Construction Incrémentale de Spécifications de Systèmes Critiques intégrant des Procédures de Vérification

Hong-Viet Luong

École des mines d'Alès
LGI2P

École doctorale MITT
IRIT - UMR 5505

Construction Incrémentale de Spécifications de Systèmes Critiques intégrant des Procédures de Vérification

THÈSE

présentée et soutenue publiquement le 1-10-2010

pour l'obtention du

Doctorat de l'université Paul Sabatier – Toulouse 3

(spécialité informatique)

par

Hong-Viet LUONG

Composition du jury

Président :	Jean-Paul BODEVEIX	Professeur, Université de Toulouse 3
Rapporteurs :	Yves LE TRAON	Professeur, Université de Luxembourg
	Guy LEDUC	Professeur, Université de Liège
Examinateurs :	Madeleine FAUGÈRE	Ingénieur de recherche, Thalès
	Mireille BLAY - FORNARINO	Professeur, Université de Nice
Directeur de thèse :	Christian PERCEBOIS	Professeur, Université de Toulouse 3
Encadrants :	Anne-Lise COURBIS	Maître Assistante, École des mines d'Alès
	Thomas LAMBOLAIS	Maître Assistant, École des mines d'Alès

Mis en page avec la classe thloria.

Remerciements

Je sais qu'il est difficile de remercier les organisations ainsi que toutes les personnes qui m'ont aidé et soutenu pendant ces années de préparation de thèse, mais je vais essayer de faire de mon mieux.

Je remercie tout d'abord Christian Percebois, professeur à l'université de Toulouse 3, d'avoir accepté d'être mon directeur de thèse. Je le remercie également de m'avoir aidé tout au long de ces trois années, de nous avoir accueilli, moi-même et mes encadrants de proximité, régulièrement et chaleureusement à Toulouse, d'avoir pris le temps de suivre mes travaux et fait des remarques précieuses pour ma thèse.

Je remercie la direction de l'école des mines d'Alès, et son représentant, M. Dorison, ainsi que la direction du LGI2P, M. Vasquez, pour m'avoir accueilli pendant trois ans au sein de l'EMA en m'offrant une allocation de doctorant, me permettant ainsi de mener mes travaux de recherche dans les meilleures conditions.

J'adresse mes immenses remerciements à Anne-Lise Courbis, pour son encadrement. Je suis reconnaissant de sa disponibilité permanente et de son aide sympathique pendant mon séjour à Nîmes. Elle m'a apporté avec enthousiasme la rigueur nécessaire à la rédaction scientifique. Ses instructions sur le français et sur le style de rédaction ont été et restent toujours très utiles pour moi.

Mes profonds remerciements vont aussi à Thomas Lambolais qui m'a montré un domaine de recherche intéressant et m'a donné de son temps ainsi que des conseils précieux pendant toutes ces années. Les discussions avec lui ont été une source abondante d'idées intéressantes. Je tiens également à le remercier pour m'avoir intégré dans l'équipe d'enseignement de l'école des mines d'Alès.

Je remercie M. Yves Le Traon, Professeur de l'université de Luxembourg, d'avoir accepté de rapporter mes travaux de thèse et de participer au jury. C'est pour moi un honneur que M. Guy Leduc, Professeur de l'université de Liège, ait été rapporteur de cette thèse. Mes principales idées tout au long de mes travaux sont basées sur ses travaux de thèse.

Je tiens à remercier Iulian Ober d'avoir participé à nos discussions, donné ses conseils inestimables pour notre recherche. De grands remerciements sont destinés aux examinateurs : Madame Mireille Blay-Fornarino, de l'université de Nice, M. Jean-Paul Bodeveix, de l'université de Toulouse 3 et Madame Madeleine Faugère, ingénieur de recherche à Thalès.

Je remercie tout le personnel du site nimois de l'école des mines d'Alès, et plus particulièrement les membres du laboratoire LGI2P. Ils m'ont aidé pendant toutes ces années de thèse. Un grand remerciement à Annie, Daniel, Vincent, Sylvain, Christelle, pour avoir contribué, par leurs remarques constructives, à la mise au point de ma soutenance de thèse. Je remercie mes camarades thésards, Saber, Sylvain, Ouaël, Abdelhak, Jorge, Liêm, Thao, Dung, Hang, Yên,... et les élèves de l'école.

Enfin, je suis reconnaissant envers ma famille, mon amie Hiên-Anh et mes amis pour m'avoir encouragé pendant ces années qui ont été quelques fois difficiles du fait de mon éloignement.

À mes parents.

Résumé

Cette thèse porte sur l'aide à la construction de machines d'états UML de systèmes réactifs. Elle vise à définir un cadre théorique et pragmatique pour mettre en œuvre une approche incrémentale caractérisée par une succession de phases de construction, évaluation et correction de modèles. Ce cadre offre des moyens de vérifier si un nouveau modèle est conforme à ceux définis durant les étapes précédentes sans avoir à demander une description explicite des propriétés à vérifier. Afin de pouvoir analyser les machines d'états, nous leur associons une sémantique LTS ce qui nous a conduit à définir une procédure de transformation automatique de machines d'états en LTS. Dans un premier temps, nous avons défini et implanté des techniques de vérification de relations de conformité de LTS (red, ext, conf, et confrestr). Dans un second temps, nous nous sommes intéressés à la définition d'un cadre de construction incrémentale dans lequel plusieurs stratégies de développement peuvent être mises en œuvre en s'assurant que le modèle final élaboré sera une implantation conforme à la spécification initiale. Ces stratégies reposent sur des combinaisons de raffinements qui peuvent être de deux types : le raffinement vertical pour éliminer l'indéterminisme et ajouter des détails ; le raffinement horizontal pour ajouter de nouvelles fonctionnalités sans ajouter d'indéterminisme. Enfin, nous transposons la problématique de construction incrémentale d'une machine d'états à la construction d'architectures dont les composants sont des machines d'états. Des conditions sont définies pour assurer la conformité entre des architectures dans le cas de la substitution de composants.

Mots-clés: Construction incrémentale, machine d'états UML, relation de conformité, raffinement, extension, réduction, graphe d'acceptance, architecture de modèles comportementaux.

Abstract

This thesis focuses on supporting construction of UML state machines of reactive systems. It aims at developing a theoretic and pragmatic framework to implement an incremental approach characterized by a succession of construction, evaluation and correction of models. This framework provides the means to verify whether a new model is consistent with those defined in the previous steps without requiring an explicit description of the properties to be verified. To analyze the state machines, we associated with them a LTS semantics which led us to define a procedure for automatic transformation of state machines in LTS. Initially, we have defined and implemented verification technique of conformance relations on LTS (red, ext, conf and confrestr). In a second step, we have defined a framework for incremental construction in which several development strategies can be implemented ensuring that the final developed model will be an implementation consistent with the initial specification. These strategies are based on combination of refinements that may be of two types : vertical refinement to eliminate nondeterminism and add details, and the horizontal refinement to add new features without adding nondeterminism. Finally, we transpose the problem of incremental construction of state machines to the construction of architectures whose components are state machines. Conditions are defined to ensure conformance between architectures in the case of substitution of components.

Keywords: Incremental construction, UML state machine, conformance relation, refinement, extension, reduction, acceptance graph, architecture.

Table des matières

Introduction

Contexte

La conception est l'utilisation de principes scientifiques, d'informations techniques et de l'imagination dans la définition d'une structure, d'une machine ou d'un système, pour effectuer des fonctions préalablement spécifiées avec le maximum d'économie et d'efficacité [Jon70]. De nos jours, dans le domaine du génie logiciel, la conception est considérée comme une phase clé permettant de décrire de façon non-ambiguë et suffisamment complète le fonctionnement d'un système tout en répondant à des exigences non-fonctionnelles (fiabilité, disponibilité, robustesse, performance...). L'élaboration et l'analyse de modèles tiennent une place majeure dans le processus de conception. Elles permettent de détecter et de prévenir de nombreuses erreurs de conception. Nous nous situons dans ce contexte et plus précisément nous nous intéressons au développement de modèles comportementaux de systèmes critiques, exprimés sous forme de machines d'états UML.

Systèmes critiques

Il existe de nombreuses définitions du terme systèmes critiques [Rus94], mais de façon intuitive, on qualifie un système de critique si une défaillance peut conduire à des conséquences inacceptables en termes humains, économiques et environnementaux. L'exemple de l'incendie du vol 501, qui a provoqué la destruction de la fusée Ariane 5 dont le coût de développement est estimé à une valeur de 370 millions de dollars est le plus coûteux de l'histoire [Lio96]. La cause est une erreur logicielle dans réutilisation du système de navigation. Il existe de plus en plus de logiciels dans les systèmes critiques et leur processus de conception et de développement doivent être rigoureux et conduire à des modèles fiables.

Notre intérêt ne se limite pas à la conception de systèmes critiques et s'étend à une définition plus large de systèmes qualifiés de *réactifs*. Un système réactif est un système qui répond de façon répétitive à des entrées sollicitées par son environnement en émettant des sorties [HP85]. Autrement dit, un système réactif doit réagir à un environnement qui ne peut pas l'attendre, à la différence d'un système interactif qui communique de façon permanente avec son environnement chacun évoluant à sa propre vitesse [Hal98]. Les concepts impliqués sont donc le parallélisme, la communication et les aspects temporels. Les systèmes réactifs sont généralement modélisés comme des ensembles de processus concurrents.

UML

Depuis sa standardisation comme langage de modélisation unifié par l'organisation OMG (*Object Management Group*) en 1997, le langage UML (*Unified Modeling Lan-*

guage [OMG09]) est devenu une norme *de facto* de l'industrie. UML a été standardisé pour fusionner différents langages visuels de modélisation : la méthode Booch, OMT, les statecharts de Harel... Il a été conçu dans l'objectif d'être le plus général possible et se veut extensible afin de prendre en charge des contextes non prévus. La définition syntaxique et sémantique de ce langage est continûment développée. Il regroupe un ensemble de notations graphiques pour créer des modèles visuels de systèmes. UML est largement reconnu et utilisé pour des applications variées telles que la modélisation, la documentation, la génération de code et le test.

