycle de vie utilisé pour les composants dans la thèse.

#### 1.1.1.2 Cycle de vie des composants

Le processus de développement d'un composant se déroule en plusieurs phases. La première phase est la définition des services que l'on désire offrir : la spécification de services. Ensuite les services sont regroupés et forment la spécification de composant. Puis on enrichit cette spécification par ce que le composant pourra requérir pour effectuer ses services : c'est l'implantation abstraite de composant. Enfin, le composant est codé dans un langage en respectant son implantation abstraite.

**Spécification de service** Une spécification de service décrit ce qu'un composant peut fournir, en utilisant les définitions de types et les opérations. Une opération peut avoir des contraintes comme les types, les pré and post-conditions et les propriétés comportementales et temporelles. Cette définition est proche de celle des Web services. Un service est défini comme la possibilité

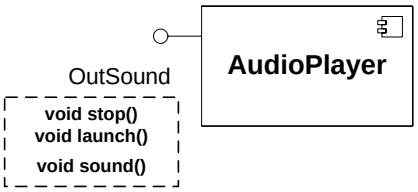



FIG. 1.1 – Spécification de composant

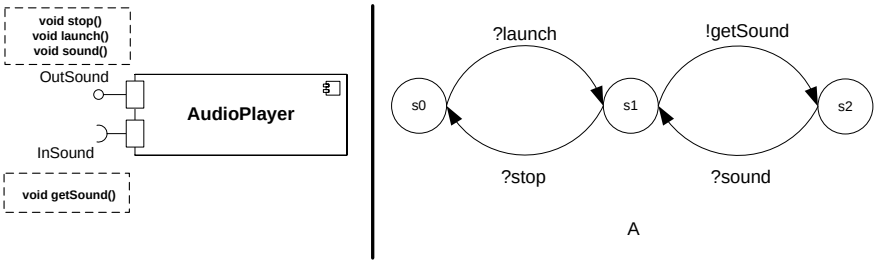



FIG. 1.2 – Implantation abstraite de composant

d'effectuer un ensemble spécialisé de tâches. Un service fourni a pour but de résoudre une préoccupation précise de l'application.

**Une spécification de composant** Une spécification de composant est l'ensemble des services fournis par le composant. Les interfaces fournies contiennent les caractéristiques des services offerts par le composant.

La Figure 1.1 donne la représentation en UML 2 d'une spécification de composant. Le composant propose trois services regroupés dans une interface fournie d'un port.

**Implantation abstraite de composant** Une implantation abstraite de composant est conforme à une spécification de composant. L'implantation fournit les informations sur les services requis par le composant. L'enchaînement entre les services requis et fournis est donné par un automate de comportement.

Une implantation abstraite fournit les informations nécessaires à l'assemblage du composant tout en cachant les opérations internes au composant. Cette entité est utilisée lors de l'assemblage de composants. Elle doit donc contenir les informations nécessaires et suffisantes pour la vérification d'assemblage.

La Figure 1.2 donne la représentation d'une implantation abstraite de composant. Le composant offre trois services et en requiert un. L'enchaînement des services est donné par l'automate $A$.

**Implantation concrète de composant** Une implantation concrète de composant est une entité au niveau code. Elle est exécutable et est conforme à l'implantation abstraite de composant. Elle fournit les services et doit respecter les contrats établis lors de l'assemblage du composant.

### 1.1.2 Contrats

La conception par contrats (DbC pour *Design by Contrat*) [Mey92a] est une méthode de conception logicielle. Elle est basée sur la théorie des types de données abstraits [Rua84] et sur la métaphore du contrat au sens juridique qui lie deux parties. Le DbC met en avant l'intérêt de spécifier précisément les interfaces d'un module logiciel en termes de pré-conditions, post-conditions et invariants.

Bien que le terme DbC ait été proposé par Bertrand Meyer en 1992 [Mey92a], des travaux sur les contrats remontent à Floyd [Flo67] et Hoare [Hoa69]. Ces travaux montrent l'intérêt de l'utilisation des assertions dans la spécification de programmes. Une assertion est définie comme une expression booléenne qui doit être vraie pour qu'un code spécifié s'exécute correctement. Reprises dans la spécification des interfaces de classe ou de composant, les assertions permettent le raffinement de contraintes de typage. D'un point de vue conceptuel, les contrats vont au-delà des mécanismes d'assertions. Ces dernières ne sont qu'un mécanisme de base qui n'a que peu en commun avec les aspects méthodologiques et l'intégration au modèle objet qu'offrent les contrats. Cependant, plusieurs langages, qui ne proposent pas de programmation par contrats dans leur version de base, utilisent le mécanisme d'assertion, lorsqu'il est disponible, pour mettre en oeuvre les contrats avec des bibliothèques, des préprocesseurs ou d'autres moyens spécialisés.

Le paradigme de contrat a été très étudié par la communauté de génie logiciel. En effet, face à des modules logiciels devenus trop complexes pour comprendre aisément leur fonctionnement global, les contrats sont utilisés comme un sous-ensemble de la spécification du système permettant de fixer un certain nombre d'invariants que le système s'engage à respecter ou que l'environnement doit respecter. Un contrat est le résultat d'un accord entre deux entités du système. La première requiert des fonctionnalités que le seconde peut offir. Le contrat est le résultat d'une négociation afin que l'offre corresponde à la demande. Un contrat ne porte que sur une partie de la spécification du système. Le contrat contrairement à la spécification n'est pas nécessairement complet dans le sens où il n'est pas possible de comprendre entièrement le système à partir des contrats.

L'idée centrale du DbC est que les entités logicielles ont des responsabilités envers les autres entités avec lesquelles elles interagissent. Ces responsabilités sont alors fondées sur des règles formalisées entre ces entités. Des spécifications fonctionnelles, ou « contrats », sont créées pour chaque module dans le système avant que ces modules ne soient codés. Les interactions entre les divers modules de l'application sont alors bornées par ces contrats.

Le travail le plus connu dans ce domaine est probablement celui autour d'Eiffel [Mey92b]. Eiffel utilise des assertions dans des pré et post-conditions et dans des invariants de classes pour décrire des contrats entre l'utilisateur et une classe. D'autres approches plus récentes comme le langage OCL [omg03] pour UML ou JML [LLP+00] pour Java fournissent maintenant des mises en œuvre étendues de la conception par contrats.

L'utilisation des concepts du DbC dans le développement d'une application entraîne de très nombreux bénéfices :

– une meilleure séparation des préoccupations : si les approches par contrats cherchent avant tout à favoriser la réutilisation de modules logiciels, la déclaration explicite d'un certain nombre d'informations relatives à l'interaction permet de clarifier le code des

méthodes. Ce code contient alors uniquement le code fonctionnel lié aux méthodes et n'est pas envahi par des parties de code vérifiant la correction des paramètres envoyés ou retournés au cours d'une interaction,
- possibilité d'outillage : grâce à une formalisation précise des langages de contrats, il est possible d'outiller convenablement l'approche afin de garantir statiquement certaines propriétés, de générer du code pour garantir à l'exécution le respect des contrats,
- documentation : le contrat est une documentation naturelle, car il répond à la question « *qu'est-ce que cet élément requérant un service est supposé faire ?* ». Plutôt que de se contenter de l'interface, on peut donc utiliser l'interface et le contrat (invariants et pré/post-conditions) comme élément de base de la documentation d'un élément logiciel. Ceci est utile lors des spécifications, mais aussi lors de la génération automatique de la documentation à partir du code et de ses commentaires, comme le permettent des outils disponibles dans de nombreux langages.

Les approches par composants visent à favoriser la réutilisation d'entités logicielles. Dans ce sens, il est très intéressant de porter ces concepts au niveau des interfaces de composant.

#### 1.1.2.1 Niveaux de contrats

Nous utilisons la classification de [BJPW99] pour les niveaux de contrats. Quatre niveaux sont définis selon les informations contenues par ceux-ci. Les quatre niveaux sont :

- le niveau *syntaxique* correspond aux opérations effectuables par le composant, aux paramètres d'entrées/sorties du composant et aux exceptions levables par le composant. Les IDL, *Interface Description Language*, peuvent servir à décrire ce niveau de contrat.
- le niveau *comportemental* correspond aux pré et post conditions et aux invariants. Les services sont considérés comme atomiques. Les types de langages utilisables pour ce niveau sont entre autres OCL, Eiffel ou les iContract pour Java.
- le niveau *synchronisation* spécifie le comportement global des objets. Ce comportement correspond à la synchronisation des appels de services. Ceci permet d'exprimer les dépendances entre les services fournis par un composant comme la séquence, le parallélisme ou le choix arbitraire. Les langages utilisable pour ce niveau peuvent être des BPEL, *Business Process Execution Language* pour les services ou des automates à états.
- le niveau *qualité de service* correspond à ce que le composant offre quantitativement. Ceci correspond à des propriétés temporelles, comme temps de réponse, à la qualité du résultat ou à la gestion des ressources. Ces propriétés font souvent appel à une tierce partie qui n'a pas de garantie de performance mais seulement une estimation de performance. Ces contrats sont les plus difficiles à vérifier. Les langages utilisable pour ce niveau sont des profils UML grâce à des annotations comme SPT ou MARTE.

Les niveaux de contrats sont de plus en plus négociables. Un composant peut plus difficilement demander un changement de type qu'un paramètre de qualité de service. Les travaux de cette thèse portent sur le quatrième niveau de contrat : la qualité de service. Nous nous intéressons plus particulièrement aux propriétés temporelles de qualités de service.

#### 1.1.2.2 Cycle de vie des contrats

L'établissement d'un contrat a lieu lors de l'assemblage de composants. Ce contrat est construit à partir d'informations fournies par les composants. Ces informations sont contenues dans un contrat de composition attaché aux interfaces du composant. Ce contrat de composition est organisé selon les quatre niveaux de contrats. Lors de l'assemblage de composants, les contrats de composition présents sont comparés et forment un contrat s'ils sont compatibles. La vérification de compatibilité se fait niveau par niveau. Si l'une des parties ne satisfait plus le contrat, celui-ci est violé et peut être renégocié selon le niveau du contrat falsifié. La Figure 1.3 présente le cycle de vie d'un contrat. La renégociation d'un contrat peut être statique ou dynamique. Une renégociation statique correspond à une renégociation lors de la phase de spécification. Une renégociation dynamique s'effectue à l'exécution quand les composants sont déployés.

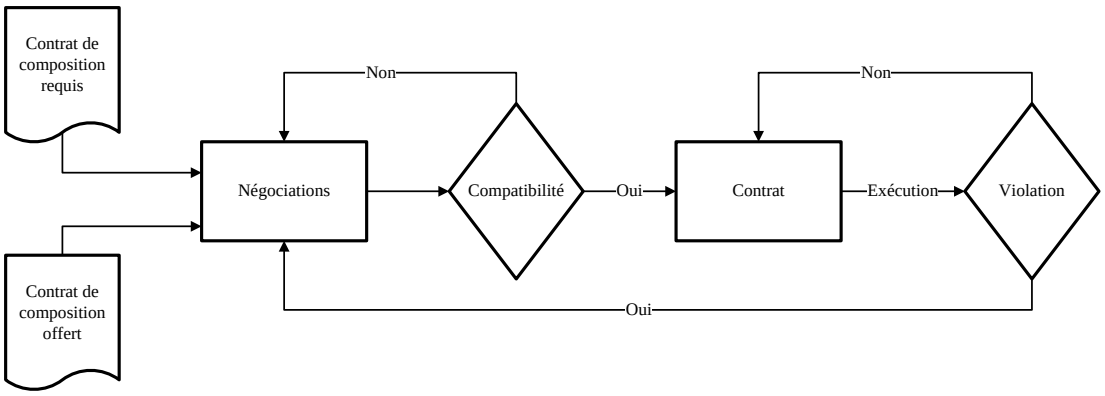



FIG. 1.3 – Cycle de vie d'un contrat

### 1.1.3 Ingénierie dirigée par les modèles

L'Ingénierie dirigée par les modèles (IDM) [EFB+05] propose des solutions permettant aux organisations de surmonter la mutation des exigences du développement de logiciel. L'ingénierie des modèles est une forme d'ingénierie générative, par laquelle les modèles permettent de générer une application informatique ou des parties de celles-ci. Les idées de base de cette approche sont voisines de celles de nombreuses autres approches du génie logiciel, comme la programmation générative, les langages spécifiques de domaines ((DSL pour *domain-specific language*) [vDKV00, CM98], le MIC (*Model Integrated Computing*), les usines à logiciels (Software Factories) [GSC+04] entre autres.

L'intérêt pour l'Ingénierie dirigée par les modèles (IDM) a été fortement amplifié, en novembre 2000, lorsque l'OMG (*Object Management Group*) a rendu publique son initiative MDA [StOs00] qui définit un cadre de normalisation pour l'IDM. Il existe cependant bien d'autres alternatives technologiques aux normes de l'OMG, par exemple *Ecore* dans le domaine technologique d'Eclipse, mais aussi les grammaires, les schémas de bases de données,

les schémas XML. En somme, l'ingénierie des modèles dépasse largement le MDA, pour se placer plutôt à la confluence de différentes disciplines.

Dans cette nouvelle perspective, les modèles occupent une place de premier plan parmi les artefacts de développement des systèmes, et doivent en contrepartie être suffisamment précis afin de pouvoir être interprétés ou transformés par des machines. Le processus de développement des systèmes peut alors être vu comme un ensemble de transformations de modèles partiellement ordonné, chaque transformation prenant des modèles en entrée et produisant des modèles en sortie, jusqu'à obtention d'artefacts exécutables.

Pour donner aux modèles cette dimension opérationnelle, il est essentiel de spécifier leurs points de variations sémantiques, et donc aussi de décrire de manière précise les langages utilisés pour les représenter. On parle alors de métamodélisation.

L'ingénierie des modèles peut jouer un rôle de médiation entre différentes disciplines, telles que la théorie des langages, la programmation générative, les bases de données, etc. L'utilisation de l'ingénierie des modèlesdans le cadre de cette thèse permet le passage de la spécification, vue comme un PIM, *Platform Independant Model*, vers le moniteur de qualité de service, vu comme un PSM, *Platform Specific Model*.

## 1.2 Formalismes

Plusieurs formalismes sont utilisés dans la thèse pour représenter les différents concepts : les composants et les contrats de composition. Nous utilisons un méta-modèle de composants inspiré de celui d'UML 2. Afin de représenter le temps, nous utilisons deux formalismes. Les automates temporisés permettent de spécifier un comportement temporel, le contrat de composition offert. La logique temporelle temporisée TCTL permet l'écriture de propriétés temporelles bornées dans le temps, le contrat de composition requis. Ces formalismes sont présentés dans la suite de cette section.

### 1.2.1 Méta-modèle de composants

Nous utilisons un méta-modèle de composants permettant de décrire les trois premiers niveaux du développement du composant. Notre méta-modèle est proche de celui défini dans UML 2. Les *statecharts* d'UML, représentant le comportement du composant, ne proposent pas d'extension correcte pour le temps. Le comportement de composant est représenté par une algèbre de processus liée à un automate à E/S pour la vérification. Pour assembler les composants, nous introduisons les contrats de composition pour le composant.

#### 1.2.1.1 Eléments structurels du méta-modèle à composants

Notre méta-modèle à composants est dérivé des concepts du diagramme d'architecture d'UML 2.0 pour la partie structurelle. Cependant, pour limiter la complexité du langage manipulé par l'architecte, nous avons supprimé certains concepts et enlevé tous les points de variations sémantiques d'UML 2.0.

Dans notre modèle à composants, un composant fournit des *services* et peut requérir des services d'autres composants. Les *services* sont accessibles par des ports uniquement. Un *port*

est un point de connexion sur un composant définissant deux ensembles d'*interfaces* : *fournies* et *requises*.

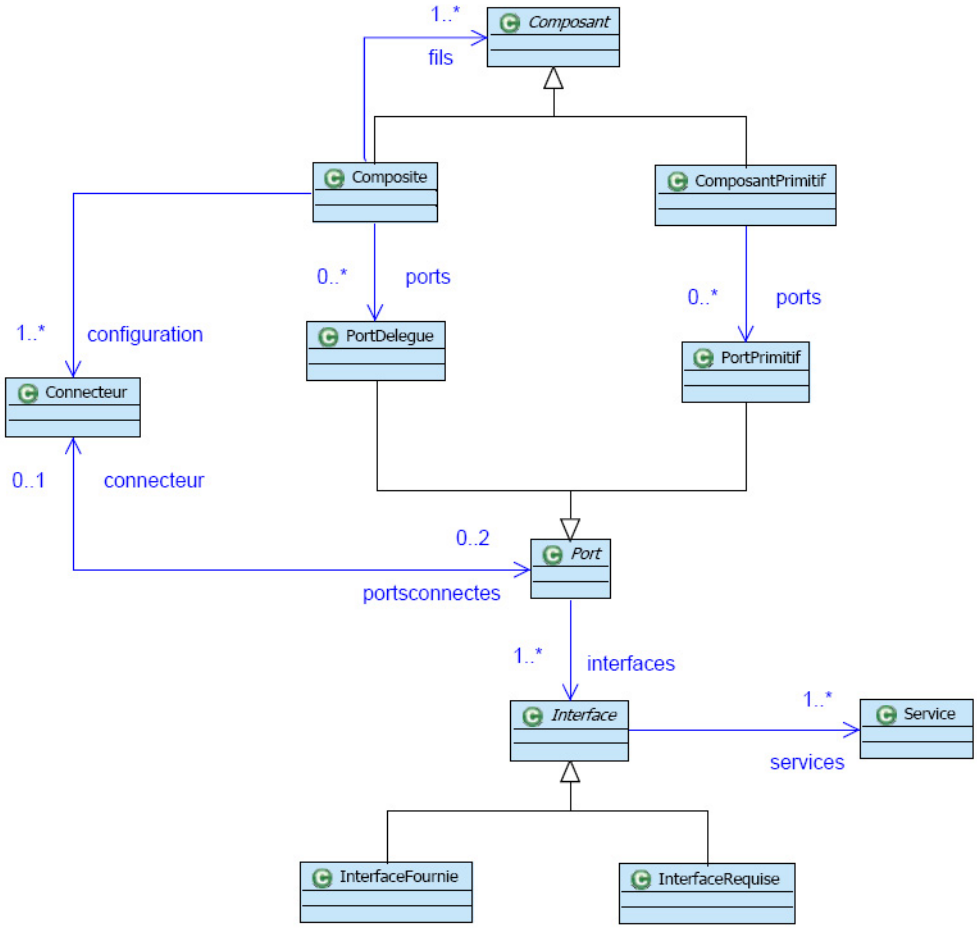



FIG. 1.4 – Partie structurelle du méta-modèle de composant

Notre modèle de composant distingue deux types de composants : *primitif* et *composite*. Les composants *primitifs* contiennent du code exécutable et sont les briques de base de construction dans les assemblages de composants logiciels. Les *services* fournis et requis par un composant primitif sont accessibles grâce à des *ports primitifs* qui sont les seuls points d'entrée d'un composant primitif.

Les *composites* sont utilisés comme mécanisme pour gérer un groupe de composants comme un tout, en cachant certaines fonctionnalités des composants de ce groupe. Notre modèle de composant n'impose pas de limite dans le nombre de niveaux de composition. Il existe deux façons de définir l'architecture d'une application : utiliser les *connecteurs* entre les ports de composants ou utiliser un composite pour encapsuler un groupe de composants, appelées aussi respectivement composition horizontale ou composition verticale. Un *connecteur* associe le port d'un composant avec un port situé sur un autre composant. Deux ports peuvent être liés

ensemble seulement si les interfaces requises de l'un sont fournies par l'autre et vice-versa. Les services fournis et requis par un composant fils d'un composant composite sont accessibles grâce à des *ports délégués* qui sont aussi les seuls points d'entrée d'un composant composite. Un *port délégué* d'un composite est connecté à un et un seul port de composant fils. La partie structurelle est présentée sur la figure 1.4.

#### 1.2.1.2 Comportement du composant

Le méta-modèle de composant d'UML 2 utilise les statecharts pour représenter le comportement des composants avec son environnement. Ce formalisme est limité pour représenter les informations de temps. Une transition peut avoir une propriété temporelle mais celle-ci est uniquement exprimable avec $after$ correspondant à un écoulement minimum de temps. Cette limitation nous a conduits à changer de formalisme pour représenter le comportement du composant. Nous avons choisi un formalisme extensible pour intégrer une gestion du temps plus riche et utilisable lors de la vérification de l'assemblage.

Avec les définitions d'interface et de service, le composant déclare des éléments structurels sur les services fournis et requis. La spécification de comportement définit l'interaction du composant avec son environnement. Le comportement est décrit par une algèbre de processus pour laquelle nous utilisons le modèle des automates à entrées/sorties [LT89] pour vérifier le système.

**L'algèbre de processus.** Pour spécifier le comportement du composant, nous utilisons une algèbre de processus simple, inspirée de FSP [Mag99]. L'algèbre de processus est fondée sur une expression décrivant un ensemble de traces (séquences d'évènements). Appliqué à un composant, un événement est une abstraction d'un appel de service ou d'une réponse à un appel. Par exemple, un appel de $m1$ sur l'interface $i1$ du port $p1$ est noté par $p1.i1.m1$ et une réponse à ce service $p1.i1.m1\$$. Chaque évènement est émis par un composant et reçu par un autre. L'appel de $m1$ par l'interface $i1$ du port $p1$ est vu comme l'émission de $!p1.i1.m1$ par le composant $C1$ (notée par un évènement $!C1.p1.i1.m1$). La réception de $p3.i2.m1$ est vue par $C2$ comme $?p3.i2.m1$ (notée par $?C2.p3.i2.m1$ du point de vue de $C2$).

Les opérateurs utilisés dans les protocoles de comportement sont : $\rightarrow$ pour la séquence, $|$ le choix 'alterné' et $*$ la répétition finie. Cette algèbre sert à représenter le comportement des composants primitifs.

**Le modèle d'automate à entrée/sortie.** En plus de l'algèbre de processus, nous utilisons le formalisme des automates à entrée/sortie pour effectuer la vérification. Chaque comportement défini avec l'algèbre de processus est transformé en automate à E/S.

**Définition 1.1** *(Automate à entrée/sortie) Un automate à entrée/sortie est un n-uplet* $(S, L, T, s_0)$ *avec :*

- $S$ *est un ensemble fini non vide d'états ;*
- $L$ *est un ensemble fini non vide de labels.* $L = I \bigcup O$ *où* $I$ *est un ensemble d'entrées et* $O$ *les sorties et* $I \bigcap O = \emptyset$ *;*

- $T \subseteq S(L \bigcup \{\tau\})S$ *est un ensemble fini de transitions où* $\tau$ *est une action interne non observable, ;*
- $s_0 \in S$ *est l'état initial.*

**Composition d'automates à entrée/sortie** La composition de composants dans notre modèle est fondée sur la synchronisation d'une sortie d'un composant avec l'entrée du composant qui lui est connectée. La composition de deux automates à entrée/sortie $B_1 = (S_{B_1}, L_{B_1}, T_{B_1}, s_{0_1})$ et $B_2 = (S_{B_2}, L_{B_2}, T_{B_2}, s_{0_2})$ est définie par l'automate $B = (S_B, L_B, T_B, s_{0_B})$ où :

- $S_B = S_{B_1} \times S_{B_2}$,
- $L_B = I_B \bigcup O_B$ où $I_B = (I_{B_1} \bigcup I_{B_2}) \backslash (O_{B_1} \bigcup O_{B_2})$ et $O_B = O_{B_1} \times O_{B_2}$,
- $s_0 = (s_{0_1}, s_{0_2})$,
- L'ensemble des transitions $T_B$ de $B_1 \parallel B_2$ est défini par les règles suivantes :
  - $a \in (I_{B_1} \backslash O_{B_2}) \bigcup \{\tau\} \wedge (s_1, a, s'_1) \in T_{B_1} \Rightarrow \forall s_2 \in S_{B_2}, ((s_1, s_2), a, (s'_1, s_2)) \in T_B$
  - $a \in (I_{B_2} \backslash O_{B_1}) \bigcup \{\tau\} \wedge (s_2, a, s'_2) \in T_{B_2} \Rightarrow \forall s_1 \in S_{B_1}, ((s_1, s_2), a, (s_1, s'_2)) \in T_B$
  - $a \in (O_{B_1} \backslash I_{B_2}) \bigcup \{\tau\} \wedge (s_1, a, s'_1) \in T_{B_1} \Rightarrow \forall s_2 \in S_{B_2}, ((s_1, s_2), a, (s'_1, s_2)) \in T_B$
  - $a \in (O_{B_2} \backslash I_{B_1}) \bigcup \{\tau\} \wedge (s_2, a, s'_2) \in T_{B_2} \Rightarrow \forall s_1 \in S_{B_1}, ((s_1, s_2), a, (s_1, s'_2)) \in T_B$
  - $a \in I_{B_1} \bigcap O_{B_2} \wedge (s_1, a, s'_1) \in T_{B_1} \wedge (s_2, a, s'_2) \in T_{B_2} \Rightarrow ((s_1, s_2), a, (s'_1, s'_2)) \in T_B$
  - $a \in I_{B_2} \bigcap O_{B_1} \wedge (s_1, a, s'_1) \in T_{B_1} \wedge (s_2, a, s'_2) \in T_{B_2} \Rightarrow ((s_1, s_2), a, (s'_1, s'_2)) \in T_B$

La composition des automates à entrée/sortie est associative et commutative. Quand l'architecte compose plusieurs composants, l'ordre de composition n'est donc pas important.

Pour rester cohérent avec le sous-ensemble UML2 sélectionné, cette algèbre de processus peut être vue comme une représentation textuelle d'un sous-ensemble de diagrammes de séquences où les rôles, identifiés dans le diagramme, sont les ports du composant.

#### 1.2.1.3 Contrats de composition

Nous ne définissons pas de formalismes particuliers pour les trois premiers niveaux de contrat. Nous supposons qu'un mécanisme d'assemblage de composants existe et prend en compte les contrats lors de la vérification de l'assemblage.

### 1.2.2 Automates temporisés

Les automates temporisés sont un modèle de systèmes réactifs à temps continu introduit par [AD94]. Un automate temporisé est un automate étendu avec des horloges qui sont un ensemble de variables augmentant uniformément avec le temps.

#### 1.2.2.1 Syntaxe

Un automate temporisé est constitué de localités. Un ensemble de transitions permet de passer d'une localité à une autre. Une transition peut porter un label, une garde et une initialisation. Chaque localité a un invariant. Nous supposons les automates temporisés déterministes, ils ont donc une seule localité initiale. Formellement, un automate temporisé est défini comme suit :

**Définition 1.2** *(Automate temporisé)*
*Un automate temporisé est un n-uplet* $A = < S, S_0, X, L, T, I, P >$*où :*

- $S$ *est un ensemble fini de localités ;*
- $S_0$ *est la localité initiale ;*
- $X$ *est un ensemble fini d'horloges. A chaque horloge, nous assignons une estimation* $v \in V$, $v(x) \in R^+$ *pour tout* $x \in X$ *;*
- $L$ *est un ensemble fini de labels ;*
- $T$ *est un ensemble fini de transitions.* $T \subseteq S \times C(X) \times L \times 2^X \times S$*, où* $C(X)$ *définit l'ensemble des conjonctions des contraintes d'horloges de la forme* $x \sim c$ *où* $x \in X$ *et* $\sim \in \{<, \leq, =, \geq, >\}$ *et* $c \in N$*.* $2^X$ *est l'ensembles des horloges à réinitialiser lors de la transition ;*
- $I$ *est l'invariant de A,* $I \in C(X)$ *;*
- $P$ *associe un ensemble de propositions atomiques à un état.*

Nous limitons les gardes à des comparaisons d'horloges seulement avec des entiers et non entre horloges. Cette limitation a pour but d'éviter l'explosion combinatoire présentée par [Bou03].

La Figure 1.5 présente un automate temporisé modélisant un train traversant d'un passage à niveau. L'automate temporisé est constitué de quatre localités, quatre transitions et d'une horloge. La localité initiale a la propriété $IDLE$. La première transition, portant $approach$, correspond à l'approche du passage à niveau. Une fois cette transition tirée, le train a 5 unités de temps pour le franchir et s'en éloigner. Cette condition est représentée par les invariants dans les localités. La seconde transition, portant $enter$, est gardée par la condition que le train doit entrer dans le passage à niveau au moins 2 unités de temps après son signal d'approche. La sortie et l'éloignement du train doivent se faire en respectant les invariants des localités.

#### 1.2.2.2 Sémantique

Un automate temporisé s'interprète à l'aide d'un système fini de transitions $S(A) =< Q, Q_0, L, R >$défini par :

**Définition 1.3** *Sémantique des automates temporisés*

- $Q$ *est l'ensemble des états. Un état est un couple* $(s, v)$*.* $s$ *est une localité et* $v$ *est une valuation d'horloges. Une valuation d'horloges est une fonction qui associe une valeur à une horloge.*
- $Q_0$ *est l'état initial du système. Son couple localité/valuation est la localité initiale et une valuation donnant* 0 *à toutes les horloges.*
- $L$ *est un ensemble fini de labels, le même que celui de l'automate temporisé.*
- $R$ *est la relation de transitions. Elle est définie par :*
  - *transition temporelle :* $(s, v) \rightarrow^c (s, v')$ *si* $v' = v + c$ *et pour tout* $0 \leq e \leq c, v + e$ *vérifie la contrainte* $I(s)$*.*
  - *transition discrète :* $(s, v) \rightarrow^l (s', v')$ *s'il existe* $< s, l, \psi, \lambda, s' > \in T$ *tel que v satisfait* $\psi$ *et* $v = v[\lambda := 0]$*.*

$v[\lambda := 0]$ est défini par :

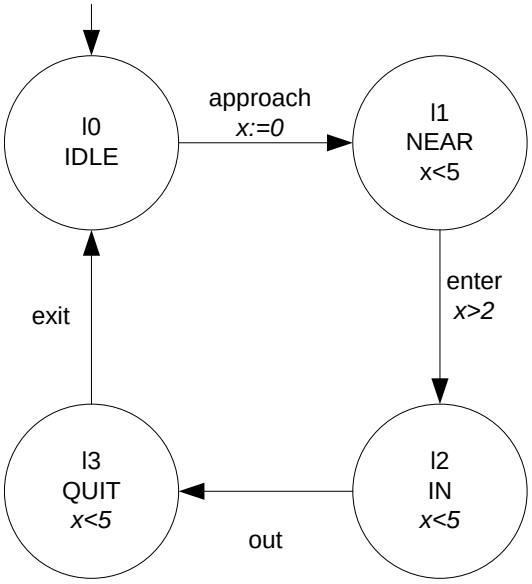



FIG. 1.5 – Exemple d'automate temporisé.

**Définition 1.4** *Réinitialisation d'horloges*

$$v[\lambda := 0](x) \begin{cases} x := 0, \textit{ si } x \in \lambda \\ v(x), \textit{ sinon} \end{cases}$$

Cette relation permet la réinitialisation d'un ensemble d'horloges.

La relation $R$ contient deux types de transitions : discrète ou temporelle. Les transitions discrètes sont instantanées et ne laissent pas le temps s'écouler. Les transitions temporelles permettent l'écoulement du temps mais ne portent pas de label.

Un $q$-chemin est une suite d'états de l'automate temporisé ayant pour état initial $q = (s, v)$ avec une valuation d'horloge nulle. Formellement, il est défini par :

**Définition 1.5** *$q$-chemin*

*Etant donné un automate temporisé $A = < S, S_0, X, L, T, I, P >$, $s \in S$ et $q = (s, v)$ un $q$-chemin est une séquence infinie de la forme :*

*$q$-chemin :* $(s, v) \rightarrow_{l_1}^{c_1} (s_1, v_1) \rightarrow_{l_2}^{c_2} (s_2, v_2) \ldots$

*où :*

- $v(x) = 0$, $x \in X$
- *pour tout $i \geq 1$, il existe une transition $t \in T$ de la forme $(s_{i-1}, g_i, l_i, \lambda_i, s_i)$ telle que $(v_i + c_i - c_{i-1})$ satisfait $g_i$ et $v_i$ est égale à $(v_i + c_i - c_{i-1})[\lambda_i = 0]$.*

La Figure 1.6 présente deux systèmes de transitions interprétant l'automate temporisé du train. Les deux systèmes illustrent le passage du train dans le passage à niveau. Dans le premier, le train arrive près de la barrière puis entre dans le passage après l'écoulement de 3 unités de temps, il en ressort et s'éloigne après 1 unité de temps. Dans le second, le train s'approche après 2 unités de temps et entre dans le passage après 4 unités de temps puis sort et s'éloigne.

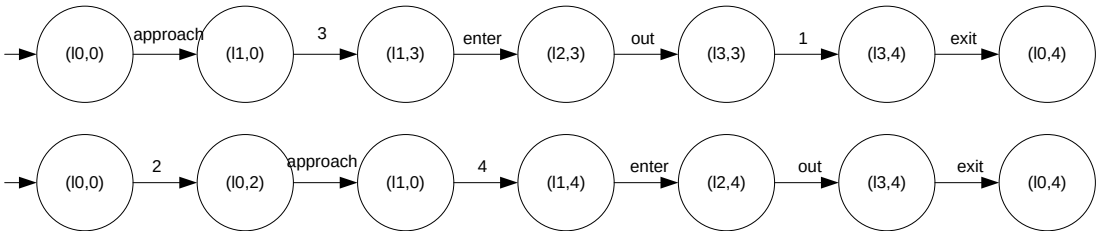



FIG. 1.6 – Système de transitions de l'automate temporisé du train.

Selon les communautés, la traduction des termes anglais peut varier. Une localité peut être appelée place, un état appelé configuration et un système de transitions une trajectoire.

Nous utiliserons dans cette thèse des automates temporisés à entrées/sorties. L'ensemble des labels $L$ est divisé en deux ensembles disjoints $I$ et $O$ représentant respectivement les entrées et les sorties. Une entrée est précédée du symbole ? et une sortie de !. Selon l'utilisation faite de l'automate temporisé, les deux ensembles seront mélangés dans $L$ ou séparés. Lors de la spécification les deux ensembles seront utilisés afin de synchroniser les composants. Si l'automate correspond au comportement global d'une application, les entrées/sorties sont des actions internes donc il n'y a plus de notion d'envoi ou de réception mais seulement d'occurence de message.

### 1.2.3 TCTL

TCTL [ACD93], *Time Computationnal Tree Logic*, est une extension de la logique temporelle CTL [CES86], *Computationnal Tree Logic*, avec des opérateurs quantitatifs temporels. En CTL, la formule $\exists\Diamond p$ exprime que le prédicat $p$ peut devenir vrai le long de certains chemins d'exécution, mais sans information sur le moment où il devient vrai. L'extension TCTL peut répondre à cette deuxième question : on peut enrichir la formule précédente afin de préciser le moment où $p$ devient vrai. Par exemple la formule $\exists\Diamond_{<5}p$ est vraie le long des chemins d'exécution où $p$ est vraie avant 5 unités de temps.

#### 1.2.3.1 Syntaxe TCTL

Soient $P$ un ensemble de propriétés et $N$ l'ensemble des entiers naturels.

**Définition 1.6** *(Syntaxe TCTL)*
*Les formules $\psi$ en TCTL sont définies par :*
$\psi := p|false|\psi_1 \rightarrow \psi_2|\exists\psi_1 U_{\sim c}\psi_2|\forall\psi_1 U_{\sim c}\psi_2$
*où $p \in P$, $c \in N$, et $\sim \in \{<, \leq, =, \geq, >\}$.*

Des abréviations sont définies par :
- $\exists\Diamond_{\sim c}\psi$ pour $\exists trueU_{\sim c}\psi$ (possibilité),
- $\forall\Diamond_{\sim c}\psi$ pour $\forall trueU_{\sim c}\psi$ (des localités le long de toutes les exécutions),
- $\exists\Box_{\sim c}\psi$ pour $\neg\forall\Diamond_{\sim c}\neg\psi$(toutes les localités le long d'une exécution),
- $\forall\Box_{\sim c}\psi$ pour $\neg\exists\Diamond_{\sim c}\neg\psi$ (toutes les localités le long de toutes les exécutions).

#### 1.2.3.2 Sémantique

La relation de satisfaction $\models$ d'une formule TCTL en un état $(s, v)$ du système de transition $ST$ d'un automate temporisé est définie par :

**Définition 1.7** *(Sémantique TCTL)*
*Etant donné un système de transition* $S(A) =< Q, Q_0, L, R >$ *et un état* $q = (s, v)$ *tel que* $q \in Q$*, la relation de satisfaction est :*

- $q \models p$ *si* $p \in P(s)$*, si* $p$ *est vraie dans la localité* $s$*,*
- $q \not\models false$
- $q \models \psi_1 \rightarrow \psi_2$ *si* $\neg q \models \psi_1 \vee q \models \psi_2$
- $q \models \exists \psi_1 U_{\sim c} \psi_2$ *si et seulement si il existe un* $q$*-chemin tel qu'il existe un état* $q_i$ *du chemin tel que* $q_i \models \psi_2$ *et que pour tout* $0 \leq j < i, q_j \models \psi_1$
- $q \models \forall \psi_1 U_{\sim c} \psi_2$ *si et seulement si pour tous les* $q$*-chemin, il existe un état* $q_i$ *du chemin tel que* $q_i \models \psi_2$ *et que pour tout* $0 \leq j < i, q_j \models \psi_1$

La logique TCTL permet d'écrire des propriétés temporelles avec une quantification du temps. Nous avons choisi cette logique car elle est décidable, *PSPACE-complete*, pour les automates temporisés et l'algorithme de satisfaction est implantée dans des model-checkers.