Motivations et objectifs

En cherchant à corriger les faiblesses d'UML, l'OMG souhaite en étendre son utilisation. Néanmoins, la productivité obtenue par la modélisation UML n'a pas encore répondu aux besoins des industriels car il manque des supports d'analyse, d'exploitation ou de vérification des modèles. À l'heure actuelle, les machines d'états UML sont difficiles à construire. Plusieurs raisons expliquent cette difficulté. Tout d'abord, la sémantique des machines d'états n'est pas suffisamment documentée et reste ambiguë. De plus, les machines d'états ne sont pas modulaires : on ne dispose pas d'opérateurs de construction modulaire comme il en existe dans les langages de programmation ou de processus. Bien que les machines d'états soient un sous-ensemble du langage UML dont l'un des objectifs est d'être un support du concept orienté objet, elles ne possèdent pas réellement les principales caractéristiques attendues des objets telles que l'héritage, la généricité ou l'encapsulation. De surcroît, il existe peu de méthodes d'aide à la construction des machines d'états UML, ce qui a pour conséquence qu'il existe peu d'outils commerciaux d'évaluation, de vérification et de génération de code à partir de machines d'états UML. De ce fait, les industriels ne trouvent pas encore la productivité escomptée dans l'utilisation des machines UML pour la conception de composants temps-réel ou critiques.

Notre objectif est d'aider à la construction de machines d'états UML, en fournissant des techniques et outils de détection d'erreurs pendant les phases de spécification et de conception des systèmes. Nous ciblons notre étude sur la détection d'erreurs d'interaction telles que les inter-blocages et le refus. Dans un premier temps, nous considérons le problème de l'aide à la construction de machines d'états dans le cadre de l'élaboration d'un composant unique en interaction avec son environnement. Un travail antérieur [Gou06] a étudié des méthodes de vérification et de validation de ces modèles. Ce travail a posé les bases d'une démarche de *construction incrémentale* de machines d'états UML. Il a mis en évidence la pertinence de l'utilisation de relations de *conformité*, préalablement définies pour le test, pour assurer la cohérence des modèles obtenus à différentes étapes des processus de construction. Ce travail reste une contribution théorique et aucune mise en œuvre pratique n'a été proposée. L'objet de notre travail est de proposer des définitions plus précises de *raffinement incrémental* et d'en réaliser une mise en œuvre.

Problématique

Approche incrémentale

Nous souhaitons définir des techniques de construction incrémentale de machines d'états UML intégrant des vérifications formelles. Dans les méthodes formelles, et pour certains langages formels particulièrement, par exemple Z et B, la démarche de construction et la vérification des modèles par raffinements successifs sont reconnues. Plusieurs applications industrielles, en particulier dans le secteur ferroviaire, ont été réalisées [BDM98]. Le raffinement en B consiste à développer des spécifications en plusieurs étapes : à partir du cahier des charges, une première spécification formelle abstraite et globale est élaborée. Celle-ci doit couvrir l'ensemble du système, mais sans donner de détails de réalisation. Cette spécification est alors raffinée de proche en proche, de manière à ajouter des détails et à déterminer les fonctions souhaitées tout en assurant que le modèle raffiné reste conforme au modèle initial.

L'idée de la démarche de construction incrémentale que nous proposons s'inspire à la fois du raffinement en B, mais également d'approches pragmatiques telles que le *prototypage rapide* cherchant à obtenir rapidement une version réduite d'un système satisfaisant un ensemble minimum d'exigences. Cette version est ensuite enrichie progressivement pour répondre à des besoins plus complets. La démarche incrémentale doit ainsi intégrer des notions de *raffinement* et des notions d'*extension*. La construction incrémentale vise à être une démarche d'élaboration de spécifications par étapes successives et itératives permettant, à partir d'une spécification incomplète, d'obtenir une spécification complète et d'un niveau de détail estimé suffisant pour la phase de réalisation. Cette démarche doit assurer que tout modèle établi à une étape donnée est cohérent avec le modèle de l'étape précédente. L'intérêt de la construction incrémentale est double :

- d'un point de vue pratique, il permet d'avoir rapidement des versions opérationnelles du système en contrôlant les coûts et temps de développement ;
- d'un point de vue conceptuel, il permet de mettre en œuvre un raisonnement communément admis pour résoudre un problème de modélisation complexe. Ce raisonnement fait appel à des concept d'abstraction, de décomposition, d'enrichissement et d'élimination d'incohérences.

Les vérifications menées durant le développement cherchent alors à s'assurer que lors de l'extension des modèles, les comportements et fonctionnalités définis dans les étapes précédentes sont préservés. Certaines approches consistent à vérifier des modèles comparativement à des propriétés définies *in extenso* (par exemple sous forme de pré ou post conditions). Cela implique de formaliser explicitement les propriétés à satisfaire, ce que nous voulons éviter, car la formalisation de propriétés nécessite l'utilisation des langages tels que la logique temporelle difficiles à maîtriser. Nous voulons proposer des techniques de comparaison globale de modèles. Nous sommes donc à la recherche de relations traduisant à la fois le raffinement et l'extension de machines d'états.

Pour définir ces relations, nous devons associer aux machines d'états une sémantique non ambiguë. Nous choisissons les LTS (*Labelled Transition Systems*) car ils sont appropriés à la modélisation de l'interaction de processus et de nombreux travaux d'analyse des LTS ont été développés. En se basant sur cette sémantique, nous proposons une transformation automatique des machines d'états en LTS.

Relations de comparaison utilisées pour l'approche incrémentale

Après l'analyse de la problématique, nous avons étudié les relations de conformité proposées par Brinksma [BS86] sur les LTS telles que la réduction (red), l'extension (ext) et la conformité (conf). Une étude approfondie a montré que ces relations sont intéressantes pour les approches de raffinement et de construction incrémentale. Cependant, leur calculabilité n'a jamais été établie. Nous montrons que ces relations sont calculables grâce à la transformation des LTS en graphes d'acceptance. Les résultats obtenus sont illustrés sur un cas d'étude formalisé à la fois en LTS et en UML. Nous explorons une nouvelle piste sur des relations plus appropriées pour le raffinement. Grâce à ces relations, nous proposons des solutions pour la construction incrémentale de machine d'états UML, consistant à suivre une démarche itérative de construction, d'évaluation et de correction.

De plus, nous définissons de nouveaux algorithmes d'implantation des relations de raffinement confrestr, red* et ct. Ces relations s'avèrent appropriées au raffinement incrémental. Ceci nous conduit à formaliser un cadre de construction incrémentale intégrant les relations de raffinement selon deux aspects : développement de fonctionnalités et développement de détail.

Plan du mémoire

- **Le chapitre 1** est consacré à l'étude des approches formelles utilisées pour élaborer des spécifications. Après l'analyse des avantages et des inconvénients de ces méthodes, nous introduisons notre approche de construction incrémentale. Afin de la mettre en œuvre, nous devons définir des relations de comparaison. Cela nous conduit à étudier des relations formelles existantes qui servent à la vérification formelle. Une étude approfondie des relations nous permet de mettre en évidence les relations appropriées pour l'approche incrémentale.
- **Le chapitre 2** traite d'un autre aspect de notre étude : la modélisation des machines d'états UML. Nous fixons une sémantique d'exécution qui servira de base à leur traduction en LTS. Nous étudions également les travaux existants sur le raffinement dans l'approche objet ainsi que sur la vérification des machines d'états.
- Dans **le chapitre 3**, nous démontrons la calculabilité des relations d'extension, de réduction et de conformité sur des systèmes de transitions étiquetées. Nous présentons l'algorithme, l'évaluation et également la méthode d'amélioration de calcul.
- **Le chapitre 4** illustre par une étude de cas l'utilisation des relations de conformité dans le cadre d'une démarche incrémentale. Dans ce chapitre, nous présentons les règles de transformation des machines d'états en LTS, le transformateur ainsi que le prototype IDCM développé pour analyser les relations de conformité.
- **Le chapitre 5** formalise un cadre de construction incrémentale permettant de mettre en œuvre plusieurs stratégies de raffinement basées sur deux axes de développement : l'un, assurant l'ajout de détails et l'autre, l'ajout de fonctionnalités. Nous proposons également des conditions sur les relations de raffinement permettant d'établir l'équivalence des stratégies. Cela nous conduit à définir de nouvelles relations de raffinement dont la calculabilité et l'implantation sont présentées dans ce chapitre.

– **Le chapitre 6** énonce certains problèmes liés à la construction incrémentale d'architectures. Nous étudions s'il est possible d'étendre les travaux développés pour un composant à un assemblage de composants. Nous illustrons les problèmes posés au travers d'un cas d'étude et nous donnons des pistes de solutions mettant en évidence des travaux futurs à mener.
– **La conclusion** fait un bilan de nos travaux et présente des perspectives de recherche.
– **L'annexe A** présente quelques algorithmes et preuves supplémentaires du chapitre 5.
– **L'annexe B** illustre des schémas de transformation de machines d'états en LTS.

Notations

Définitions formelles

Nous adoptons les notations propositionnelles et ensemblistes suivantes.

Notation	*signification*
$\{a, b, c, \ldots\}$	l'ensemble constitué des éléments $a, b, c, \ldots$
$\neg a$	négation d'une proposition a
$a \wedge b$	conjonction logique de deux propositions
$a \vee b$	disjonction logique de deux propositions définie par $\neg(\neg a \wedge \neg b)$
$a \Rightarrow b$	implication logique équivalente à $\neg a \vee b$
$a \Leftrightarrow b$	équivalence logique définie par $a \Rightarrow b \wedge b \Rightarrow a$
$\varnothing$	ensemble vide
$E_1 \cup E_2$	union des ensembles E_1 et E_2
$E_1 \cap E_2$	intersection des ensembles E_1 et E_2
$E_1 - E_2$	différence des ensembles E_1 et E_2
$E_1 \subseteq E_2$	inclusion de l'ensemble E_1 dans E_2
$E_1 \subset E_2$	inclusion stricte de l'ensemble E_1 dans E_2
$e \in E$	appartenance de e à l'ensemble E
$\{e \in E \mid P(e)\}$	ensemble des éléments e de E qui satisfont la propriété $P(e)$
$E_1 \times E_2$	produit cartésien des ensembles E_1 et E_2
E^*	ensemble des séquences ayant aucun ou plusieurs éléments de E
E^+	ensemble des séquences ayant un ou plusieurs éléments de E
2^E ou $\mathcal{P}(E)$	ensemble des sous-ensembles de E
ssi	si et seulement si
$\stackrel{def}{=}$	est défini par

Relations

Avant de détailler les définitions des relations d'ordre dans le chapitre suivant, on donne quelques définitions des relations de préordre, d'équivalence qui sont les concepts de base pour la théorie des ordres.

Définition. *Une relation sur un ensemble E est la donnée d'une partie $\mathcal{R}$ de $E \times E$. Pour indiquer qu'une paire (a, b) de $E \times E$ est dans cette partie $\mathcal{R}$, on utilisera l'une ou l'autre des notations suivantes :* $(a, b) \in \mathcal{R}$, $a \mathcal{R} b$, $\mathcal{R}(a, b)$.

Propriétés des relations

Une relation binaire $\mathcal{R}$ est :

- *réflexive* si $\forall a \in E,\ (a,a) \in \mathcal{R}$,
- *irréflexive* si $\forall a,b \in E,\ (a,b) \in \mathcal{R} \Rightarrow a \neq b$,
- *symétrique* si $\forall a,b \in E,\ (a,b) \in \mathcal{R} \Rightarrow\ (b,a) \in \mathcal{R}$,
- *antisymétrique* si $\forall a,b \in E,\ (a,b) \in \mathcal{R} \wedge (b,a) \in \mathcal{R} \Rightarrow\ a = b$,
- *asymétrique* si $\forall a,b \in E,\ (a,b) \in \mathcal{R} \Rightarrow \neg(b,a) \in \mathcal{R}$,
- *transitive* si $\forall a,b,c \in E,\ (a,b) \in \mathcal{R} \wedge (b,c) \in \mathcal{R} \Rightarrow\ (a,c) \in \mathcal{R}$

Définition. *Un contexte $C[.]$ est une expression avec un paramètre formel $[.]$. $C[a]$ est un contexte où toute occurrence de '.' est remplacée par a.*

Relations d'ordre

- La relation identité est notée Id : $\forall a,a' \in E\,(a,a') \in Id \Leftrightarrow a = a'$
- Une relation de préordre est une relation binaire réflexive et transitive.
- Une relation d'équivalence une relation symétrique, réflexive et transitive.
- Une relation de préordre $\mathcal{R}$ est une *pré-congruence* (préordre congruent) si : $\forall a,b \in E,$ on a $a\ \mathcal{R}\ b \Leftrightarrow C[a]\ \mathcal{R}\ C[b]$ pour tout contexte $C[.]$.
- Une relation d'équivalence $\mathcal{R}$ est une *congruence* si : $\forall a,b \in E,$ on a $a\ \mathcal{R}\ b \Leftrightarrow C[a]\ \mathcal{R}\ C[b]$ pour tout contexte $C[.]$.

Opérations sur les relations

- La relation inverse de $\mathcal{R}$ est notée $\mathcal{R}^{-1}$: $\mathcal{R}^{-1} \stackrel{def}{=} a\ \mathcal{R}^{-1}\ b \Leftrightarrow b\ \mathcal{R}\ a$
- $\mathcal{R}_1 \circ \mathcal{R}_2$ dénote la composition de $\mathcal{R}_1$ et $\mathcal{R}_2$, définie par :

$$\mathcal{R}_1 \circ \mathcal{R}_2 \stackrel{def}{=} \{(x_1,x_2) \in X \times X \mid \exists x \in X.\ x_1\ \mathcal{R}_1\ x \wedge x\ \mathcal{R}_2\ x_2\}$$

On dit qu'une relation $\mathcal{R}_1$ est plus forte qu'une relation $\mathcal{R}_2$ si $\mathcal{R}_1$ est contenue dans $\mathcal{R}_2$.

Première partie

État de l'art

Chapitre 1

Méthodes formelles et vérification comportementale

Sommaire

Les logiciels sont devenus de plus en plus complexes avec l'augmentation de puissance des ordinateurs. Cet accroissement de complexité et de taille s'accompagne souvent d'une augmentation du nombre d'erreurs.

Les méthodes formelles ont été proposées comme techniques mathématiques visant à réduire ces erreurs. Ce chapitre examine en particulier l'utilisation des méthodes formelles pour la spécification et la vérification de systèmes critiques.

Notre intérêt est le suivant : *existe-t-il des méthodes de spécification et de vérification comportementales qui peuvent aider à la construction des machines d'états UML ?* Dans ce chapitre, nous verrons dans les premiers paragraphes les objectifs et les avancées actuelles des méthodes formelles et des techniques de vérification des aspects dynamiques des systèmes. Nous identifions en particulier l'utilisation de relations de comparaison dans les langages de processus communicants : une vérification par comparaison vise à s'assurer que plusieurs processus partagent un ensemble de propriétés, sans avoir à définir et vérifier explicitement ces propriétés. Parmi ces relations, nous identifions celles qui sont appropriées à une démarche de construction incrémentale.

1.1 Méthodes formelles pour le développement de spécifications

En référence à l'idée d'un *langage idéal* tel que proposé par Frege à la fin du 19e siècle [GG91], une logique formelle permet, par manipulation de la *forme* seule du langage, de conduire des raisonnements visant à affirmer des propositions. La notion de langage formel s'oppose à celle de langue naturelle et vise à en supprimer toute ambiguïté. Par l'utilisation d'un texte formel, l'espoir est donc de pouvoir obtenir automatiquement (mécaniquement) un certain nombre d'analyses telles que la vérification de propriétés ou la détection d'incohérences.

Sous l'appellation *méthodes formelles*, on désigne des techniques informatiques de *spécification* et de *vérification* de systèmes logiciels ou électroniques. Elles sont investiguées depuis les années 70 [Hoa69, Mil71, Hoa78, QS82]. Dans le développement logiciel, l'application des méthodes formelles a été étudiée des étapes amont (ingénierie des exigences, spécification, conception) aux étapes aval (réalisation, phases de test).

Par *spécification*, on entend un énoncé du système à réaliser (ce que le système peut ou doit faire ainsi que ce qu'il ne doit pas faire), sans indiquer comment ce système fonctionne. Les modèles manipulés peuvent être décrits à différents niveaux d'abstraction et éventuellement dans différents langages. La *vérification* consiste à comparer un modèle formel à un autre. Il peut s'agir de vérifier des propriétés logiques (décrites dans une logique donnée) sur un autre modèle. Il peut s'agir également de comparer un modèle détaillé à un modèle plus abstrait, tous deux décrits dans le même langage.

De manière plus ambitieuse, la *validation* d'une description ou d'une solution, cherche à s'assurer que cette description ou cette solution sont « les bonnes », à savoir que la description décrit effectivement les exigences souhaitées, ou que la solution répond effectivement au problème posé. Il s'agit d'une comparaison d'un modèle (formel ou non) à des exigences ou un problème informels.

Parmi les techniques de vérification que nous souhaitons utiliser dans le cadre du développement de machines d'états UML, nous nous orientons en priorité :

- vers celles prenant en compte les concepts comportementaux ;
- vers celles n'utilisant qu'un seul langage de formalisation : en effet, s'agissant d'une aide à la spécification de machines d'états, nous ne souhaitons pas que le spécifieur doive décrire, dans un autre langage, des propriétés ou un autre modèle utile à la spécification de la machine d'états visée. Nous n'envisageons pas non plus d'extraire automatiquement d'une machine d'états plusieurs modèles de langages différents.

Les paragraphes suivants présentent plusieurs schémas possibles d'utilisation des méthodes formelles, avec les avantages et inconvénients qu'ils comportent. En premier lieu, nous situons quels intérêts principaux nous voyons aux méthodes formelles, vis-à-vis des objectifs globaux de sûreté de fonctionnement. En second, nous examinons deux « cadres » possibles d'application de techniques de modélisation et de vérification en évaluant leurs intérêts et inconvénients. Ceci permet de mettre en évidence les objectifs et intérêts principaux d'une démarche de *construction incrémentale*.

1.1.1 Intérêts et limites des méthodes formelles par rapports aux objectifs de sûreté de fonctionnement

Pour le développement des logiciels en général et de systèmes critiques en particulier, l'exigence d'un « bon » système pose plusieurs défis. Les pressions économiques des marchés en évolution rapide demandent des cycles courts. Ce sont souvent les premières phases (ingénierie des exigences et développement des spécifications) qui en souffrent. Les modèles amont sont alors souvent négligés, alors même que les erreurs non détectées dans les phases amont sont les plus tenaces et les plus coûteuses [Lam01]. Il apparaît également que les modèles de cahiers des charges et d'exigences sont parmi les plus difficiles à réaliser. Lamsweerde et Darimont estiment que la place de l'ingénierie des exigences devrait être de 10% dans le développement complet d'un système.

Intérêts des méthodes formelles

L'un des premiers intérêts des méthodes formelles consiste à accorder aux modèles préliminaires de spécification et de conception une place importante. L'effort même de modélisation et formalisation, sans encore prendre en compte les moyens d'analyse et d'automatisation qu'il offre, est déjà un apport considérable dans l'analyse et la compréhension du cahier des charges.

Afin de situer l'apport des méthodes formelles en terme de qualités obtenues sur le système, on peut se référer aux critères définis en *sûreté de fonctionnement*. Prenons par exemple le cadre défini par Laprie [ALRL04, LAB$^+$95], où la sûreté de fonctionnement est vue comme « la propriété d'un système permettant aux utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre ». Les attributs de sûreté de fonctionnement concernent les notions de :

- *Fiabilité*, comme étant la capacité d'un système à exécuter ses fonctions requises dans des conditions déterminées pour une période de temps déterminée. Ceci se mesure par exemple avec le temps moyen entre la mise en service et la première défaillance ;
- *Sécurité innocuité*, qui est l'absence de conséquences catastrophiques pour des utilisateurs et pour l'environnement ;
- *Disponibilité*, c'est le fait d'être prêt à l'utilisation et s'exprime par le pourcentage de temps où le service est assuré. Elle se mesure par exemple avec le temps moyen entre deux défaillances ;
- *Intégrité*, qui est l'absence d'altérations du système ;
- *Maintenabilité*, comme étant la capacité à subir des réparations et des évolutions techniques ;
- *Confidentialité*, qui concerne la non divulgation d'informations.