### 1.2.4 Giotto

Giotto [HKH03] est une plateforme proposant une abstraction de temps en le séparant des fonctionnalités. Cette abstraction est écrite à l'aide d'un langage spécifique à Giotto. Giotto a été développé pour des applications temps-réel pouvant s'exécuter dans un environnement distribué. Un programme Giotto permet de spécifier les interactions possibles entre les acteurs du système et leur environnement. Ce programme est composé de quatre concepts principaux :

- les *ports*,
- les *tasks*,
- les *drivers*,
- les *modes*.

Les *ports* permettent d'effectuer les communications acteurs/environnement. Trois types de *ports* sont définis selon leur rôle : *sensors*, *actuators*, *inputs*, *outputs* et *privates*. Les *sensors* sont mis à jour par l'environnement, les autres le sont par Giotto. Les *inputs* et les *outputs* sont les paramètres des *tasks*. Les *privates* sont utilisés pour la communication entres des *tasks* concurrentes.

Les *tasks* représentent des appels à des fonctions. Ils peuvent avoir des paramètres définis à l'aide de *ports*. Ces fonctions sont écrites dans le langage de l'implantation, Java ou C.

Les *drivers* permettent d'effectuer des opérations gardées. Ils font appel dans un premier temps à une fonction booléenne et selon la valeur retournée, appellent ou non la seconde fonction. Ils ont des paramètres pouvant être passés à l'une ou l'autre ou les deux fonctions le composant.

Les *modes* sont les éléments majeurs d'un programme en Giotto. A un instant donné, un programme Giotto ne peut se trouver que dans un seul *mode*. Il est composé d'une période, d'un ensemble d'*Outputs*, d'invocations de *tasks*, de mise à jour d'*actuators* et d'un ensemble

de changement de *mode*. Les invocations, les mises à jour et les changements ont une fréquence et seront appelés pendant la période selon cette fréquence. Un changement de *mode* décrit la transition d'un *mode* à un autre. Il est composé d'une fréquence, du *mode* cible et d'un *driver*. La garde du *driver* est évaluée selon la fréquence du changement.

Une fois le programme Giotto écrit, il est compilé afin d'obtenir un ordonnancement des actions à effectuer à l'exécution. Cet ordonnancement est appelé *Ecode*.

Les différents éléments de Giotto font appel à des fonctions extérieurs pour l'exécution des *tasks* et des *drivers*. Ces fonctions, codées en Java ou C, doivent respecter un format défini par Giotto : retour de valeur, héritage...

**Exemple** Le code de la Figure 1.7 réprésente le contrôle d'un ascenseur en Giotto. Le programme est composé de deux *sensors* permettant de connaître la position de l'ascenseur et de savoir si des boutons ont été pressés. Deux *actuators* permettent d'activer les moteurs de la porte et de l'ascenseur. Cinq *tasks* correspondent aux fonctionnalités de l'ascenseur : monter, descendre, ouvrir la porte, fermer la porte et attendre. Les trois *drivers* de changement de *modes* ont des gardes permettant de savoir si l'ascenseur doit monter, descendre ou ouvrir la porte. Le programme commence dans le *mode* $idle$. Chaque *mode* a une période de $500$ ms. Chacun met à jour les deux *actuators* puis regarde si on doit effectuer un changement. Par exemple, $idle$ a trois changements possibles : monter, descendre ou ouvrir la porte. Si aucune de ces actions n'est possible alors on appelle la *task* $IDLE$ et on recommence la période. Si l'ascenseur a été appelé, la cage monte ou descend, puis quand elle est arrivée au bon étage, ouvre la porte. La fermeture de la porte est automatique, le *driver* $True$ rendant toujours vrai.

La plateforme Giotto offre une abstraction de temps permettant la séparation des fonctionnalités du comportement temporel. Les fonctionnalités peuvent être écrite en C ou en Java et ne sont donc pas restreints à un langage propriétaire.

## 1.3 Etat de l'art

Cette thèse a pour cadre la qualité de service temporelle. De nombreux travaux existent pour représenter la qualité de service et plus particulièrement le temps. Des travaux s'intéressent à la qualité de service au niveau de la spécification de l'application et à l'écriture de contrats de qualité de service. D'autres travaux s'intéressent à la vérification des la qualité de service pendant l'exécution.

### 1.3.1 Spécification de temps

#### 1.3.1.1 CQML

**Présentation** CQML [Aag01], *Component Quality Modelling Language*, est un langage lexical pour la spécification de la qualité de service. La QdS est représentée sous forme de caractéristiques. Chaque caractéristique correspond à un aspect de QdS et peut être quantifiable. Les caractéristiques peuvent être groupées en catégories. Une caractéristique peut être une durée mais elle n'est pas contrainte. Une catégorie permet d'exprimer que la durée doit être infé-

```
sensor
   PortButtons  buttons  uses GetButtons;
   PortPosition position uses GetPosition;

actuator
   PortMove     motion   uses PutMoveMotor;
   PortDoor     door     uses PutDoorMotor;

output
   PortMove     tmotion := InitPortMove;
   PortDoor     tdoor   := InitPortDoor;
   bool_port    openwin := Elevator;

task Idle(PortButtons b) output (tmotion, tdoor) state () {
   schedule TaskIdle(b, tmotion, tdoor)}

task Up(PortButtons b) output (tmotion, tdoor) state () {
   schedule TaskUp(b, tmotion, tdoor)}

task Down(PortButtons b) output (tmotion, tdoor) state () {
   schedule TaskDown(b, tmotion, tdoor)}

task Open(PortButtons b) output (tmotion, tdoor) state () {
   schedule TaskOpen(b, tmotion, tdoor)}

task Close(PortButtons b) output (tmotion, tdoor) state () {
   schedule TaskClose(b, tmotion, tdoor)}

// Actuator driver
driver Move(tmotion) output (PortMove m) {
  if constant_true() then copy_PortMove(tmotion, m)}

driver Door(tdoor) output (PortDoor d) {
  if constant_true() then copy_PortDoor(tdoor, d)}

// Input driver
driver getButtons (buttons) output (PortButtons b) {
  if constant_true() then copy_PortButtons(buttons, b)}

// Mode switch driver
driver PGTC(buttons, position) output () {
   if CondPosGTCall(buttons, position) then dummy()}

driver PLTC(buttons, position) output () {
   if CondPosLTCall(buttons, position) then dummy()}

driver PEQC(buttons, position) output () {
   if CondPosEQCall(buttons, position) then dummy()}
driver True() output () {if constant_true() then dummy()}

start idle {
        mode idle() period 500 {
                actfreq 1 do motion(Move);
                actfreq 1 do door(Door);
                exitfreq 1 do up(PLTC);
                exitfreq 1 do down(PGTC);
                exitfreq 1 do open(PEQC);
                taskfreq 1 do Idle(getButtons);
        }
        mode up() period 500 {
                actfreq 1 do motion(Move);
                actfreq 1 do door(Door);
                exitfreq 1 do open(PEQC);
                taskfreq 1 do Up(getButtons);
        }
        mode down() period 500 {
                actfreq 1 do motion(Move);
                actfreq 1 do door(Door);
                exitfreq 1 do open(PEQC);
                taskfreq 1 do Down(getButtons);
        }
        mode open() period 500 {
                actfreq 1 do motion(Move);
                actfreq 1 do door(Door);
                exitfreq 1 do close(True);
                taskfreq 1 do Open(getButtons);
        }
        mode close() period 500 {
                actfreq 1 do motion(Move);
                actfreq 1 do door(Door);
                exitfreq 1 do idle(True);
                taskfreq 1 do Close(getButtons);
        }
}
```

FIG. 1.7 – Code Giotto pour un ascenseur.

rieure à une valeur donnée. Une catégorie représente une partie de la QdS d'une application. Les catégories sont ensuite regroupées en profils pour être liées à un composant du système.

**Conclusion** CQML permet de décrire les contrats de qualité de service fournis et requis par un composant. Un composant est responsable de sa QdS offerte et attend des garanties sur ceux qu'il requiert. CQML permet la description de contrats de QdS de niveau syntaxique. Le niveau sémantique est principalement expliqué en anglais sans bases formelles. CQML est aussi peu implanté dans des outils permettant la vérification de contrat lors de l'assemblage de composants. Le manque de sémantique ne permet une validation formelle des contrats lors de la spécification de l'application.

#### 1.3.1.2 Profil Marte

**Présentation** Le profil Marte, *UML profile for Modeling and Analysis of Real-Time and Embedded systems* a pour objectif d'étendre UML 2 afin de pouvoir l'utiliser dans une approche dirigée par les modèles temps-réel. Ce profil remplace le profil SPT, *Schedulability, Performance and Time*, défini pour UML 1.4. Les attentes de l'OMG pour ce profil étaient :

- une modélisation unifiée du système pour le logiciel et le matériel,
- avoir une interopérabilité entre les outils de développement au niveau spécification, vérification et génération de code,
- pouvoir faire des prévisions quantitatives dans des modèles en tenant compte des caractéristiques logiciel et matériel.

Le temps est divisé en quatre sous-domaines :

- *TimeStructure* est constitué d'ensembles d'instants partiellement ordonnés,
- *TimeAccess* regroupe les moyens d'accès à la structure de temps,
- *TimeValueSpecification* permet de dénoter les instants et les durées,
- *TimeUsage* introduit les éléments de modélisation des événements, comportements et contraintes temporels.

Marte a été voté en juin 2007 à l'OMG.

**Conclusion** Le profil Marte permet de combler les lacunes du profil SPT et est conforme à UML 2. La sémantique de MARTE est seulement exprimable en langage naturel, en anglais technique. Elle n'est pas liée à un modèle mathématique bien formé. L'absence de sémantique formelle ne permet pas d'effectuer de vérification formelle sur les propriétés introduites dans le profil.

#### 1.3.1.3 Uppaal

**Présentation** Uppaal [YPD94, LPY97] est un environnement permettant de modéliser, valider et vérifier des systèmes temps-réel représentés par des réseaux d'automates temporisés étendus avec des types de données (entiers bornés, tableaux). Les applications typiques sont celles où le temps est critique comme les contrôleurs temps-réel et les protocoles de communication. Uppaal est composé de trois parties : un langage de description, un simulateur et un model-checker. Le langage de description est un langage de commande non-déterministe avec

des types de données. Il sert de langage de modélisation ou de spécification pour décrire le comportement du système par des réseaux d'automates étendus avec des horloges et des variables de données. Le simulateur est un outil de validation permettant l'étude des exécutions dynamiques possibles d'un système lors de la phase de spécification. Ceci permet d'obtenir une détection de faute à moindre coût avant l'utilisation du model-checking. Le model-checker peut vérfier les invariants et l'atteignabilité de propriétés en explorant les états du système, c'est-à-dire une analyse d'atteignabilité en termes d'états symboliques représentés par des contraintes. Afin de faciliter le débogage, le model-checker peut générer une trace de diagnostic expliquant pourquoi une propriété est satisfaite ou non par la description d'un système. Cette trace est visualisable dans le simulateur.

Uppaal a des extensions permettant des analyses de meilleur coût, la génération de test ou la vérification de jeux temporisés.

**Conclusion** Uppaal permet de représenter le comportement des différents acteurs du système que l'on souhaite développer. Une fois la spécification effectuée, le comportement global du système peut être simulé et des propriétés peuvent être vérifiées. Cependant, Uppaal s'adresse à des architectes ayant une expérience avec les formalismes temporels comme la compréhension d'un réseau d'automates temporisés et l'écriture de propriétés en logiques temporelles. L'utilisation d'Uppaal requiert la connaissance des automates temporisés et des logiques temporelles. Enfin Uppaal permet la vérification d'un sous-ensemble de TCTL, on ne peut pas imbriquer plusieurs formules TCTL.

#### 1.3.1.4 ConFract

**Présentation** ConFract [CRCR05] représente par des contrats la spécification et la vérification de propriétés fonctionnelles et extra-fonctionnelles de composants logiciels Fractal [BCL$^+$04]. Lors de la spécification, ConFract établit les contrats lors de l'assemblage des composants. Ces contrats seront utilisés comme base lors de la configuration et à l'exécution des composants. ConFract distingue trois types de contrats : interface, composition externe et composition interne. Une extension permet de prendre en compte les violations de contrats afin d'avoir une négociation plus fine lors d'une violation.

**Conclusion** ConFract permet la construction de contrats lors de la spécification qui seront utilisés à l'exécution. L'écriture des spécifications est effectué à l'aide d'un langage dédié à Java et Fractal mais peut permettre une écriture simplifiée. ConFract est pour l'instant spécifique au modèle à composant Fractal même si son utilisation pourrait être possible sur d'autres modèles à composant. Enfin, ConFract ne prend pas en compte des propriétés de qualité de service de temps dans ses contrats.

#### 1.3.1.5 Synthèse sur la spécification

De nombreux autres travaux et profils existent sur la spécification de temps lors de la phase de conception. La plupart se base sur une syntaxe permettant d'exprimer les propriétés temporelles mais peu ont une base sémantique forte afin d'effectuer une réelle vérification de

ces propriétés. Ces approches s'adressent particulièrement à des architectes ayant de bonnes connaissances dans les formalismes temporels ou dans l'utilisation de profils UML. Nous souhaitons pouvoir aider les architectes voulant compléter leur application avec des propriétés temporelles sans avoir à devenir spécialistes en temps.

### 1.3.2 Temps à l'exécution

#### 1.3.2.1 BIP

**Présentation** BIP [BBS06], *Behavior, Interaction, Priority*, a pour objectif de développer une théorie, des méthodes et des outils pour la construction de systèmes temps-réel constitués de composants hétérogènes. L'outil permet une composition incrémentale des composants hétérogènes. Les sources d'hétérogénéité sont liées à l'interaction, l'exécution et l'abstraction. Les systèmes développés sont corrects par construction pour des propriétés comme l'exclusion mutuelle ou l'absence de verrous. Cette construction permet de minimiser la validation *a posteriori*. L'intégration des composants et la génération de code glue sont automatisées en respectant des exigences données.

**Conclusion** BIP permet de modéliser des composants et de les valider par construction. L'utilisation de BIP demande une bonne connaissance des mécanismes utilisés lors de l'implantation de composants et ne peut donc pas être utilisée par des architectes s'intéressant uniquement aux fonctionnalités offertes. La modélisation du temps par BIP doit être effectuée explicitement et les composants doivent se synchroniser à chaque écoulement de temps afin de continuer leur exécution.

#### 1.3.2.2 Approche de Chan/Poernomo

**Présentation** Les travaux de [CPSJ05] définissent une méthode permettant la génération d'un moniteur de qualité de service à partir de modèles UML en utilisant une approche MDA, *Model Driven Architecture*. La spécification est le PIM, *Platform Independant Model*, et est décrite à l'aide d'un profil UML. Les propriétés de qualité de service sont exprimées dans une logique temporelle probabiliste, PCTL (*Probabilistic Computational Tree Logic*). Ces propriétés sont dynamiquement vérifiées par le moniteur durant l'exécution de l'application. Ce moniteur est implanté dans .NET, le PSM (*Platform Specific Model*) et généré à partir des spécifications par une transformation de modèles.

**Conclusion** L'approche permet la génération automatique d'un moniteur de qualité de service à partir de la spécification d'une application. La vérification des contrats est effectuée uniquement à l'exécution. Aucune vérification n'est effectuée lors de la spécification de l'application.

#### 1.3.2.3 Times

**Présentation** Times [AFP+02] est un outil de modélisation et d'analyse d'ordonnancement pour des systèmes embarqués temps-réel. Times permet la création d'automates temporisés

étendus avec des taches grâce à un éditeur graphique. Le système peut être simulé afin de vérifier le comportement dynamique. Un vérificateur permet l'analyse d'ordonnancement. Il vérifie que les états accessibles du système sont ordonnançables. Ce vérificateur s'appuie sur celui d'Uppaal. Un générateur de code permet d'obtenir du code exécutable pour le firmware brickOS en C. Le code fourni a, sous hypothèses que le vérificateur a trouvé un ordonnancement, le comportement défini lors de la spécification.

**Conclusion** Times permet le développement de systèmes embarqués en fournissant une suite d'outils de simulation vérifiant l'ordonnancement et générant du code. Cet outil vise particulièrement des architectes à l'aise avec le formalisme utilisé et ayant une bonne connaissance de la future implantation du système. En effet, dès le début de la spécification, l'architecte doit savoir si les tâches sont préemptives ou pas et connaître les durées d'exécution afin d'obtenir un ordonnancement. Cette démarche s'applique bien pour les systèmes embarqués qui n'ont pas de qualité de service de temps mais un temps critique.

#### 1.3.2.4 Synthèse sur le temps à l'exécution

La vérification des propriétés de temps lors de l'exécution doit permettre de s'assurer que l'application respecte bien sa spécification. Deux approches permettent de s'en assurer : avoir un code contraint par le temps ou avoir un moniteur surveillant l'exécution. Un code contraint par le temps est nécessaire pour les applications embarquées mais ne convient pas lorsque le temps fait parti uniquement de la qualité de service. En effet, la fonctionnalité doit être fournie même si la qualité est médiocre et donc continuer si le temps est écoulé. L'utilisation d'un moniteur permet de surveiller cette qualité de service. Sa génération permet de réduire les erreurs de développement et de pouvoir le reproduire si un changement a lieu dans la spécification.

### 1.3.3 Conclusion sur l'état de l'art

La qualité de service temporelle permet d'avoir des informations sur les fonctionnalités offertes par l'application. La spécification de la qualité de service de temps doit permettre une vérification formelle des propriétés lors de la spécification de l'application. Les notations uniquement syntaxiques ne permettent pas cette vérification. Une fois la vérification effectuée, l'architecte souhaite savoir si les propriétés, définies lors de la spécification, sont correctes lors de l'exécution de l'application. Cette vérification peut être faite en contraignant l'application au niveau du temps ou en surveillant l'exécution. La contrainte est intéressante pour les logiciels ou le temps est prépondérant. Nous plaçons notre approche dans les applications où le temps n'est pas critique. Nous souhaitons surveiller à l'exécution les propriétés de qualité de service. Cette surveillance est effectuée à l'aide d'un moniteur généré à partir des spécifications de l'application.

# Chapitre 2

# Ajout de propriétés temporelles dans les composants

Lors de la spécification d'un système, l'architecte peut désirer introduire des informations de temps dans les composants. Si le système développé est un système temps-réel, l'architecte utilise directement des formalismes temporisés pour représenter les propriétés temporelles qu'il souhaite intégrer. Dans le cas où le temps ne fait pas parti des fonctionnalités demandées par le cahier, l'architecte n'est pas toujours à l'aise avec les formalismes temporels. Celui-ci peut vouloir ajouter des propriétés temporelles afin d'obtenir une information sur la qualité de service de temps. L'ajout des propriétés temporelles ne peut être intéressant que si celles-ci sont vérifiées lors de la phase de conception. Notre objectif est d'aider l'architecte en lui fournissant une méthode pour ajouter facilement du temps dans les composants à partir d'éléments simples. Nous utilisons une approche à base de motifs. Un motif permet de représenter une propriété temporelle de façon simple, le temps de réponse d'une fonctionnalité par exemple. L'utilisation d'un motif permet à l'architecte de ne pas avoir besoin de connaître la représentation de la propriété mais uniquement ce qu'elle signifie. Lors de la spécification, l'architecte choisit un ensemble de motifs correspondants aux propriétés qu'il souhaite intégrer. Cet ensemble est automatiquement ajouté et est utilisé lors de l'assemblage de composants pour vérifier que les propriétés sont bien respectées.

Ce chapitre a pour objectif de définir une méthode pour ajouter des propriétés extra-fonctionnelles de temps, appelées dans la suite propriétés temporelles, dans des applications à composants. Ces propriétés sont souvent orthogonales aux propriétés fonctionnelles et elles sont difficiles à ajouter. Elles représentent des informations de qualité de service et ne sont pas toujours essentielles à l'exécution du système. Ces propriétés peuvent être violées mais cela n'empêchera pas le système de fonctionner correctement. Dans le cas d'une violation, l'architecte peut décider de conserver l'assemblage de composants avec une mauvaise qualité de service ou remplacer des composants afin d'obtenir une meilleur qualité de service. La gestion du temps dans la spécification n'est pas aisée car le temps est une ressource particulière qui ne peut être arrêtée ou modifiée. De plus, si l'architecte veut effectuer des vérifications sur les propriétés temporelles, la représentation du temps doit avoir une sémantique permettant cette vérification.

L'objectif de l'ajout des propriétés de temps est de pouvoir les intégrer facilement et sans modifier les fonctionnalités déjà définies pour les composants. Les informations temporelles sont ajoutées dans les composants et doivent être utilisées lors de la composition des composants. En effet, les propriétés temporelles correspondent à ce qu'un composant peut requérir et offrir. Elles doivent donc être vérifiées lors d'une composition pour savoir si la qualité de service est correcte entre les deux composants. Le temps est ajouté à deux endroits : sur le comportement du composant lui-même et dans les contrats attachés aux interfaces requises. L'ajout dans les contrats permet d'ajouter des propriétés temporelles que le composant requiert lors de l'assemblage. Les contrats temporels sont les contrats de composition requis. L'ajout dans le comportement permet lui d'intégrer le temps dans ce que le composant fournit lors de l'assemblage. Le comportement est le contrat de composition offert.

Ce chapitre présente dans un premier temps l'ajout de temps dans le comportement. Cet ajout se fait à l'aide de motifs de temps. Un motif de temps représente une propriété temporelle que l'on souhaite intégrer au comportement. Le comportement temporel du composant est représenté à l'aide d'un automate temporisé résultant de l'ajout des motifs. Le comportement est le contrat de composition offert par un composant. Dans un second temps, nous présentons l'ajout de temps dans les contrats des interfaces requises par le composant. Un contrat temporel est un conjonction de formules de logique temporelle écrites en TCTL. Enfin, nous expliquons comment utiliser les informations temporelles lors de l'assemblage des composants pour valider les assemblages déjà définis. Nous comparons les contrats de compositions requis, les formules TCTL, avec les contrats de compositions offerts, les automates temporisés.

## 2.1 Ajout de propriétés temporelles dans le comportement des composants

Lors de la spécification de composant, l'architecte décrit ce que que le composant fournit et comment s'organisent les services fournis. L'enchaînement de ces services correspond au comportement du composant. Nous allons ajouter des informations temporelles dans ce comportement pour enrichir ce que le composant propose. Afin d'aider l'architecte dans l'ajout des propriétés temporelles, nous définissons un ensemble de motifs temporels correspondant aux propriétés. Chaque motif correspond à une propriété temporelle et est paramétré par les éléments propres à la propriété : le ou les messages, la garde et l'intégration du motif. Dans cette partie, nous allons présenter quatre motifs temporels : temps de réponse, temps d'exécution, période et délai. Puis nous illustrerons l'intégration de ces motifs sur le comportement d'un composant.

### 2.1.1 Motifs temporels

Afin de faciliter l'ajout de temps dans le comportement, nous définissons un ensemble de motifs temporels s'inspirant des travaux de [GO03] : temps de réponse, délai, temps d'exécution, période d'appel de service, durée, etc. Ils définissent un ensemble de propriétés nécessaires pour représenter le temps dans un profil UML. Nous présentons la définition d'un motif temporel puis les quatre premiers motifs, cet ensemble n'est pas exhaustif.

#### 2.1.1.1 Définition d'un motif

Un motif temporel requiert trois types de paramètres : le ou les services concernés, l'opérateur de comparaison $\sim$ compris dans l'ensemble $\{<, \leq, =, \geq, >\}$ et la borne de comparaison de la propriété temporelle. Un motif va ajouter dans le comportement une horloge et des propriétés atomiques qui seront utilisées lors la vérification des contrats. Les propriétés atomiques permettent de savoir quel message vient d'être reçu. Dans le cas où plusieurs transitions arrivent dans le même état, nous voulons être sûr du message qui vient d'être émis ou reçu. Afin de garantir ceci, chaque transition, portant un message concerné par un motif, est transformée en deux transitions et une localité. La première transition est la transition portant le message mais a comme cible la nouvelle localité. Cette localité contiendra la propriété désirée et aura comme invariant un temps nul. La seconde transition a comme cible, la localité cible de la transition d'origine et ne portera ni garde, ni message, ni réinitialisation d'horloges. Cette transition est donc instantanée et ne modifie pas le comportement de l'automate. La Figure 2.1 illustre cette transformation.

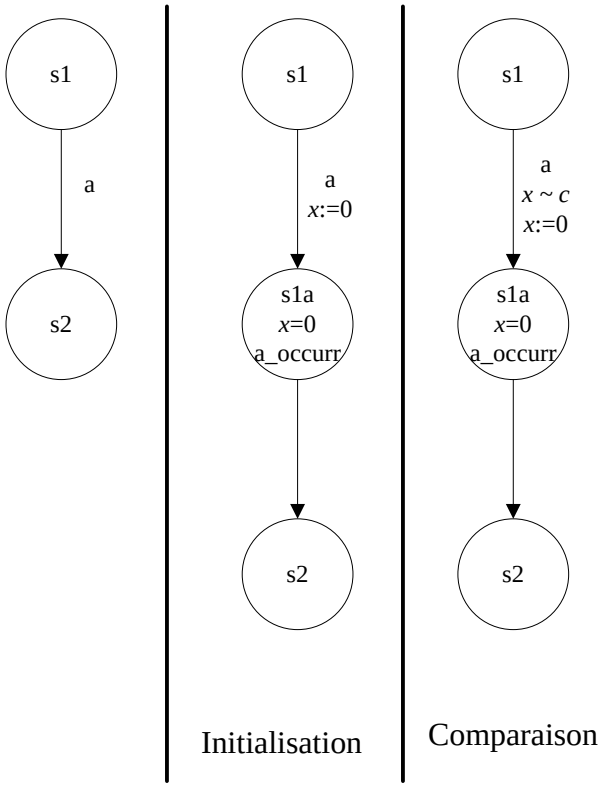



FIG. 2.1 – Dépliage instantané d'une transition.

#### 2.1.1.2 Durée

Le motif *(*durée) (D) permet de représenter le temps écoulé entre deux messages. Les paramètres de ce motif sont les deux messages $m1$ et $m2$, l'opérateur de comparaison $\sim$ et la borne de comparaison $c$.

**Description** La Figure 2.2 présente le motif générique $D$ pour la propriété temporelle durée. Ce motif exprime une durée entre deux labels $m1$ et $m2$. Les paramètres sont les deux labels, l'opérateur de comparaison et la valeur de la durée. Le motif se compose de trois localités, de deux transitions et d'une horloge. La transition entre les deux premières localités initialise l'horloge lors de l'occurrence du label $m1$. La seconde localité a la proposition atomique $m1\_occur$. La transition entre les deux dernières localités compare l'horloge avec la valeur de la durée lors de l'occurrence du second label $m2$. La troisième localité a la proposition atomique $m2\_occur$.

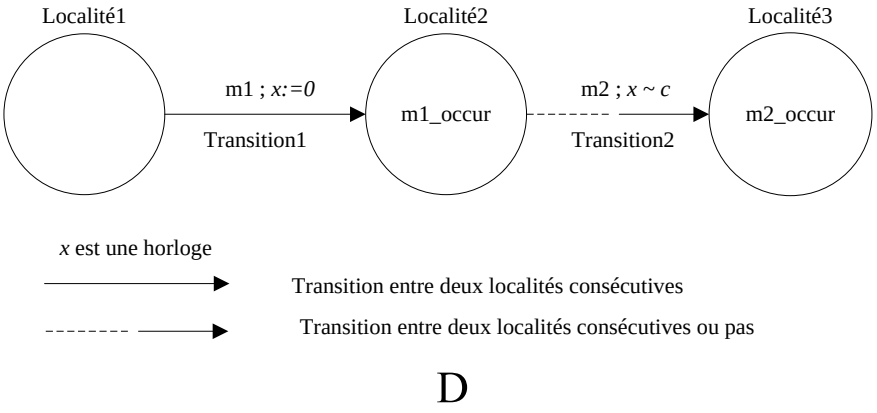



FIG. 2.2 – Motif durée.

**Application** Le motif *durée* (D) s'applique en vérifiant dans un premier temps si les messages $m1$ et $m2$ appartiennent bien à l'automate du composant. Les deux transitions portant les deux messages ne sont pas obligatoirement consécutives dans l'automate, d'autres transitions peuvent être présentes entre elles. Ceci est représenté graphiquement par la ligne pointillée entre la seconde localité et la seconde transition. Une nouvelle horloge $x$ est ajoutée à l'automate. Ensuite, la transition portant le label $m1$, notée $t1$, est sélectionnée et l'horloge $x$ est initialisée sur cette transition. La transition $t1$ est ensuite dépliée et la localité ajoutée est enrichie de la proposition $m1\_occur$. L'ensemble des transitions portant le label $m2$ est sélectionné. Une sélection est ensuite effectuée sur cet ensemble pour ne sélectionner que les transitions accessibles à partir de $t1$ respectant les conditions suivantes :

- (1) il n'existe pas d'autres transitions portant $m1$ entre $t1$ et la transition. Cette condition permet de s'assurer que la propriété portera bien entre la dernière occurrence du message $m1$.

– (2) il n'existe pas d'autres transitions portant $m2$ entre $t1$ et la transition. Cette condition permet de s'assurer que la propriété portera bien sur la première occurrence du message $m2$.

Un cas particulier est une transition portant le message $m1$ ou $m2$ et ayant comme source et cible la même localité. Dans ce cas, un dépliage de un pas doit être effectué dans l'automate. En effet, si la transition boucle sur la localité, la propriété atomique deviendra vraie avant l'occurrence réelle du message.

Ces conditions sont illustrées par la Figure 2.3.

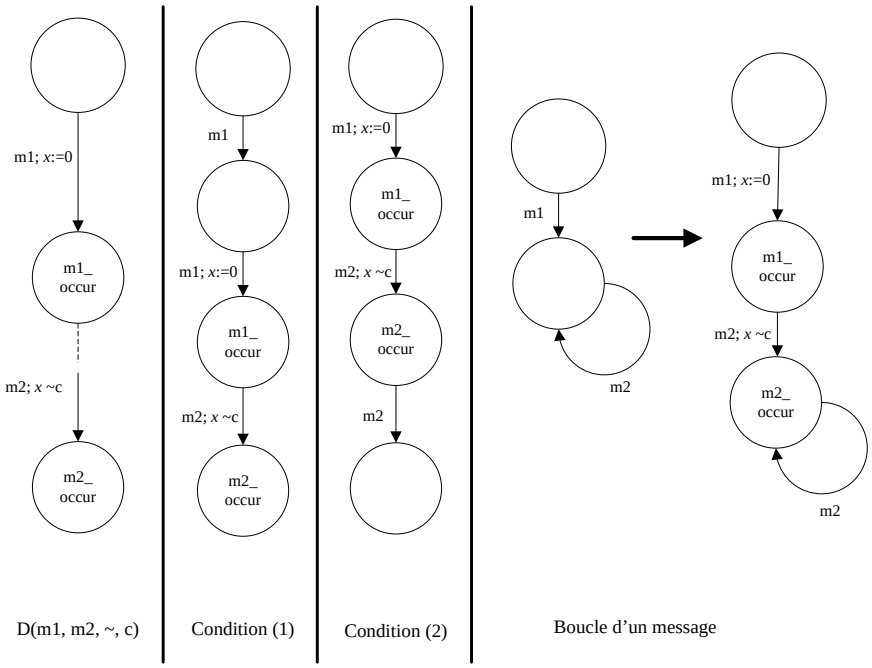



FIG. 2.3 – Motif durée.

La garde $x \sim c$ est ajoutée à l'ensemble des transitions satisfaisant ces conditions. Les transitions complétées par la garde sont dépliées et la localité ajoutée est enrichie de la proposition $m2\_occur$. L'ensemble des localités des chemins respectant les conditions (1) et (2) menant d'une localité cible de la transition $m1$ à une localité source d'une transition portant $m2$ a son invariant complété par la garde $x \sim c$. Ceci permet d'exprimer qu'entre l'occurrence des deux messages le temps est contraint par la garde.

**Avantages/Inconvénients** Ce motif est le plus général car il permet de représenter une écoulement de temps entre deux messages quelconques de l'automate. Il peut être utilisé pour représenter un temps de réponse entre un appel de service et son acquittement. Un motif spécifique a été défini permettant de ne pas préciser l'acquittement par l'architecte. Le motif *durée* ne peut pas être utilisé pour modéliser une période pour un même message ou un temps d'exécution, nécessitant seulement un seul message et ayant une application spécifique.

#### 2.1.1.3 Temps de réponse

Le motif *temps de réponse* (RT) décrit la propriété temporelle correspondant au temps entre un message et l'obtention de sa réponse, son acquittement. C'est un cas particulier du motif *durée*. Par exemple, pour exprimer un temps de réponse sur le service $getSound$, une horloge doit être initialisée lors de l'appel à $getSound$ et doit être comparée à une valeur lors de la réception de son acquittement. Ce motif requiert trois paramètres : le service à appeler $service$, l'opérateur de comparaison $\sim$ et la borne de comparaison $c$. Le motif permet d'exprimer qu'entre l'occurrence de $service$ et de son acquittement $service\$$ il s'écoule une quantité de temps conforme à $\sim c$.

**Description** L'automate $RT$ de la Figure 2.4 représente le motif générique du temps de réponse. Le motif est composé de trois localités, deux transitions et une horloge. Cette horloge est initialisée lors de la première transition avec l'appel de service et est vérifiée lors de la seconde transition avec l'acquittement de l'appel. La localité cible de la première transition est enrichie avec la propriété $service\_occurr$ et la localité cible de la seconde transition avec la propriété $service\$\_occurr$.

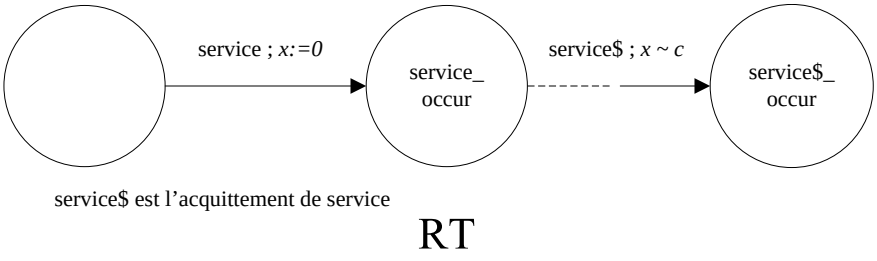



FIG. 2.4 – Motif temps de réponse.

**Application** Lors de l'ajout de ce motif au comportement du composant, nous vérifions dans un premier temps que le service et son acquittement sont bien présents dans l'automate. Le service et son acquittement peuvent ne pas être consécutifs dans l'automate. Sur la transition portant le message $service$, nous ajoutons une nouvelle horloge qui est initialisée. Cette transition est dépliée. La proposition $service\_occurr$ est ajoutée dans la localité cible de la transition. L'ajout sur les transitions portant l'acquittement $service\$$ suit les mêmes règles que le motif *durée* ainsi que l'ajout des invariants.

**Avantages/Inconvénients** Ce motif permet d'intégrer simplement un temps de réponse sur un service offert par le composant. Ce motif ne s'ajoute que sur une seule occurrence de l'appel et toutes les réponses accessibles depuis cette occurrence. Cette limitation vient du fait que si tous les appels sont pris en compte, l'initialisation d'horloge s'effectue à chaque occurrence de l'appel. Si on peut trouver dans l'automate deux appels successifs, celui à prendre en compte doit être choisi par l'architecte.

#### 2.1.1.4 Temps d'exécution

Le motif *temps d'exécution* (ET) représente le temps pris par le traitement d'une information. Par exemple, après la réception d'un message, le composant peut avoir besoin d'un certain temps pour traiter cette réponse. Le motif a trois paramètres : le message $message$, l'opérateur de comparaison $\sim$ et la borne de comparaison $c$.

**Description** L'automate $ET$ de la Figure 2.5 représente le motif générique de temps d'exécution. Le motif comprend trois localités, deux transitions et une horloge. L'horloge est initialisée lors de l'occurrence de $message$ sur la première transition et comparée sur la seconde sans aucun label. La seconde localité a la propriété $message\_occurr$ et la troisième $message\_execend$.