Il est certain qu'aucune méthode ne garantit toutes ces qualités. Les méthodes formelles contribuent essentiellement à l'amélioration des critères de fiabilité, de disponibilité, de sécurité-innocuité et d'intégrité. Elles visent à limiter les défauts de spécification et de conception, en apportant à la fois des techniques visant à construire de manière correcte *a priori*, et des techniques de vérification pour analyser des failles *a posteriori*.

Nous pouvons considérer le langage UML comme semi-formel, ainsi que nous le verrons par la suite (chapitre 2). Notre problématique étant d'apporter des aides à la construction

et à l'évaluation de machines d'états UML, les méthodes formelles susceptibles de se coupler aux machines d'états sont en premier lieu celles décrivant les aspects comportementaux, parallèles, voire temps-réel des systèmes [Hal90, AHR02, BFLW09]. Nous attendons deux apports des méthodes formelles :

1. permettre de détecter ou d'éviter un certain nombre d'erreurs de spécification et de conception. Les erreurs à détecter sont par exemple celles concernant l'impossibilité d'appeler un ou plusieurs services donnés, pouvant correspondre à un blocage général ou provisoire, ainsi qu'à des inter-blocages et *live-locks* entre différents composants parallèles ou avec l'environnement (risque de faille de fiabilité et de disponibilité) ; ou les erreurs d'accès concurrents à une ressource partagée (risque de faille d'intégrité).
2. apporter (ou consolider) une démarche et une méthode de construction. Nous présentons par la suite le cadre de la démarche de construction par raffinements successifs.

Nous constatons que peu d'environnements offrent des outils couvrant ces deux aspects.

Limites et inconvénients des méthodes formelles

Pourtant, les méthodes formelles sont souvent critiquées. D'une part, elles sont jugées difficiles pour des non-experts, tels que les clients. La difficulté de formalisation s'accompagne d'une difficulté de lecture et relecture pour les intervenants extérieurs. Ceci limite les échanges possibles entre clients et concepteurs pendant les phases initiales importantes qui décident du succès ou de l'échec d'un projet. De plus, les clients peuvent participer aux étapes de validation des spécifications.

D'autre part, ces méthodes ont un champ d'application limité et ne couvrent que certains aspects des systèmes, et de plus, n'offrent pas nécessairement une garantie totale sur ces aspects. D'un point de vue théorique, il existe une limitation d'indécidabilité faisant que plusieurs propriétés ne peuvent être vérifiées. D'un point de vue pratique, comme nous le montrons dans les paragraphes suivants, une activité de modélisation et de vérification formelles ne peut faire l'impasse sur une activité de *validation informelle* des modèles proposés. Cette validation (conduite par relectures, simulations informelles ou tests) est alors sujette aux failles. L'existence d'une preuve formelle sur les modèles peut laisser une illusion de correction, alors que cette preuve n'a pas d'intérêt si les modèles eux-mêmes ne sont des modèles fidèles de ce qu'ils sont censés représenter.

1.1.2 Usages de la vérification formelle dans le développement logiciel

Nous montrons ici deux schémas possibles d'application des méthodes formelles. Ces schémas sous-tendent différentes démarches de formalisation et de vérification. Afin de conserver autant que possible les avantages de chacune de ces approches, mais aussi de répondre à certaines de leurs limites, nous convergeons ensuite (paragraphe 1.1.3) vers le schéma que nous préconisons pour la suite des développements.

Approche de formalisation et de vérification « non intégrée »

La figure 1.1 présente un cadre général d'utilisation des techniques formelles de modélisation et de vérifications. La partie gauche (informelle) représente un cycle de développement classique, partant des besoins et formulant un cahier des charges puis une conception

pour aboutir à une réalisation. La réalisation s'exécute dans un environnement. La partie droite (formelle) représente les modèles formels correspondant aux étapes informelles : modèle de la spécification correspondant à une formalisation d'une partie du cahier des charges (S) ; modèle de la réalisation correspondant à une formalisation de la conception et de la réalisation (R), mais aussi modèle de l'environnement (E). Ces modèles formels couvrent au mieux une partie des descriptions informelles correspondantes. Tout comme plusieurs langages et techniques de développement peuvent être utilisés du côté informel, plusieurs langages peuvent être utilisés pour la partie formelle.

Les techniques de vérification formelle peuvent alors chercher à s'assurer que la réalisation R, dans son environnement E, répond aux exigences S. Ce qui pourrait s'écrire par :

$$(R \,||\, E) \models S$$

en notant ici $||$ un opérateur de composition parallèle et $\models$ une relation de satisfaction.

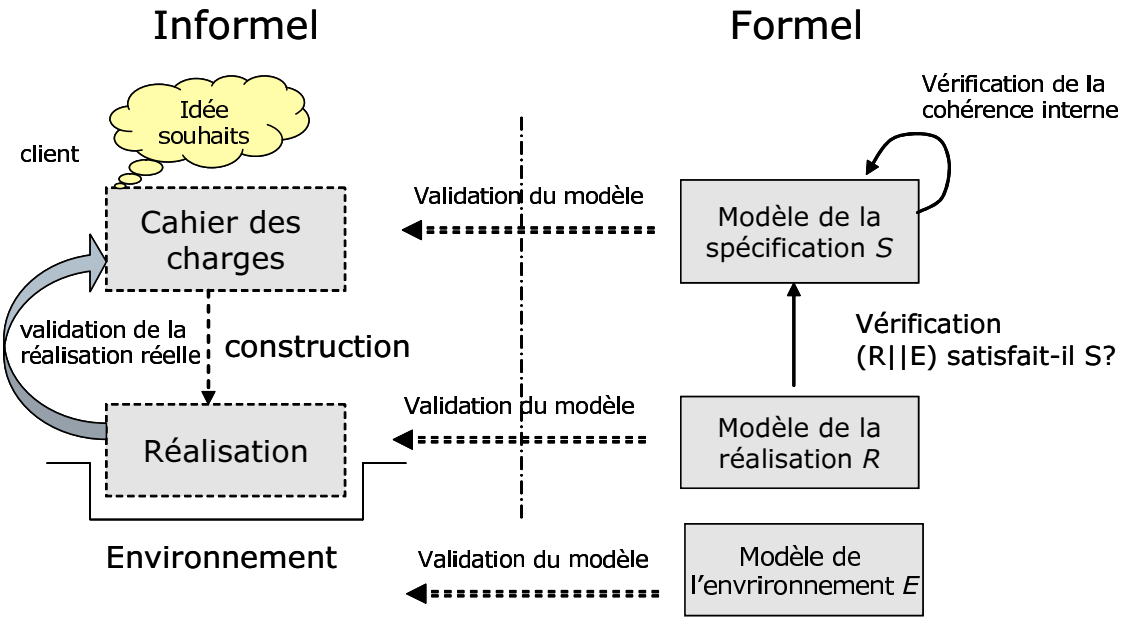


FIGURE 1.1 – Schéma d'application général des méthodes formelles pour la vérification [LG05] : approche non intégrée.

D'autres vérifications (internes) de S, R et E sont possibles. Par exemple, on peut s'assurer que les propriétés de la spécification S ne sont pas contradictoires, que la réalisation R ne comporte ni *deadlock* ni *livelock* ou encore qu'elle est *déterministe*.

Cette approche « non intégrée » présente plusieurs limites :

- les aspects formels ne couvrent en général qu'une partie des fonctionnalités et services visés ou développés dans le système réel. Le schéma ne le montre pas, mais l'ensemble de la partie de droite est en général beaucoup plus faible que la partie gauche (en termes de fonctionnalités couvertes) ;
- les modèles formels sont réalisés « en double » par rapport aux descriptions informelles de gauche. Ceci entraîne un sur-coût important, mais aussi multiplie les risques

d'incohérence entre les deux parties formelles et informelles ;

- si les modèles formels ne sont pas *validés* (informellement) par rapport au cahier de charges et à la réalisation, le travail de vérification devient inutile ou presque. Cette approche peut laisser l'illusion du caractère absolu d'une preuve formelle, alors que les modèles et/ou les propriétés vérifiées ne sont pas en accord avec le système modélisé ;
- les démarches de modélisation sont libres. Cette approche ne préconise pas de démarche particulière. Les modèles formels peuvent ainsi être réalisés avant ou après leurs correspondants réels. Mais les modèles de spécification peuvent également être réalisés après les modèles de réalisation ;
- la partie formelle de droite est en général multi-langage et multi-technique. Si ceci multiplie les possibilités de modélisation et de vérification, cela entraîne également des difficultés (risque d'incohérences et difficulté de maîtrise technique et de compréhension).

Les approches de raffinement répondent à certaines de ces limites. Elles intègrent les deux processus de développement (modèles formels et réalisation) en un seul, tout en proposant une démarche de développement. C'est le cas par exemple de la méthode B [Abr96].

Approche de formalisation et de vérification « intégrée » : démarche de raffinement

La notion de *raffinement* a été introduite dans les années 1970 par Wirth [Wir71], dans laquelle un programme est progressivement mis au point dans une séquence d'étapes de raffinement. Cette notion est ensuite abordée dans plusieurs travaux, par Dijkstra [Dij76], puis formalisée par Back [Bac78] dans les années 1980. Plusieurs auteurs ont ensuite développé cette notion, en particulier Abadi et Lamport [AL88] et Abrial [Abr96]. Le raffinement est un moyen de construire de manière progressive des programmes corrects. Le raffinement est vu comme le passage de la spécification (ce que fait le logiciel) au programme (comment fait le logiciel) tout en préservant la correction des propriétés validées par rapport aux spécifications intermédiaires.

D'après Gervais [GFL05], le raffinement est classifié sous plusieurs formes : le raffinement de séquences d'opérations (où les machines et les types de données abstraits sont raffinés) et le raffinement de transitions d'états en B événementiel.

Certaines propriétés du raffinement dans B telles que la *transitivité* et la *monotonie* sont des propriétés importantes et essentielles. La transitivité permet d'assurer que les spécifications intermédiaires ou finales possèdent les mêmes propriétés que la spécification initiale. Le raffinement en B [Abr96, GFL05] est basé sur trois aspects : raffinement de données, raffinement de contrôle et raffinement algorithmique. Le raffinement en Z [DB01, ACGW94], est centré sur le raffinement de données et d'opérations. Dans [ACGW94], Ainsworth et al ont présenté le concept de « co-raffinement » consistant à développer dans différents points de vue les données et les opérations.