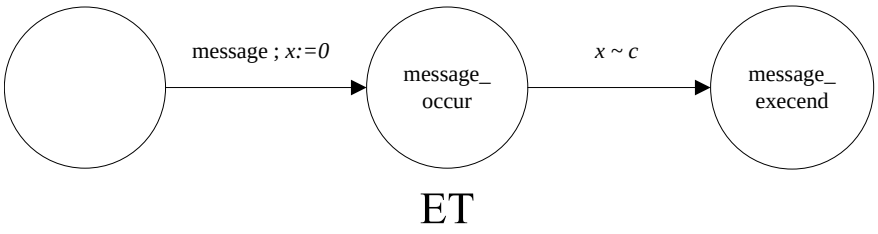



FIG. 2.5 – Motif temps d'exécution.

**Application** Ce motif a plusieurs façons de s'appliquer selon le composant : bloquant ou non-bloquant.

Le temps de traitement peut s'effectuer en bloquant le composant et donc en empêchant le composant de recevoir ou d'émettre des messages. Dans ce cas, l'ajout du motif doit simuler ce blocage en interdisant les échanges pendant la garde après l'occurrence de $message$. Nous sélectionnons la transition portant $message$ en parcourant l'automate. La propriété $message\_occurr$ est ajoutée dans la localité cible, notée $s$. Aucune transition portant un message ne peut partir de cette localité à cause du traitement. On peut simuler cette attente en créant une transition vers une nouvelle localité $s\_exec$ avec la garde du motif $x \sim c$. Cette nouvelle localité aura la propriété $message\_execend$. Elle n'est atteignable que si la garde est vérifiée et le composant peut reprendre ses échanges de messages. Les transitions sortantes de $s$ sont donc transférées sur $s\_exec$. Les localités source et cible de la transition portant $message$ ont leur invariant complété par la garde $x \sim c$.

Si le composant peut recevoir ou émettre des messages pendant le traitement, la façon d'intégrer le motif est différente, le composant n'étant pas bloqué dans ses échanges de messages. L'architecte doit alors préciser soit le nombre de messages pouvant être gérés par le composant, soit un message exigeant que le traitement soit terminé. Nous sélectionnons la transition portant $message$ en parcourant l'automate, on note $s$ la localité cible de la transition et elle est dépliée. La propriété $message\_occurr$ est ajoutée dans la localité ajoutée, notée $s$. Dans le cas de la gestion de $n$ messages, la garde $x \sim c$ est ajoutée à toutes les transitions appartenant aux

chemins de longueur $n$ ayant $s$ comme état initial. La *nième* localité de chaque chemin aura la propriété $message\_execend$. Dans le cas d'un message d'arrêt, le motif *durée* sera appliqué avec les paramètres $message$, $message\_arret$, $\sim$ et $c$.

**Avantages/Inconvénients** Ce motif n'est pas applicable si l'architecte ne connaît pas à l'avance les possibilités du composant en termes de traitement : bloquant ou non-bloquant. L'utilisation du pire des cas, c'est-à-dire bloquant, peut être appliqué mais cela ne correspondra peut-être pas au comportement réel de l'implantation du composant.

#### 2.1.1.5 Période

Le motif *période*(P) permet de représenter la propriété temporelle correspondant à une période d'un message. Les paramètres de ce motif sont le message $message$, l'opérateur $\sim$ et la borne de comparaison $c$ de la période.

**Description** La représentation générique est illustrée par la Figure 2.6. Le motif comprend deux localités et une transition. La transition porte le message $message$, la garde $\sim c$ et l'initialisation de l'horloge. La seconde localité a la propriété $message\_occur$.

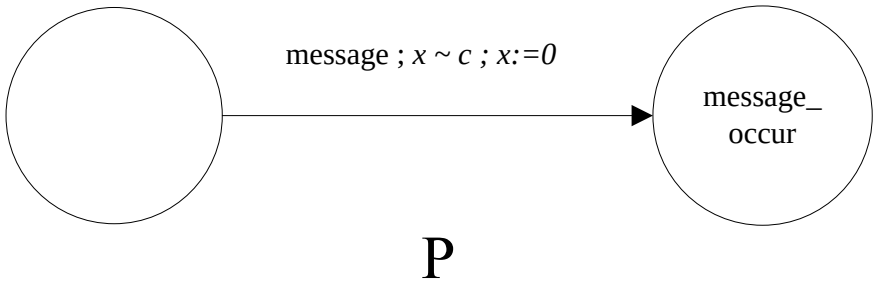



FIG. 2.6 – Motif période.

**Application** Lors de l'application, ce motif a une condition sur la structure de l'automate d'origine. En effet, pour qu'un message ait une période, il faut que ce message apparaisse dans une boucle dans le comportement, sinon ce n'est pas une période mais juste un temps entre deux occurrences du message. Dans ce cas le motif *durée* peut être utilisé.

La boucle ne peut pas contenir que la transition portant $message$. En effet, chaque message est couplé avec un acquittement. Le message du motif est donc soit un message ou un acquittement qui doivent être successifs. Une boucle ne peut pas être constitué d'une transition unique.

Dans un premier temps, le processus d'intégration vérifiera si le message appartient bien à une boucle. Si le message appartient bien à une boucle, le motif est intégré. La transition portant le message est dépliée. La première transition du dépliage est enrichie de la garde $x \sim c$ et de l'initialisation de l'horloge. La localité ajoutée par le dépliage est enrichie de la proposition

$message\_occur$. Lors du premier passage dans la boucle, l'horloge sera initialisée. Lors du second passage, la garde sera vérifiée et l'horloge réinitialisée et ainsi de suite. Au premier passage, la garde sera aussi vérifiée. L'ajout du motif doit assurer que cette vérification sera correcte.

Deux cas se présentent : la localité initiale appartient à la boucle ou non. Si la localité initiale de l'automate est dans la boucle alors la satisfaction de la garde aura les mêmes conditions qu'au second passage dans la boucle. En effet, l'horloge utilisée par le motif ne peut être mise à zéro que sur la transition ayant $message$. Sinon, une initialisation d'horloge doit être ajoutée sur toutes les transitions permettant d'entrer dans la boucle, c'est-à-dire menant d'une localité hors de la boucle à une localité appartenant à la boucle.

Toutes les localités de la boucle ont leurs invariants complétés par la garde $x \sim c$.

Si plusieurs transitions portant $message$ existent dans la boucle, alors la garde $x \sim c$ est ajoutée à toutes ces transitions.

La propriété $message\_occur\_next$ est ajoutée à l'ensemble des localités ayant comme transitions entrantes une transition provenant de la localité cible de la transition portant $message$.

**Avantages/Inconvénients** Le motif *Période* permet d'ajouter la propriété temporelle de l'occurrence d'un message avec une période donnée. Dans le cas où le message n'apparaît pas dans une boucle, on peut tenter d'appliquer le motif *durée* avec le message en premier et second paramètres. Ceci permet de contraindre les occurrences du message dans le comportement.

### 2.1.2 Application des motifs

Après avoir défini les quatre motifs de temps, nous allons maintenant illustrer leur application en les ajoutant dans un comportement d'un composant. L'architecte choisit les différents motifs qu'il souhaite ajouter au composant et renseigne les paramètres nécessaires. Une fois les motifs sélectionnés, ils seront automatiquement intégrés au comportement du composant en transformant l'automate originel en automate temporisé. Les motifs seront combinés successivement avec cet automate pour obtenir le comportement temporisé final. L'ordre d'ajout dans le comportement n'est pas important car les horloges sont toutes indépendantes les unes des autres.

Nous illustrons le processus d'ajout en déroulant pas à pas l'ajout d'un motif de chaque type défini précédemment à l'automate $TA$ de la Figure2.7. Le choix des transitions est défini par la première occurrence du label dans l'automate par défaut. L'architecte peut, s'il le veut, avoir la liste des transitions possibles et choisir celle qu'il désire.

**Ajout du motif temps de réponse** Premièrement, comme illustré Figure 2.7, le motif *temps de réponse*, représenté par l'automate $RT$ au centre, est sélectionné avec les paramètres suivants : $getSound$ pour l'appel de service, l'opérateur $<$ et la valeur $4$. L'automate de comportement d'origine est l'automate $TA$. L'automate résultant de l'ajout est $TA1$. Une horloge $x1$ est d'abord ajoutée à l'automate. Celle-ci est initialisée sur la transition $?getSound$ partant de la localité $s4$. La transition est dépliée. La propriété $getSound\_occurr$ est ajoutée à la localité $s4a$ ajoutée lors du dépliage. La transition portant l'acquittement de l'appel de service

!$getSound$\$ est ensuite sélectionnée et dépliée. La garde $x1 < 4$ est ajoutée à cette transition et la propriété $getSound\$\_occurr$ est ajoutée à la localité cible $s6a$. L'invariant de la localités $s5$ est $x1 < 4$.

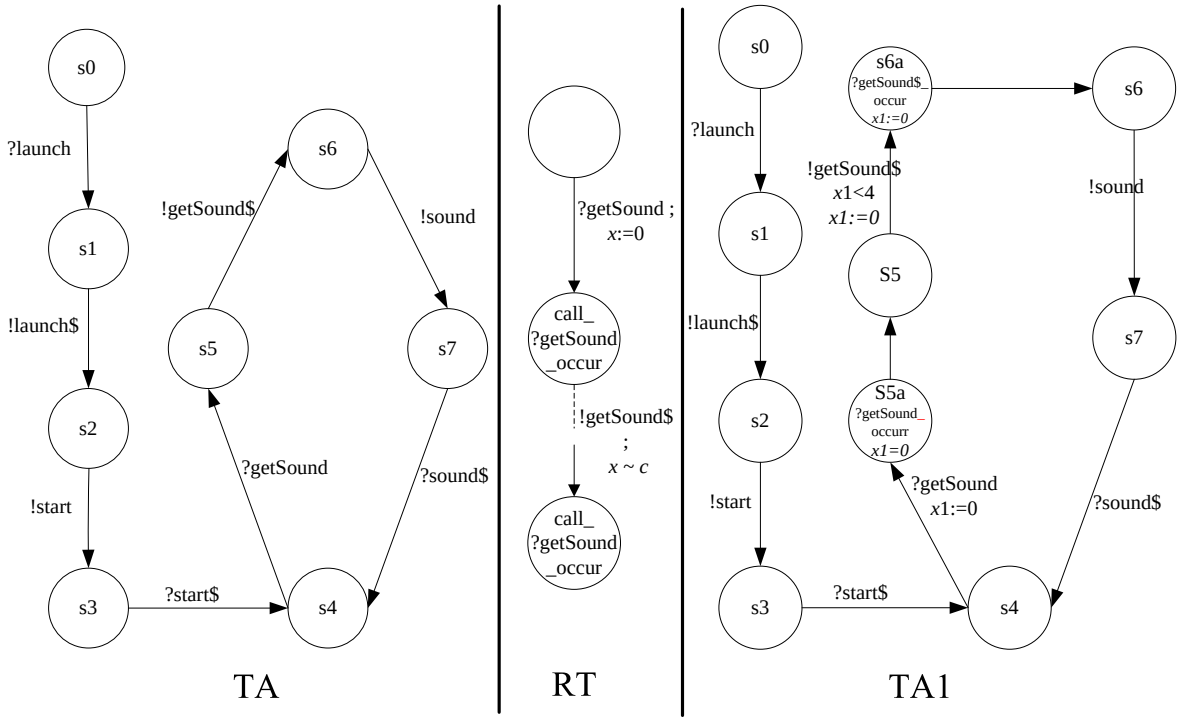



FIG. 2.7 – Ajout du motif temps de réponse

**Ajout du motif temps d'exécution** Nous appliquons ensuite le motif *temps d'exécution* avec les paramètres !$getSound$\$, l'opérateur $<$ et la valeur 2. Nous nous plaçons dans la cas où le composant ne peut pas recevoir de message pendant le traitement. Une deuxième horloge $x2$ est ajoutée et initialisée sur la transition portant !$getSound$\$. Le dépliage pour cette transition n'est pas nécessaire car il a déjà été effectué lors du motif précédent. L'ajout d'une nouvelle localité n'est pas nécessaire car la transition sans message joue le même rôle. Nous ajoutons la garde sur cette transition et la propriété $getSound\$\_execend$ est ajoutée à sa localité cible. L'invariant de la localité $s6a$ est supprimé car on ne veut plus contraindre la transition sortante. Le nouvel automate du comportement est montré par l'automate $TA2$ de la Figure 2.8.

**Ajout du motif période** L'ajout du motif *période* se fait avec les paramètres !$sound$, $<$, 7. Le message !$sound$ appartient à un cycle dans l'automate. Une nouvelle horloge $x3$ est ajoutée dans l'ensemble des horloges de l'automate. La transition ayant comme label !$sound$ est dépliée puis complétée avec la garde $x3 < 7$ et l'initialisation de l'horloge $x3$. L'état initial n'appartient pas au cycle donc une initialisation de $x3$ est ajoutée à la transition reliant les

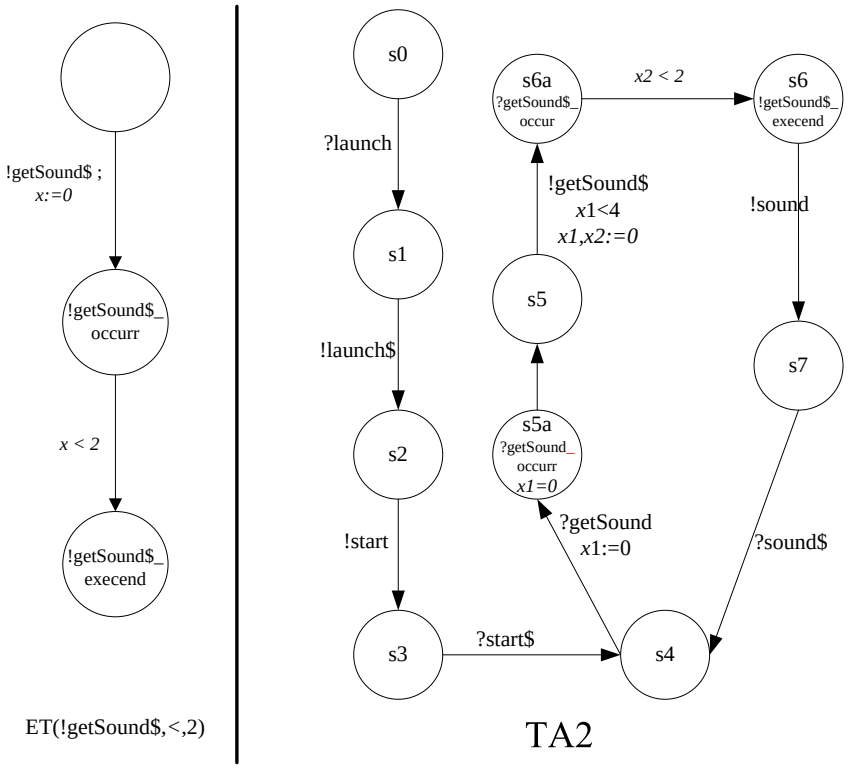



FIG. 2.8 – Ajout du motif temps d'exécution

localités $s3$ et $s4$. Les invariants des localités constituant le cycle sont complétés par la garde $x3 < 7$. L'automate résultant de cet ajout est l'automate $TA3$ de la Figure 2.9.

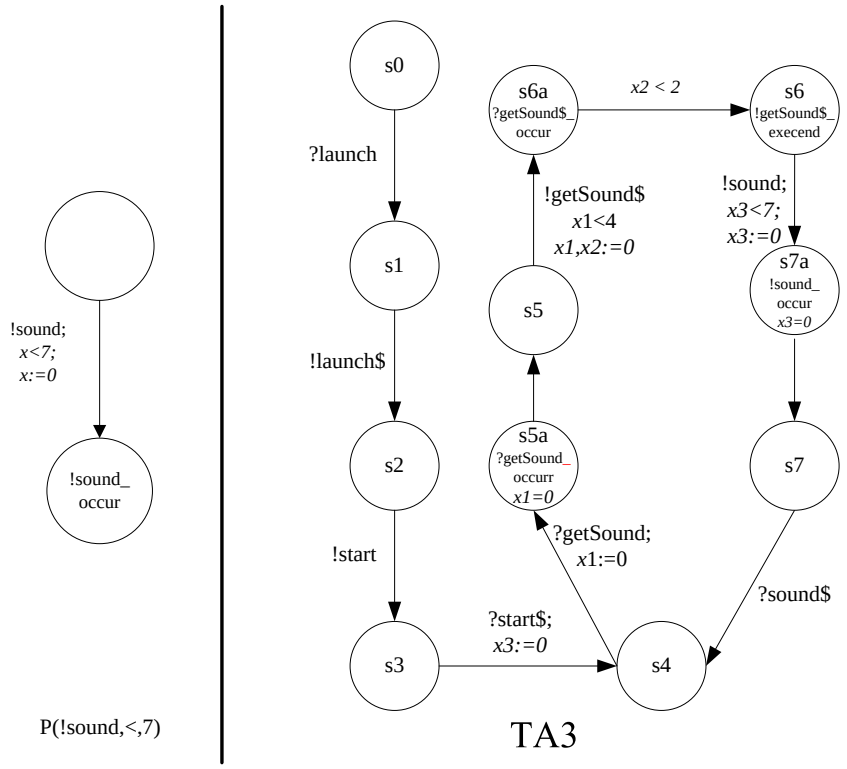



FIG. 2.9 – Ajout du motif période

**Ajout du motif durée** L'ajout du motif durée est effectué avec les paramètres $?launch$, $!start$,$<$ et 1. Une horloge $x4$ est ajoutée à l'ensemble des horloges. La transition portant le label $?launch$ est dépliée puis la réinitialisation de $x4$ est ajoutée. La garde est ensuite ajoutée sur la transition avec comme label $!start$ après l'avoir dépliée. Les invariants de localités $s1$ et $s2$ sont complétés par la garde $x4 < 1$. L'automate obtenu par l'ajout du motif durée est représenté par l'automate $TA4$ de la Figure 2.10.

### 2.1.3 Conservation du comportement

L'ajout de propriétés de qualité de service ne doit pas modifier le comportement d'origine du composant. L'automate temporisé résultant de l'ajout des motifs temporels doit être équivalent au niveau des traces à l'automate d'origine. Les motifs de temps n'ajoutent pas de comportement supplémentaire au niveau des motifs et n'en ajoutent pas lors de la création de localités. Une projection de l'automate temporisé vers un automate classique en supprimant les horloges permet de réobtenir l'automate d'origine. Ceci est effectué par l'outil Kronos [BDM+98] qui fournit un automate non temporisé au format $.aut$ de l'outil ALDEBA-

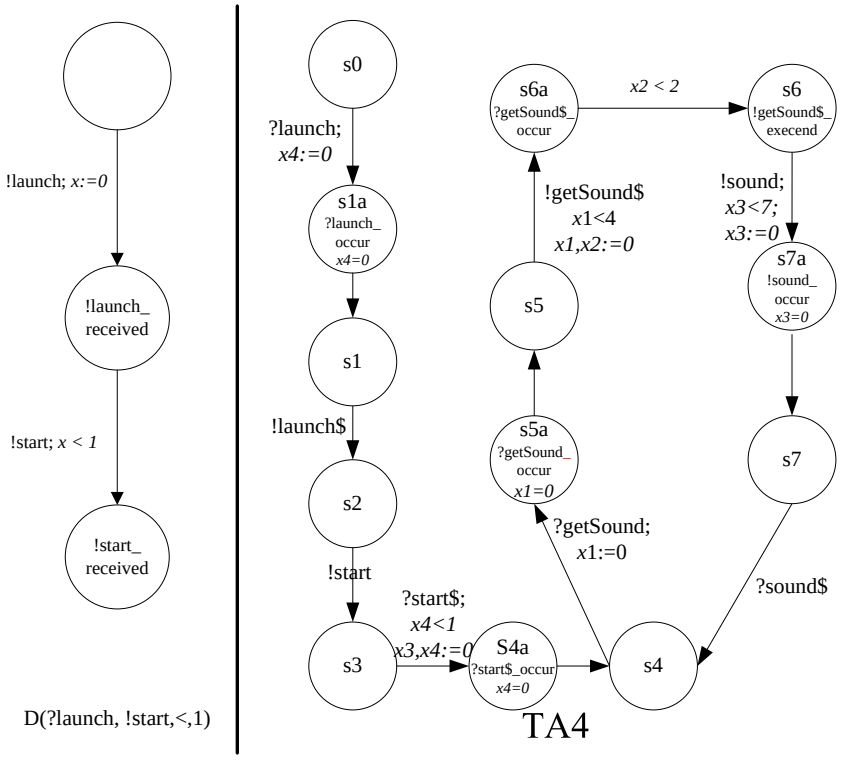



FIG. 2.10 – Ajout du motif durée

RAN [Fer]. Celui-ci permet de comparer des automates. Nous comparons l'automate d'origine avec l'automate obtenu avec Kronos. Les transitions sans labels sont labelisés par le message $tau$ et ne seront pas utilisés lors de la comparaison.

Par exemple, la suppression des informations temporelles de l'automate temporisé permet d'obtenir l'automate $TA4$ de la Figure 2.11 qui est équivalent à $TA$ qui est l'automate d'origine.

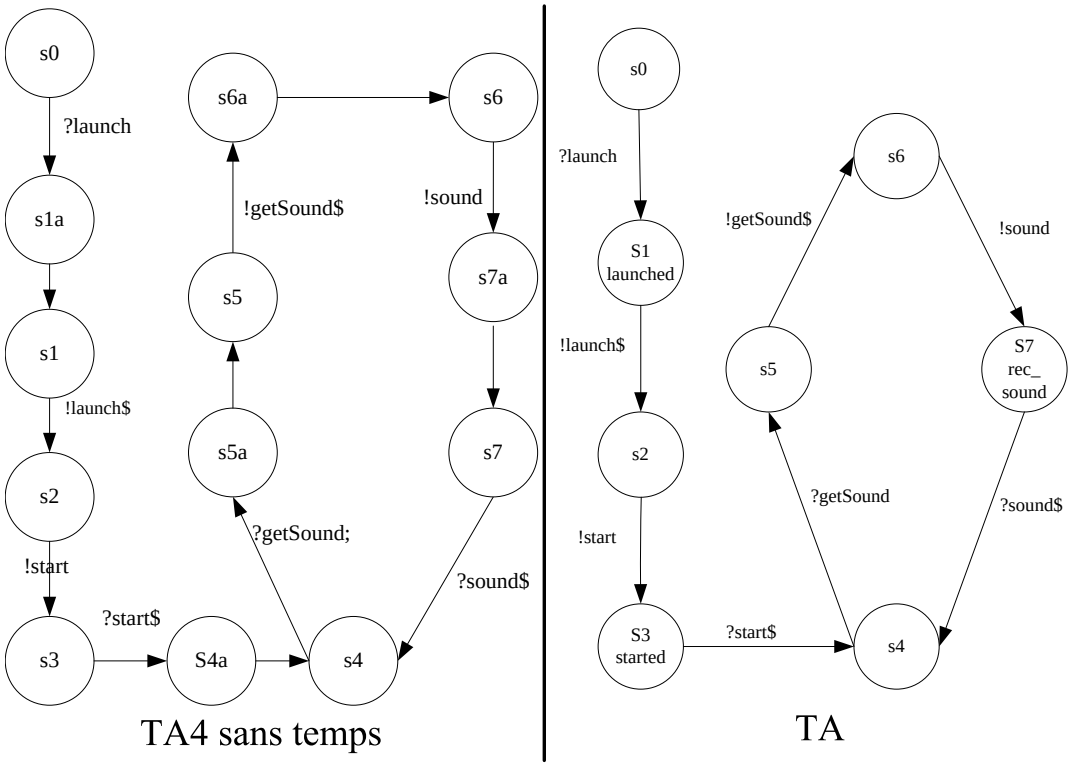



FIG. 2.11 – Projection automate temporisé

### 2.1.4 Conclusion sur le comportement

La définition d'un ensemble de motifs de temps permet l'ajout de propriétés de temps dans le comportement des composants. L'utilisation de motifs permet à l'architecte de ne pas ajouter manuellement les propriétés temporelles dans le comportement. Il peut ainsi ne pas avoir à connaître parfaitement les automates temporisés pour ajouter des informations de qualité de service temporels. L'architecte choisit les motifs qu'il souhaite ajouter avec les paramètres demandées et les propriétés temporelles sont automatiquement intégrés au comportement du composant choisi. Cette intégration se fait avec une forte séparation des préoccupations. Les motifs ne sont pas dépendants du comportement originel du composant mais des fonctionnalités de celui-ci. Si le comportement vient à changer, l'ajout des motifs peut être rejoué sans changement si les fonctionnalités n'ont pas changé.

## 2.2 Ajout de temps dans les contrats

Lors de la conception d'un composant, lorsque les fonctionnalités sont définies, l'architecte renseigne ce que le composant requiert pour bien fonctionner. Ces renseignements sont attachés aux interfaces requises du composant à l'aide de contrats. Les contrats répartissent les informations requises selon quatre niveaux définis dans 1.1.2. Les trois premiers niveaux de contrats sont utilisés pour assembler les composants en respectant les propriétés syntaxiques, comportementales et de synchronisation. L'objectif de cette section est d'enrichir les contrats de composition requis avec le quatrième niveau de contrats pour la qualité de service de temps. Ces contrats de composition sont utilisés lors de la composition des composants. Ils seront confrontés aux automates temporisés du contrat de composition offert par l'autre composant. Afin de vérifier le contrat de composition offert, une logique temporelle peut être utilisée, un model-checker vérifiant la satisfaction de la formule par l'automate temporisé. Nous voulons vérifier des propriétés temporelles donc la logique temporelle doit pouvoir quantifier le temps. Plusieurs logiques temporelles temporisées existent : MTL [Koy90], TCTL [ACD93] ou encore RTGIL [MRK$^+$97]. Nous choisissons TCTL qui dispose d'outils de vérification pour les automates temporisés : UPPAAL [YPD94, LPY97], Kronos [BDM$^+$98] et Hytech [AHH93]. Afin d'ajouter simplement les contrats temporels, nous définissons, comme pour le comportement, un ensemble de motifs correspondants à des propriétés temporelles. Nous présentons dans un premier temps les structures permettant de faire la correspondance entre un pseudo langage et des formules TCTL. Ensuite, nous présenterons comment former à partir de ces structures des contrats temporels.

### 2.2.1 Motifs de spécifications temps-réels

L'écriture de formules de logique temporelle n'est pas évidente pour des personnes peu habituées à les manipuler. Les travaux de [KB05] définissent une grammaire en anglais structuré afin d'écrire des formules dans plusieurs logiques temporelles : LTL [MP92], CTL [CES86], MTL [Koy90], TCTL, RTGIL [MRK$^+$97]. La grammaire, définie dans la Figure 2.12, permet d'écrire une propriété. Une propriété est composée d'une portée et d'une spécification. La portée définit quand la propriété doit être vraie lors de l'exécution du programme :

- globally : tout au long de l'exécution,
- before R : le long de l'exécution jusqu'à R,
- after Q : le long de l'exécution à partir de Q,
- between Q and R : le long de l'exécution entre Q et R,
- after Q until R : le long de l'exécution à partir de Q jusqu'à R, même si R ne devient jamais vrai.

La spécification peut être de type qualitatif ou temps-réel. Le type qualitatif permet d'écrire des formules de logiques temporelles classiques. Le type temps-réel vise à écrire des formules en logique temporelle temporisée en précisant le temps de manière quantitative grâce aux catégories.

Lorsqu'une formule est écrite, il faut la transformer dans la logique choisie. Pour chaque combinaison de spécifications temps-réel, une correspondance entre les portées et la logique choisie est établie. Par exemple, la spécification comprenant les règles $scope, 18, 22, 23$, signi-

| | | | |
|---|---|---|---|
| **Start** | 1: property | ::= | *scope* “, ” *specification* “.” |
| **Scope** | 2: scope | ::= | “Globally” \| “Before ” R \| “After ” Q \| “Between ” Q “ and ” R \| “After ” Q “ until ” R |
| **General** | 3: specification | ::= | *qualitativeType* \| *realtimeType* |
| **Qualitative** | 4: qualitativeType | ::= | *occurrenceCategory* \| *orderCategory* |
| | 5: occurrenceCategory | ::= | *absencePattern* \| *universalityPattern* \| *existencePattern* \| *boundedExistencePattern* |
| | 6: absencePattern | ::= | “it is never the case that ” P “ holds” |
| | 7: universalityPattern | ::= | “it is always the case that ” P “ holds” |
| | 8: existencePattern | ::= | P “ eventually holds” |
| | 9: boundedExistencePattern | ::= | “transitions to states in which ” P “ holds occur at most twice” |
| | 10: orderCategory | ::= | “it is always the case that if ” P “ holds” (*precedencePattern* \| *precedenceChainPattern1-2* \| *precedenceChainPattern2-1* \| *responsePattern* \| *responseChainPattern1-2* \| *responseChainPattern2-1* \| *constrainedChainPattern1-2*) |
| | 11: precedencePattern | ::= | “, then ” S “ previously held” |
| | 12: precedenceChainPattern1-2 | ::= | “ and is succeeded by ” S “, then ” T “ previously held” |
| | 13: precedenceChainPattern2-1 | ::= | “, then ” S “ previously held and was preceded by ” T |
| | 14: responsePattern | ::= | “, then ” S “ eventually holds” |
| | 15: responseChainPattern1-2 | ::= | “, then ” S “ eventually holds and is succeeded by ” T |
| | 16: responseChainPattern2-1 | ::= | “ and is succeeded by ” S “, then ” T “ eventually holds after ” S |
| | 17: constrainedChainPattern1-2 | ::= | “, then ” S “ eventually holds and is succeeded by ” T “, where ” Z “ does not hold between ” S “ and ” T |
| **Real-time** | 18: realtimeType | ::= | “it is always the case that ” (*durationCategory* \| *periodicCategory* \| *realtimeOrderCategory*) |
| | 19: durationCategory | ::= | “once ” P “ becomes satisfied, it holds for ” (*minDurationPattern* \| *maxDurationPattern*) |
| | 20: minDurationPattern | ::= | “at least ” c “ time unit(s)” |
| | 21: maxDurationPattern | ::= | “less than ” c “ time unit(s)” |
| | 22: periodicCategory | ::= | P “ holds ” *boundedRecurrencePattern* |
| | 23: boundedRecurrencePattern | ::= | “at least every ” c “ time unit(s)” |
| | 24: realtimeOrderCategory | ::= | “if ” P “ holds, then ” S “ holds ” (*boundedResponsePattern* \| *boundedInvariancePattern*) |
| | 25: boundedResponsePattern | ::= | “after at most ” c “ time unit(s)” |
| | 26: boundedInvariancePattern | ::= | “for at least ” c “ time unit(s)” |

FIG. 2.12 – Grammaire en anglais structurée.

fiant avoir toujours la propriété $P$ au moins toutes les $c$ unités de temps, a une correspondance de portée pour la logique TCTL définie par :

- globally : $\forall\Box(\forall\Diamond_{\leq c}P)$,
- before $R$ : $\forall[((\forall\Diamond_{\leq c}(P \vee R)) \vee \forall\Box(\neg R))UR]$
- after $Q$ : $\forall\Box(Q \rightarrow \forall\Box(\forall\Diamond_{\leq c}P))$
- between $Q$ et $R$ : $\forall\Box((Q \wedge \neg R) \rightarrow (\forall[((\forall\Diamond_{\leq c}(P \vee R)) \vee \forall\Box(\neg R))UR]))$
- after $Q$ until $R$ : $\forall\Box((Q \wedge \neg R) \rightarrow (\forall[(\forall\Diamond_{\leq c}(P \vee R)UR)]))$

L'ensemble spécification et correspondance forme un motif de spécifications temps-réel.

### 2.2.2 Contrats temporels

A l'instar des contrats des trois premiers niveaux, un contrat de qualité de service est attaché aux interfaces requises. Afin de faciliter la définition des contrats temporels, nous définissons un ensemble de squelettes de contrats temporels à partir des motifs de spécifications temps-réel. Ces squelettes seront complétés par l'architecte en fournissant les paramètres nécessaires et produiront des formules TCTL. L'architecte a aussi la possibilité d'écrire les contrats directement en TCTL.

#### 2.2.2.1 Adaptation des motifs

Les motifs de Konrad et al [KB05] permettent d'obtenir des formules en TCTL avec des propriétés contenues dans les localités. Les motifs temporels définis pour le comportement s'expriment en utilisant des labels apparaissant dans l'automate. Des propriétés sont ensuite ajoutées dans les localités. Nos contrats temporels porteront donc eux aussi sur les labels à l'origine. Le lien avec les propriétés dans les localités sera fait automatiquement.

Comme pour les motifs temporels, nous avons définis un ensemble de squelettes de contrats. Nous allons expliquer la construction des squelettes des trois motifs (*durée*, *temps de réponse* et *période*) à partir des motifs de spécifications temps-réel. Le motif *temps d'exécution* est spécifique au comportement. Si un composant requiert un certain temps pour effectuer un traitement c'est pour recevoir une réponse sous la forme d'un acquittement donc le motif utilisé est celui du *temps de réponse*.

**Durée** Le squelette de contrat *durée* permet de requérir que le temps entre l'occurrence de deux messages soit comparé à une valeur. Les occurrences des deux messages sont notées respectivement $m1\_occur$ et $m2\_occur$. Cette propriété s'écrit en anglais structuré avec les règles 1,2,3,18,24,25 : *Globally, it is always the case that if* $m1\_occur$ *holds, then* $m2\_occur$ *holds after at most x units of time*. Cette formule exprime que après l'occurrence de $m1\_occur$, $m2\_occur$ doit être vraie après au maximum $x$ unités de temps. Cette formule s'écrit en TCTL par :
$\forall\Box(m1\_occur \rightarrow \forall\Diamond_{\leq x}m2\_occur)$

Le squelette ne doit pas seulement exprimer la durée avec l'opérateur $\leq$, nous l'étendons donc à tous les opérateurs. La généralisation de cette formule s'écrit en TCTL par :

$\forall\Box(m1\_occur \rightarrow \forall\Diamond_{\sim x} m2\_occur)$

**Temps de réponse** Le squelette de contrat *temps de réponse* permet de requérir que le temps entre la demande et la réponse, l'acquittement, soit comparé à une valeur. L'occurrence de la demande est notée par $service\_occur$ et la réponse $service\$\_occur$. Cette propriété s'écrit en anglais structuré avec les règles 1,2,3,18,24,25 : *Globally, it is always the case that if service_occur holds, then service$_occur holds after at most x units of time*. Cette formule exprime que après l'occurrence de $service\_occur$, $service\$\_occur$ doit être vraie après au maximum $x$ unités de temps. Cette formule s'écrit en TCTL par :
$\forall\Box(service\_occur \rightarrow \forall\Diamond_{\leq x} service\$\_occur)$

La généralisation de cette formule s'écrit en TCTL par :
$\forall\Box(service\_occur \rightarrow \forall\Diamond_{\sim x} service\$\_occur)$

**Période** Le squelette de contrat *période* permet de requérir qu'un message doit apparaître avec une période donnée. L'occurrence du message est notée par la proposition $message\_occur$ dans la localité cible de la transition. Le motif période s'écrit en anglais structuré avec les règles 1,2,3,18,22,23 et s'écrit : *Globally, it is always the case that message_occur hold at least every x time units*. Cette formule exprime le fait que $message\_occur$ doit être vrai au moins une fois pendant chaque période de $x$ unités de temps. Cette formule s'écrit en TCTL par :
$\forall\Box(\forall\Diamond_{\leq x} message\_occur)$

La formule permet de rester dans la même localité afin de satisfaire la propriété. Nous souhaitons que ce cas ne se présente pas donc nous ajoutons une condition dans la formule. Cette condition est $message\_occur\_next \rightarrow$. La formule est donc :
$\forall\Box(message\_occur\_next \rightarrow \forall\Diamond_{\leq x} message\_occur)$

Nous généralisons cette formule en ajoutant la possibilité d'avoir tous les opérateurs possibles. La formule TCTL du squelette de contrat *période* est donc la suivante :
$\forall\Box(message\_occur\_next \rightarrow \forall\Diamond_{\sim x} message\_occur)$

La vérification de cette formule ne peut être effectuée avec un résultat correct que si l'automate temporisé a été créé avec les motifs temporels définis précedemment.

#### 2.2.2.2 Exemples de contrats

Nous illustrons la création de contrats en enrichissant, dans le composite de la Figure 2.13, le composants $Decoder$ d'un contrat temporel sur son interface $IDinsound$ et en ajoutant un contrat temporel requis par l'environnement.