La figure 1.2 illustre ainsi une démarche développement par raffinements successifs. On peut trouver ce type de raffinement explicitement dans certaines méthodes comme B ou Z [SD01], l'environnement Specware [SJ95], le langage CO-OPN/2 [Ser99] ou implicitement dans d'autres langages tels que Coq ou FoCaLize [BC04]. Dans ce cadre, à partir

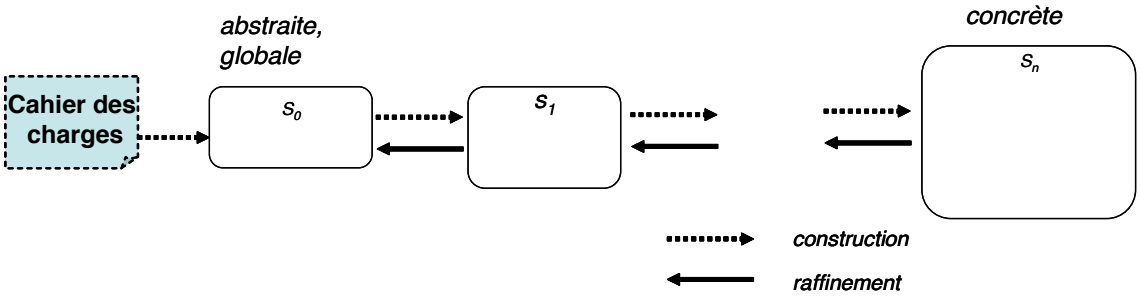


FIGURE 1.2 – Développement par raffinements successifs

du cahier des charges, on introduit une première spécification qui est très abstraite, en général indéterministe, mais qui doit couvrir l'ensemble du système sans donner de détail de réalisation. Cette spécification est alors raffinée de proche en proche, de manière à la détailler graduellement afin de s'orienter vers une implantation, tout en préservant les invariants de raffinement des étapes précédentes. La méthode B classique (par opposition au B événementiel), préserve ainsi les propriétés de sûreté au cours du développement. En effet, ces propriétés sont des invariants souhaités, de la forme « $\Box\neg p$ », c'est-à-dire « à tout instant de l'exécution du système, p est toujours faux ».

Cette démarche offre alors les intérêts et limites suivants :

- cette démarche progressive permet d'intégrer les aspects formels et informels dans un seul processus et permet de réduire le sur-coût des développements formels ;
- contrairement à l'approche précédente, celle-ci préconise une démarche particulière, imposant l'ordre d'élaboration des modèles. Ceci offre à la fois un guide de modélisation, en faisant abstraction de nombreux détails au début et en réduisant progressivement l'indéterminisme, mais facilite aussi les travaux de preuve en les réalisant sur des étapes intermédiaires ;
- le cadre formel assure une plus grande cohérence de langages : les raffinements successifs ainsi que les vérifications sont en général menés avec un seul formalisme, même s'il couvre de nombreux aspects comme dans le cas de la méthode B ;
- là aussi, comme pour l'approche non intégrée, seule une partie des fonctionnalités et services réels peut être développée par ce type d'approche ;
- le problème de la validation des modèles formels se concentre sur la spécification initiale. Il n'est plus utile de valider séparément les modèles intermédiaires. Seule reste à valider la spécification de départ. Par rapport à l'approche précédente, ce travail est donc facilité. Néanmoins, bien qu'il s'agisse d'une spécification globale et abstraite, le travail de validation reste délicat et source d'erreurs ;
- si la validation de la spécification initiale est facilitée, la construction de cette première spécification ne l'est pas forcément. Cette approche suppose que la spécification initiale est suffisamment large : il s'agit d'une représentation abstraite et globale de l'ensemble du système. Adopter un point de vue suffisamment général et abstrait pour couvrir toutes les fonctionnalités attendues du système, sans les détailler, n'est pas chose aisée ;

– par rapport à des objectifs industriels, cette démarche retarde la mise à disposition d'une version exécutable. Les spécifications intermédiaires ne correspondent pas à des versions exploitables et exécutables du futur système. Pour l'équipe de développement et surtout pour les clients, le fait de retarder la mise au point d'une première version n'est pas un élément rassurant. D'un point de vue économique, le fait de ne pouvoir exploiter commercialement des versions intermédiaires ne permet pas facilement de rentabiliser cette approche ;
– par son côté descendant, cette approche s'adapte assez mal à des changements dans les besoins des clients, exigeant de nombreuses mises à jour.

1.1.3 Vers un processus de développement incrémental

En s'inspirant des démarches de raffinement mais aussi de pratiques industrielles telles que le prototypage, nous proposons une démarche plus pragmatique, que nous qualifions de *démarche incrémentale*. À la différence du raffinement en B, la démarche incrémentale part d'une spécification partielle dans laquelle seules quelques fonctionnalités (supposées *essentielles*) sont prises en compte, constituant ainsi un *noyau* initial. Le développement consiste à mener conjointement le développement des spécifications successives avec celui des modèles des implantations. Chaque implantation doit être conforme à sa spécification et également à la spécification de l'étape précédente. Les spécifications intermédiaires obtenues à chaque étape décrivent de nouvelles fonctionnalités ou comportements mais elles ne doivent pas dégrader les spécifications des étapes précédentes. Dans ce cadre de développement : « *toute implantation du modèle raffiné doit être une implantation valide du modèle initial* ».

Rappelons que dans le processus de développement de spécifications, une *implantation* ne signifie pas une implantation physique ou un programme mais plutôt une spécification suffisamment détaillée pour être exploitable afin de réaliser le système [BS86]. La relation de « raffinement incrémental » doit être transitive pour assurer que la spécification finale préserve les propriétés de la spécification initiale.

À la différence d'une relation de raffinement *stricte* comme en B classique, on peut imaginer que la relation de raffinement incrémental ne préservera pas nécessairement les propriétés de sûreté pour les nouveaux comportements autorisés. En effet, s'il s'agit de comportements et de fonctionnalités non définis initialement, ils ne peuvent être comparés à une version de référence. Par exemple, l'ajout d'une procédure d'urgence pour ouvrir les portes d'un train n'a pas lieu de satisfaire un préalable invariant de sûreté demandant à ce que les portes soit fermées tant que le train est en mouvement. En revanche, la relation de raffinement incrémental doit préserver les propriétés de vivacité, de la forme « $p \Rightarrow \Diamond q$ » (si p est vrai, alors q devra être satisfait), garantissant que les fonctionnalités et comportements nécessaires d'une spécification doivent être préservés dans son raffinement incrémental. Un raffinement incrémental doit toujours être capable de faire (ou de garantir) ce que l'original doit faire.

Cette approche présente les avantages et limites suivants :

– comme le raffinement en B, cette démarche progressive permet d'intégrer à la fois des aspects formels et informels conduisant à réduire le sur-coût des développements formels ;

- cette approche offre non seulement les avantages de l'approche de raffinement tels que la démarche progressive, interactive et l'intégration d'un développement formel, mais elle permet aussi de dériver, de manière pratique plutôt que théorique cette fois, une version « exécutable » à chaque étape. Ces versions exécutables permettent d'avoir des retours du client dès le début du développement ; ceci par exemple afin d'identifier des problèmes au plus tôt.
- contrairement à l'approche de raffinement en B, cette démarche combine l'usage des langages formels (LOTOS, CCS, CSP...) et semi-formels tels que diagrammes comportementaux UML (machines d'états UML ...) ;
- le problème de la validation des modèles formels concerne non seulement la spécification initiale, mais il est aussi réparti tout au long des étapes d'élaboration des modèles ;
- à la différence de la démarche de raffinement, cette démarche s'applique sur une spécification partielle contenant des fonctionnalités jugées principales. La construction de cette première spécification est facilitée par rapport à celle du raffinement classique, mais l'identification des fonctionnalités essentielles n'est pas aisée ;
- contrairement au raffinement en B, cette démarche s'inspire du prototypage rapide consistant à mener conjointement le développement des spécifications successives avec celui des implantations. Cette démarche conduit à différents modèles de versions exécutables.
- contrairement à la démarche de raffinement, cette approche s'adapte à des changements (même imprévus) des besoins des clients, exigeant de nombreuses mises à jour.

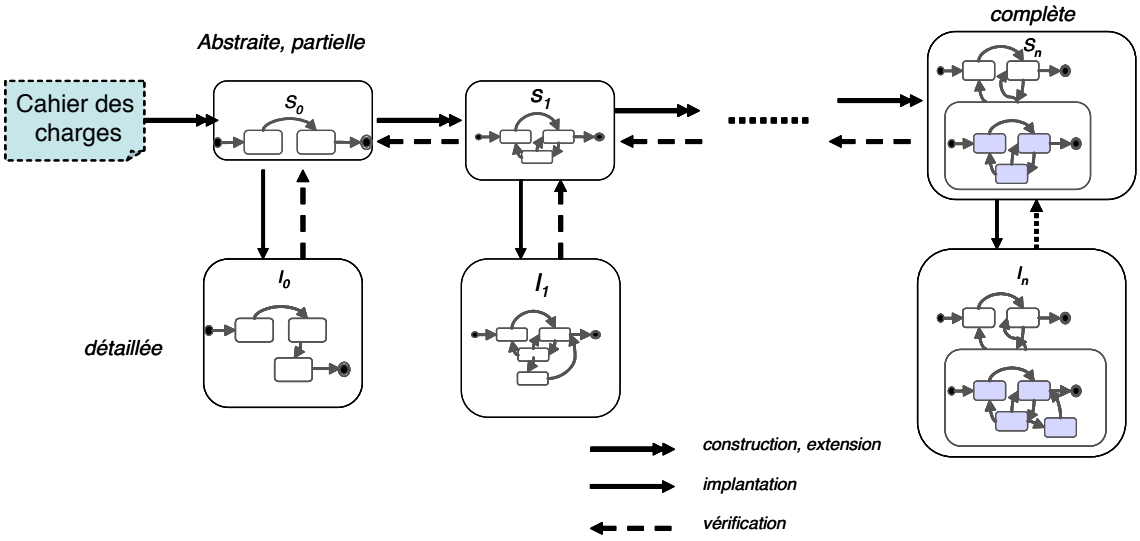


FIGURE 1.3 – Développement incrémental intégrant la vérification

Le processus de construction incrémentale fait lui-aussi appel à des relations d'implantation et de raffinement. Les relation d'implantation doivent être réflexives car une spécification est l'implantation valide d'elle-même. Des relations de raffinement doivent

être transitives pour assurer que la spécification finale préserve les propriétés de la spécification initiale. De manière analogue au raffinement classique, la technique de raffinement incrémental cherchée consiste elle-aussi à ajouter des détails concrets et à réduire l'*indéterminisme*.