Le composant $Decoder$ émet le message $start$ vers le composant $Extraction$ et désire qu'il soit opérationnel en au plus une unité de temps, c'est-à-dire que l'acquittement soit reçu

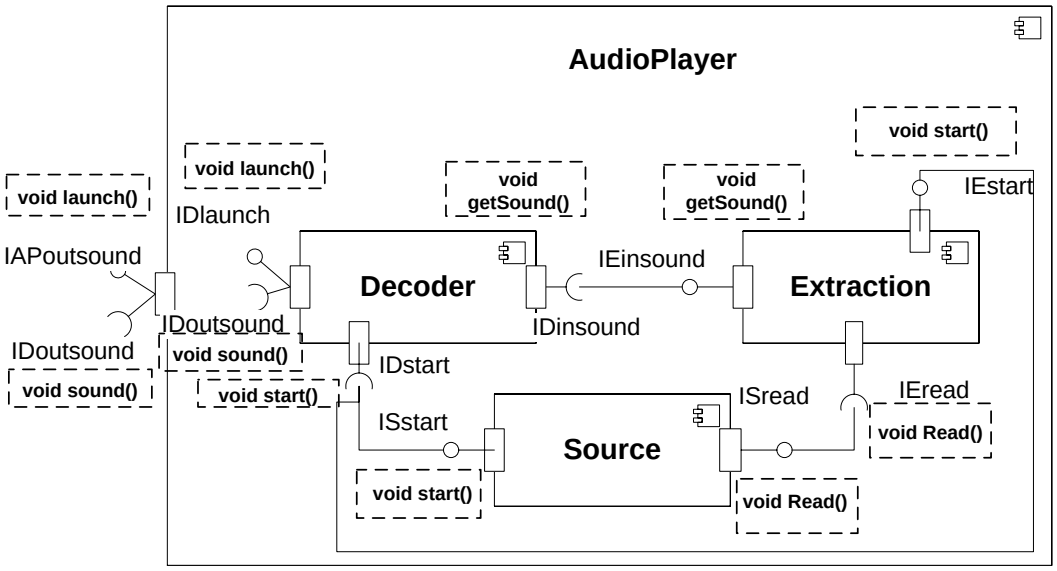



FIG. 2.13 – Exemple.

en au plus une unité de temps. Cette exigence correspond au motif *temps de réponse*. Un contrat temporel est ajouté à l'interface requise contenant le service $start$. Ce contrat s'écrit en TCTL avec la formule suivante :

$\forall\Box(start\_occurr \rightarrow \forall\Diamond_{<1} start\_occurr)$

L'environnement requiert qu'entre le moment où il envoie la demande de lancement de l'application $launch$, il puisse recevoir le premier échantillonage en moins de $8$ unités de temps. Ce contrat correspond au motif *durée* entre l'envoi de $launch$ et la réception de $sound$. Ce contrat s'écrit en TCTL avec la formule suivante :

$\forall\Box(launch\_occurr \rightarrow \forall\Diamond_{<8} sound\_occurr)$

#### 2.2.2.3 Contrats implicitement créés

Certains contrats sont automatiquement créés lorsque des motifs de temps sont ajoutés par l'architecte. Par exemple, lors du choix du motif de temps de réponse sur un appel de service externe, ($!getSound$ par exemple), un contrat est implicitement créé avec ce motif. En effet ce type de propriétés temporelles mettant en jeu des acteurs extérieurs au composant peut se répercuter sur le comportement et sur ce que le composant requiert. La formule TCTL est donc automatiquement créée avec les paramètres du motif. Lors de l'ajout du motif comportemental *temps de réponse* sur l'appel de service $!getSound$, un contrat temporel est implicitement créé et est lié à l'interface contenant $getSound$ afin d'être vérifié lors de l'assemblage de composants.

### 2.2.3 Conclusion sur les contrats

Nous avons défini une méthode afin d'ajouter des contrats temporels de qualité de service dans les composants. Les contrats temporels sont attachés aux interfaces requises des composants et sont ainsi utilisés par le processus de composition. L'approche à base de motifs permet à l'architecte d'écrire les contrats de façon simplifiée sans avoir à savoir comment le contrat est réellement écrit. Ces contrats sont ensuite traduits dans la logique temporelle TCTL. L'utilisation d'une logique temporelle permet d'avoir un contrat avec une sémantique utilisable lors de la composition. L'ensemble de motifs utilisables n'est pas exhaustif et peut être complété : succession de plusieurs messages dans un intervalle de temps par exemple.

## 2.3 Vérifier les propriétés de temps lors de la composition de composants

Lors de l'assemblage de composants, nous vérifions que les services requis sont satisfaits par rapport aux services fournis. Après l'ajout des propriétés temporelles, nous allons utiliser ces propriétés pour valider les assemblages existants. Les propriétés temporelles sont de la qualité de service donc non essentielles à l'assemblage. En effet les propriétés temporelles étant négociables, elles peuvent être falsifiées lors d'un assemblage. Cette vérification permettra de savoir si la qualité de service a un niveau satisfaisant. Une mauvaise qualité de service ne remettra pas obligatoirement en cause les assemblages existants. L'architecte peut décider alors de conserver les contrats établis avec une mauvaise qualité de service ou de les modifier en changeant les contrats de composition non respectés.

Lors de l'assemblage de composants, nous regardons si ce qui est fourni est compatible avec ce qui est requis. Dans le cas des propriétés temporelles, nous regardons si le contrat temporel est correct par rapport au comportement offert. Afin de vérifier cette compatibilité, nous devons vérifier si la formule en TCTL est vérifiée par l'automate temporisé. Le contrat de comportement du composant fournissant le service est l'automate temporisé projeté sur les services de l'interface contenant le service. Le contrat du composant requiérant le service est le contrat temporel attaché à l'interface contenant le service.

Nous utilisons le *model-checker* Kronos [BDM$^+$98] qui permet de vérifier si un automate temporisé satisfait une formule TCTL ou non. Le comportement de chaque composant primitif est décrit par un automate temporisé et les contrats temporels sont exprimés à l'aide de la logique TCTL. Lorsque l'automate temporisé ne satisfait pas la formule, Kronos identifie les localités où la formule n'est pas vérifiée.

Kronos ne permet pas de vérifier des formules contenant les opérateurs de comparaison $>$ et $\geq$ car cela ne correspond pas à des formules d'atteignabilité. Nous modifions ces formules de vivacité en leur équivalentes en sûreté. A la place de demander d'atteindre une propriété en plus d'une quantité de temps, nous demandons que cette propriété ne doit pas être vraie avant la borne désirée. La formule $\forall\Box_{>c}$ est transformé en $\neg\exists\Diamond_{\leq c}$ et $\forall\Diamond_{>c}$ en $\neg\exists\Box_{\leq c}$($\geq$ est transformé en $<$).

Nous allons vérifier le contrat de l'environnement défini dans la section 2.2.2.2 sur le composant $Decoder$. Le contrat de composition offert est l'automate temporisé de $Decoder$. Le

contrat de composition requis est le contrat temporel suivant :
$\forall\Box(launch\_received \rightarrow \forall\Diamond_{<8} sound\_received)$

Les deux contrats de composition sont donnés au model-checker. La réponse est $true$ donc le niveau de qualité de service est correct et le composant est validé. Si le contrat le composition requis avait eu une borne de comparaison inférieure ou égale à 5 par exemple, le model-checker n'aurait pas pas validé la satisfaction de ce contrat par l'automate temporisé. La réponse aurait été que les localités sur un chemin menant à une localité ayant $sound\_received$ peuvent avoir un invariant ne satisfiant pas la contrainte.

Lors d'une composition validée par Kronos, le contrat résultant est l'automate du contrat offert. Cet automate représente le comportement attendu de l'implantation du système en respectant la qualité de service.

## 2.4 Conclusion

L'ajout de temps dans les composants permet l'introduction de propriétés temporelles utilisables lors de l'assemblage de composants. L'ajout se fait dans le comportement et les contrats des composants. Une approche à base de motifs est utilisée, chacun représentant une propriété temporelle. Les motifs permettent de définir de façon abstraite la propriété désirée. Ces motifs sont ensuite traduis dans le formalisme correspondant afin d'être utilisables lors de la composition des composants. L'utilisation de motifs permet à l'architecte d'ajouter les propriétés temporelles sans avoir à écrire des formules de logiques temporelles ou manipuler les automates temporisés. Elle permet aussi un ajout identique pour un même type de propriétés et n'est pas dépendant de la façon de faire d'un architecte. Les propriétés temporelles sont ensuite vérifiées lors de l'assemblage de composants. Ceci est possible car les informations temporelles ne sont pas uniquement syntaxiques mais sont fondées sur des formalismes avec une sémantique bien définie, les automates temporisés et TCTL.

Une fois l'application validée par l'architecte, l'application a un comportement temporel qui est le comportement attendu de l'application. L'application est développée puis exécutée. Lors de l'exécution, il faut s'assurer que le comportement spécifié est bien respecté. Afin d'effectuer cette vérification, nous utilisons le comportement temporel attendu pour générer automatiquement un moniteur de qualité de service grâce à une transformation de modèles.

# Chapitre 3

# De la spécification à la réalisation

Une fois la spécification définie par l'architecte, les informations sont données aux programmeurs afin d'obtenir l'implantation du système. La qualité de service définie lors la spécification doit être vérifiée lors de l'exécution du système afin de vérifier si elle est correcte. Pour la vérifier, un système de monitorat peut être mis en place afin de surveiller l'exécution. Cette instrumentation peut se faire de façon manuelle par un programmeur ou être générée à partir des spécifications. Une instrumentation manuelle est souvent source d'erreurs et ne peut pas toujours être effectuée car les fonctionnalités sont prioritaires. Une instrumentation générée permet de fournir une surveillance sans ou avec peu d'interventions manuelles. Cette génération est effectuée à partir des résultats de la spécification.

Ce chapitre présente la génération d'un moniteur de qualité de service pour le temps à partir du comportement temporel d'une application à composants. La vérification des propriétés de temps lors de l'exécution de l'application n'est pas toujours aisée avec les langages utilisés pour l'implantation. Vérifier le temps implique l'ajout par le programmeur de tests dans le code ou d'un module spécifique pour cette vérification. Notre approche consiste à effectuer cette incorporation de la vérification de façon automatique et indépendante du code développé. Ceci permet une bonne séparation entre le code et les propriétés de qualité de service de temps. Nous utilisons l'automate temporisé complet de l'application comme base du moniteur. Pour réaliser le moniteur nous avons choisi une plate-forme permettant une gestion du temps simplifié. Le langage Giotto permet une abstraction de temps. Ce chapitre présente la réalisation de ce moniteur de qualité de service. Le moniteur est obtenu par une transformation de modèles conformes au méta-modèle des automates temporisés vers des modèles conforme au méta-modèle de Giotto. Le moniteur est ensuite généré à partir du résultat de la transformation. Nous présentons dans un premier temps l'objectif du moniteur à l'exécution. Dans un second temps, nous présentons la spécification utilisée pour la génération. Nous comparons les deux algorithmes et dans quels cas utiliser l'un ou l'autre. Enfin, nous présentons les algorithmes de transformation.

## 3.1 Objectif de la réalisation d'un moniteur

La vérification de la qualité de service à l'exécution permet de savoir si les spécifications sont bien respectées. Mais intégrer la vérification manuellement par le programmeur n'est pas aisée car elle n'est pas toujours compatible avec les fonctionnalités de l'application. L'objectif du moniteur est de notifier l'utilisateur ou le manageur de qualité de service que les propriétés de temps sont respectées. Pour cela, nous devons identifier dans le comportement temporel à quels moments il peut être violé. Le moniteur vérifiera à l'exécution que les messages sont bien émis conformément aux propriétés. Nous souhaitons donc générer un moniteur permettant de surveiller si les propriétés de qualité de service de temps sont correctes à l'exécution. Pour cela, nous utilisons le comportement temporel des composants défini lors de la phase de spécification afin d'obtenir le moniteur.

La Figure 3.1 présente le processus de génération du moniteur à partir de la spécification. La spécification est un PIM qui est est transformé dans un PSM, le moniteur de qualité de service.

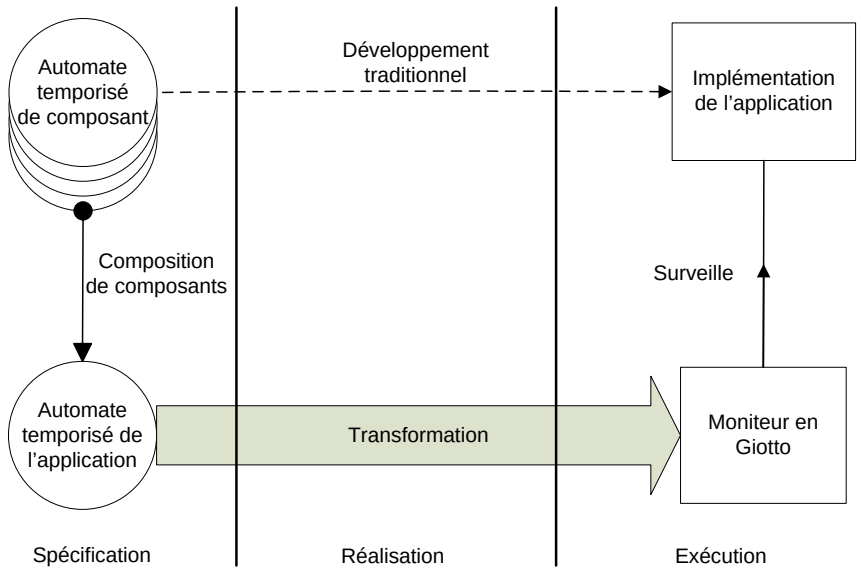



FIG. 3.1 – De la spécification à la réalisation.

Les composants notifient le moniteur grâce à un ensemble de fichiers intermédiaires. Le moniteur vérifiera l'occurrence des messages en appelant des opérations de ces fichiers. La Figure 3.2 présente la communication entre Giotto, les composants et l'utilisateur. La vérification des fichiers intermédiaires s'effectuent à l'aide des *drivers*.

Plusieurs types de moniteurs peuvent être générés selon le lieu d'utilisation de la vérification. Un seul moniteur peut être généré pour surveiller l'ensemble de l'application ou plusieurs moniteurs si l'on veut surveiller des composants/composites particuliers. Dans tous les cas, le processus de génération produit le moniteur ou les moniteurs à partir du comportement du composite de plus haut niveau que l'on souhaite surveiller.

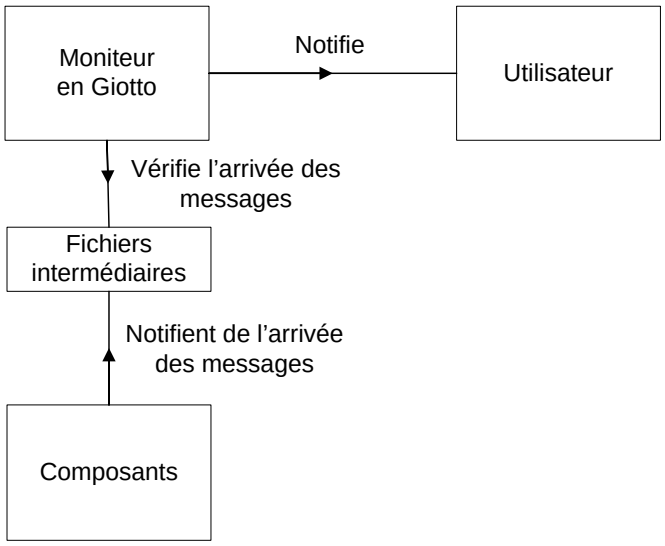



FIG. 3.2 – Communication à l'exécution.

## 3.2 Spécification de l'application

Lors du cycle de vie d'une application, une fois la spécification validée par l'architecte, les spécifications sont données aux développeurs qui ont en charge de réaliser les composants. Les informations liées à la qualité de service de temps ne sont pas toujours facilement utilisables par les développeurs. De plus, elles ne font pas partie des fonctionnalités essentielles aux composants donc peuvent être développées que dans un second temps. Nous proposons d'utiliser les résultats de la spécification de l'application pour générer la partie de l'application gérant la qualité de service de temps. L'application validée a un comportement temporel attendu. Nous allons utiliser ce comportement temporel pour générer le moniteur. L'ensemble des propriétés temporelles sont satisfaites dans l'application au niveau de la spécification. Le moniteur va permettre de vérifier qu'elles le sont toujours lors de l'exécution. Il ne surveillera que les propriétés temporelles et pas le comportement complet de l'application : si un message n'est pas essentiel à satisfaction d'une propriété temporelle, il n'est pas utile de surveiller son occurrence. Le moniteur ne surveillera donc que les occurrences des transitions ayant des opérations temporelles, c'est-à-dire une garde ou réinitialisant une horloge. Une étape préliminaire à la transformation est de ne conserver dans l'automate que ces transitions utiles. Cette étape est une projection de l'automate temporisé de l'application sur ses transitions ayant une opération temporelle.

La projection de l'automate temporisé par rapport aux transitions temporelles permet d'avoir uniquement les transitions que l'on souhaite surveiller. Durant cette projection nous simplifions les gardes afin de restreindre l'ensemble des opérateurs de comparaison à $\{<, =, >\}$. Les gardes du type $x \leq c$ sont équivalentes à $x < c + 1$. Les gardes du types $x \geq c$ sont équivalentes à $x > c - 1$. Ces transitions sont remplacées par leur équivalent dans les gardes correspondantes.

Le résultat de cette projection est présentée sur la Figure 3.3. L'automate ne contient plus que six localités.

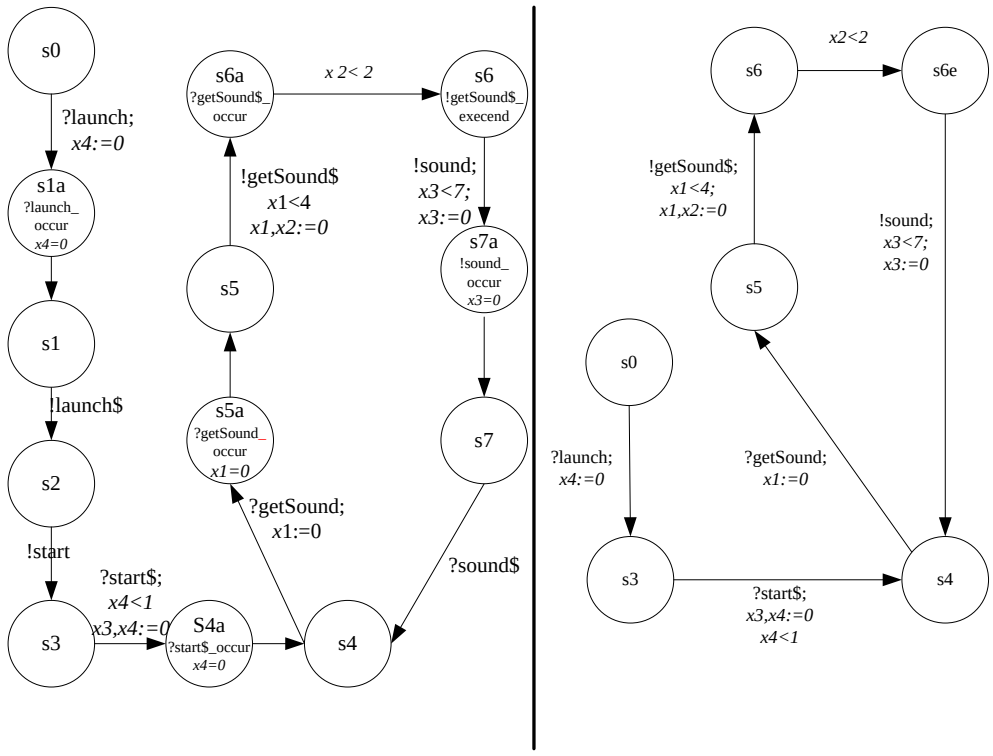



FIG. 3.3 – Projection sur les transitions temporelles.

Nous avons développé deux algorithmes pour implanter l'automate et identifier les violations. La première méthode consiste à discrétiser l'automate temporisé et identifier pendant cette discrétisation les possibles violations. La seconde méthode implante l'automate temporisé tel quel et l'identification des violations est effectuée lors de la réalisation. Ces deux méthodes sont présentées dans la suite de ce chapitre.

## 3.3 Comparaison des deux méthodes

Nous avons développé deux algorithmes pour transformer les automates temporisés en Giotto.

Le premier algorithme effectue la transformation en deux étapes : une discrétisation et une transformation vers Giotto. L'étape de discrétisation permet de passer du temps continu au temps discret. Pour l'automate temporisé, cela correspond à la représentation sous forme de système de transitions. Cet algorithme a été développé afin de pouvoir étendre la transformation vers Giotto à d'autres formalismes comme les automates hybrides utilisant un temps discret. La discrétisation peut produire un nombre d'états proportionnel aux bornes de comparaison

des gardes. Le code Giotto obtenu est spécifique à un automate temporisé donné. Si une valeur d'horloge est modifiée dans la spécification, la transformation doit être refaite.

Le second algorithme effectue la transformation en une seule étape et est spécifique aux automates temporisés. Le code Giotto obtenu est spécifique à un automate donné mais pas à la valeur des bornes de comparaison. Ces bornes ne sont pas stockées dans le code Giotto mais dans un autre fichier. Une modification des bornes est donc possible sans changer le code Giotto et donc sans avoir à refaire la transformation.

Les deux algorithmes sont utilisables pour les automates temporisés. Le second est plus intéressant que le premier pour une extension possible de la gestion de la qualité de service. Une négociation des contrats à l'exécution est possible en modifiant les bornes de comparaison sans avoir à redémarrer le système. L'avantage du premier est d'être applicable sur un autre formalisme que les automates temporisés, les automates hybrides par exemple.

## 3.4 Réalisation avec discrétisation

Un automate temporisé comporte généralement plusieurs horloges évoluant uniformément mais réinitialisées à différents moments. Les horloges ont donc une valeur différente à un instant donné. Lors de la composition des composants, l'automate temporisé du composite contient donc différentes horloges. Cet automate temporisé sert à produire le moniteur de qualité de service. A l'exécution, les composants n'utilisent qu'une seule horloge. Afin de produire le moniteur, il faut donc modifier les horloges de l'automate temporisé pour obtenir une seule horloge. Ce passage est possible car les comparaisons d'horloges ont été interdites lors de l'ajout des motifs comportementaux lors de la spécification. Si la spécification n'a pas été faite en utilisant les motifs comportementaux, la transformation s'assurera qu'il n'existe pas de comparaison d'horloges. Cette restriction permet d'éviter l'explosion combinatoire introduite par [Bou03]. Les horloges ne sont comparées qu'à des entiers. Cette transformation produit un moniteur se comportant exactement comme l'application doit se comporter.

Durant l'exécution, le moniteur devra savoir à quel moment le comportement de l'application n'est plus conforme à la spécification, c'est-à-dire que la qualité de service a été violée. Pour effectuer ceci, il doit pouvoir identifier à quel moment la violation est intervenue. L'automate temporisé est le comportement attendu de l'application et donc de la qualité de service attendue. Lorsque l'application ne respecte pas le comportement décrit par l'automate, le moniteur doit identifier cette violation et avertir l'utilisateur.

Le nombre d'horloges peut être important après l'utilisation des motifs comportementaux. Lors de la discrétisation, il n'y a plus de différences entre les horloges car elles n'apparaissent plus. Avant de lancer la discrétisation, nous réduisons le nombre d'horloges. Deux algorithmes de réduction ont été développés par [DY96]. Les horloges qui n'ont aucune relation entre elles sont regroupées. Ceci est effectué en regardant dans une localité quelles horloges seront utilisées dans les chemins issus de la localité. Cette réduction est effectuée grâce à l'outil Optikron [Daw98] faisant partie de Kronos. Des réinitialisations d'horloges peuvent être faites en prenant la valeur d'une autre horloge. Le nouvel automate a le même comportement mais ne contient plus que deux horloges au lieu de quatre.

Nous effectuons la transformation de l'automate vers le moniteur en deux étapes. La pre-

mière est la discrétisation des horloges en identifiant les possibles violations de qualité de service. La seconde est la correspondance entre ce nouvel automate et Giotto.

### 3.4.1 Discrétisation de l'automate temporisé

Un automate temporisé peut être décomposé en état, un couple localité et valuation d'horloges. La passage d'un état à un autre se fait grâce à deux types de transitions : temporisée et discrète. Une transition temporisée correspond à une incrémentation de la valeur des horloges sans changer de localité. Une transition discrète se fait sans changement de valuation mais en changeant de localité en prenant une transition. Nous nous inspirons de cette définition pour discrétiser l'automate temporisé.

Nous conservons les notions de transitions dicrètes et temporisées mais nous ne conservons qu'une horloge dans l'automate. Le passage de plusieurs horloges à une seule est possible car les gardes n'autorisent pas de comparaison d'horloges. En adressant la qualité de service, nous voulons savoir si, à chaque fin d'unité de temps, les événements attendus sont arrivés. Les transitions temporelles seront donc toutes avec un pas d'une unité de temps. La transformation porte sur les transitions. Nous définissons selon le type de transition, l'ensemble de la transformation. Nous distinguons les transitions non gardées et les transitions avec une garde de type $<$, $=$ et $>$. Nous présentons la transformation puis nous illustrerons la discrétisation d'un automate temporisé sur un exemple.

#### 3.4.1.1 Discrétisation de l'automate

L'algorithme de discrétisation transforme l'automate temporisé en un automate discret. L'algorithme parcourt les localités de l'automate et ajoute les états correspondants en fonction des transitions sortantes de la localité courante. L'algorithme ne s'arrête que lorsqu'il a parcouru toutes les localités de l'automate temporisé et que rien n'est ajouté dans l'automate discret. La structure de l'algorithme est donné par l'algorithme 3.1.

**Algorithme 3.1** Discrétisation de l'automate temporisé

**ENTRÉES:** $A =< S, X, L, T, \iota, P >$
**SORTIES:** $Ad =< S, Tt, Td >$
  nouvel automate discret $Ad1$
  $Ad1.init = (init, X = 0)$
  **tantque** $Ad1 ! = Ad$ **faire**
    $Ad = Ad1$
    **pour tout** $s \in A.S$ **faire**
      **pour tout** $state \in \{dom(Ad1.S) == s\}$ **faire**
        **pour tout** $t \in s.Tsortantes$ **faire**
          traitement des transitions
        **fin pour**
      **fin pour**
    **fin pour**
  **fin tantque**

Chaque état créé est constitué d'une valuation des horloges de l'automate temporisé. Il contient aussi une référence à la localité dont il est issu. Nous présentons le traitement des transitions dans la suite de cette partie.

#### 3.4.1.2 Transition non gardée

Une transition non gardée est une transition sans écoulement de temps. C'est une transition discrète. Cette transition apparaît toujours dans l'automate temporisé si elle contient une réinitialisation d'horloges. La transition est transformée en autant de transitions discrètes que son état source a d'états dans l'automate discretisé. Pour chaque transition, les horloges à réinitialiser le sont. Chaque transition a pour état cible un nouvel état avec une nouvelle valuation d'horloges. Les états ayant les mêmes valuations sont fusionnés. L'algorithme est le suivant :

**Algorithme 3.2** Discrétisation de transition non gardée

**si** t.garde=null **alors**
  création-transition-discrète(t.source,t.target,t.label)
  $Ad1.S+ = (t.cible, init(state.valuation, t.initialize)$
**finsi**

#### 3.4.1.3 Transition gardée avec l'opérateur $<$

Une transition gardée avec $x < c$ impose que l'application doit rester un temps maximum de $c$ unités de temps dans la localité source de la transition. L'horloge $x$ a une valeur initiale $c_i$ en arrivant dans la localité d'où est tirée la transition. Cette valeur $c_i$ ne peut être égale à $c$ sinon le comportement temporel n'est pas correct par rapport à la qualité de service. Ce cas peut avoir lieu si les contrats temporels ne sont pas assez complets pour couvrir tous les comportements possibles de l'application. Cela peut arriver lors de la composition de composants si les contrats de comportement requis n'ont pas de lien avec la transition étudiée. La génération du moniteur ne peut continuer car le moniteur détecte automatiquement une violation de service lors de cette transition. Dans le cas où $c_i \geq c$, la discrétisation est stoppée et l'architecte est prévenu de cette situation avec le préfixe permettant d'arriver à cette erreur.

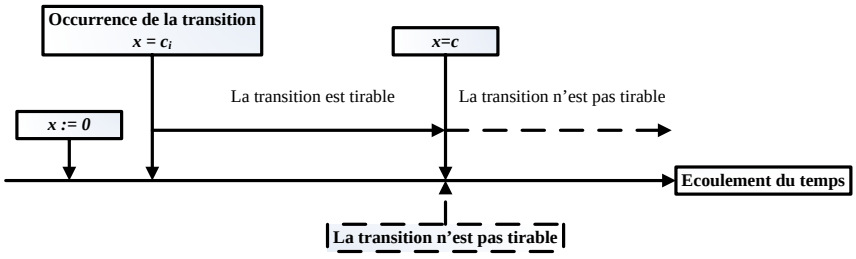



FIG. 3.4 – Ecoulement du temps pour $<$.

La Figure 3.4 présente l'écoulement du temps et les possibilités pour tirer la transition en fonction de la valeur de $c$. L'intervalle de temps pour tirer la transition est donc $[c_i..c[$.

L'objectif de la discrétisation est de d'obtenir pour chaque état, une durée de une unité de temps. La localité va être transformée en $c - c_i$ états correspondant chacun à une unité de temps. Ces états sont reliés par des transitions temporelles incrémentant le temps de une unité.

La transition peut avoir ou non un label. Si la transition porte un label alors le label peut avoir lieu en respectant la garde. Une transition discrète est ajoutée dans tous les états créés, ceux-ci respectant la garde de la transition. Un nouvel état est créé pour chaque transition discrète avec la même valuation que l'état source sauf si une réinitialisation est requise par la transition. Dans ce cas, les horloges à réinitialiser le sont. Les états se trouvant avec une même valuation sont fusionnés.

Si la transition ne porte pas de label, une transition temporaire est créée partant de tous les états créés vers un état correspondant à la localité cible de la transition. Aucune réinitialisation d'horloges ne peut être effectuée. En effet, lors de l'ajout des motifs comportementaux, les réinitialisations d'horloges ne sont possibles que sur une transition portant un label. Les états sources et cibles ont donc la même valuation. Cette transition temporaire sera supprimée à la fin de la discrétisation. Elle permet pendant la transformation de séparer les états en fonction de la localité à laquelle ils correspondent. La suppression entraînera la fusion des états source et cible qui ont bien la même valuation.

L'algorithme de discrétisation des transitions avec l'opérateur $<$ est le suivant :

**Algorithme 3.3** Discrétisation de transition avec l'opérateur $<$

```
si t.garde.operateur==< alors
  pour tout t.source faire
    i := c_i
    etat-traité :=t.source
    tantque i<c-1 faire
      etat-traité :=création-transition-temporisé(etat-traité,t.valuation)
      création-transition-discrète(etat-traité,t.target,t.label)
      i++
    fin tantque
    création-transition-discrète(etat-traité,t.target,t.label)
  fin pour
finsi
```

La Figure 3.5 présente la discrétisation d'une transition avec la garde $x < 3$ et le label $message$. Le label, pour respecter la garde, doit avoir lieu dans l'intervalle de temps $[0..3[$. La valeur de l'horloge $x$ est $0$ dans la première localité. La première localité est transformée en trois états avec une valeur d'horloge valant respectivement $0, 1, 2$. De chacun de ces états, une transition discrète portant le label $message$ mène dans un état correspondant à la localité cible de la transition transformée. Ces états ont la même valuation d'horloge que dans l'état source. Une transition temporelle est ajoutée entre le premier et le second état ainsi qu'entre le second et le troisième. Le troisième état ne peut évoluer que par une transition discrète pour satisfaire la garde.

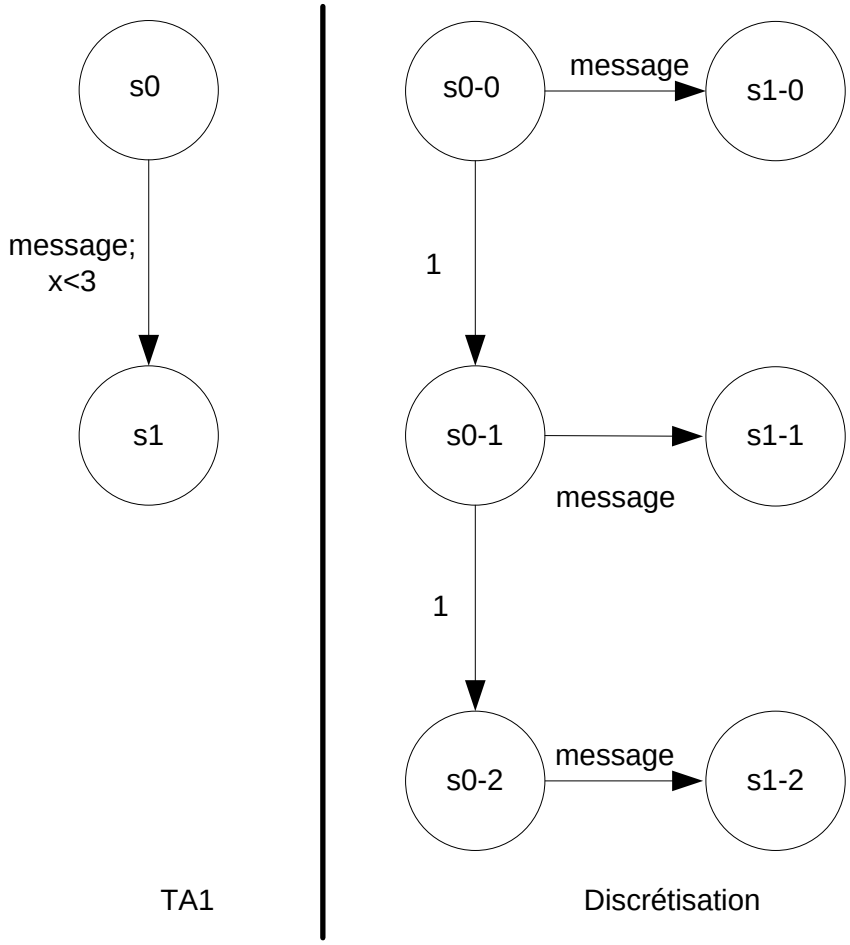



FIG. 3.5 – Discrétisation de $<$.

#### 3.4.1.4 Transition gardée avec l'opérateur $>$

Une transition gardée avec $x > c$ impose qu'elle ne peut être tirée qu'après $c$ unités de temps. La Figure 3.6 présente l'écoulement du temps et les possibilités pour tirer la transition en fonction de la valeur de $c$. L'horloge $x$ a une valeur initiale $c_i$ en arrivant dans la localité d'où est tirée la transition. Cette valeur $c_i$ peut être égale à $c$. Dans ce cas la transition est tirable de suite. L'intervalle de temps où la transition ne peut pas être tirée pendant la période $[c_i..c]$. La localité va être transformée en $c - c_i + 1$ états. Ces états sont reliés par des transitions temporelles. Une transition temporelle est ajoutée au dernier état pour mener dans un état dont la valuation satisfait la garde. Une transition temporelle bouclant sur cet état est ajoutée. Cet état a une valuation spéciale qui n'est pas un entier mais un ensemble dont la valeur minimale est $c + 1$. Cet ensemble est noté $(c + 1)+$.

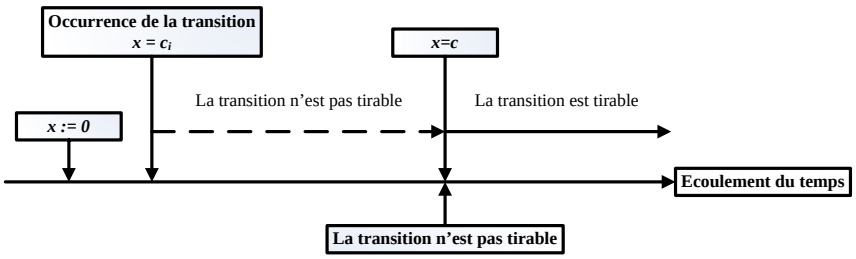



FIG. 3.6 – Ecoulement du temps pour $>$.

La transition peut avoir ou non un label. Si la transition porte un label alors le label peut avoir lieu en respectant la garde. Une transition discrète est ajoutée dans le dernier état ajouté. Un nouvel état est créé avec la même valuation que l'état source sauf si une réinitialisation est requise par la transition. Dans ce cas les horloges à réinitialiser le sont.

Si la transition ne porte pas de label, une transition temporaire est créée partant du dernier état créé vers un état correspondant à la localité cible de la transition. Aucune réinitialisation d'horloges ne peut être effectuée. Les états source et cible ont donc la même valuation. Cette transition temporaire sera supprimée à la fin de la discrétisation.

L'algorithme de discrétisation des transitions avec l'opérateur $>$ est donné par l'algorithme 3.4.

La Figure 3.7 présente la discrétisation d'une transition avec la garde $x > 2$ et le label $message$. Le label, pour respecter la garde, doit avoir lieu après l'intervalle de temps $[0..2]$. La valeur de l'horloge $x$ est $0$ dans la première localité. La première localité est transformée en trois états avec une valeur d'horloge valant respectivement $0, 1, 2$. Le dernier état a une transition temporelle bouclant sur l'état. Du dernier de ces états, une transition discrète portant le label $message$ mène dans un état correspondant à la localité cible de la transition transformée. Cet état a la même valuation d'horloge que dans l'état source.