Déterminisme. Nous considérons qu'un système ou un composant d'un système est déterministe si à tout instant, son comportement est entièrement déterminé par les *événements* qu'il perçoit ou qu'il a perçu en interaction avec son environnement. Autrement dit, un système est déterministe si à chaque fois qu'il peut exécuter plusieurs actions différentes, le choix entre ces actions est déterminé par son environnement. Cette définition est en accord avec les définitions de Hoare [Hoa78] et de Milner [Mil89] : « *si l'on effectue la même expérience deux fois sur un système déterminé, en commençant à chaque fois dans son état initial, alors nous nous attendons à obtenir le même résultat, ou comportement, à chaque fois* ». Cette formalisation est interprétée par De Nicola et Hennessy [NH83] comme : « *un processus est déterministe si et seulement s'il y a équivalence entre le fait qu'il* peut *réussir un test et le fait qu'il* doit *passer ce test* ».

Dans notre contexte, nous cherchons des relations de raffinement qui respectent les critères suivants :

i. réduction de l'indéterminisme ;

ii. ajout d'actions (ou de traces pour étendre des fonctionnalités) ;

iii. transitivité.

Les deux premières propriétés sont discutées dans le document de von der Beeck [vdB00] sur le développement des spécifications dans lequel le raffinement consiste en une réduction de l'indéterminisme et une réduction de la partialité. La réduction de la partialité signifie que l'on passe d'un système incomplet à un système plus complet. Trois relations sont analysées pour le raffinement : le raffinement de traces, le préordre d'échec [BHR84] et le préordre de spécification [CS90]. Aucune d'entre elles ne satisfait tous les critères. Il propose le double préordre de réduction qui satisfait la réduction de l'indéterminisme et la réduction de partialité. Toutefois ce préordre n'est pas adéquat dans notre contexte car il ne satisfait pas la propriété de raffinement (voir paragraphe.1.1.3).

Boiten et Bujorianu [BB03] ont exploré le raffinement au travers de l'unification des diagrammes d'états-transitions UML. Ils leur associent une sémantique Z et effectuent l'unification en définissant des exigences par des pré-conditions. Ils définissent que la spécification S_2 raffine la spécification S_1 comme suit :

$$\mathit{Refine}(S_1, S_2) \stackrel{def}{=} \forall P.\ \mathit{Conf}(P, S_2) \Rightarrow \mathit{Conf}(P, S_1) \tag{1.1}$$

où $\mathit{Conf}(P, S)$ représente la conformité entre un programme P et sa spécification S.

Cette définition rejoint notre compréhension du raffinement, disant que « *toute implantation du modèle raffiné doit être une implantation valide du modèle initial* ».

Pour aider la construction incrémentale des machines d'états UML, nous allons chercher des méthodes de vérification comportementale. Les machines d'états UML sont proches des LTS souvent utilisés comme modèle sémantique des algèbres de processus, dans lesquelles il existe plusieurs relations pour comparer deux modèles. Dans le cadre de notre travail, nous nous intéressons particulièrement à la vérification *a priori* utilisant ces relations.

1.2 Relations de comparaison dans des langages de processus

1.2.1 Préliminaires

Les LTS sont des descriptions d'automates simples et abstraites, ne précisant ni si les actions sont en entrée ou en sortie (il n'y a pas de distinction entre événements et actions), ni dans quelles circonstances une action est préférable à une autre (il n'y a pas de garde), ni combien de temps leur exécution demande ou après combien de temps leur exécution sera déclenchée. Ce sont des descriptions fortement indéterministes, à la fois d'un point de vue temporisé, mais aussi dans le sens où plusieurs exécutions d'une même séquence d'actions observables à partir de l'état initial peuvent conduire à des états différents. Certaines actions peuvent-être des synchronisations du processus avec son environnement, ou la réception d'un signal transmis par l'environnement. Ces actions ne peuvent se produire que si l'environnement coopère.

Soit $Act = \mathcal{L} \cup \{\tau\}$ l'ensemble de toutes les actions, où $\mathcal{L}$ est l'ensemble des actions observables et τ l'action interne. Soit $\mathcal{P}$ un ensemble de noms d'états.

Définition (Systèmes de transitions étiquetées [Mil99])**.** *Un LTS $\langle P, A, \rightarrow, p\rangle$ est un quadruplet formé :*

- *d'un ensemble non vide $P \subseteq \mathcal{P}$ d'états (ou processus),*
- *d'un ensemble $A \subseteq Act$ de noms d'actions ($A = L \cup \{\tau\}$, où $L \subseteq \mathcal{L}$ est l'ensemble des actions visibles du LTS)*
- *d'une relation de transitions $\rightarrow \subseteq P \times A \times P$*
- *d'un état initial $p \in P$.*

On désigne aussi par p tout LTS $\langle P, A, \rightarrow, p\rangle$. Soient p, p', q des LTS, $a, a_{i,0\leqslant i\leqslant n}$ des actions de Act et $\sigma \in \mathcal{L}^*$ une suite d'actions observables. Si $(p, a, p') \in \rightarrow$, on note $p \xrightarrow{a} p'$. Ceci peut s'interpréter en disant que le processus (ou le LTS) p peut exécuter a et passer de p à p', ou encore que p se transforme en p' en exécutant a.

Nous définissons :

$$
\begin{aligned}
p \xrightarrow{a_1 \cdots a_n} p' &\stackrel{def}{=} \exists p, \ldots, p_n.\ p = p \xrightarrow{a_1} \ldots \xrightarrow{a_n} p_n = p' \\
p \xrightarrow{a_1 \cdots a_n} &\stackrel{def}{=} \exists p'.\ p \xrightarrow{a_1 \cdots a_n} p' \\
p \stackrel{\varepsilon}{\Rightarrow} p' &\stackrel{def}{=} p = p' \text{ ou } p \xrightarrow{\tau \cdots \tau} p' \\
p \stackrel{a}{\Rightarrow} p' &\stackrel{def}{=} \exists p_1, p_2.\ p \stackrel{\varepsilon}{\Rightarrow} p_1 \xrightarrow{a} p_2 \stackrel{\varepsilon}{\Rightarrow} p' \\
p \xRightarrow{a_1 \cdots a_n} p' &\stackrel{def}{=} \exists p, \ldots, p_n.\ p = p \xRightarrow{a_1} \ldots \xRightarrow{a_n} p_n = p' \\
p \stackrel{\sigma}{\Rightarrow} &\stackrel{def}{=} \exists p'.\ p \stackrel{\sigma}{\Rightarrow} p'
\end{aligned}
$$

$$
\begin{aligned}
\text{Traces (observables)} : Tr(p) &\stackrel{def}{=} \{\sigma \in L^* \mid p \stackrel{\sigma}{\Rightarrow}\} \\
p \text{ after } \sigma &\stackrel{def}{=} \{p' \mid p \stackrel{\sigma}{\Rightarrow} p'\} \\
D(p,a) &\stackrel{def}{=} \{p' \mid p \stackrel{a}{\rightarrow} p'\} \\
Out(p) &\stackrel{def}{=} \{a \in L \mid p \stackrel{a}{\rightarrow}\} \\
Out(p,\sigma) &\stackrel{def}{=} \bigcup_{p' \in p \text{ after } \sigma} Out(p') \\
S(p) &\stackrel{def}{=} \{a \mid \exists p'.\ p \stackrel{a}{\rightarrow} p'\} \\
\mathsf{stable}\ (p) &\stackrel{def}{=} \nexists p'.\ p \stackrel{\tau}{\rightarrow} p'
\end{aligned}
$$

$D(p,a)$ est l'ensemble des états que l'on peut obtenir à partir de l'état p après avoir exécuté l'action a. $Out(p,\sigma)$ est l'ensemble des actions observables de p après la trace σ et $S(p)$ est l'ensemble des actions que p peut faire.

1.2.2 Sémantique des processus

Dans la littérature consacrée à la sémantique des processus communicants [vG90] et également de logique temporelle [EH86], on distingue le *temps linéaire* et le *temps arborescent*. En temps linéaire, les exécutions possibles d'un processus sont déterminées par ses traces. En temps arborescent, on prend en considération la structure interne de branchement des processus. Considérons par exemple pour deux processus p et q suivants (nous utilisons la syntaxe simplifiée de LOTOS) $p := a;b;stop \,[]\, a;c;stop$ et $q := a;(b;stop \,[]\, c;stop)$. En sémantique de temps linéaire, on ne peut pas les distinguer car ils ont les même traces, $\{ab, ac\}$ tandis qu'en sémantique de temps arborescent, on fait une distinction entre ces deux processus, grâce à deux structures arborescentes différentes [dBBKM83]. Deux équivalences connues représentant ces deux sémantiques sont respectivement l'équivalence de traces et l'équivalence observationnelle [Mil89]. La sémantique de temps arborescent joue un rôle très important dans la concurrence parce qu'elle permet de modéliser plus finement les risques de blocages tandis que cela est une faiblesse de la sémantique temps linéaire.

Van Glabbeek a étudié les sémantiques dans le cadre de l'algèbre de processus uniforme BCCSP qui ne contient que des opérateurs algébriques de base de CCS et CSP. Dans cette étude, il a classifié douze sémantiques connues définies en termes de relations d'actions [vG90]. Nous rappelons globalement cette classification (cf. figure 1.4) sans chercher à l'expliquer ici en détails, afin de situer les quatre sémantiques qui nous intéresseront par la suite. Dans son spectre temps linéaire–temps arborescent, on trouve comme sémantique la plus grossière (la plus faible), la sémantique de traces. La plus fine (la plus forte) est la sémantique de bisimulation, proposée par Milner [Mil89]. Les flèches présentent le chemin d'une sémantique plus fine vers une sémantique plus grossière. Cela signifie que toutes les relations (de préordre ou d'équivalence) de la sémantique plus fine (en haut) impliquent (ou sont plus fortes que) les relations de la sémantique plus grossière (en bas).