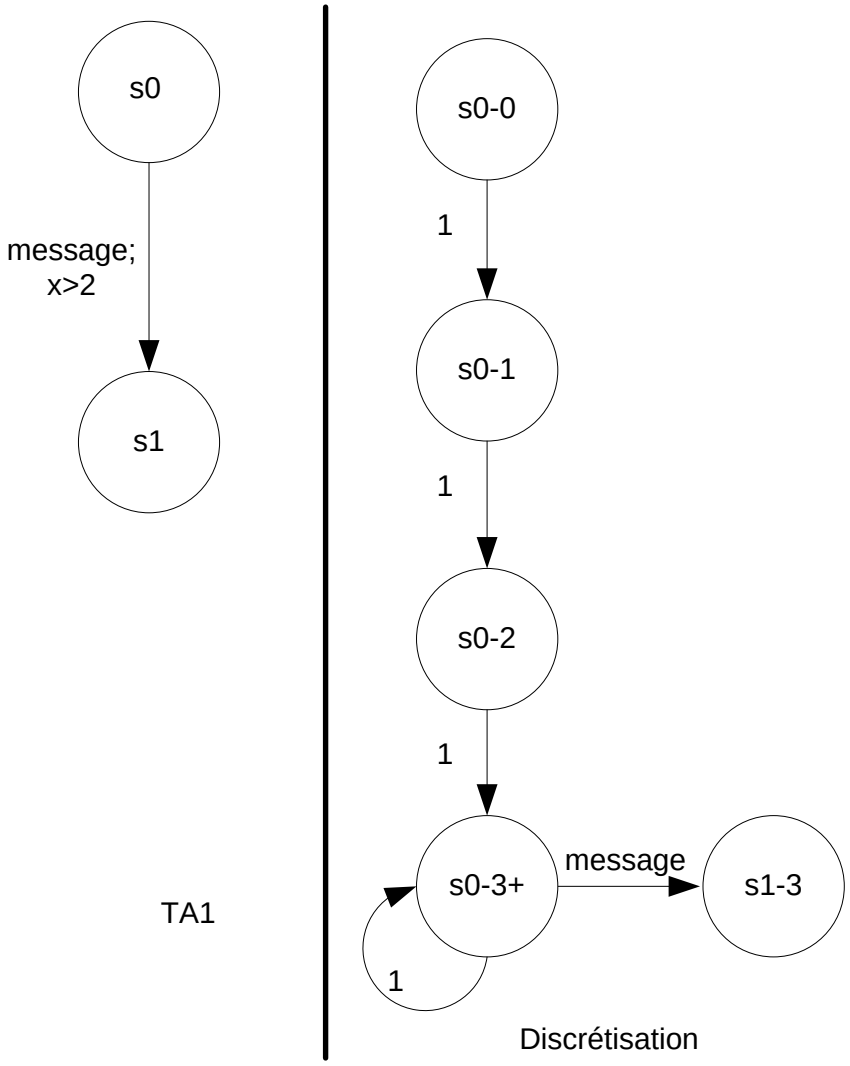



FIG. 3.7 – Discrétisation de $>$.

**Algorithme 3.4** Discrétisation de transition avec l'opérateur $>$

```
si t.garde.operateur== > alors
  pour tout t.source faire
    i := c_i
    etat-traité :=t.source
    tantque i<c-1 faire
      etat-traité :=création-transition-temporisé(etat-traité,t.valuation)
      i++
    fin tantque
    création-boucle-temporisé(etat-traité)
    création-transition-discrète(etat-traité,t.target,t.label)
  fin pour
finsi
```

#### 3.4.1.5 Transition gardée avec l'opérateur $=$

Une transition gardée avec $x = c$ impose qu'elle ne peut être tirée que lorsque la valeur de l'horloge vaut exactement $c$ unités de temps. La Figure 3.8 présente les possibilités pour tirer la transition. L'horloge $x$ a une valeur initiale $c_i$ en arrivant dans la localité d'où est tirée la transition. Cette valeur $c_i$ peut être égale à $c$. Dans ce cas la transition est tirable de suite. L'intervalle de temps où la transition ne peut pas être tirée pendant la période $[c_i..c[\bigcup]c..\infty[$. La localité va être transformée en $c - c_i + 1$ états. Ces états sont reliés par des transitions temporelles. Le dernier état de la suite est le seul qui satisfait la garde. Aucune transition temporelle ne sera plus ajoutée.

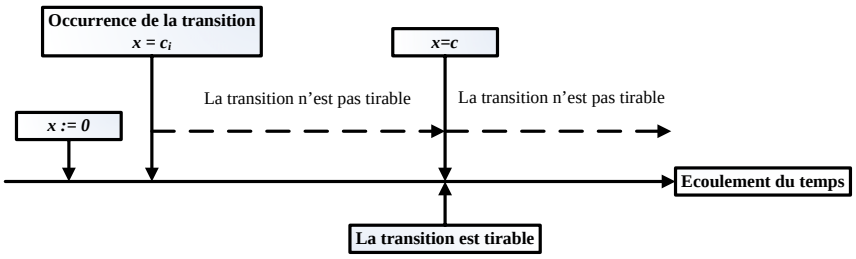



FIG. 3.8 – Ecoulement du temps pour $=$.

La transition peut avoir ou non un label. Si la transition porte un label alors le label peut avoir lieu en respectant la garde. Une transition discrète est ajoutée dans le dernier état ajouté. Un nouvel état est créé avec la même valuation que l'état source sauf si une réinitialisation est requise par la transition. Dans ce cas les horloges à réinitialiser le sont ceux.

Si la transition ne porte pas de label, une transition temporaire est créée partant du dernier état créé vers un état correspondant à la localité cible de la transition. Aucune réinitialisation d'horloges ne peut être effectuée. Les états source et cibles ont donc la même valuation. Cette transition temporaire est supprimée à la fin de la discrétisation.

L'algorithme de discrétisation des transitions avec l'opérateur $=$ est présenté par l'algorithme 3.5.

**Algorithme 3.5** Discrétisation de transition avec l'opérateur $=$

```
si t.garde.operateur= = alors
  pour tout t.source faire
    i := c_i
    etat-traité :=t.source
    tantque i<c-1 faire
      etat-traité :=création-transition-temporisé(etat-traité,t.valuation)
      i++
    fin tantque
    création-transition-discrète(etat-traité,t.target,t.label)
  fin pour
finsi
```

La Figure 3.9 présente la discrétisation d'une transition avec la garde $x = 2$ et le label $message$. Le label, pour respecter la garde, doit avoir lieu quand l'horloge vaut 2. La valeur de l'horloge $x$ est 0 dans la première localité. La première localité est transformée en trois états avec une valeur d'horloge valant respectivement $0, 1, 2$. Du dernier de ces états, une transition discrète portant le label $message$ mène dans un état correspondant à la localité cible de la transition transformée. Cet état a la même valuation d'horloge que dans l'état source.

#### 3.4.1.6 Transition avec garde quelconque

Les transitions n'ont pas toujours des gardes simples mais peuvent avoir une garde qui est une conjonction de comparaisons simples. La seule restriction est qu'une horloge n'apparaisse qu'une fois dans cette conjonction. En effet un motif n'ajoute qu'une comparaison par horloge. La discrétisation pour une conjonction se fait en regardant pour chaque membre de la garde, l'intervalle de temps correct. L'union des intervalles obtenus permet d'obtenir l'intervalle de temps pour la conjonction. Cet intervalle de temps est noté $]deb, fin[$, avec $fin > deb$. La borne supérieure doit être strictement supérieure à la borne inférieure car sinon la transition n'est jamais tirable. Si tel est le cas, l'architecte est informé de l'erreur et la discrétisation est suspendue.

L'agorithme applique la création d'états en s'appuyant sur l'algorithme pour une garde $>$ en utilisant $deb$ comme borne puis ensuite celui pour une garde $<$ en utilisant $fin$ comme borne. Ceci va permettre d'interdire la création de transition en dehors de l'intervalle requis.

L'algorithme de discrétisation pour une garde quelconque est donné par l'algorithme 3.6.

La Figure 3.10 présente la discrétisation d'une garde $x1 > 1 \wedge x2 < 3$. L'intervalle de temps pour $x1 > 1$ est $[1.. + \infty[$ et celui de $x2 < 3$ est $[0..3[$. Lintervalle pour la garde est donc $]0..3[$. La transition peut donc être tirée après 1 ou 2 unités de temps mais pas avant ni après. On s'occupe d'abord de la borne inférieur. La transition n'est pas tirable directement donc nous ajoutons une transition temporelle. Le nouvel état $s0 - 1$ permet de tirer la transition donc n'importe quel incrément de temps permettra le tirage de la transition. L'état devient donc

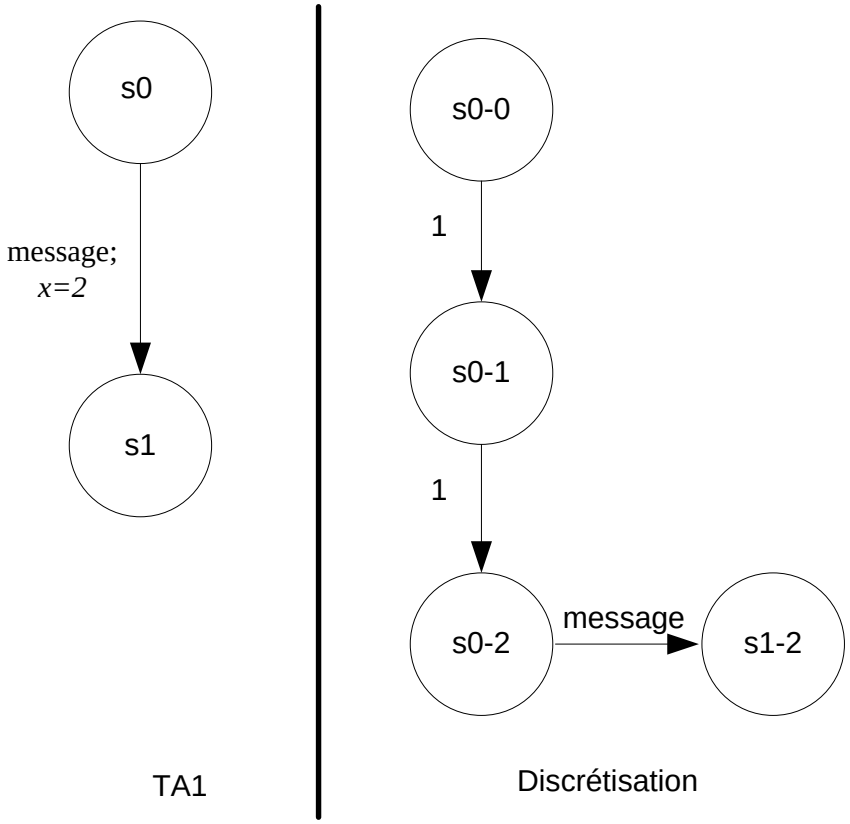



FIG. 3.9 – Discrétisation de $=$.

**Algorithme 3.6** Discrétisation de transition avec conjonction

**si** t.garde.operateur= $and$ **alors**
  **pour tout** t.source **faire**
    intervalleglobal+=membregauche.intervalle
    **tantque** membredroit.operateur=$and$ **faire**
      intervalleglobal+=membregauche.intervalle
    **fin tantque**
    intervalleglobal+=membredroit.intervalle
    creation transition garde quelconque
  **fin pour**
**finsi**

$s0 - 1+$. Une transition portant le message est créée vers un nouvel état $s1 - 1+$.

On traite maintenant la borne supérieure. Une transition temporelle est ajoutée de $s0 - 1+$ vers $s0 - 2$. $s0 - 1+$ devient donc $s0 - 1$ car une transition temporelle l'a comme source. L'état $s1 - 1+$ est transformé en $s1 - 1$ car la transition menant dans cet état n'a plus un état $+$ comme source. Une transition portant message est créée vers un nouvel état $s1 - 2$. La borne supérieure est atteinte donc la discrétisation est terminée.

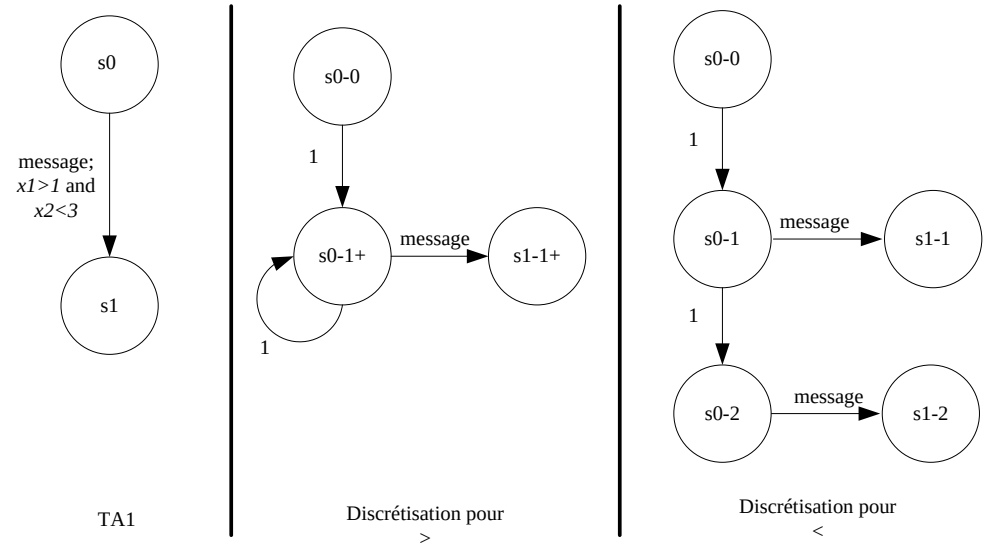



FIG. 3.10 – Discrétisation d'une conjonction.

#### 3.4.1.7 Suppression des transitions temporaires

Des transitions temporaires ont pu être créées lors de la discrétisation. Ce sont des transitions sans label ni incrément de temps. Ces transitions sont des transitions instantanées entre deux états. La suppression de ces transitions se font en fusionnant les deux états attachés à la transition $t$. Les transitions, dont la source est l'état cible $ec$ de $t$, sont dupliquées en autant de transitions qu'il y a de transitions temporaires entrant en $ec$. Ces transitions ont la même cible et leur source est la source de la transition temporaire ayant entraîné leur création.

Des états vont maintenant se retrouver avec plusieurs transitions temporelles sortantes. Les états cibles de ces transitions temporelles vont être fusionnés car ils correspondent au même écoulement de temps.

La dernière étape est de regrouger les transitions pouvant arriver dans la même unité de temps. Un regroupement de transitions est possible quand plusieurs transitions discrètes sont tirables sans transitions temporisées et sans rencontrer deux fois le même message. L'algorithme du regroupement est présenté par l'algorithme 3.7

La Figure 3.11 présente la discrétisation de l'automate temporisé avec la suppression des transitions temporaires et le regroupement des transitions.

**Algorithme 3.7** Regroupement des transitions

**ENTRÉES:** $Ad1 =< S, Tt, Td >$
**SORTIES:** $Ad2 =< S, Tt, Td >$

nouvel automate Ad2
Ad2.init :=Ad1.init
**tantque** $s \in Ad2.S$, $ssanstransitionssortantes$ **faire**
  **si** $correspondant(s) \in Ad1.S$ avec transition temporisé **alors**
    ajout transition temporisé à $s$
  **finsi**
  **pour tout** transition discrète de $correspondant(s) \in Ad1.S$ **faire**
    chemindiscret($correspondant(s)$)
    création transition discrète avec le chemindiscret
    **pour tout** préfixe du chemindiscret **faire**
      **si** transition temporisé possible après préfixe **alors**
        création transition discrète avec le préfixe
      **finsi**
    **fin pour**
  **fin pour**
**fin tantque**

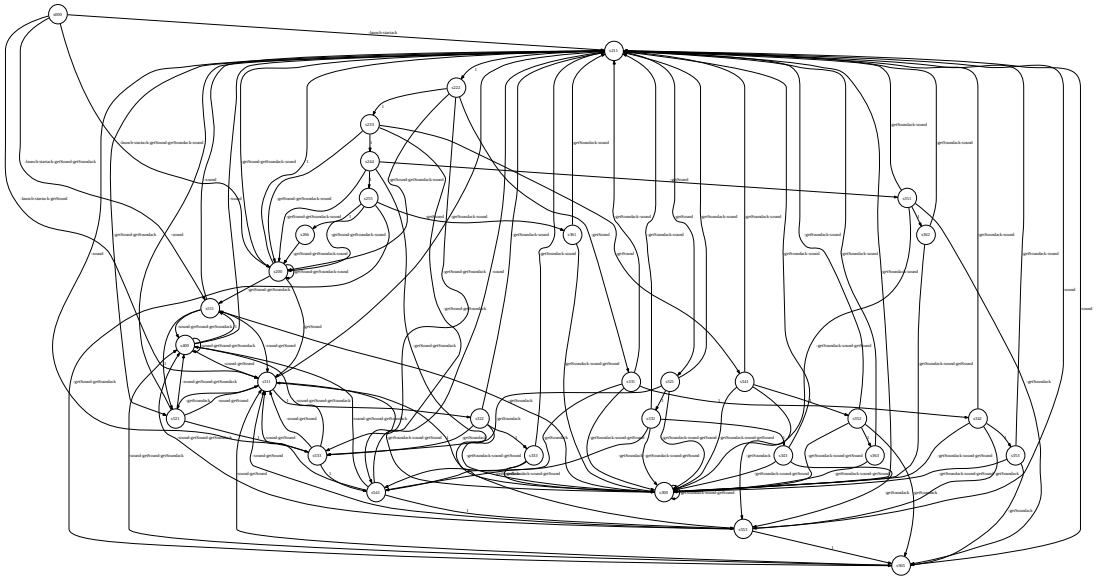


FIG. 3.11 – Discrétisation complète.

L'automate obtenu après discrétisation est un automate représentant le comportement temporel de l'application unité de temps par unité de temps. Il sert de base pour le moniteur mais n'est pas complet. En effet, le moniteur doit indiquer à l'utilisateur si l'exécution est correcte, donc si celle-ci ne viole pas le comportement de la spécification. Le moniteur doit donc connaître les violations possibles du comportement afin de notifier l'utilisateur.

### 3.4.2 Identification des violation possibles

L'identification des violations permet de savoir quand le comportement attendu n'est plus respecté. Une violation apparaît si un message n'arrive pas en respectant sa garde sur la transition le portant. Nous effectuons cette identification de la discrétisation de l'automate temporisé. L'automate discrétisé aura deux types de transitions : discrète et temporisé. Pour chacun de ces types nous définissons les violations possibles du contrat de la qualité de service.

#### 3.4.2.1 Violation pour les transitions discrètes

Une transition discrète est par définition une transition instantanée. Pour qu'une transition discrète viole la qualité de service, il faut qu'elle s'effectue avant son exécution attendue. Ce cas de figure peut soit se produire quand aucun message n'est attendu ou qu'un message arrive avant le message attendu. Le premier cas correspond à une discrétisation d'une garde du type $>$. Le second correspond à une transition discrète de plusieurs messages et dont des messages sont arrivés avant leur prédécesseurs. Afin de notifier l'utilisateur d'un tel cas, nous ajoutons une transition discrète à l'état.

Dans le cas où aucune transition discrète n'était présente, la violation est de recevoir un message. Une transition portant l'ensemble des labels de l'alphabet de l'automate est ajoutée vers un état de violation. Une transition temporisée permet de reprendre le comportement initial.

Dans le cas d'un mauvais ordre d'arrivée, l'ensemble des labels, excepté le premier message de chaque transition discrète, est ajouté à une transition menant vers un état de violation. L'ensemble des transitions sortantes de l'état est dupliqué sur l'état de violations.

Les deux cas peuvent se présenter pour un même état.

La Figure 3.12 représente la complétion lors de la transformation pour une arrivée de messages à la place de celui attendu.

#### 3.4.2.2 Violation pour les transitions temporisés

Une transition temporisée permet un écoulement du temps de une unité de temps. La violation d'une propriété correspond à un écoulement anormal du temps. En effet, si un message arrive en retard par rapport à ce qui était prévu, le temps s'écoulera. Un état de l'automate discret doit évoluer soit par un événement discret ou, si l'évènement n'est pas présent, par une incrémentation d'horloges. Lors de la discrétisation, une transition temporelle est ajoutée tant que la garde est correcte. Le dernier état a une valuation d'horloge correcte mais si l'on augmente d'une unité alors on viole la garde. Cet état n'a qu'une transition discrète et doit être prise pour satisfaire la propriété. La violation de la garde est de pouvoir prendre une transition temporelle. Cette situation est visible dans l'automate discrétisé quand un état $s$ ne peut

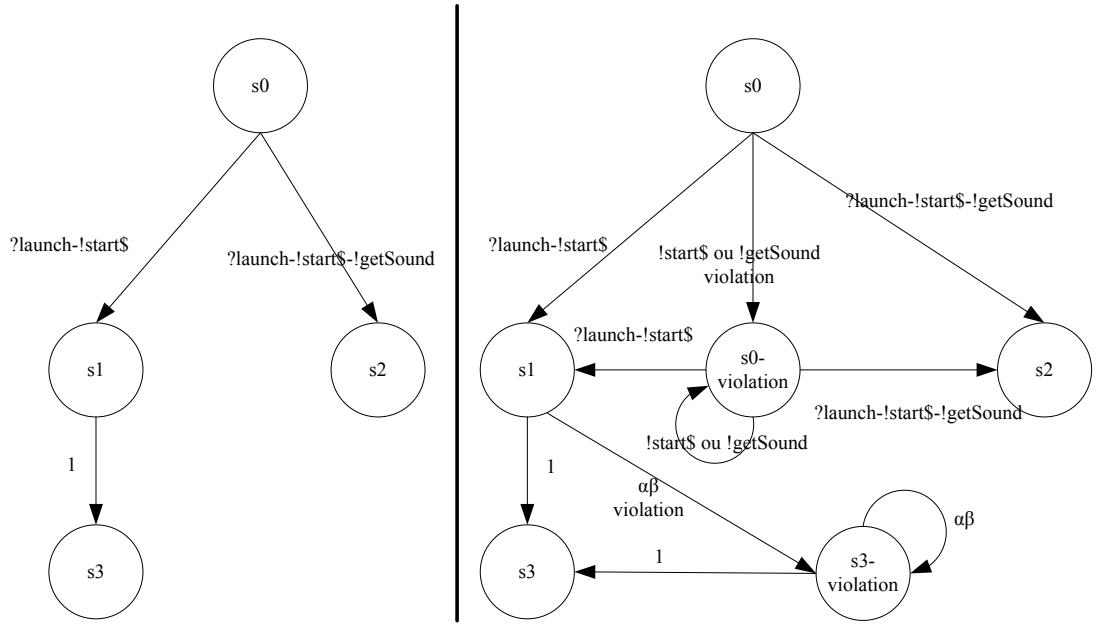



FIG. 3.12 – Violations des transitions discrètes.

effectuer de transitions temporisées donc seulement des transitions discrètes. Une transition temporisée est ajoutée à cet état vers un état de violation. Cette transition est marquée comme transition de violation. Tout incrément de temps dans cet état fait que l'on reste dans cet état. Toutes les transitions discrètes issues de $s$ sont dupliquées dans l'état de violation vers leur cible d'origine. La Figure 3.13 représente la complétion d'un état n'ayant pas de transition temporelle dans ses transitions sortantes.

### 3.4.3 Transformation en Giotto

La seconde étape de la transformation est de traduire l'automate discret en Giotto. L'objectif du programme Giotto est de surveiller si les composants s'exécutent correctement par rapport à leur comportement attendu. L'automate discrétisé représente ce comportement unité de temps par unité de temps. Un état de l'automate discretisé permet de savoir ce qui doit se passer ou pas pendant une période de une unité de temps. Le moniteur en Giotto doit vérifier les évènements durant cette unité de temps.

Un programme Giotto est composé de modes. Un mode permet d'exécuter un ensemble de fonctions pendant une période. Le programme peut quitter un mode pour aller dans un autre grâce à un changement de mode.

Chaque état de l'automate discrétisé représente le comportement attendu pour une unité de temps. Les transitions sortantes sont les changements d'états possibles et les informations portées par les transitions sont les conditions de ce changement. Chaque état de l'automate discret correspond à un mode en Giotto. La période du mode sera de une unité de temps. A la fin de la période, le moniteur changera obligatoirement de mode. En effet, on ne peut rester dans

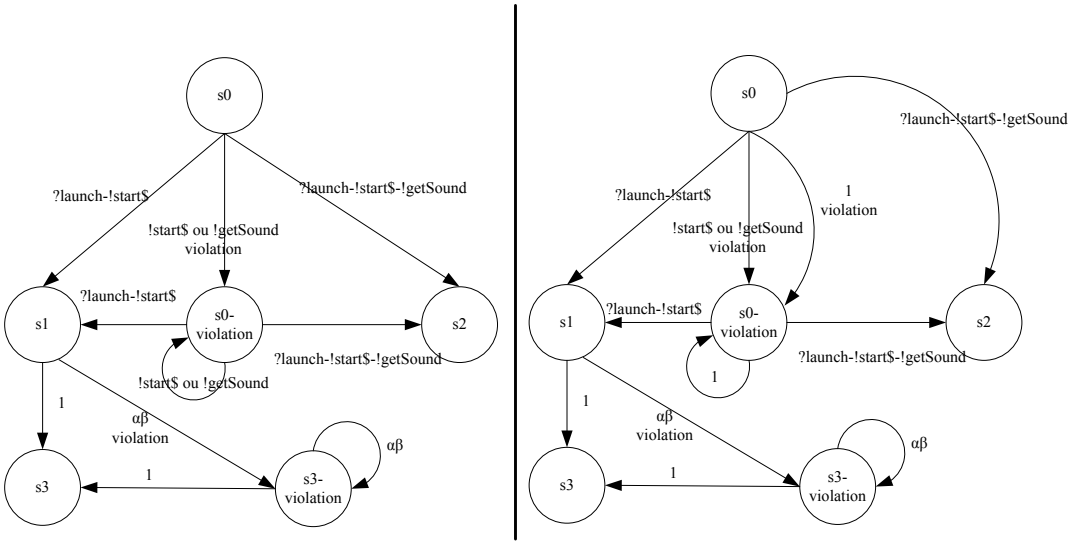



FIG. 3.13 – Violations des transitions temporisées.

un état de l'automate discret plus de une unité de temps. Une transition de l'automate temporisé correspond à un changement de mode conditionné par les informations portées par la transition. On change d'état soit par une transition discrète soit par une transition temporisée. Chacun de ces types de transitions aura une transformation particulière. Nous présentons la transformation pour les états et pour chacun de ces types de transitions. Nous présenterons ensuite la génération des fichiers intermédiaires nécessaires à la communication entre le moniteur et les composants.

#### 3.4.3.1 Transformation des états

L'algorithme transforme tous les états en exactement un mode. Ce mode à une période de une unité de temps. L'algorithme construit dans un premier temps tous les modes en parcourant tous les états de l'automate. Si l'état est l'état initial de l'automate alors son mode est le mode de départ du moniteur. Ensuite il reparcourt tous les états pour traiter les transitions. L'algorithme de transformation des états est le suivant est décrit par l'algorithme 3.8.

#### 3.4.3.2 Transformation des transitions discrètes

Une transition discrète ne peut être tirée que si l'ensemble des labels portés par cette transition a effectivement eu lieu. Une transition discrète est transformée en un changement de mode dont le driver vérifiera l'occurrence des labels. Le driver fait appel à un fichier intermédiaire qui retourne vrai si les messages sont bien arrivés. Le driver a une conséquence si la transition est une transition violant la qualité de service. Cette conséquence fait appel à un fichier pour notifier l'utilisateur. Si un état à plusieurs transitions sortantes, l'algorithme les classe par taille de l'ensemble de messages. La transition portant le plus de messages est préférée à une transition avec un seul message. A l'exécution, cet ordre est important car le moniteur traite d'abord

**Algorithme 3.8** Transformation des états

**ENTRÉES:** $Ad =< S, init, Tt, Td >$
  **pour tout** $s \in S$ **faire**
    création mode(s,période)
    **si** $s == init$ **alors**
      mode de départ
    **finsi**
  **fin pour**
  **pour tout** $s \in S$ **faire**
    traitement des transitions sortantes
  **fin pour**

le premier changement de mode puis, si il le faut, le second et ainsi de suite.

#### 3.4.3.3 Transformation des transitions temporisées

Une transition temporisée n'est tirée que si aucune transition discrète n'est tirable. Le changement de mode associé sera donc le dernier du mode à être exécuté car ce changement de mode s'effectue sans condition. En effet si aucun changement de mode dû à une transition discrète ne peut être effectué alors le temps s'incrémente. La conséquence sera vide si aucune violation n'est provoquée par cet incrément de temps. Si il y a violation, le driver associé au changement de mode fait appel à un fichier pour notifier l'utilisateur de la violation.

La Figure 3.14 représente le mapping pour l'état initial de l'automate discret.

#### 3.4.3.4 Génération des fichiers intermédiaires

Les composants notifient le moniteur de l'arrivée des messages. Cela s'effectue grâce à un ensemble de fichiers intermédiaires. Ces fichiers sont générés en même temps que le fichier en Giotto. La discrétisation génère deux types de fichiers : vérification des messages et notification à l'utilisateur.

Le premier type, la vérification, permet aux composants de notifier l'arrivée de messages et au moniteur de le vérifier. Ces fichiers fournissent donc deux opérations.

Le second type, la notification, permet au moniteur de notifier à l'utilisateur des violations possibles de la qualité de service. Ces fichiers n'offrent qu'une opération.

### 3.4.4 Conclusion

L'algorithme de réalisation d'un moniteur en discrétisant l'automate permet d'obtenir le comportement attendu unité de temps par unité de temps. Cet algorithme est capable, en le modifiant légèrement, de transformer d'autres formalismes comme les automates hybrides. La modification porte sur la suppression de la phase de discrétisation, la période d'un mode et la fréquence de vérification des changements de mode. Le moniteur généré permet bien de notifier l'utilisateur de la violation de la qualité de service. Si une réaction à cette violation à été prévue, elle peut être intégrée au moniteur. Par contre, une modification dynamique des

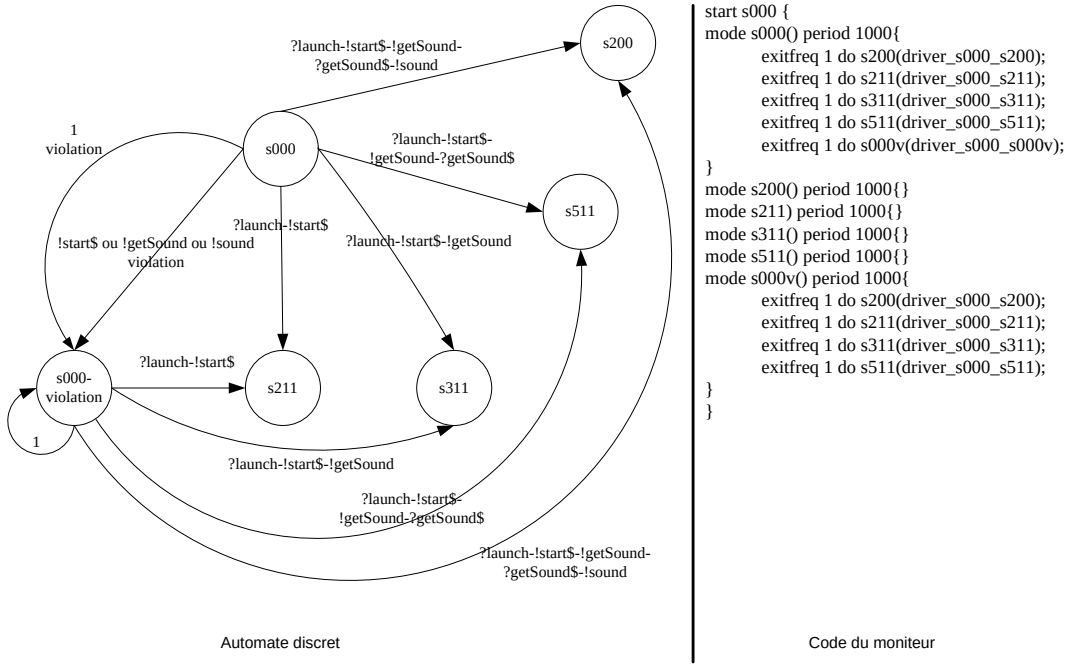



FIG. 3.14 – Mapping automate discret.

propriétés temporelles car le comportement du moniteur est fixe par transformation et ne peut donc pas être modifié à l'exécution.

Le moniteur a une taille proportionnelle aux bornes supérieurs des comparaisons d'horloges. Cette taille peut être importante si l'unité de temps est mal choisi, unité de temps en milliseconde alors que les propriétés sont en seconde, ou si il existe une grande différence d'échelle entre les différentes horloges. Dans ce cas, la génération de plusieurs moniteurs, un pour chaque niveau d'échelle, est une solution plus intéressante.

## 3.5 Réalisation de l'automate temporisé

Le second algorithme de transformation a pour objectif de transformer directement l'automate temporisé en Giotto sans étape intermédiaire. L'exécution du moniteur se comportera comme une simulation de l'automate temporisé en notifiant l'utilisateur des possibles violations de services. La transformation introduit les informations de violations car elles ne sont pas présentes dans l'automate d'origine. Nous décomposons l'algorithme en trois phases. La première est la transformation simple de l'automate temporisé en Giotto. Cette phase permet d'obtenir un code Giotto ayant la même structure que l'automate temporisé. La seconde est l'identification des violations possibles de la qualité de service. Le code Giotto sera enrichi par la possibilité de notifier l'utilisateur des violations possibles. La dernière est la génération des fichiers intermédiaires nécessaires à la communication entre Giotto et les composants. Ces fichiers n'auront pas la même structure que le premier algorithme. En effet, ils devront prendre

en compte les horloges. Le méta modèle de Giotto a été étendu pour tenir compte de cette situation. Cette section présente successivement les trois phases de cet algorithme.

### 3.5.1 Transformation de l'automate temporisé vers Giotto

La première phase de la transformation est d'obtenir un code Giotto ayant la même structure que l'automate temporisé. Dans cette transformation, nous allons considérer une localité de l'automate comme un pas d'exécution. Un mode sera donc créé pour chaque localité. Les transitions seront le moyen d'avancer dans l'exécution. Elles seront donc représentées par les changements de modes. Dans un automate temporisé, l'écoulement du temps est sous-entendu et donc aucune incrémentation du temps n'est représentée. Cette incrémentation devra donc être explicitement représentée dans le code généré en Giotto. Nous allons présenter la transformation de l'automate temporisé en reprenant ces trois étapes : transformation des localités, transformation des transitions et incrément du temps.

#### 3.5.1.1 Transformation des localités

Chaque localité de l'automate temporisé est transformée exactement en un mode en Giotto. Ces modes auront tous la même période. L'unité de base d'un programme Giotto est la milliseconde. Dans notre spécification, l'unité de temps est la seconde. Afin de ne pas rester trop longtemps dans une localité après l'occurrence d'un message, chaque mode aura une période de $100$ millisecondes. Les horloges sont converties pour tenir compte de ce changement d'unité de temps. L'algorithme parcourt ensuite l'ensemble des localités de l'automate et créé un mode portant le nom de l'état et une période de $100$. Si la localité est la localité initiale de l'automate temporisé alors le mode correspondant sera le mode de départ du programme Giotto. L'algorithme de transformation pour les localités est le suivant :

**Algorithme 3.9** Transformation des localités

**ENTRÉES:** $A =< S, X, L, T, \iota, P >$
  **pour tout** $l \in L$ **faire**
    création mode
    **si** l=init **alors**
      mode de départ
    **finsi**
  **fin pour**

Cette transformation est illustrée par le passage de l'automate temporisé au code Giotto de la Figure 3.15.

#### 3.5.1.2 Transformation des transitions

Une transition dans l'automate permet de passer d'une localité à une autre si la garde est vérifiée et le label présent. Les localités sont des modes en Giotto et le passage d'un mode à un autre en Giotto s'effectue par des changements de mode *exitfreq* qui font appel à un driver. Ce driver vérifie si les conditions nécessaires au changement de mode sont valides. Chaque

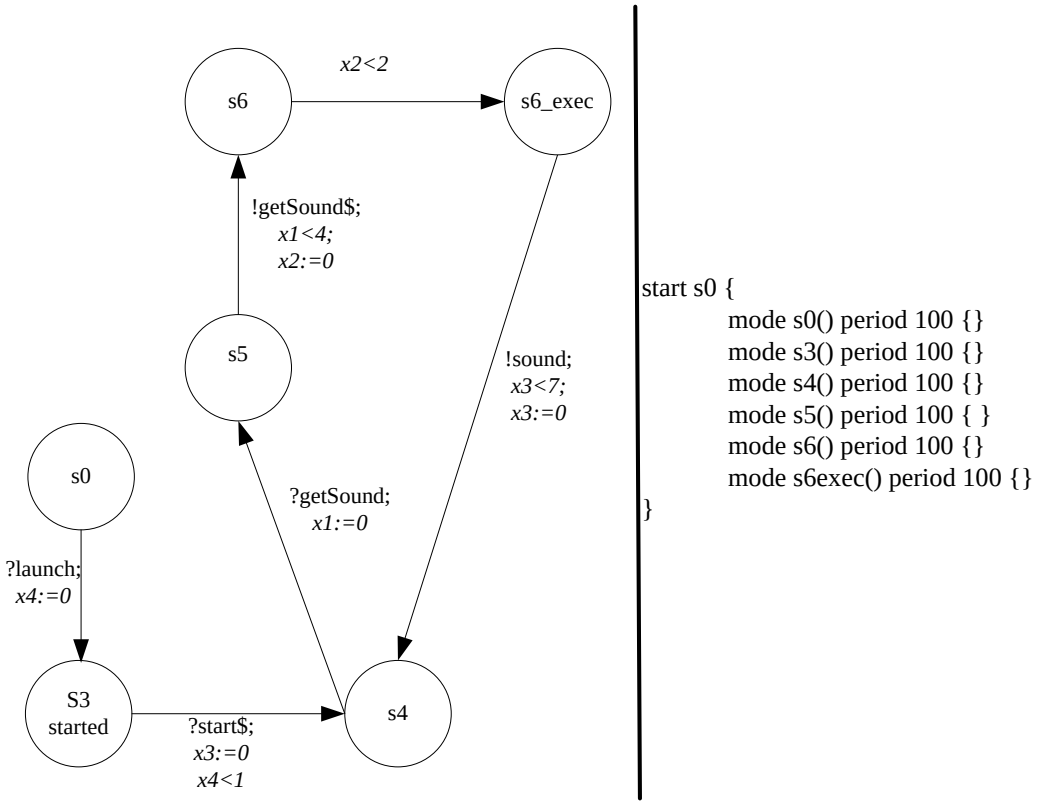



FIG. 3.15 – Transformation des localités.

transition est transformée en un changement de mode qui aura un *driver* vérifiant la garde et le label. L'algorithme parcourt les localités et effectue la transformation pour les transitions sortantes de chaque localité est décrit par l'algorithme 3.10.

**Algorithme 3.10** Transformations des transitions

**ENTRÉES:** $A =< S, X, L, T, \iota, P >$
**pour tout** $l \in L$ **faire**
**pour tout** $t \in T$ **faire**
creation du driver(nomfichier,constant_true)
creation du exitfreq(driver,localité cible de la transition)
**fin pour**
**fin pour**

La Figure 3.16 présente le code obtenu après la transformation des transitions. Chaque changement de mode a un *driver* associé.

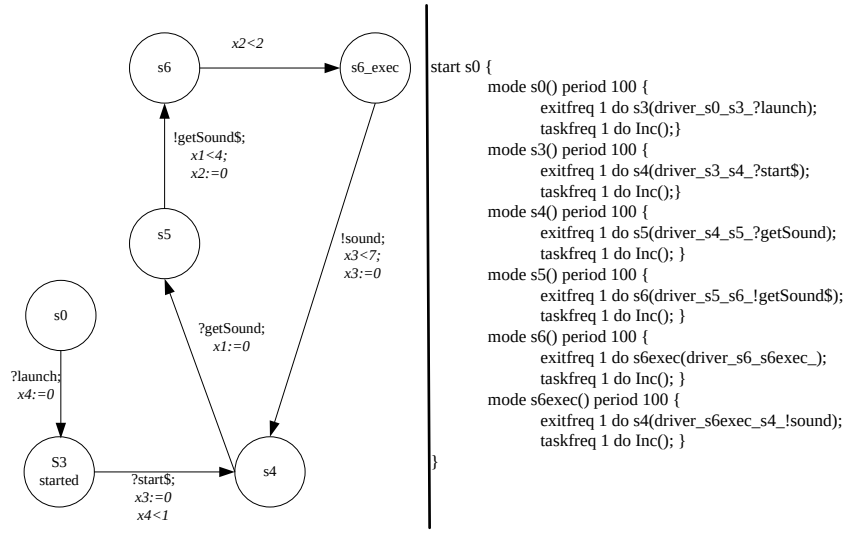



FIG. 3.16 – Transformation des transitions.

#### 3.5.1.3 Incrémentation des horloges

Dans un automate temporisé, les horloges croissent uniformément et ce de façon implicite. Lors de l'exécution du code Giotto cette incrémentation implicite n'est pas possible et doit donc être explicitement effectuée. Chaque mode créé à partir de l'automate temporisé a la même période qui est l'unité de mesure du temps. A chaque vérification des changements de modes possibles, les horloges sont augmentées d'une unité de temps. L'incrémentation des horloges est donc effectuée dans les modes à la fin de chaque période. Les tâches en Giotto permettent d'effectuer une action une ou plusieurs fois par période. Une tâche est créée pour

effectuer cette incrémentation une fois par période. Cette tâche fait appel à un driver qui augmentera toutes les horloges d'une unité de temps. L'ajout de cette tâche se fait lors de la création des modes. L'algorithme complet de la transformation de l'automate temporisé est décrit par l'algorithme 3.11.

**Algorithme 3.11** Transformation de l'automate temporisé

**ENTRÉES:** $A =< S, X, L, T, \iota, P >$
  **pour tout** $l \in L$ **faire**
    création mode
    création tâche d'incrémentation
    **si** l=init **alors**
      mode de départ
    **finsi**
  **fin pour**
  **pour tout** $l \in L$ **faire**
    **pour tout** $t \in T$ **faire**
      création du driver(nomfichier,constant_true)
      création du exitfreq(driver,localité cible de la transition)
    **fin pour**
  **fin pour**

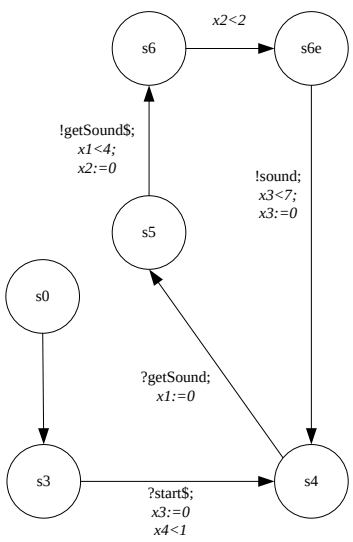
La Figure 3.17 représente la transformation complète de l'automate temporisé de gauche en code Giotto de droite. Le code Giotto obtenu peut être exécuté et s'exécutera correctement si le comportement des composants respecte la spécification. L'algorithme doit donc compléter le code Giotto pour notifier l'utilisateur des violations de la qualité de service.

### 3.5.2 Identification des violations possibles

L'algorithme direct a transformé l'automate temporisé en Giotto. L'objectif du moniteur est de surveiller si la qualité de service est violée à l'exécution. L'algorithme doit donc être enrichi afin d'identifier à quels endroits de l'automate temporisé la qualité de service peut être violée. Dans un premier temps, nous allons définir à quels moments la qualité de service peut être violée. Ensuite nous présenterons l'algorithme de transformation en tenant compte de la possible violation de qualité de service.

#### 3.5.2.1 Violations de qualité de service dans l'automate temporisé

L'automate temporisé représente le comportement attendu du composant. Il exprime ce qui doit se passer et justement donc pas ce qui ne doit pas se passer. Les violations de la qualité de service doivent donc être définies afin de les identifier dans l'automate. Nous nous intéresserons seulement aux violations temporelles de l'automate et non aux violations structurelles. Une violation temporelle est, par exemple, l'arrivée tardive d'un message. Une violation structurelle est l'arrivée d'un message non attendu.




```
start s0 {
        mode s0() period 100 {
                exitfreq 1 do s2(driver_s0_s2_?launch);
                taskfreq 1 do Inc();}
        mode s2() period 100 {
                exitfreq 1 do s3(driver_s2_s3_!start);
                taskfreq 1 do Inc();}
        mode s3() period 100 {
                exitfreq 1 do s4(driver_s3_s4_?start$);
                taskfreq 1 do Inc();}
        mode s4() period 100 {
                exitfreq 1 do s5(driver_s4_s5_?getSound);
                taskfreq 1 do Inc();}
        mode s5() period 100 {
                exitfreq 1 do s6(driver_s5_s6_!getSound$);
                taskfreq 1 do Inc();}
        mode s6() period 100 {
                exitfreq 1 do s6exec(driver_s6_s6exec_);
                taskfreq 1 do Inc();}
        mode s6exec() period 100 {
                exitfreq 1 do s4(driver_s6exec_s4_!sound);
                taskfreq 1 do Inc();}

}
```

FIG. 3.17 – Transformation complète de l'automate temporisé.

Nous définissons un ensemble de violations possibles dans une localité. Les violations sont définies sur les transitions gardées sortantes de cette localité. Une transition est violée si sa garde n'est pas vérifiée à l'exécution et son label non présent.

Nous distinguons les transitions avec labels et les transitions sans labels. Une transition avec label attend son label puis s'assure que la garde est correcte. A l'exécution, si le label arrive et que la garde n'est pas vérifiée nous devrons changer de mode en notifiant l'utilisateur de la violation de la qualité de service. Un changement de mode doit donc être ajouté au mode pour tenir compte du label avec la garde invalidée. Ce changement de mode est ajouté après les changements de mode ne violant pas la qualité de service. Le driver associé au changement de mode vérifiera seulement la condition d'arrivée du label puis notifiera l'utilisateur de la violation.

Dans le cas d'une transition sans label, il faut s'assurer que la violation ne va pas entraîner un blocage de l'exécution. Les gardes pouvant entraîner un blocage sont $<, =$. En effet, la garde $>$ ne peut pas être violée car on attendra que le temps passe pour tirer cette condition. Dans le cas $<, =$, la violation vient du fait que l'on peut arriver dans la localité en violant déjà la garde. Pour tenir compte de cela, un changement de mode est ajouté pour cette transition. Le driver associé aura comme condition $true$ et comme action la notification de la violation.

#### 3.5.2.2 Algorithme complet

L'algorithme est enrichi pour ajouter les informations relatives à la violation de la qualité de service. Deux tests sont ajoutés à la partie de l'algorithme traitant des transitions. Ils permettront de tenir compte des deux cas possibles de violation. Le premier, présenté par l'algorithme 3.12, permet de détecter les transitions gardées avec label et ajoute un changement de mode pour la violation.

**Algorithme 3.12** Violation des transitions gardées

**si** label !=null et garde !=null **alors**
  création du exitfreq(driver-violation,localité cible de la transition)
**finsi**

Le second test, , présenté par l'algorithme 3.13, permet de détecter les transitions gardées avec les opérateurs $<, =$. Un driver est ajouté pour notifier l'utilisateur de la violation.

**Algorithme 3.13** Violation des transitions gardées

**si** label==null et garde.operateur $\in \{<, = \{$ **alors**
  création du exitfreq(driver-violation,localité cible de la transition)
**finsi**

L'algorithme prenant compte la notification de la violation de service est le suivant :

**Algorithme 3.14** Transformation de l'automate temporisé

**ENTRÉES:** $A =< S, X, L, T, \iota, P >$
**pour tout** $l \in L$ **faire**
  création mode
  création tâche d'incrémentation
  **si** l=init **alors**
    mode de départ
  **finsi**
**fin pour**
**pour tout** $l \in L$ **faire**
  **pour tout** $t \in T$ **faire**
    création du driver(nomfichier,constant_true)
    création du exitfreq(driver,localité cible de la transition)
    **si** label !=null et garde !=null **alors**
      création du exitfreq(driver-violation,localité cible de la transition)
    **finsi**
    **si** label==null et garde.operateur $\in \{<, = \{$ **alors**
      création du exitfreq(driver-violation,localité cible de la transition)
    **finsi**
  **fin pour**
**fin pour**

Le code Giotto en prenant en compte la violation de la qualité de service est donné par la Figure 3.18.

L'algorithme créé des drivers liés à des fichiers permettant de savoir si les gardes sont vérifiées. Ces fichiers seront générés en même temps que le code Giotto. Etant donné que ces fichiers doivent vérifier des informations venant des composants en fonction de la spécification, les fichiers sont générés en fonction de cette spécification.

```
start s0 {
        mode s0() period 1000 {
                exitfreq 1 do s2(driver_s0_s2_?launch);
                taskfreq 1 do Inc();
        }
        mode s2() period 1000 {
                exitfreq 1 do s3(driver_s2_s3_!start);
                exitfreq 1 do s3(driver_s2_s3_!startviolation);
                taskfreq 1 do Inc();
        }
        mode s3() period 1000 {
                exitfreq 1 do s4(driver_s3_s4_?start$);
                taskfreq 1 do Inc();
        }
        mode s4() period 1000 {
                exitfreq 1 do s5(driver_s4_s5_?getSound);
                taskfreq 1 do Inc();
        }
        mode s5() period 1000 {
                exitfreq 1 do s6(driver_s5_s6_!getSound$);
                exitfreq 1 do s6(driver_s5_s6_!getSound$violation);
                taskfreq 1 do Inc();
        }
        mode s6() period 1000 {
                exitfreq 1 do s6exec(driver_s6_s6exec_);
                exitfreq 1 do s6exec(driver_s6_s6exec_violation);
                taskfreq 1 do Inc();
        }
        mode s6exec() period 1000 {
                exitfreq 1 do s4(driver_s6exec_s4_!sound);
                exitfreq 1 do s4(driver_s6exec_s4_!soundviolation);
                taskfreq 1 do Inc();
        }
}
```

FIG. 3.18 – Code Giotto complet.

### 3.5.3 Génération des fichiers intermédiaires

Une fois la génération du code Giotto effectuée, le fichier est créé afin d'être compilé. Lors de cette création, les fichiers intermédiaires utilisés pour effectuer les traitements sont aussi créés. Différents types de fichiers sont générés selon les types de concepts de Giotto ils se rapportent. La transformation génère trois types de fichiers :

- tâche : il correspond à l'incrément de l'horloge,
- condition : il permet de savoir si les gardes lors d'un changement de mode sont vérifiées,
- conséquences : il correspond à l'opération effectuée lors d'un changement de mode : réinitialiser des horloges ou notifier l'utilisateur.

Afin de générer ces fichiers, les informations des gardes doivent être présentes dans le modèle généré par l'algorithme. Le méta-modèle de Giotto doit donc être complété pour tenir compte de ces informations.

#### 3.5.3.1 Méta-modèle pour la génération des fichiers intermédiaires

Le méta-modèle Giotto ne permet que la connaissance du nom du fichier relatif à un driver. Ce fichier est habituellement écrit par le programmeur. Notre approche va générer ce fichier automatiquement. Afin d'effectuer ceci, nous devons représenter les informations nécessaires à cette génération dans le méta-modèle de Giotto. Ces fichiers ont besoin de connaître les gardes associés aux transitions utilisant leur driver associé.

Nous enrichissons le méta-modèle avec les mêmes informations qu'une transition contient : une garde, un label et un ensemble d'horloges à réinitialisées. Nous ajoutons aussi les informations de la violation de la qualité de service. Ces informations sont attachées aux drivers.

La Figure 3.19 présente le méta-modèle Giotto prenant en compte la génération des fichiers utilisés lors de l'exécution. On ajoute un ensemble d'horloges qui correspondra à l'ensemble des horloges de l'automate. Chaque driver a une transition associée contenant les informations de la garde.

#### 3.5.3.2 Génération des fichiers intermédiaires

Les fichiers intermédiaires sont générés en même temps que le fichier Giotto. Ils sont écrits soit en Java soit en C selon le langage souhaité. En Java, chaque driver aura un fichier associé. En C, chaque driver aura un ensemble de fonctions associées. Nous présentons ici la génération pour Java. Le fichier C avec les ensembles de fonctions est présenté en annexe. Pour chaque type de fichier, nous présentons les informations nécessaires pour leur création et le squelette de code associé.

**Tâche** Notre transformation ne génère qu'une seule tâche : l'incrément des horloges. Les horloges sont stockées en tant que variable globale. Le fichier de tâche incrémentera les horloges. Il n'a pas besoin de connaître d'informations spécifiques. Le fichier contenant les variables a besoin de connaître le nombre d'horloges. Il propose en ensemble de fonctions permettant de :

- incrémenter les horloges,
- réinitialiser un ensemble d'horloges,
- obtenir la valeur d'une horloge.

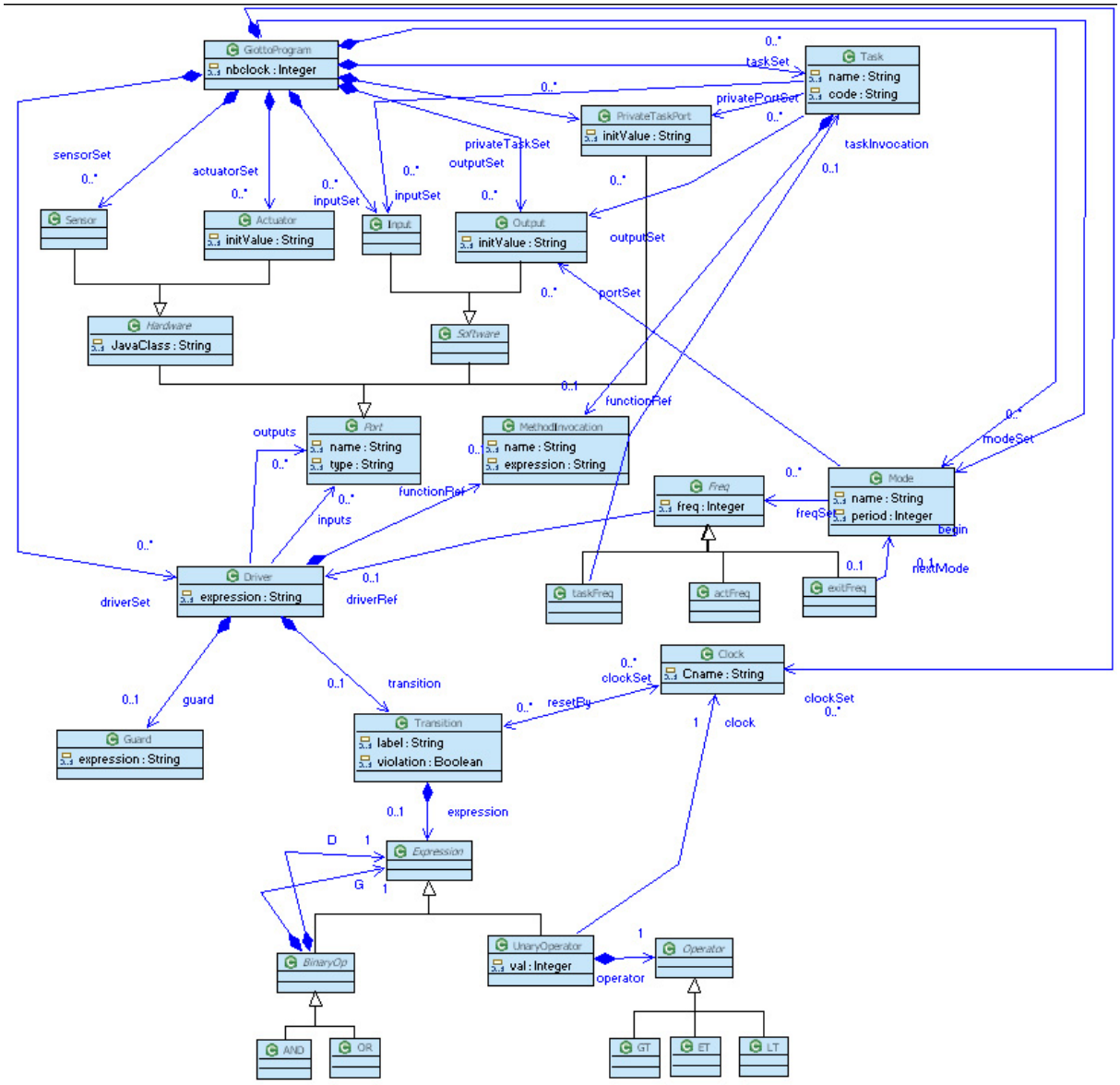



FIG. 3.19 – Méta modèle de Giotto pour l'implantation des automates temporisés.

La Figure 3.20 présente les deux classes représentant la tâche d'incrément des horloges et le stockage des horloges pour un nombre quelconque $x$ d'horloges. Cette variable est connue lors de la transformation.

```
package giotto.functionality.code.generation;
import java.io.Serializable;

public class Clock implements Serializable {
        static int[] clocks=new int[x];
        static public int getValue(int i) {
                return clocks[i];
        }
        static public void Inc() {
                for(int i=0;i<x;i++) clocks[i]=clocks[i]+1;
        }
        static public void Init(int[] I) {
                for(int i=0;i<I.length;i++) clocks[I[i]]=0;
        }
}

package giotto.functionality.code.generation;
import giotto.functionality.interfaces.TaskInterface;
import giotto.functionality.table.Parameter;
import java.io.Serializable;
public class Inc implements TaskInterface, Serializable {
        public void run(Parameter parameter) {
                Clock.Inc();
        }
}
```

FIG. 3.20 – Classes d'incrément et de stockage d'horloges.

**Condition** Une condition associée à un driver permet de savoir si la garde de la transition est satisfaite. Une garde peut contenir un label et une conjonction de comparaison d'horloges. La classe doit permettre la notification de l'arrivée du label. L'arrivée du label est un booléen initialement à faux. Une méthode publique pour notifier l'arrivée du label est ajoutée. La méthode $run$ retourne si la garde est bien vérifiée.

La Figure 3.21 représente la classe générique pour une condition avec comme label $message$ et une garde $g$. La variable $expression$ est une inéquation d'horloges et est générée lors de la création du fichier. Un exemple d'expression est : $Clock.getValue(0) < 5$. La classe portant le nom du message permet de savoir si une occurrence du message a eu lieu. C'est cette classe qui est appelé pour notifier l'arrivée du message.

```
package giotto.functionality.code.generation;
import giotto.functionality.code.BaseCondition;
import giotto.functionality.interfaces.ConditionInterface;
import giotto.functionality.table.Parameter;
import java.io.Serializable;
public class Condnear extends BaseCondition
implements ConditionInterface, Serializable {
    public boolean run(Parameter parameter) {
                return message.occur()&&guard();
    }
        public static boolean guard(){
                return=expression;
        }
}
package giotto.functionality.code.robot;
import java.io.Serializable;
public class message implements Serializable  {
        static boolean message=false;
        public static void messageoccurs(){
            message=true;
    }
    public static void messageinit(){
        message=false;
    }
        public static boolean occur(){
                return message;
        }
}
```

FIG. 3.21 – Classes de condition.

**Conséquence** Une conséquence associée à un driver permet de réinitialiser les horloges devant l'être. Les messages utilisés pour la condition précédant cette conséquence sont réinitialisés en appelant $messageinit()$ sur la classe correspondante. L'autre type de conséquence est la notification d'une violation de la qualité de service. Cette conséquence fait appel soir à une méthode générique soit une méthode fournie par le programmeur.

La Figure 3.22 représente la classe générique pour la réinitialisation d'un ensemble d'horloges $X$ entre les localités $l1$ et $l2$.

```
package giotto.functionality.code.generation;
import giotto.functionality.interfaces.DriverInterface;
import giotto.functionality.table.Parameter;
import java.io.Serializable;

public class Initfroml1tol2
implements DriverInterface, Serializable {
    public void run(Parameter parameter) {
        Clock.Init(X);
    }
}
```

FIG. 3.22 – Classe de conséquence de réinitialisation.

La Figure 3.23 représente la classe générique pour la notification d'une violation avec le message $message$ qui est initialisé avec un message d'erreur. Le message générique est : *Violation entre les localités l1 et l2 pour la transition portant le label et la garde*. Cette classe effectue aussi la réinitialisation d'un ensemble d'horloges $X$ qui était prévue entre les localités $l1$ et $l2$.

```
package giotto.functionality.code.generation;
import giotto.functionality.interfaces.DriverInterface;
import giotto.functionality.table.Parameter;
import java.io.Serializable;

public class Violatefroml1tol2
implements DriverInterface, Serializable {
        String message="";
    public void run(Parameter parameter) {
        Violation.send(message);
                Clock.Init(X);
    }
}
```

FIG. 3.23 – Classe de conséquence de violation.

### 3.5.4 Conclusion

L'algorithme de réalisation d'un moniteur en transformant directement l'automate temporisé en Giotto permet la vérification du comportement des composants. L'utilisateur est averti des violations de qualité de service et pourra s'il le juge nécessaire de modifier les contrats temporels à l'exécution. Cette modification est réalisable car les propriétés ne sont pas dans le code Giotto mais dans les fichiers intermédiaires. L'inconvénient de cet algorithme est qu'il est spécifique aux automates temporisés et n'est pas applicable à d'autres formalismes.

## 3.6 Ajout de la notification dans les composants

Le moniteur doit être informé de l'occurrence des messages par les composants lors de l'exécution afin de vérifier le comportement temporel de l'application. Cette notification se fait à l'aide d'un appel de fonction. La fonction est contenue dans un des fichiers générés en même temps que le moniteur. L'ensemble des messages nécessitant une notification correspond à l'alphabet de l'automate projeté 3.3 utilisé pour la génération du moniteur. Chaque élément de l'alphabet est couplé avec l'appel de fonction correspondant. Cet appel est de la forme $message.messageoccurs()$. L'ensemble des couples label-fonction est donné aux programmeurs responsables de l'implantation des composants afin qu'à chaque occurrence du label, la fonction soit appelée.

## 3.7 Conclusion sur la génération de moniteur

La génération automatique de moniteur de qualité de service de temps permet de fournir une instrumentation pour la vérifiaction à l'exécution des propriétés temporelles. L'automatisation offre l'avantage de pouvoir être rejoué lors d'une modification de la spécification sans intervention de l'architecte. Ceci permet également de réduire la possible introduction d'erreurs lors d'une instrumentation manuelle. Le premier algorithme permet de ne pas cibler uniquement le formalisme des automates temporisés mais aussi les formalismes autorisant le temps discret. Le second algorithme permet d'étendre l'approche avec une politique de gestion de qualité de service à l'exécution. Celle-ci permettra de pouvoir renégocier les contrats dynamiquement.

L'automatisation de la génération du moniteur de qualité de service permet l'utilisation d'un processus générique réutilisable dans d'autres applications. Le moniteur de qualité de service permet de vérifier que l'implantation de l'application respecte sa spécification.

Une preuve pour les transformations reste cependant à effectuer afin de s'assurer que les transformations n'ajoutent pas ou n'enlèvent pas de comportement lors du passage de la spécification au moniteur.

# Chapitre 4

# Mise en place du processus MDE par l'outillage *Thot*

L'ensemble de méthodes présentées dans la thèse a été implanté dans l'outil *Thot*. *Thot* permet à partir des spécifications des composants d'un système l'ajout de propriétés de temps dans le comportement des composants puis de générer un moniteur surveillant l'exécution des composants. Nous avons choisi une approche à base de modèles et de transformation de modèles. A partir des spécifications, un modèle du comportement de chaque composant est créé et enrichi avec les propriétés de temps. Une fois l'application validée, un moniteur est généré en effectuant une transformation de modèle conforme au méta-modèle des automates temporisés vers un modèle conforme au méta-modèle de Giotto. Enfin, le modèle du moniteur permet de créer les fichiers nécessaires à l'exécution du moniteur. Le processus est automatisé mais peut demander l'intervention de l'architecte en cas de problèmes lors de la spécification.

Ce chapitre présente dans un premier temps les langages utilisés pour implanter notre outil. Sintaks [MFF+06] permet de passer des spécifications fournies par l'architecte aux méta-modèles. Kermeta [MFJ05] est utilisé pour la création et l'utilisation des méta-modèles et l'écriture des transformations des modèles. Ensuite, nous présentons les méta-modèles utilisés lors des différentes phases de la méthode. Enfin, nous expliquerons l'utilisation de l'outil.

## 4.1 Outils utilisés pour la mise en oeuvre de *Thot*

Deux outils ont été utilisés pour implanter l'approche. Nous utilisons un développement dirigé par les modèles donc notre choix s'est porté vers des langages de modélisation et de transformation. Le passage des spécifications textuelles aux modèles a aussi été outillé. Une implantation en Java aurait été possible en écrivant un parser spécifique pour chaque fichier texte en entrée. L'utilisation d'un outil spécialisé dans le passage des textes aux modèles permet de créer directement les modèles nécessaires pour la transformation. Sintaks offre cette fonctionnalité. Le choix de Kermeta pour les méta-modèles et les transformations permet d'avoir des algorithmes proches de ceux écrits en langage de description. L'outil Sintaks permet le passage d'un format texte à un modèle.

### 4.1.1 Kermeta

Kermeta [MFJ05] est une plateforme open-source de méta-modélisation développée au sein de l'équipe Triskell. Le langage Kermeta est une extension du langage EMOF [EMO05] (Essential Meta-Object Facilities). L'extension est un langage d'actions permettant de spécifier une sémantique et un comportement pour les méta-modèles. Le langage d'action est impératif et orienté-objet. Il est utilisé pour fournir une implantation des opérations définies dans les méta-modèles. Le langage d'actions de Kermeta est particulièrement adapté à la manipulation des modèles. Il inclut à la fois des concepts objets et des concepts modèles. Kermeta comporte le typage statique, l'héritage multiple et la redéfinition avec une politique de liaison dynamique.

Nous avons choisi d'utiliser Kermeta pour différentes raisons. Premièrement, Kermeta propose une représentation graphique et textuelle pour représenter les méta-modèles utilisés dans cette thèse. Ensuite, le langage permet d'implanter les algorithmes de modification dans le méta-modèle lui-même. Il offre des expressions et des fermetures (comme each, detect ou select) analogue à OCL(Object Constraint Language). Enfin Kermeta est compatible avec le framework de méta-modélisation d'Eclipse EMF(Eclipse Modeling Framework). Ceci permet d'utiliser Eclipse pour éditer, stocker et visualiser les modèles.

### 4.1.2 Sintaks

Sintaks [MFF+06] est un outil permettant de définir des ponts entre les syntaxes concrètes (fichiers textes) et abstraites (modèles). Un pont correspond à un modèle Syntaks, un fichier avec une extension $.sts$, définissant la façon d'analyser un texte pour obtenir le modèle correspondant en respectant un méta-modèle donné. Ce pont définit aussi comment explorer un modèle pour écrire la représentation textuelle correspondante. L'application de ces fonctionnalités est intégrée dans Kermeta dans le menu contextuel des fichiers texte et modèle.

## 4.2 *Thot*

L'outil développé permet à partir des spécifications textuelles la génération des fichiers nécessaires pour la composition des composants et la génération du moniteur. L'ensemble du processus est automatique et ne requiert l'intervention de l'architecte qu'en cas de problème. La Figure 4.1 présente l'architecture de *Thot*et décrit les différents fichiers pendant le processus de transformation. Le processus est séparé en deux phases : spécification et réalisation. Chacune de ces deux phases peut être effectuée l'une après l'autre ou indépendammment.

### 4.2.1 Méta-modèles

Pour chacun des formalismes utilisés dans la thèse, un méta-modèle a été créé en Kermeta.

**Automate temporisé** Un méta-modèle a été développé correspondant à la syntaxe des automates temporisés définie en 1.2.2. La Figure 4.2 présente ce méta-modèle. Il est écrit en Kermeta et est enrichi d'opérations permettant d'écrire les modèles vers le model-checker Kronos par exemple. Deux fichiers $kmt$ sont utilisés. Le premier permet de décrire la structure du

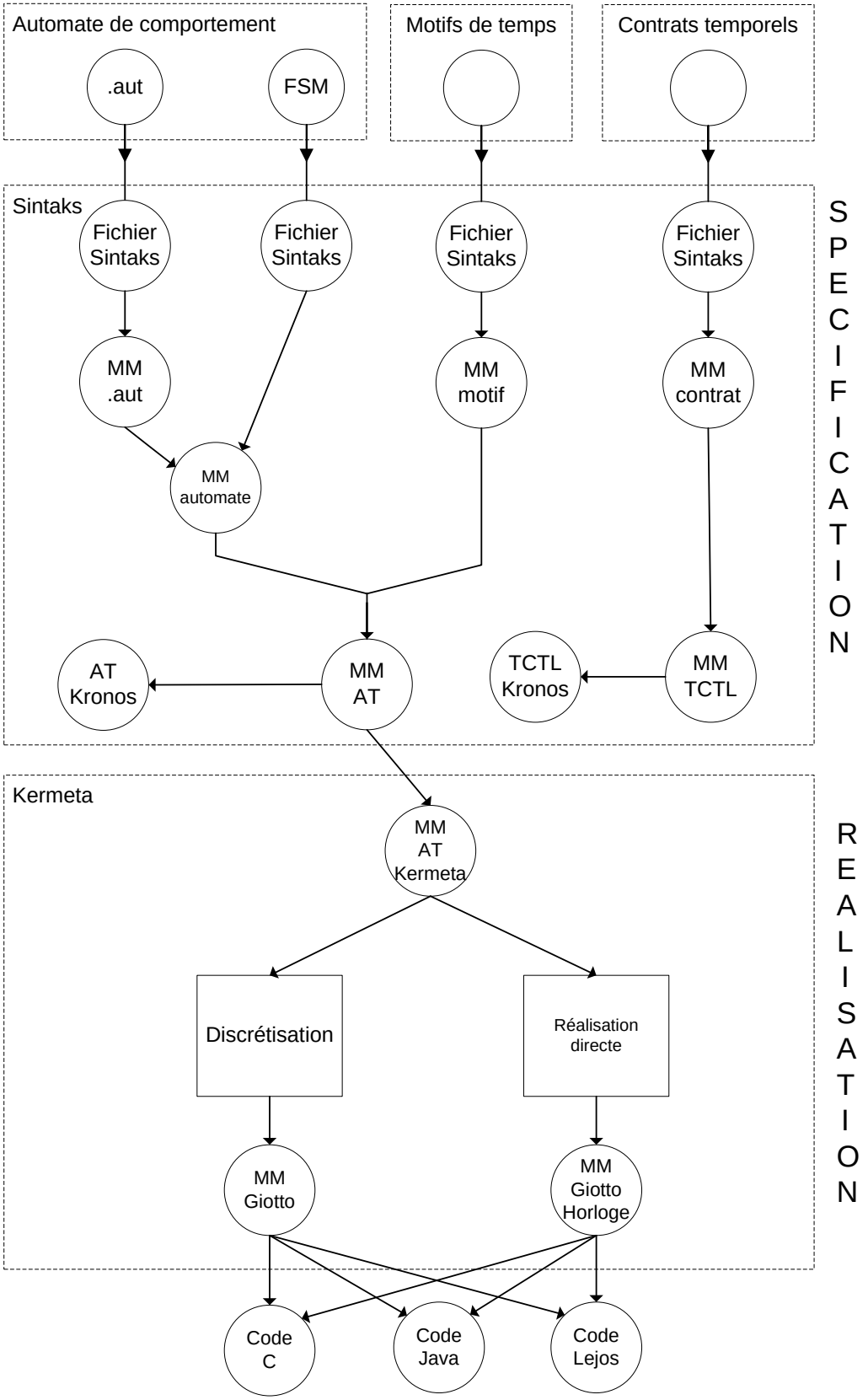



FIG. 4.1 – Processus de génération.

méta-modèle et ainsi d'obtenir un fichier $ecore$ sans informations superflues. Le second étend le premier en ajoutant des opérations comme l'ajout des propriétés de temps ou un pretty-printer vers Kronos.