Dans ce chapitre, nous n'allons aborder que quatre sémantiques qui sont marquées par des ellipses en pointillés : bisimulation, simulation, échec et trace. Les relations concernant ces quatre relations sont présentées dans le paragraphe suivant.

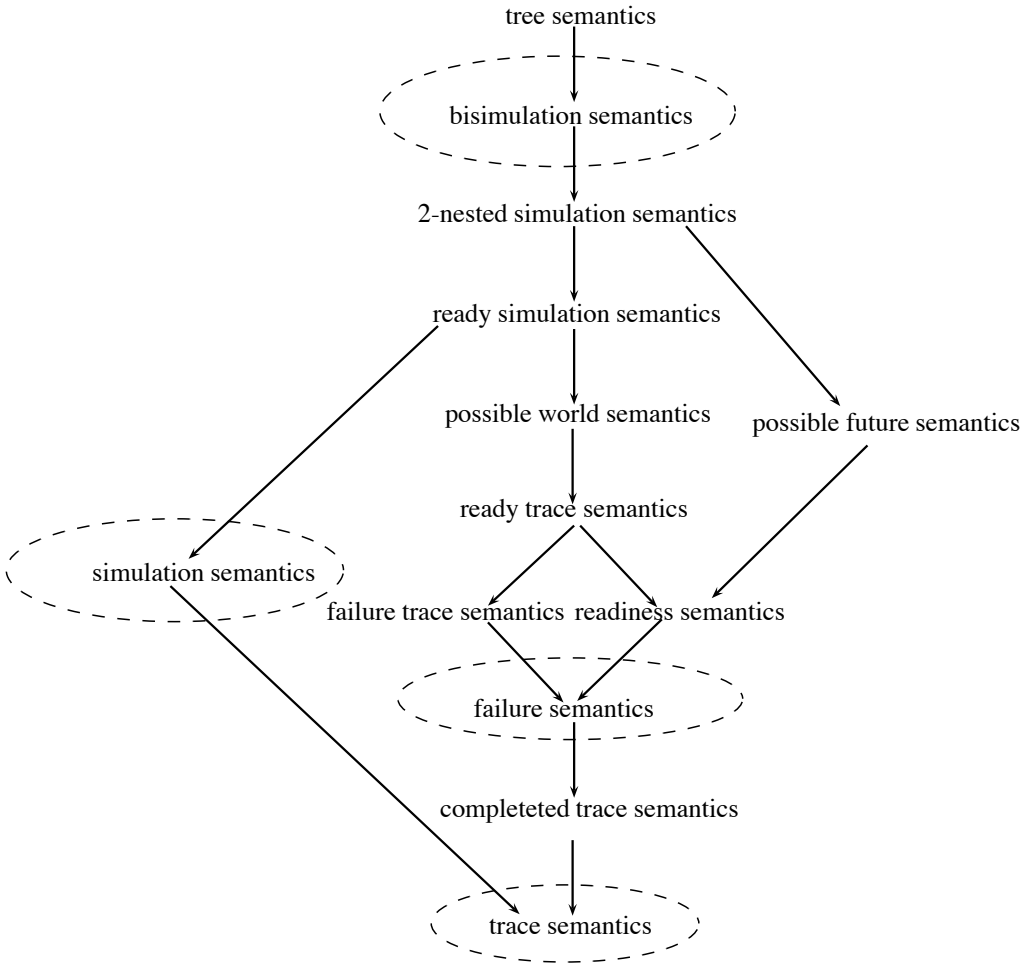


FIGURE 1.4 – Quatre sémantiques dans le spectre temps linéaire-arborescent de van Glabbeek [vG90]

1.2.3 Observabilité

En langages de concurrence tels que CCS, LOTOS, CSP, un système est considéré comme un processus p qui interagit avec son environnement. p peut être décrit par une composition de processus ou de manière monolithique. Son environnement peut lui-aussi être décrit par d'autres processus. On parlera *d'observateur* lorsqu'on se place au sein de l'environnement de p. Les interactions entre processus sont constituées d'unités atomiques appelées actions (en LOTOS, elles sont appelées portes). Un processus p peut évoluer en processus q, en exécutant l'action a ce qui se représente par $p \xrightarrow{a} q$. Une action a peut être observable ou non. Les interactions (communications ou synchronisations) entre processus ne peuvent se faire que sur des actions observables. Les actions internes d'un processus p sont toutes cachées à son environnement : dans le LTS correspondant à p, elles apparaissent toutes sous le même nom. En CCS, sa notation est τ, et i en LOTOS. Il n'y a pas de notation pour l'action silencieuse en CSP, mais ce langage possède un opérateur de choix indéterministe pouvant exprimer $p \sqcap q$ où p, q sont deux processus. Ce choix peut

être exprimé par : $\tau.p + \tau.q$

Le mécanisme abstraction est l'un des plus importants dans les algèbres de processus, car il nous fournit un mécanisme permettant de cacher les actions qui ne sont pas observables [vGW96]. Dans les langages tels que CCS, CSP, LOTOS, il existe un opérateur permettant de cacher des actions données d'un processus afin qu'elles soient invisibles ou silencieuses. En LOTOS, l'opérateur de masquage est « hide » et dans CCS ou CSP, la notation est $\backslash\{a_1, a_2 \ldots\}$. Dans ces trois langages de processus, l'opérateur de masquage vise à éviter à l'environnement d'interagir sur ces actions.

1.2.4 Relations de simulation, bisimulation forte et observationnelle

La notion de simulation proposée par Milner [Mil71] est la solution traditionnelle pour vérifier des systèmes concurrents.

Définition 1.2.1 (Simulation forte [Mil89])**.** *Une relation $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ est une simulation forte si $p\ \mathcal{S}\ q$ entraîne, $\forall a \in A$,*

pour tout p' tel que $p \xrightarrow{a} p'$, il existe q' tel que $q \xrightarrow{a} q'$ et $p'\ \mathcal{S}\ q'$.

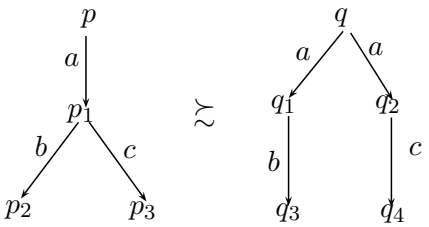


FIGURE 1.5 – p simule q

Premièrement, la simulation forte a une « sémantique totale abstraite » : q simule p si chaque arbre de calcul intégré dans le déroulement p peut également être intégré dans le déroulement de q [HKR02]. D'autre part, la simulation entre les processus a une caractérisation logique : q simule p si et seulement q et p satisfont le même ensemble de formules de logique temporelle arborescente CTL [GL94].

Nous considérons les deux machines de la figure 1.5. Dans cet exemple, p simule q et p est plus déterministe que q. Mais cette propriété n'est pas toujours garantie. L'exemple de la figure 1.7 montre que la préservation du déterminisme n'est pas satisfaite.

Définition 1.2.2 (Plus grande simulation)**.** *$p \precsim q$ s'il existe une simulation forte $\mathcal{R}$ telle que $p\mathcal{R}q$.*

Définition 1.2.3 (Bisimulation forte [Mil89])**.** *Une relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ est une bisimulation forte si $p\ \mathcal{R}\ q$ entraîne $\forall a \in A$*

- *Pour tout p' tel que $p \xrightarrow{a} p'$, il existe q', tel que $q \xrightarrow{a} q'$ et $p'\ \mathcal{R}\ q'$,*
- *Pour tout q' tel que $q \xrightarrow{a} q'$, il existe p', tel que $p \xrightarrow{a} p'$ et $p'\ \mathcal{R}\ q'$.*

Le concept de bisimulation a été défini à l'origine par David Park [Par81].

Définition 1.2.4 (Équivalence forte)**.** *p et q sont fortement équivalents, ce que l'on note $p \sim q$, s'il existe une bisimulation forte $\mathcal{R}$ telle que $p\,\mathcal{R}\,q$.*

Proposition 1.2.1. *$p \sim q$ ssi $\forall a \in A$:*

- *Pour tout p' tel que $p \xrightarrow{a} p'$, il existe q' tel que $q \xrightarrow{a} q'$ et $p' \sim q'$.*
- *Pour tout q' tel que $q \xrightarrow{a} q'$, il existe p' tel que $p \xrightarrow{a} p'$ et $p' \sim q'$.*

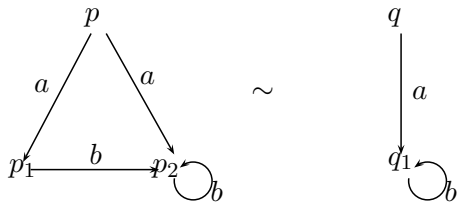


FIGURE 1.6 – $q \sim p$

L'exemple de la figure 1.6 présente une bisimulation forte entre deux processus q et p.

Remarque : $p \gtrsim q$ et $q \gtrsim p$ n'entraîne pas nécessairement $p \sim q$ (cf. figure 1.7). C'est-à-dire que si p simule q et q simule p, p et q ne sont pas forcément bisimilaires. Dans l'exemple 1.7, q présente un deadlock après la trace a tandis que p ne le présente pas. q et p ne sont pas équivalents.

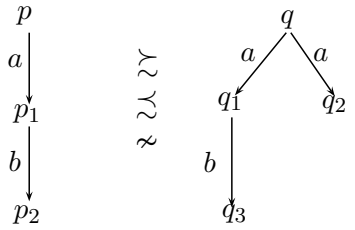


FIGURE 1.7 – $q \gtrsim p$ et $q \precsim p$ mais $q \nsim p$

Définition 1.2.5 (Simulation faible [Mil99])**.** *Une relation $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ est une simulation faible si $p\mathcal{S}q$ entraîne $\forall a \in A$*

pour tout p', $p \xrightarrow{a} p'$, il existe q' tel que $q \stackrel{a}{\Rightarrow} q'$ et $p'\ \mathcal{S}\ q'$

La simulation faible est similaire à la simulation forte, à la différence qu'elle prend en considération les mouvements silencieux des processus. Comme la simulation forte, la préservation du déterminisme n'est pas garantie.