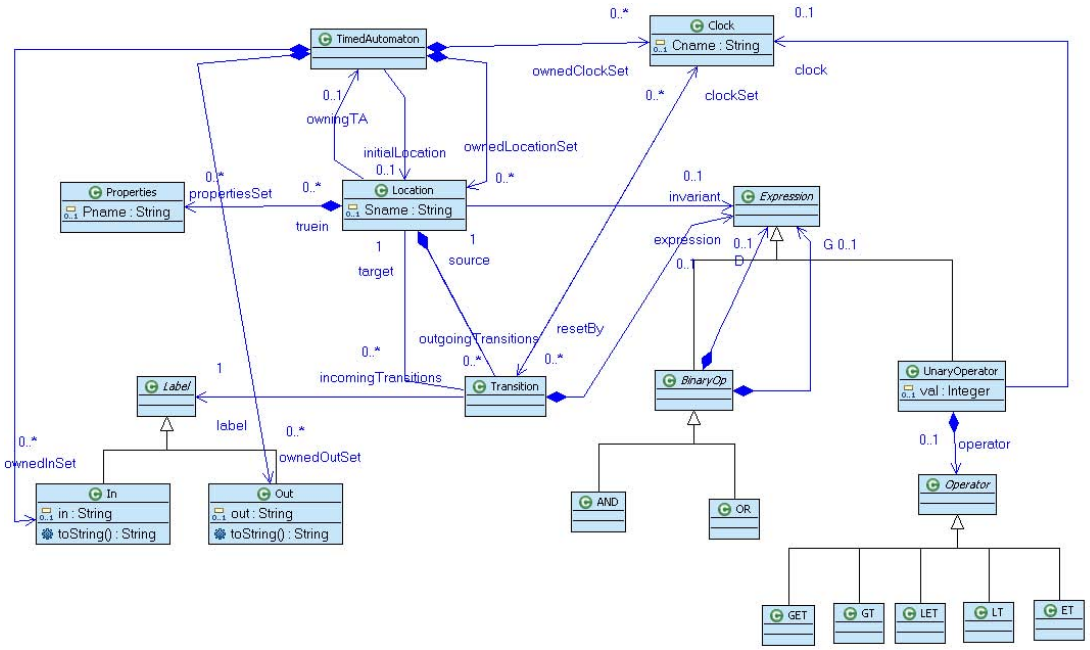



FIG. 4.2 – Méta-modèle des automates temporisés.

Une opération d'ajout de propriétés de temps permet d'ajouter à un automate temporisé l'un des motifs de temps définis 2.1. Quatre opérations ont été créées. Chacune d'elle modifie l'automate temporisé en respectant les possibles applications des motifs définies.

Une opération permet la projection de l'automate temporisé sur ses transitions temporelles. La réduction du nombre d'horloges créée un fichier $tg$ pour Optikron puis récupère le résultat de la réduction pour reconstruire l'automate temporisé.

**TCTL** Un méta-modèle a été développé correspondant à la syntaxe de la logique TCTL définie dans la section 1.2.3. La Figure 4.3 présente ce méta-modèle. Une opération permet de fournir le fichier Kronos.

**Motifs comportementaux** Un méta-modèle a été créé pour les motifs comportementaux. Chaque motif a une classe où les attributs sont les paramètres du motif. Chaque motif hérite d'une classe abstraite motif. La racine du méta-modèle est une classe pouvant contenir un ensemble de motifs. Un modèle de ce type est créé pour chaque composant du système et représente les motifs comportementaux qui lui sont attachés. La Figure 4.4 présente ce méta-modèle.

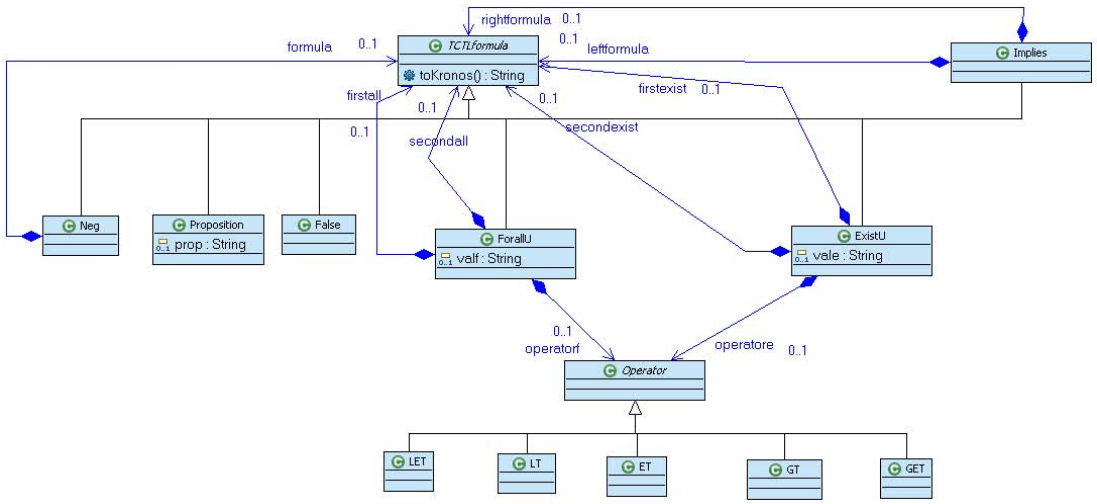



FIG. 4.3 – Méta-modèle de TCTL.

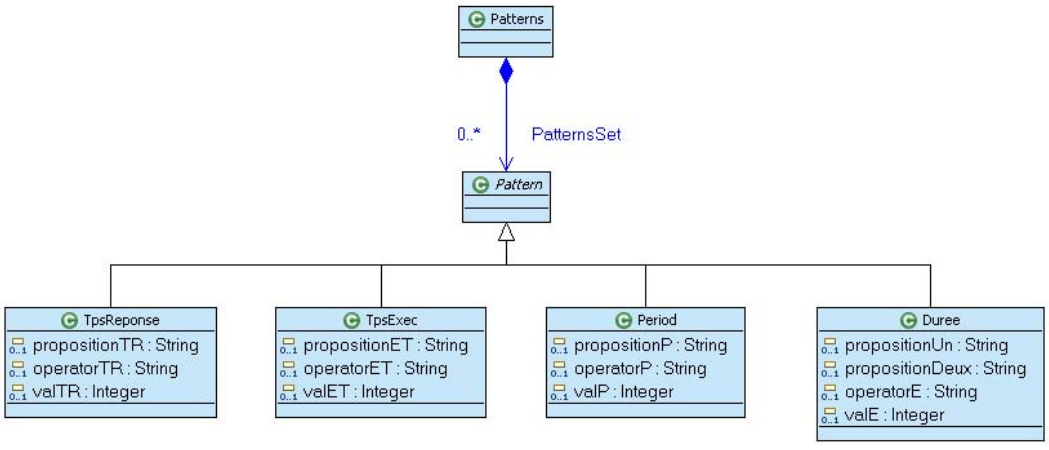



FIG. 4.4 – Méta-modèle des motifs comportementaux.

**Contrats temporels** Un méta-modèle a été créé pour les contrats temporels. Chaque contrat a une classe où les attributs sont les paramètres du contrat. Chaque contrat hérite d'une classe abstraite contrat. La racine du méta-modèle est une classe pouvant contenir un ensemble de contrats. Un modèle de ce type est créé pour chaque interface requise des composants du système qui ont des contrats temporels associés. La Figure 4.5 présente ce méta-modèle.

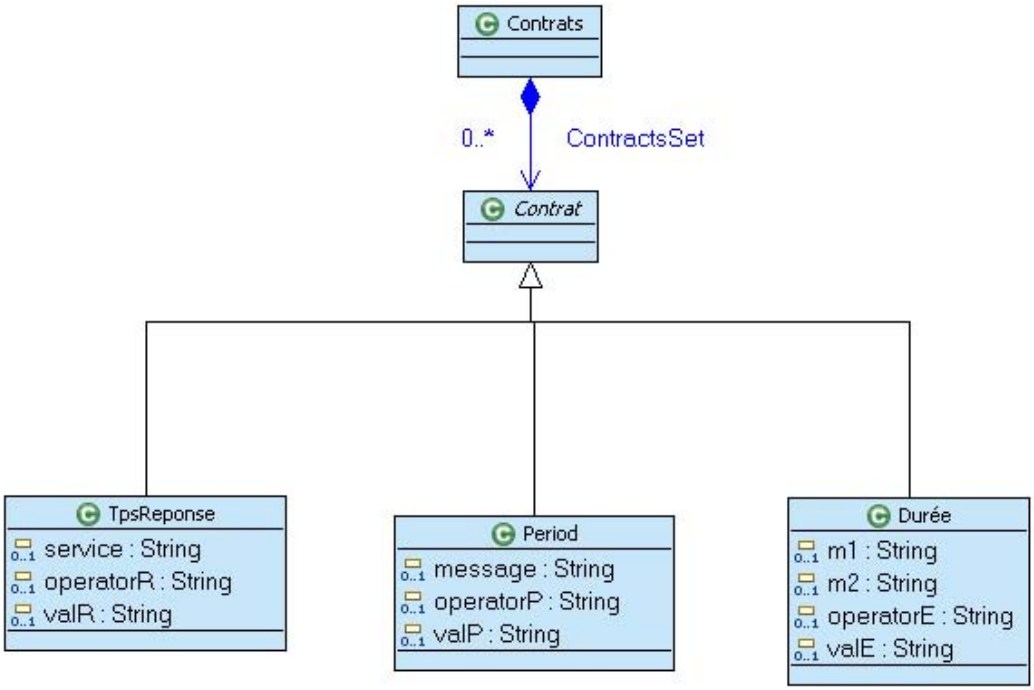



FIG. 4.5 – Méta-modèle des contrats temporels.

**Giotto** Un méta-modèle correspondant à la syntaxe Giotto a été développé. Il représente les concepts utilisés par Giotto. Une extension de ce méta-modèle a été effectuée pour permmettre de représenter les informations nécessaires à la génération de fichiers intermédiaires décris dans la section 3.5.3. Une opération permet la génération de l'ensemble des fichiers du moniteur vers le langage choisi.

### 4.2.2 Spécification

La phase de spécification se décompose en trois parties : récupération du système non-temporisé, l'écriture des contrats et l'intégration du temps dans le comportement. Ces trois parties ont comme point d'entrée un fichier texte. Elles sont écrites en langage naturel puis transformées en modèle grâce à Sintaks. L'intégration du temps dans le comportement est faite grâce à Kermeta.

Les spécifications comprennent l'automate de comportement du composant, les motifs de temps et les contrats de qualités de services. L'écriture de ces spécifications peut se faire à l'aide d'une grammaire Sintaks ou directement en utilisant un éditeur de modèle. Dans les deux cas, les spécifications obtenues se présentent sous la forme de fichiers $xmi$. Pour un composant, il y a un fichier pour le comportement, un pour les motifs comportementaux et un par interfaces requises pour les contrats temporels. Ce processus est illustré par la Figure 4.6.

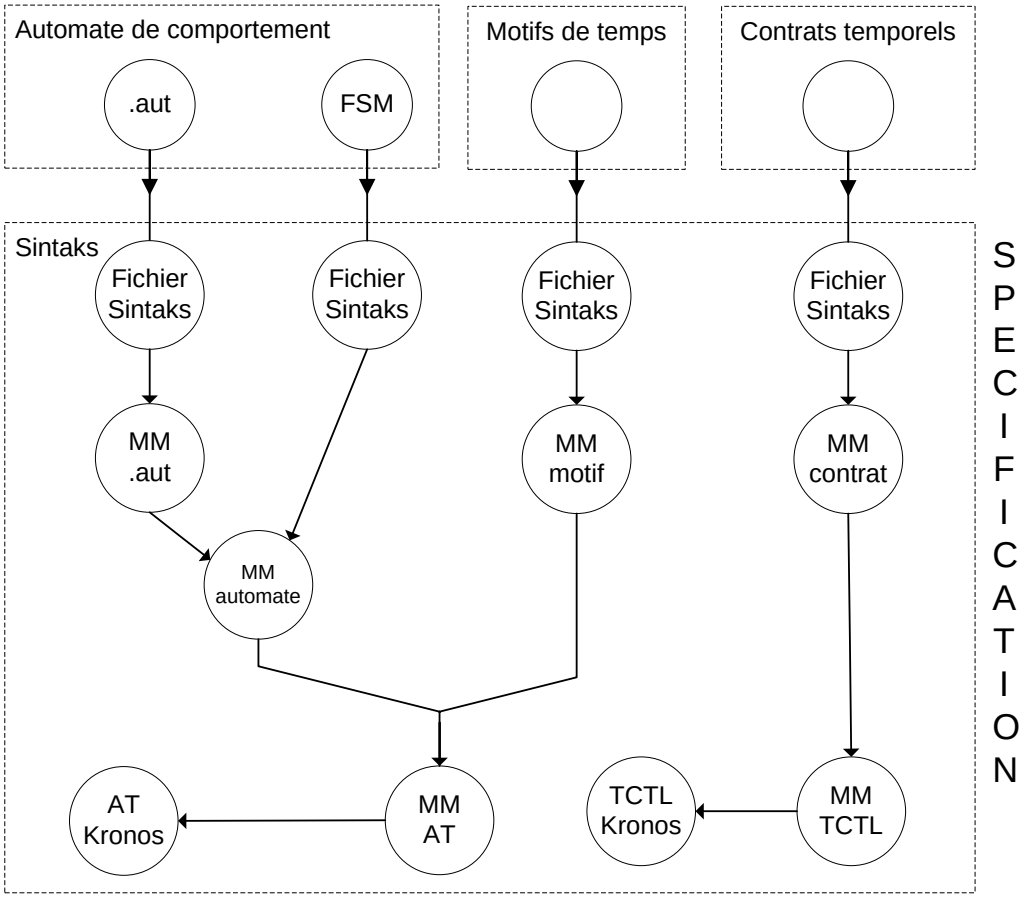



FIG. 4.6 – Phase de spécification.

#### 4.2.2.1 Création de l'automate

L'automate de comportement du composant peut être créé à partir de plusieurs formalismes de description d'automate. Le méta-modèle d'automate pour notre application est celui des automates à entrées/sorties. L'automate a un seul état initial et est déterministe. Ce méta-modèle est une restriction de celui des automates temporisés sans les horloges, les gardes et les réinitialisations. Nous utilisons donc le méta-modèle des automates temporisé comme cible des fichiers Sintaks. Pour chaque formalisme, un fichier Sintaks est créé pour effectuer le passage

du formalisme désiré vers le méta-modèle des automates temporisés. Nous avons écrit deux fichiers Sintaks. Le premier prend en compte le formalisme d'automates des fichiers $.aut$. Le second est celui de l'exemple Kermeta des FSM. Un exemple pour ces deux formalismes est donné par la Figure 4.7. Le formalisme $.aut$ donne sur la première ligne, l'état initial, le nombre d'états et le nombre de transitions. Il décrit ensuite l'ensemble des transitions du systèmes où $(s1, label, s2)$ représente la transition de $s1$ vers $s2$ portant le label $label$. Le formalisme des FSMs fournit l'ensemble des états, l'état initial puis les transitions.

Formalisme $.aut$ :

```
des(0,8,8)
(0,"?launch",1)
(1,"!launch$",2)
(2,"!start",3)
(3,"?start$",4)
(4,"?getSound",5)
(5,"!getSound$",6)
(6,"sound",7)
(7,"?sound$",4)
```

Formalisme $FSM$ :

```
state s0;
state s1;
state s2;
state s3;
state s4;
state s5;
state s6;
state s7;

initial s0;

transition from s0 to s1 label ?launch;
transition from s1 to s2 label !launch$;
transition from s2 to s3 label !start;
transition from s3 to s4 label ?start$;
transition from s4 to s5 label ?getSound;
transition from s5 to s6 label !getSound$;
transition from s6 to s7 label !sound;
transition from s7 to s4 label ?sound$;
```

FIG. 4.7 – Exemple de fichiers d'entrée.

Le premier formalisme nécessite une étape intermédiaire entre le fichier Sintaks et le méta-modèle des automates temporisés. Ceci permet d'avoir une grammaire Sintaks simple permet-

tant de passer de la syntaxe des fichiers $aut$ à leur méta-modèle. Une transformation de modèle permet d'obtenir l'automate pour le méta-modèle souhaité. Cette étape intermédiaire est constituée d'une grammaire simple en Sintaks et d'une transformation courte à la place d'une grammaire complexe pour effectuer le passage en une fois.

Le second formalisme permet d'obtenir directement le fichier $xmi$ décrivant l'automate.

D'autres fichiers Sintaks existent permettant de transformer des fichiers Kronos dans notre formalisme d'automates temporisés après l'étape de réduction d'horloges lors de la transformation avec discrétisation.

#### 4.2.2.2 Motifs de temps

Un méta-modèle permet de décrire les motifs de temps. Il représente un ensemble de motifs qui seront ajoutés au comportement. Chaque motif contient comme attributs les paramètres qui le compose. Une grammaire en Sintaks permet de passer d'un fichier texte à ce méta-modèle. Une transformation en Kermeta permet d'intégrer ces motifs à l'automate de comportement. Les deux fichiers $xmi$ sont chargés puis les motifs sont appliquées et un fichier $xmi$ contenant l'automate de comportement temporisé. Ce fichier peut fournir le fichier Kronos afin de l'utiliser lors de la composition du composant.

#### 4.2.2.3 Ecritures des contrats

Les contrats sont écrits en langage naturel puis ensuite transformés dans le méta-modèle de TCTL. Ce passage se fait en deux étapes à l'aide d'un méta-modèle intermédiaire représentant les contrats temporels. C'est une abstraction des formules TCTL conservant uniquement les variables renseignés par l'architecte. Il est comparable à celui des motifs temporels. Sintaks permet le passage du format texte à ce méta-modèle intermédiaire. Une transformation permet de passer de ce méta-modèle à celui de TCTL.

L'ensemble des contrats temporels d'une interface d'un composant est écrit dans un fichier qui est transformé par Sintaks. Chaque motif temporel a un template en Sintaks. Ce template est composé de terminaux et de valeurs. Chacune de ces valeurs apparaît dans le méta-modèle intermédiaire auquel le fichier Sintaks est lié.

Le modèle créé par Sintaks est ensuite transformé en TCTL. Cette transformation est écrite en Kermeta et permet l'écriture des contrats en TCTL. Les contrats sont sauvés dans un fichier $xmi$. Ce fichier est attaché à l'interface requise du composant à laquelle les contrats temporels sont attachés. Il peut fournir les fichiers Kronos contenant chacun un contrat temporel pouvant être utilisé lors de la composition de composant.

### 4.2.3 Réalisation du moniteur

Les deux algorithmes de transformation sont implantés en Kermeta. Chacune prend en paramètre un automate temporisé et un langage cible : Java, C ou Lejos(un langage java restreint pour le développement de robots Lego). Les deux retournent un modèle Giotto. Une fois le modèle Giotto créé, une opération permet de générer l'ensemble du moniteur : le code Giotto et les fichiers intermédiaires. Le processus est illustré dans la Figure 4.8.

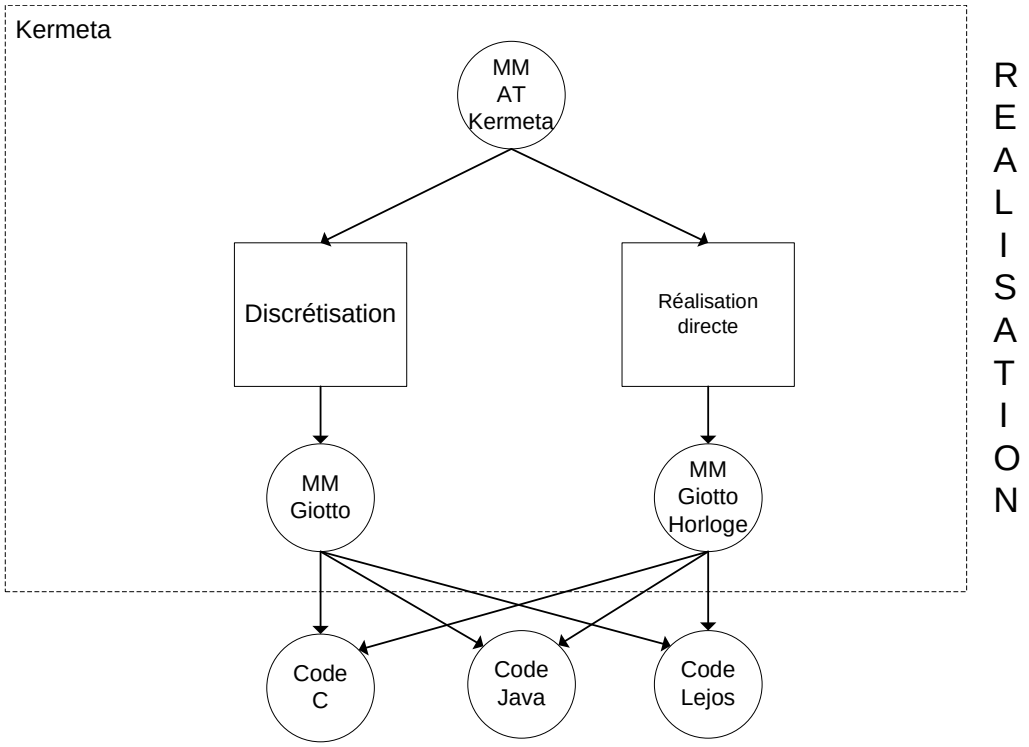



FIG. 4.8 – Phase de réalisation.

L'algorithme avec discrétisation se passe en deux phases et comprend donc deux transformations. Les deux transformations sont écrites en Kermeta. Le code Kermeta est proche de l'algorithme en langage naturel. La première transformation prend un automate temporisé en paramètre et retourne un automate discrètisé. La second transformation prend en paramètre cet automate discrètisé et retourne un modèle Giotto.

Le second algorithme prend en paramètre un automate temporisé et retourne un modèle Giotto. Le code en Kermeta est très proche de l'algorithme défini précédemment.

#### 4.2.3.1 Génération du moniteur

Cette opération prend en paramètre la cible choisie pour le moniteur et un chemin pour stocker les fichiers. Cette opération est attachée au méta-modèle Giotto. La différence de génération entre les deux transformations est la gestion des horloges. Le second méta-modèle héritant du premier, les opérations spécifiques sont réécrites pour la gestion des horloges.

L'opération de la classe principale génère le fichier Giotto en parcourant les éléments le constituant : les sensors, les actuators, les tâches, les drivers et les modes. Chacun des éléments effectue les traitements qui lui sont propres : impression des informations des générations du fichier correspondant si nécessaire. La classe principale s'occupe de la génération du fichier s'occupant des horloges. Dans le cas d'une génération vers Lejos, le fichier Giotto est compilé en faisant appel au compilateur Giotto. Le compilateur ne produira pas le fichier Ecode mais un ensemble de fichiers utilisés par la plate-forme Giotto correspondant au fichier Ecode.

## 4.3 Conclusion sur l'implantation des outils

L'ensemble de la méthode définit dans ce document a été implanté en Sintaks pour la partie spécification et en Kermeta pour la partie transformation. Tout le procesuss, une fois les fichiers de spécification écrits, est entièrement automatique et ne requiert l'intervention de l'architecte qu'en cas de problème. Cet automatisation permet de pouvoir rejouer le processus en ne modifiant que les fichiers de spécification.

L'enrichissement des formalismes d'automates en entrée du processus est possible en fournissant un fichier Sintaks permettant de créer l'automate.

# Chapitre 5

# Application

L'approche complète a été validé en développant un application modélisant le comportement d'une voiture à l'aide de Lego Mindstorms[1]. Nous définissons un ensemble de propriétés temporelles sur le comportement de la voiture. Nous présentons dans un premier temps les Lego Mindstorms et le langage d'implémentation choisi Lejos[2]. Puis nous définissons le comportement nominal de la voiture sans propriétés temporelles. Ensuite nous introduisons les propriétés de temps et générons le moniteur permettant de les surveiller. Enfin, nous étudions le résultat du déplacement du robot dans un environnement hostile.

## 5.1 Lego Mindstorms et Lejos

### 5.1.1 Lego Mindstorms

Les Lego Mindstorms sont constitués d'une brique programmable, de capteurs et de pièces permettant de construire un robot. Le programme peut interagir avec les capteurs et donc modifier son comportement. Le langage fourni avec les Lego ne permet qu'un nombre limité d'action, étant à la base un langage graphique. Nous avons modifié le firmware d'origine de la brique afin de pouvoir changer de langage de programmation. Nous avons choisi un firmware, Lejos, permettant d'exécuter une JVM réduite mais suffisante pour l'implantation de notre moniteur.

### 5.1.2 Lejos

Lejos est un firmware permettant d'écrire des programmes Java pour les charger sur la brique Lego. Le langage Java est réduit afin de pouvoir exécuter la JVM sur la brique qui a une place limitée en mémoire. La classe nous intéressant pour l'implantation de notre moniteur est présente : les processus. Le code Java de Giotto a du être modifié pour tenir compte des restrictions de Lejos : pas de tables de hachage, pas de gestion de fichier, pas de switch et pas de chargement dynamique de classes. Le développement d'un programme pour Lejos se fait en trois étapes. La première est la compilation des fichiers avec les restrictions. La seconde

[1] http ://mindstorms.lego.com/

[2] http ://lejos.sourceforge.net/

est l'éditeur de lien afin d'obtenir un fichier binaire exécutable pas la JVM. La dernière est le chargement du programme sur la brique.

## 5.2 Comportement nominal

L'exemple modélise le comportement d'une voiture sur une route. Le véhicule se déplace seul et réagit selon des événements extérieurs comme la présence d'un autre véhicule devant lui. Le véhicule se déplace sur une route avec une vitesse maximale autorisée. Une distance de sécurité doit être respectée avec les autres véhicules. Le véhicule devra ralentir lorsque sa vitesse sera trop élevée ou quand il sera trop proche d'un autre véhicule.

La Figure 5.1 présente les composants du véhicule. Le système est composé de trois composants $Radar$, $Vitesse$ et $Controleur$. Le composant $Vitesse$ permet de connaître la vitesse du véhicule. Le composant $Radar$ permet de connaître la distance séparant le véhicule de son prédécesseur. Le composant $Controleur$ est averti de la vitesse et de la distance du véhicule, il peut alors ralentir si nécessaire.

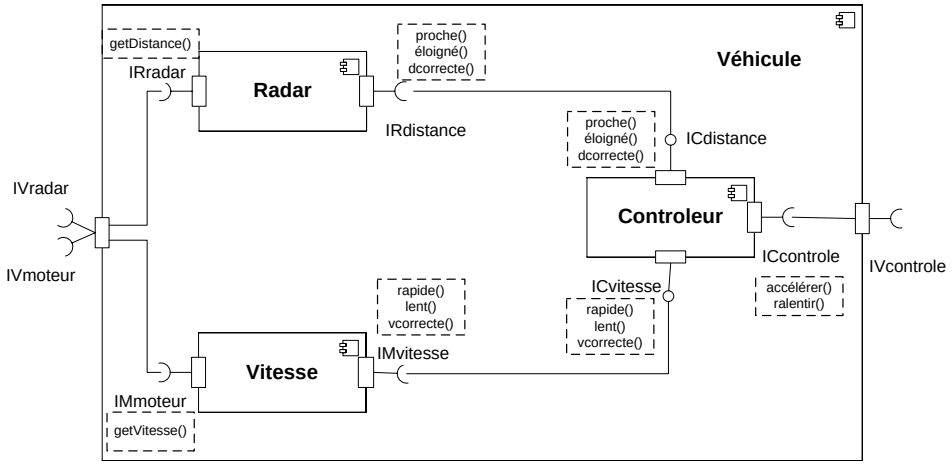



FIG. 5.1 – Le composite Véhicule

La Figure 5.2 présente le comportement du $Radar$. Celui-ci interroge le radar physique afin de connaître la distance le séparant du véhicule le précédant. Selon la distance, le $Radar$ appelle le service correspondant sur le $Controleur$ : $proche$ ou $dcorrecte$ (pour une distance correcte). Nous ne modélisons pas le choix de l'appel effectué en fonction de la distance. Nous nous intéressons uniquement au comportement possible.

La Figure 5.3 présente le comportement du composant $Vitesse$. Il interroge le moteur physique afin de connaître la vitesse actuelle du véhicule puis informe le $Controleur$ de l'état de la vitesse en appelant le service correspondant : $lent$ ou $vcorrecte$(pour une vitesse correcte). Nous ne nous intéressons pas aux bornes délimitant les critères de vitesse.

La Figure 5.4 présente le comportement du $Controleur$. Il attend la réception d'un message pour effectuer l'opération correspondante. Si les messages $proche$ ou $rapide$ arrivent alors il demande de ralentir puis envoie les acquittements. Si les messages $dcorrecte$ ou

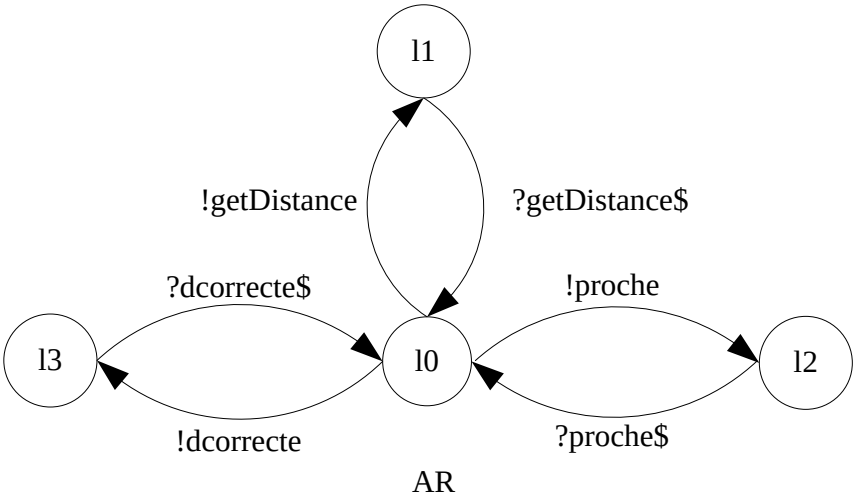



FIG. 5.2 – Comportement de Radar

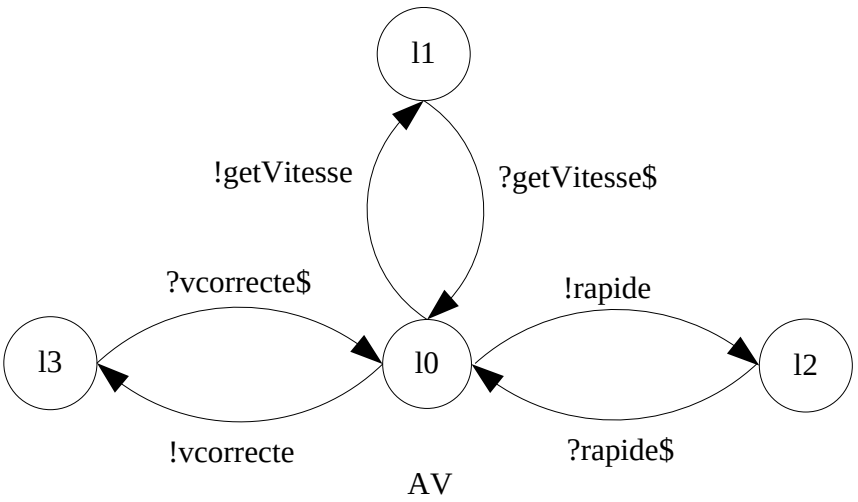



FIG. 5.3 – Comportement de Vitesse

$vcorrecte$ arrivent, il envoie un acquittement sans effectuer d'autres appels.

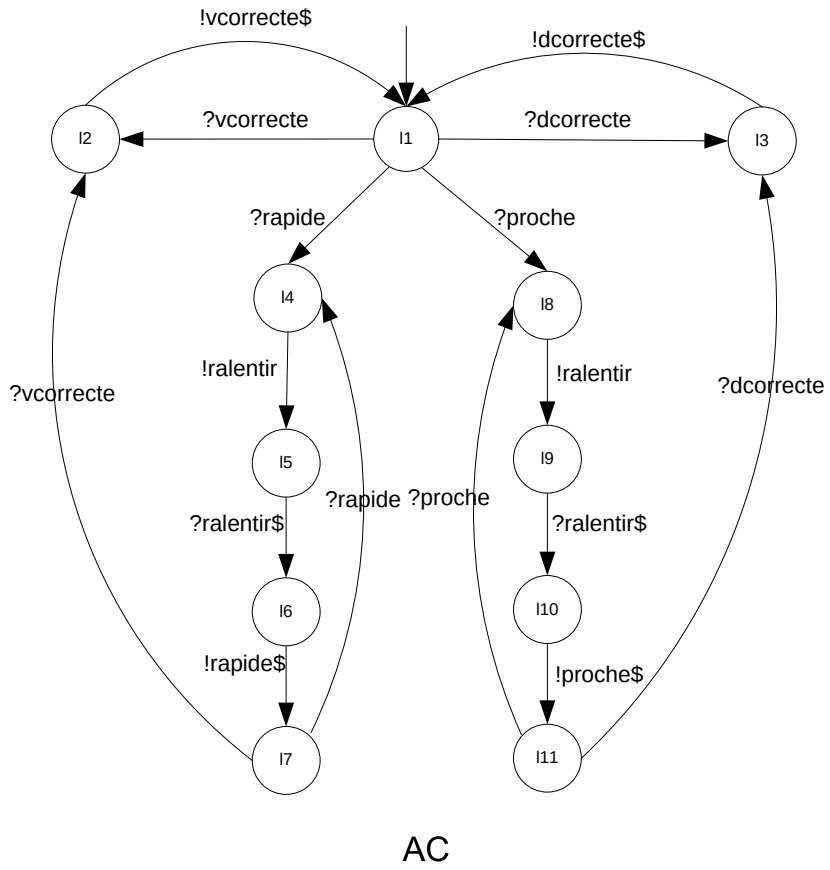



FIG. 5.4 – Comportement de Controleur

## 5.3 Ajout de temps et génération du moniteur

Nous déroulons notre approche en enrichissant les composants de propriétés temporelles puis en générant un moniteur de qualité de service.

### 5.3.1 Ajout de propriétés de temps

Nous ajoutons des propriétés temporelles aux fonctions afin de définir une qualité de service pour le véhicule. L'échantillonnage de la distance et de la vitesse doit s'effectuer au moins 2 fois par seconde. Ces propriétés sont ajoutées aux comportements des composants $Vitesse$ et $Radar$. Le véhicule ne doit pas rester pendant un temps trop long à une distance non correcte de son prédécesseur, le composant $Radar$ doit pouvoir appelé le service $dcorrecte$ après un certain temps après avoir détecté un problème. De même, le véhicule ne peut rester trop

longtemps à une vitesse incorrecte. Enfin le $controleur$ assure que sous la condition que le ralentissement s'effectue dans un temps donné, il peut traiter les messages de distance et vitesse non correctes. L'unité de temps est en milliseconde $ms$ et vaut $100ms$.

**Echantillonnage de la distance** Cette propriété correspond au motif comportemental *période* appliqué au message $?getDistance\$$ avec l'opérateur $<$ et la borne $5$. Ce motif est ajouté dans l'automate de $Radar$. L'ajout du motif est illustré par l'automate $ATR$ de la Figure 5.5. Toutes les localités de l'automate sont à la fois successeur et prédécesseur de la transition portant $?getDistance\$$. L'ensemble des localités de l'automate a donc un invariant $x1 < 5$. Une localité $l0a$ est ajoutée avec la propriété atomique $?getDistance\$\_occur$. Cette localité permet de rendre cette propriété vraie que lors de l'occurrence de $?getDistance$. La propriété $?getDistance\$\_occur\_next$ dans la localité $l0$, unique localité successeur de la localité ajoutée. Cette propriété permettra lors du model-checking, la vérification de la périodicité de $?getDistance$.

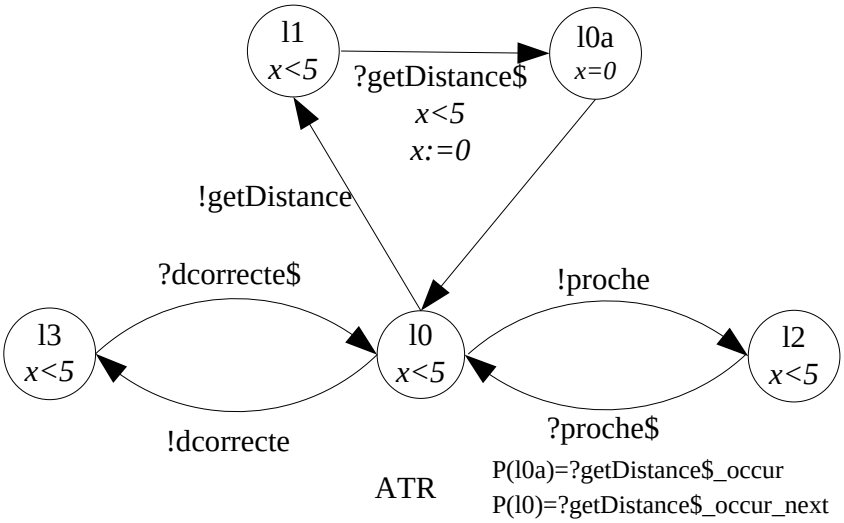



FIG. 5.5 – Comportement temporisé de Radar

**Echantillonnage de la vitesse** Cette propriété correspond au motif comportemental *période* appliqué au message $!getVitesse\$$ avec l'opérateur $<$ et la borne $5$ ajouté dans l'automate du composant $Vitesse$. L'ajout du motif est illustré par l'automate $ATV$ de la Figure 5.6. Comme pour le motif précédant, l'ensemble des localités de l'automate a un invariant $x1 < 5$.

**Réactivité du contrôleur pour le traitement des informations de distance** Cette propriété correspond à l'ajout d'un contrat temporel sur l'interface $IRdistance$ du composant $Radar$ utilisant le motif *temps de réponse*. Les paramètres du contrat temporel sont $!proche$, $<$ et 3. Ce contrat s'écrit en langage naturel et en TCTL par : le temps de réponse de $!proche$ est inférieur à 3 unités de temps, $\forall\Box(!proche\_occur \rightarrow \forall\Diamond_{<3}?proche\$\_occur)$.

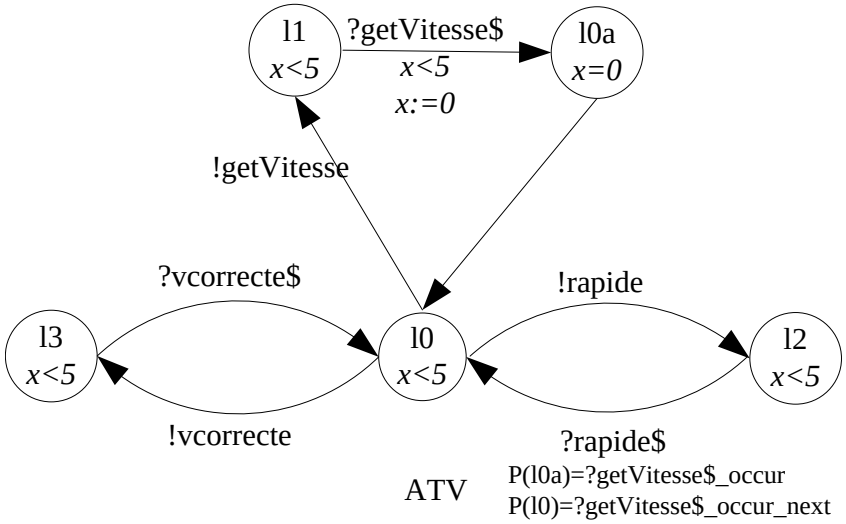



FIG. 5.6 – Comportement temporisé de Vitesse

**Réactivité du controleur pour le traitement des informations de vitesse** Cette propriété correspond à l'ajout d'un contrat temporel sur l'interface $IVvitesse$ du composant $Vitesse$ utilisant le motif *temps de réponse*. Les paramètres du contrat temporel sont !$rapide$, $<$ et 3. Ce contrat s'écrit en langage naturel et en TCTL par : le temps de réponse de !$rapide$ est inférieur à 3, $\forall\Box(!rapide\_occur \rightarrow \forall\Diamond_{<3}?rapide\$\_occur)$.

**Réactivité du ralentissement** Cette propriété correspond à l'ajout d'un contrat temporel sur l'interface $ICcontrole$ du composant $Controleur$ utilisantsle motif *temps de réponse*. Les paramètres du contrat temporel sont !$ralentir$, $<$ et 2. Ce contrat s'écrit en langage naturel et en TCTL par : le temps de réponse de !$ralentir$ est inférieur à 2, $\forall\Box(!ralentir\_occur \rightarrow \forall\Diamond_{\sim 2}?ralentir\$\_occur)$.

**Traitements des messages** Cette propriété correspond au fait que lorsque le $controleur$ reçoit un message signalant un problème sur la distance ou la vitesse, il prend les mesures nécessaires dans un délai de maximum $4$ unités de temps. Elle correspond à l'application du motif comportemental *temps de service* entre la réception du message et l'envoi de son acquittement. Quatre motifs sont ajoutés, un pour chaque message : $proche$ et $rapide$. L'ajout des deux motifs est illustré par l'automate $ATC$ de la Figure 5.7 et concerne les horloges $x1, x2$.

**Distance de sécurité** Cette propriété correspond à l'ajout d'un motif temporel *durée* entre la réception du message $proche$ et la réception du message $dcorrecte$ avec l'opérateur $<$ et la valeur $30$. L'ajout de ce motif est illustré par l'automate $ATC$ de la Figure 5.7 et concerne l'horloge $x5$. L'ensemble des localités a un invariant $x5 < 30$.

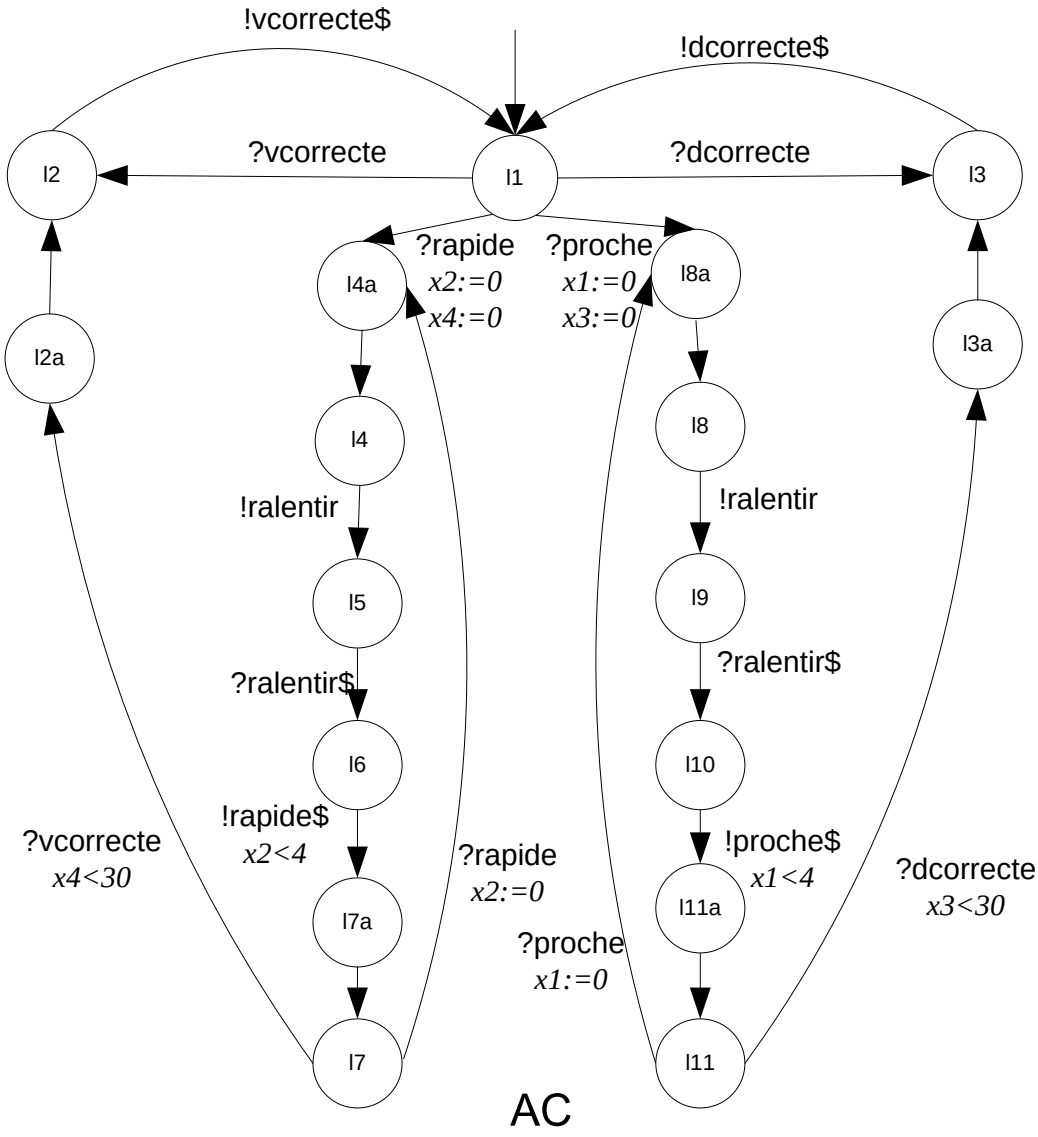



FIG. 5.7 – Comportement temporisé de Contrôleur

**Respect de la vitesse** Cette propriété correspond à l'ajout d'un motif temporel *durée* entre la réception du message $rapide$ et la réception du message $vcorrecte$ avec l'opérateur $<$ et la valeur 30. L'ajout de ce motif est illustré par l'automate $ATC$ de la Figure 5.7 et concerne l'horloge $x6$. L'ensembles des localités a un invariant $x6 < 30$.

### 5.3.2 Vérification de l'assemblage

Nous vérifions l'assemblage des trois composants en utilisant Kronos. Les trois automates produisent chacun un fichier Kronos ainsi que les contrats temporels.

**Interface IRdistance-ICdistance** Le *model-checker* vérifie que l'automate de comportement de $Controleur$ satisfait les trois contrats temporels de $IRdistance$. La réponse de Kronos est $true$ pour ce contrat, la composition $Radar\text{-}Distance$ est validée.

**Interface IMvitesse-ICvitesse** Le *model-checker* vérifie que l'automate de comportement de $Controleur$ satisfait les trois contrats temporels de $IMvitesse$. La réponse de Kronos est $true$ pour ce contrat, la composition $Vitesse\text{-}Distance$ est validée.

Des motifs temporels ne sont captés par aucun contrat. C'est le cas des motifs pour le respect de la vitesse et de la distance de sécurité. Ces motifs correspondent à des souhaits de l'architecte sur l'implantation du véhicule qu'il souhaite surveiller lors de l'exécution. De même, les contrats du $controleur$ sur l'environnement ne sont pas vérifiables lors de la modélisation.

### 5.3.3 Génération du moniteur de qualité de service

Afin de comparer les deux algorithmes décrits dans le chapitre 3, nous avons utilisé les deux pour obtenir le moniteur de qualité de service. Nous effectuons la composition synchrone des trois automates puis la projection sur les transitions temporisées. L'automate résultant est présenté sur la Figure 5.8. L'automate est composé de 5 localités, 6 horloges et 16 transitions. Il est uniquement composé de transitions avec des informations temporelles.

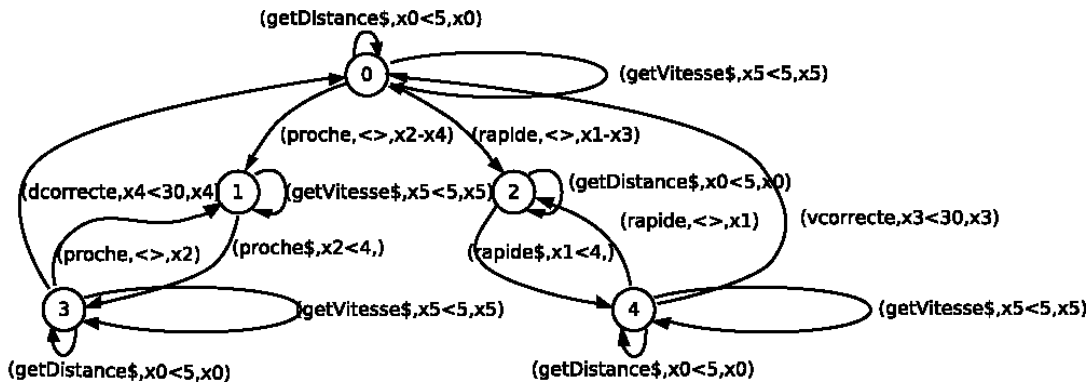



FIG. 5.8 – Produit synchronisé projeté des trois automates

**Moniteur avec discrétisation** Les horloges de l'automate peuvent être séparés en deux groupes selon la valeur des gardes qui leur sont associés. Nous produisons donc deux moniteurs, le premier pour les horloges $x3$ et $x4$ et le second avec les autres horloges. Pour chaque groupe d'horloges, nous projetons l'automate sur les horloges concernées et par rapport à l'ordre de grandeur. Nous réduisons ensuite le nombre d'horloges. L'automate résultant de la projection pour les horloges $x3$ et $x4$ est présenté sur la Figure 5.9. Il représente le comportement temporel attendu par unité de temps pour $x3$ et $x4$. L'ensemble des messages est réduit aux messages liés à ces horloges.

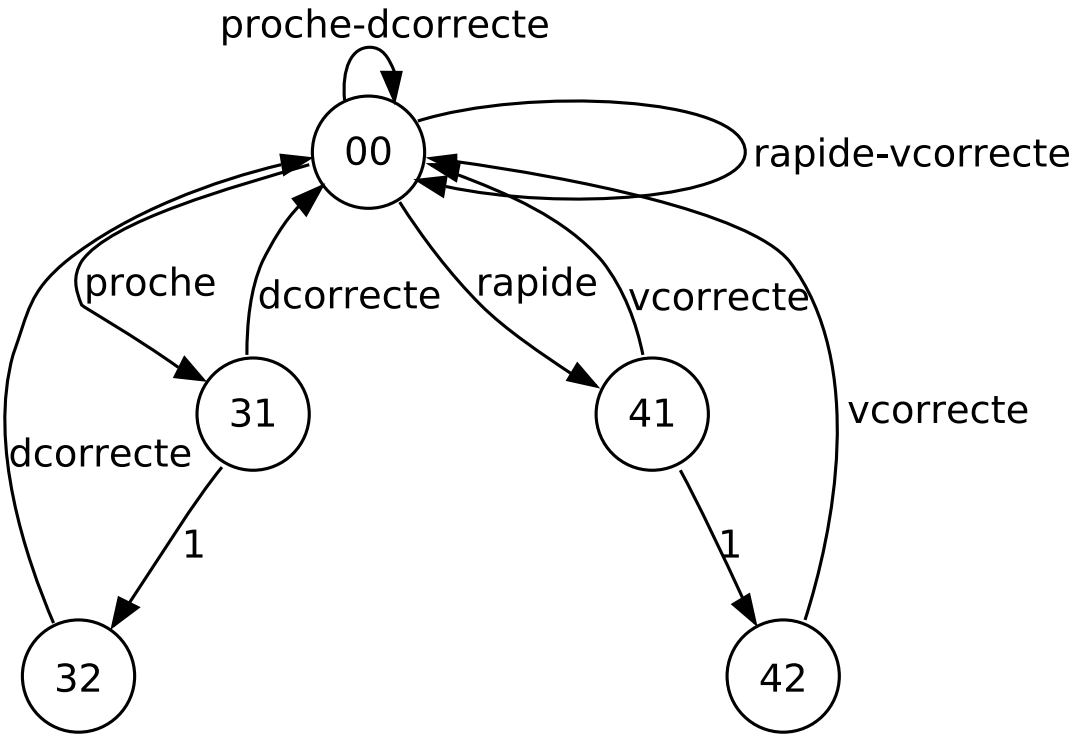



FIG. 5.9 – Automate pour le moniteur des horloges $x3$ et $x4$

Pour chacun des automates nous produisons le code Giotto et les fichiers nécessaire à la notification de l'arrivée des messages.

**Moniteur avec réalisation** La différence d'ordre de grandeur des horloges n'a pas d'influence sur la réalisation de l'automate temporisé en Giotto. Nous générons le code Giotto de la Figure 5.10 et les fichiers pour la notification.

**Insertion des notifications dans le code** Le code a été développé en suivant les automates de comportement non temporisés du véhicule. Ce code est écrit en Java avec les restrictions nécessaires pour Lejos. La liste des messages temporisé est disponible grâce à la projection de

```
start 0 {
        mode 0() period 1000 {
                exitfreq 1 do 0(driver_getDistance$0);
                exitfreq 1 do 0(vdriver_getDistance$);
                exitfreq 1 do 1(driver_proche24);
                exitfreq 1 do 0(driver_getVitesse$5);
                exitfreq 1 do 0(vdriver_getVitesse$);
                exitfreq 1 do 2(driver_rapide13);
                taskfreq 1 do Inc();
        }
        mode 1() period 1000 {
                exitfreq 1 do 3(driver_proche$);
                exitfreq 1 do 3(vdriver_proche$);
                exitfreq 1 do 1(driver_getVitesse$5);
                exitfreq 1 do 1(vdriver_getVitesse$);
                taskfreq 1 do Inc();
        }
        mode 2() period 1000 {
                exitfreq 1 do 4(driver_rapide$);
                exitfreq 1 do 4(vdriver_rapide$);
                exitfreq 1 do 2(driver_getDistance$0);
                exitfreq 1 do 2(vdriver_getDistance$);
                taskfreq 1 do Inc();
        }
        mode 3() period 1000 {
                exitfreq 1 do 3(driver_getDistance$0);
                exitfreq 1 do 3(vdriver_getDistance$);
                exitfreq 1 do 1(driver_proche2);
                exitfreq 1 do 3(driver_getVitesse$5);
                exitfreq 1 do 3(vdriver_getVitesse$);
                exitfreq 1 do 0(driver_dcorrecte4);
                exitfreq 1 do 0(vdriver_dcorrecte);
                taskfreq 1 do Inc();
        }
        mode 4() period 1000 {
                exitfreq 1 do 4(driver_getDistance$0);
                exitfreq 1 do 4(vdriver_getDistance$);
                exitfreq 1 do 0(driver_vcorrecte3);
                exitfreq 1 do 0(vdriver_vcorrecte);
                exitfreq 1 do 4(driver_getVitesse$5);
                exitfreq 1 do 4(vdriver_getVitesse$);
                exitfreq 1 do 2(driver_rapide1);
                taskfreq 1 do Inc();
        }
}
```

FIG. 5.10 – Code pour le moniteur avec réalisation

l'automate temporisé. Cette liste permet de savoir où les notifications doivent être effectuées dans le code. Pour chaque message dans la liste, une fois l'occurrence passée, un appel à une fonction est effectué pour notifier Giotto. Par exemple le code de la Figure 5.11, correspondant au $Controleur$, contient des appels aux fonctions créés lors de la génération du moniteur.

```
package giotto.functionality.code.robot
public class Radar extends Thread{
        Radarport rport=new Radarport();
        Controleur control;
        public Radar(Controleur c){
                control=c;
        }
        public void run() {
                while(true){
                        int distance=rport.getDistance();
                        getDistance$.messageoccurs();
                        if (distance<20){
                                proche.messageoccurs();
                                control.proche();
                                proche$.messageoccurs();
                        } else {
                                dcorrecte.messageoccurs();
                                control.dcorrecte();
                        }
                }
        }
}
```

FIG. 5.11 – Code tissé de $Radar$

## 5.4 Exécution

Nous utilisons deux robots se déplaçant en ligne droite. Le premier joue le rôle du véhicule précédant le véhicule que nous souhaitons observer. Le premier robot a une vitesse variable, il alterne les accélérations et les ralentissements afin de simuler un traffic routier. Le second robot contient le code du véhicule ainsi que les moniteurs de qualité de service. La photo de la Figure 5.12 montre les deux robots.

### 5.4.1 Moniteurs issus de la discrétisation

Le code du robot et les deux moniteurs sont chargés sur le robot puis les deux robots sont démarrés. La qualité de service pour l'échantillonnage de la vitesse et de la distance n'est jamais violée car les capteurs ont un temps de réponse quasi instantané.

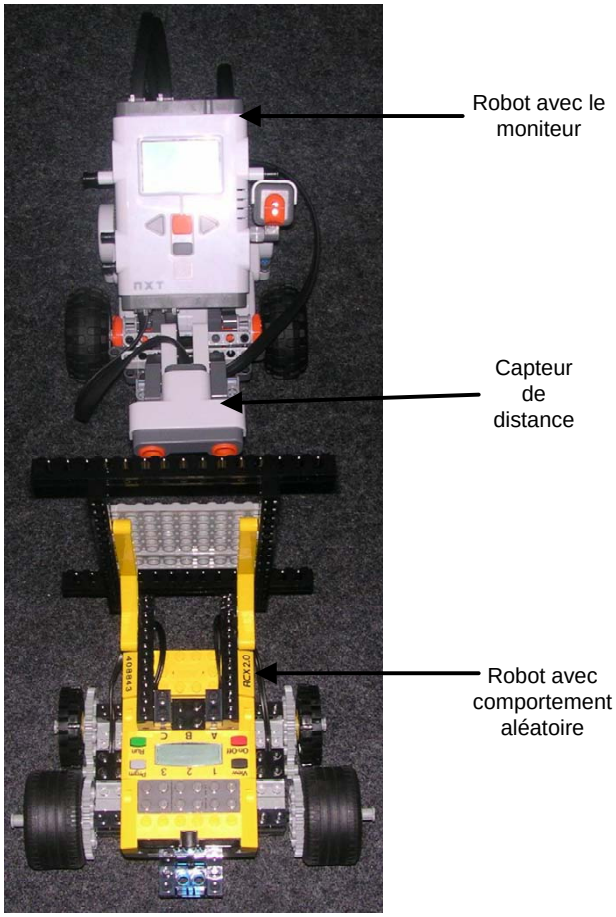



FIG. 5.12 – Les robots

La qualité de service de vitesse est violée quand le robot prend suffisamment de vitesse et que la puissance de freinage est faible.

La qualité de service pour la distance est violée quand le premier robot ralentit brusquement. La notification à l'utilisateur intervient dès le dépassement des trois secondes autorisées.

### 5.4.2 Moniteur issu de la réalisation

Le code du moniteur issu de la réalisation est chargé avec le code du véhicule sur le robot puis les deux robots sont démarrés. Comme pour les moniteurs discrets, la qualité de service pour l'échantillonnage de la vitesse et de la distance n'est jamais violée.

Comme pour le cas précédant, la qualité de service de vitesse est violée quand le robot prend suffisamment de vitesse.

La qualité de service pour la distance est violée quand le premier robot ralentit brusquement. La notification intervient dès le dépassement des trois secondes autorisées.

## 5.5 Conclusion

La validation de l'approche sur un exemple permet d'utiliser notre démarche sur une application concrète. L'utilisation de *Thot*a permis de développer les fonctionnalités sans avoir à s'occuper de la qualité de service temporelle.

Au niveau de la spécification, l'utilisation des motifs permet de pouvoir étudier plusieurs configurations de la qualité de service sans avoir à modifier manuellement les automates temporisés. Une modification du comportement du robot n'influence pas les motifs utilisés si l'alphabet de l'automate reste inchangé.

Au niveau du comportement, la génération de l'ensemble des fichiers de notification permet de se focaliser sur le développement. Pour le moniteur issu de la réalisation, $44$ fichiers sont nécessaires pour la notification des messages et des erreurs. La génération réduit le temps de développement et le risque d'erreur dans l'écriture de ces fichiers. Les appels de notifications sont les seules ajouts par rapport à un développement sans moniteur de qualité de service.

Au niveau de l'exécution, les moniteurs permettent de savoir si la qualité de service temporelle respecte la spécification de l'application. Une politique de gestion de la qualité de service pour les robots pourrait influer sur la puissance du freinage et de l'accélération en réponse à une violation de la qualité de service.

# Conclusion

Les travaux de cette thèse ont pour contexte la gestion de la qualité de service temporelle pendant le développement d'applications à composants. La qualité de service temporelle doit permettre d'estimer la qualité du logiciel aussi bien lors de sa spécification que lors de son exécution afin d'augemnter la fiabilité du logiciel. L'expression des propriétés temporelles doit être réutilisable dans le développement de différents logiciels de façon transparente. Nous avons défini un processus de gestion de la qualité de service temporelle pour le développement d'application à composants.

La vérification de la qualité de service s'effectue *a priori* lors de la spécification de l'application afin d'avoir une indication sur le niveau de la qualité de service lors de l'exécution. La vérification lors de l'exécution requiert une instrumentation spécifique. Cette instrumentation peut être produite manuellement mais est une source d'erreurs supplémentaire. Une instrumentation automatisée permet d'obtenir un outil à partir de la spécification.

Nous nous sommes intéressés dans un premier temps à l'introduction de la qualité de service de temps dans les applications à composants. Nous avons définis un ensemble de motifs temporels correspondant aux propriétés temporelles les plus courantes comme la durée ou la période. Nous avons identifié à quels endroits ces motifs doivent être ajoutés afin d'être utilisés pendant la spécification de l'application. Les motifs sont ajoutés dans le comportement du composant correspondant à ce que le composant offre et dans les contrats correspondant à ce que le composant requiert. Ainsi les informations de qualité de service de temps pourront être vérifiées lors de la composition des composant. L'utilisation de motifs temporels permet une séparation des préoccupations. Les motifs temporels ne sont pas spécifiques au logiciel développé et peuvent donc être réutilisés.

Dans un second temps, nous avons défini une méthode pour fournir une instrumentation pour la vérification de la qualité de service à l'exécution grâce à un processus fondé sur l'ingénierie des modèles. Cette instrumentation est générée à partir de la spécification de l'application afin de ne pas perdre ce qui avait été spécifié. La génération est effectuée à partir du comportement temporisé de l'application. Deux algorithmes de génération ont été developpés. Le premier permet d'obtenir une instrumentation correspondant au comportement attendu unité de temps par unité de temps. Le second est une réalisation de l'automate temporisé. Comme pour les motifs, la génération du moniteur de qualité de service n'est pas spécifique à un logiciel et est réutilisable.

## Perspectives

Les motifs définis dans cette thèse ne représentent pas l'ensemble des propriétés temporelles pouvant être utiles dans le développement d'un logiciel. Ils ne prennent pas en compte des propriétés portant sur plus de deux services. Des motifs exprimant un enchaînement ou une répitition de plusieurs messages pourraient compléter l'ensemble définis. L'ajout d'un motif temporel sera lié à l'ajout d'un contrat correspondant si il concerne une interaction avec d'autres composants.

L'application des motifs a été définie pour obtenir un comportement décrit par un automate temporisé. Une extension à d'autres formalismes temporisés, comme les réseaux de Petri temporels ou temporisés ou les automates hybrides, permettrait de ne pas restreindre le comportement de l'application à un automate à entrée/sortie. Les réseaux de Petri offrent par exemple une meilleure représentation des processus parallèles. Les automates hybrides permettent une représentation de temps continu et discret. Les contrats temporels devront être étendus pour être compatibles avec les formalismes utilisés.

La génération du moniteur de qualité de service ne prend en entrée que les automates temporisés. Le moniteur pourrait être obtenu à partir d'autres formalismes comme les réseaux de Petri ou les automates hybrides. Les algorithmes de transformation devront être intégrés à *Thot* ainsi que les grammaires Sintaks permettant la création de modèles.

Dans la version actuelle de la génération, les appels de notification doivent être ajoutés manuellement en utilisant l'alphabet des messages temporels. L'utilisation d'une approche par aspect pour intégrer ses appels permettrait d'automatiser complètement la génération du moniteur. La seule charge restant au développeur étant d'indiquer la correspondance entre l'alphabet temporel et les fonctions correspondantes.

La qualité de service n'est pas uniquement composée du temps mais peut aussi concerner la consommation de ressources ou la précision des fonctionnalités. Une transposition de l'approche à d'autres domaines que le temps, permettrait de surveiller plus de propriétés de qualité de service à l'exécution. Une extension des motifs permettrait de modéliser la consommation de ressources. Les contrats entre composants devront être capable de vérifier la consommation. Au niveau de la surveillance à l'exécution, le moniteur devra surveiller ces nouvelles propriétés. Giotto pourrait être remplacé par une plateforme spécifique pour la vérification de la qualité de service.

Les propriétés de qualité de service vérifiées sont des propriétés simples. Des propriétés probabilistes ou stochastiques. Ces propriétés devront être vérifiées lors de la spécification. Le moniteur devra estimer la future qualité de service du logiciel afin de prévenir l'utilisateur d'une possible mauvaise qualité de service future.

Le moniteur qe qualité de service effectue une notification simple lors de la violation du comportement attendu. Afin de compléter le moniteur, un processus de négociation de contrats pourrait être ajouter. Une première possibilité serait de pouvoir regénérer un nouveau moniteur de qualité de service ou de prévoir les changements possibles dès les spécifications. Une seconde possibilité serait d'utiliser des systèmes de négociation de contrats, comme ConFract, pour effectuer cette phase de négociation.

Enfin, une transposition de l'approche à d'autres types d'application que les composants est possible. Les applications orientées service ou la plateforme $.NET$ sont des pistes possible.

$.NET$ a déjà un mécanisme d'instrumentation permettant la vérification de propriétés utilisée par l'approche Chan/Poernomo. L'extension du comportement avec les réseaux de Petri permettrait l'utilisation des diagrammes d'activité d'UML 2 qui sont utilisés dans le domaine des applications orientées services.

# Bibliographie


[Aag01] J. O. Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, Department for Informatics, University of Oslo, June 2001.

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1) :2–34, 1993.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2) :183–235, 1994.

[AFP+02] Tobias Amnell, Elena Fersman, Paul Pettersson, Wang Yi, and Hongyan Sun. Code synthesis for timed automata. *Nordic J. of Computing*, 9(4) :269–300, 2002.

[AHH93] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. In *IEEE Real-Time Systems Symposium*, pages 2–11, 1993.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *SEFM '06 : Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[BCL+04] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J-B Stefani. An Open Component Model and Its Support in Java. In *ICSE'2004 - CBSE7*. Springer - LNCS, 2004.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos : A model-checking tool for real-time systems. In *Proc. 1998 Computer-Aided Verification, CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.

[BJPW99] A. Beugnard, J-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7) :38–45, 1999.

[BO92] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4) :355–398, 1992.

[Bou03] P. Bouyer. Untameable timed automata ! In *STACS '03 : Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science*, pages 620–631, London, UK, 2003. Springer-Verlag.

[BW96] A. W. Brown and K. C. Wallnan. Engineering of component-based systems. In *ICECCS '96 : Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96)*, page 414, Washington, DC, USA, 1996. IEEE Computer Society. *ISBN : 0-8186-7614-0*.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, 1986.

[CM98] C. Consel and R. Marlet. Architecturing software using a methodology for language development. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming*, volume 1490 of *LNCS*, pages 170–194, Pisa, Italy, septembre 1998.

[CPSJ05] K. Chan, I. Poernomo, H. W. Schmidt, and J. Jayaputera. A model-oriented framework for runtime monitoring of nonfunctional properties. In *QoSA/SOQUA*, pages 38–52, 2005.

[CRCR05] P. Collet, R. Rousseau, T. Coupaye, and N. Rivierre. A Contracting System for Hierarchical Components. In *ICSE'2005 - CBSE8*. Springer - LNCS, 2005.

[Daw98] Conrado Daws. Optikron : A tool suite for enhancing model-checking of real-time systems. In *Computer Aided Verification*, pages 542–545, 1998.

[DY96] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *RTSS '96 : Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, page 73, Washington, DC, USA, 1996. IEEE Computer Society.

[EFB+05] J. Estublier, J-M. Favre, J. Bézivin, L. Duchien, R. Marvie, S. Gérard, B. Baudry M. Bouzhegoub, J-M. Jézéquel, M. Blay, and M. Riveil. Action Spécifique CNRS sur l'Ingénierie Dirigée par les Modèles. Rapport de synthèse 1.1.2, CNRS, janvier 2005.

[EMO05] Meta-Object Facility (MOF) Specification, 2005. Version 2.0.

[FEBF06] Jean-Marie Favre, Jacky Establier, and Mireille Blay-Fornarino, editors. *L'ingénierie dirigée par les modèles : au-delà du MDA*. Hermes-Lavoisier, Cachan, France, February 2006.

[Fer] Jean-Claude Fernandez. Aldébaran : A tool for verification of communicating processes.

[Flo67] R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical aspects of computer science : Proc. American Mathematics Soc. symposia*, volume 19, pages 19–31, Providence RI, 1967. American Mathematical Society.

[GO03] S. Graf and I. Ober. A real-time profile for UML and how to adapt it to SDL. In *SDL Forum 2003, July 1-4, Stuttgart*, volume 2708 of *LNCS*, July 2003.

[GS93] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–40. World Scientific Publishing Company, 1993.

[GSC+04] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi. *Software Factories : Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley ; 1st edition, 2004. *ISBN : 0471202843*.

[HC01] G. Heineman and W. Councill, editors. *Component-Based Software Engineering, Putting the Pieces Together*. Addison-Westley, 2001. *ISBN : 0-201-70485-4*.

[HKH03] T.A. Henzinger, C.M. Kirsch, and B. Horowitz. Giotto : A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1) :84–99, January 2003.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, octobre 1969.

[Iec95] Iso Iec. Open distributed processing- reference model - part 2 : Foundations international standard 10746-2 itu-t recommendation x.902, 1995.

[KB05] S. Konrad and B.H.C.Cheng. Real-time specification patterns. In *ICSE27*, pages 372–381, May 2005.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4) :255–299, 1990.

[LLP+00] G. T. Leavens, K. Rustan M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML : notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106. ACM, octobre 2000.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2) :134–152, Octobre 1997.

[LT89] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3) :219–246, 1989.

[Mag99] J. Magee. Behavioral analysis of software architectures using ltsa. In *Proceedings of the 21st international conference on Software engineering*, pages 634–637. IEEE Computer Society Press, 1999. *ISBN : 1-58113-074-0*.

[Mey92a] B. Meyer. Applying "design by contract". *Computer*, 25(10), octobre 1992.

[Mey92b] B. Meyer. *Eiffel : the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. *ISBN : 0-13-247925-7*.

[MFF+06] Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, and Jean-Marc Jézéquel. Model-driven analysis and synthesis of concrete syntax. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 98–110. Springer, 2006.

[MFJ05] P-A. Muller, F. Fleurey, and J-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *MoDELS*, pages 264–278, 2005.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[MRK+97] L. E. Moser, Y. S. Ramakrishna, G. Kutty, P. M. Melliar-Smith, and L. K. Dillon. A graphical environment for the design of concurrent real-time systems. *ACM Transactions on Software Engineering and Methodology*, 6(1) :31–79, 1997.

[omg03] *UML 1.5 Object Constraint Language Specification*, mars 2003. Version 1.5.

[Rua84] L. M. Ruane. Abstract data types in assembly language programming. *ACM SIGPLAN Notices*, 19(1) :63–67, janvier 1984.

[SBLP07] Sébastien Saudrais, Olivier Barais, Duchien Laurence, and Noel Plouzeau. From formal specifications to qos monitors. In *in Journal of Object Technology, vol. 6, no. 11, Special Issue on Advances in Quality of Service Management*, pages 7–24, December 2007.

[StOs00] R. Soley and the OMG staff. MDA Model-Driven Architecture, novembre 2000. Online presentation http ://www.omg.org/mda/presentations.htm.

[Szy98] C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[vDKV00] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages : An annotated bibliography. *ACM SIGPLAN Notices*, 35(6) :26–36, June 2000.

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In Dieter Hogrefe and Stefan Leue, editors, *Proc. of the 7*th *Int. Conf. on Formal Description Techniques*, pages 223–238. North–Holland, 1994.

# Table des figures

# Liste des algorithmes

# Publications

## Journaux internationaux

1.Sébastien Saudrais, Olivier Barais, Laurence Duchien et Noël Plouzeau, From formal specifications to quality of service monitors, Journal of Object Technology.

## Conférence internationales

1. Sébastien Saudrais, Noël Plouzeau et Olivier Barais, Integration of Time Issues into Component-Based Applications, In Component-Based Software Engineering (CBSE 07), Juillet 2007.

## Workshops internationaux

1. Sébastien Saudrais, Olivier Barais, et Laurence Duchien, Using model-driven engineering to generate qos monitors from a formal specification, In Proceedings of the Aquserm 2006, Hong Kong, Chine, Octobre 2006.

## Conférences francophones

1. Sébastien Saudrais, Olivier Barais et Noël Plouzeau, Composants avec Propriétés Temporelles, Dans Conférence Francophone sur les architectures logicielles (CAL'06), In Proceedings of the CAL 2006, Nantes, France, 2006. Hermès Sciences.

2. Sébastien Saudrais, Olivier Barais, Laurence Duchien et Noël Plouzeau, Intégration de Propriétés Temporelles dans des Applications à Base de Composants, Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL 07), Juin 2007.

## Résumé

La qualité de service temporelle permet à l'utilisateur d'une application à composants d'estimer les propriétés extra-fonctionnelles l'application. Afin d'obtenir une estimation précise de la qualité, les informations de qualité de service temporelle doivent être présentes durant le processus de développement de l'application, de la spécification jusqu'à l'implantation. La contribution de cette thèse est la définition d'un processus de gestion de la qualité de service temporelle pendant le développement d'une application à composants. Nous nous intéressons dans un premier temps à l'introduction des informations temporelles lors de la spécification de l'application. Nous utilisons une approche à base de motifs pour intégrer le temps dans la spécification de l'application. Dans un second temps, nous fournissons un moniteur de qualité de service temporelle à partir de la spécification. Le moniteur s'assure, pendant l'exécution, que l'application respecte la qualité de service temporelle requise par la spécification. Finalement, nous décrivons notre outil *Thot* implantant le processus de gestion de la qualité de service temporelle et nous l'utilisons sur un cas d'application.

## Abstract

Quality of service of time helps a component based application's user to estimate the extra-functional properties of the application. In order to have a precise valuation of the quality, time quality of service information must occur along the whole application's development process, from the specification to the implementation. The contribution of this work is the definition of a time quality of service process during a component based application. First, we describe timed information's introduction during the application's specification. We use a pattern based approach in order to integrate time in the application's specification. Second, a quality of service monitor is produced from the specification. The monitor checks that, during the execution, the application obeys the time quality of service required by the specification. Finally, we describe our tool *Thot* which implements the time quality of service process and we use it on a case study.