Définition 1.2.6 (Bisimulation faible [Mil89])**.** *Une relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ est une bisimulation faible si $p\,\mathcal{R}\,q$ entraîne $\forall a \in A$*

- *Pour tout p' tel que $p \stackrel{a}{\Rightarrow} p'$, il existe q', tel que $q \stackrel{a}{\Rightarrow} q'$ et $p'\,\mathcal{R}\,q'$,*
- *Pour tout q' tel que $q \stackrel{a}{\Rightarrow} q'$, il existe p', tel que $p \stackrel{a}{\Rightarrow} p'$ et $p'\,\mathcal{R}\,q'$.*

Proposition 1.2.2. *$p \approx q$ ssi $\forall a \in A$:*

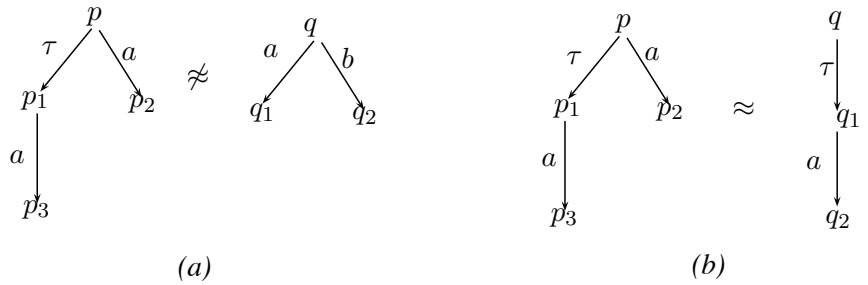


FIGURE 1.8 – (a) $q \not\approx p$ (b) $q \approx p$

– *Pour tout* p' *tel que* $p \stackrel{a}{\Rightarrow} p'$, *il existe* q' *tel que* $q \stackrel{a}{\Rightarrow} q'$ *et* $p' \approx q'$.
– *Pour tout* q' *tel que* $q \stackrel{a}{\Rightarrow} q'$, *il existe* p' *tel que* $p \stackrel{a}{\Rightarrow} p'$ *et* $p' \approx q'$.

Définition 1.2.7. *Traces (observables) :* $Tr(p) \stackrel{def}{=} \{\sigma \in L^* \mid p \stackrel{\sigma}{\Rightarrow}\}$.

Une trace est une séquence finie d'actions partielles et observables à partir de l'état initial.

Définition 1.2.8 (Équivalence de traces)**.** *Deux processus* q *et* p *sont en équivalence de traces si* $Tr(q) = Tr(p)$

Notons que la bisimulation forte implique l'équivalence de traces mais pas l'inverse comme montré dans le treillis de van Glabbeek (cf. figure 1.4). Deux processus sont équivalents au sens des traces si et seulement s'ils respectent les mêmes propriétés de logique temporelle linéaire (LTL) [HHK95].

1.2.5 Relations de conformité

Dans cette partie, nous nous intéressons à la relation de conformité. La notion de conformité est définie par Brinksma [BS86] selon l'idée suivante : « *Toute action pouvant être refusée par l'implantation doit également pouvoir être refusée par la spécification* ».

La relation conf a été proposée comme une relation d'implantation pour vérifier si un modèle d'une implantation respecte sa spécification [Led91a, Tre99]. C'est une formalisation de la notion de conformité utilisée en test de conformité. Elle traduit ainsi spécifiquement la réduction de l'indéterminisme. Mais elle ne garantit par l'ajout de traces. De plus, elle n'est pas transitive, ce qui fait qu'elle ne peut être candidate pour représenter un raffinement.

Avant de présenter en détail les définitions des relations de conformité, nous présentons les notions de base.

Définition 1.2.9 (Ensemble de refus)**.** *L'ensemble de refus de* p *après la trace* σ *est défini par :*

$$Ref(p,\sigma) \stackrel{def}{=} \left\{ X \mid \exists p \in P \ after \ \sigma.p \stackrel{e}{\not\Rightarrow}, \forall e \in X \right\}$$

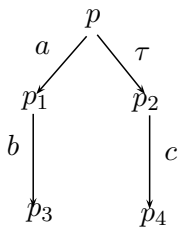


FIGURE 1.9 – Processus p

Les ensembles de refus de p décrit par la figure 1.9 après les traces ϵ, c, a et ab sont :

$$Ref(p,\epsilon) = \{\{a,b\},\{b\},\{a\},\varnothing\}$$
$$Ref(p,c) = \{\{a,b,c\},\{b,c\},\{a,c\},\{a,b\},\{a\},\{b\},\{c\},\varnothing\}$$
$$Ref(p,a) = \{\{a,c\},\{a\},\{c\},\varnothing\}$$
$$Ref(p,ab) = \{\{a,b,c\},\{b,c\},\{a,c\},\{a,b\},\{a\},\{b\},\{c\},\varnothing\}$$

La conformité d'un processus à sa spécification traduit le fait qu'un processus *doit* accepter tout ce que sa spécification doit accepter.

Définition 1.2.10 (Conformité [BS86])**.**

$$q \text{ conf } p \text{ si } \forall \sigma \in Tr(p).\ Ref(q,\sigma) \subseteq Ref(p,\sigma)$$

Nous retenons une définition de l'ensemble d'acceptance de p après une trace σ :

Définition 1.2.11 (Ensemble d'acceptance [Led91b])**.**

$$Acc_1(p,\sigma) \stackrel{def}{=} \left\{X \mid \exists p' \in p \text{ after } \sigma.\ X = Out(p',\varepsilon)\right\}$$

On peut également trouver un lien entre l'ensemble d'acceptance et l'ensemble de refus par la proposition suivante :

Proposition 1.2.3 ([Led91b])**.**

$$\forall \sigma \in Tr(p).\ Acc_1(q,\sigma) \subset\subset Acc_1(p,\sigma) \Leftrightarrow Ref(q,\sigma) \subseteq Ref(p,\sigma)$$

où on définit l'inclusion d'ensemble d'ensembles par :

Définition (Inclusion d'ensemble d'ensembles [Hen85])**.** *Soient* $A, B \subseteq 2^X$. $A \subset\subset B$ *si :*

$$\forall S \in A.\ \exists S' \in B.\ S' \subseteq S.$$

On peut donc trouver une autre définition de la relation de conformité :

Proposition 1.2.4 ([Led91b])**.**

$$q \text{ conf } p \text{ si } \forall \sigma \in Tr(p).\ Acc_1(q,\sigma) \subset\subset Acc_1(p,\sigma)$$

Les relations d'extension et de réduction [BS86, Led91a, Tre99] sont des relations de conformité combinées à des extensions ou des réductions de traces.

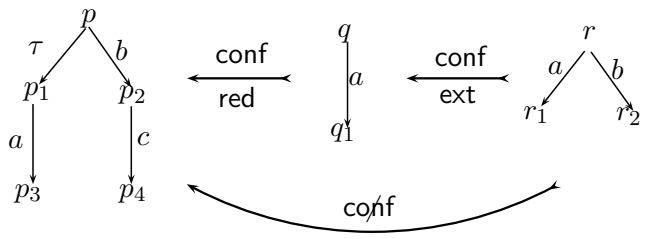


FIGURE 1.10 – Exemple de non transitivité de la conformité

Définition 1.2.12 (Réduction [BS86]). *q red p si $Tr(q) \subseteq Tr(p)$ et q conf p.*

Définition 1.2.13 (Extension [BS86]). *q ext p si $Tr(p) \subseteq Tr(q)$ et q conf p.*

Dans l'exemple 1.10, q red p et r ext q. Mais, $r \not\mathsf{conf}\ p$.

La relation conf n'est donc pas transitive, tandis que les relations red et ext sont réflexives et transitives. Cependant, ces relations (conf, red et ext) s'avèrent difficiles à vérifier en pratique [LS00].

Proposition 1.2.5 (Fermeture transitive de conf [Led91b]). $\mathsf{conf}^* = \mathsf{conf} \circ \mathsf{conf}$

où conf^* est la fermeture transitive de conf ($\mathsf{conf}^* \stackrel{def}{=} \mathsf{conf} \circ \mathsf{conf} \circ \cdots$), qui est la plus forte relation transitive plus faible que conf. De plus, $\mathsf{conf}^* = \mathsf{ext} \circ \mathsf{red}$ [Led91b]. L'exemple de la figure 1.10 montre que r est plus déterministe que p, mais $r \not\mathsf{conf}\ p$. La réduction de blocage implique la réduction de l'indéterminisme (le seul moyen d'ajouter des blocages est d'ajouter de l'indéterminisme), mais le sens inverse n'est pas garanti.

Ces relations de conformité ne sont définies que sur des systèmes sans distinction des entrées et des sorties. Une non conformité est détectée lorsqu'un système refuse une entrée qu'il était censé accepter. Or, en pratique, il est impossible de tester le refus d'un événement a en boîte noire : cela reviendrait à être capable de savoir qu'après une suite d'événements donnés, un système refusera éternellement d'accepter l'événement a. Afin de mettre en œuvre le test de conformité sur des systèmes réels, Tretmans [Tre99] a proposé de nouvelles définitions des relations de conformité dans le langage des IOTS (Input Output Transition Systems).

1.2.6 Relations de test

La phase de test représente un effort important dans le cycle de développement de logiciels. Le test est considéré comme une interaction particulière entre un observateur et le processus observé. Les relations de préordre et d'équivalence *Must* et *May* sont définies par Hennessy [NH83] pour distinguer ce qu'un processus peut faire de ce qu'il doit faire. Concrètement, deux systèmes de p et q sont liés par le préordre *May* ou *Must* si chaque fois que p « peut » (ou « doit ») passer un test, q « peut » (ou « doit ») également passer ce test.

Définition 1.2.14 (Préordre *May* [CH93]). *Soient p et q deux LTS.*

$$p \sqsubseteq_{\text{may}} q \text{ si } Tr(p) \subseteq Tr(q)$$

Le préordre *May* est l'inclusion de traces classique. Dans le cas des LTS, l'ajout de détails consiste à définir de nouvelles traces et éventuellement, de nouvelles actions. q peut faire tout ce que p peut faire. L'ajout de traces permet d'ajouter des détails ou des fonctionnalités mais la spécification étendue peut être moins déterministe que l'originale. De plus, les anciennes fonctionnalités peuvent être refusées dans la nouvelle spécification.

L'exemple de la figure 1.8(a) montre que $p \sqsubseteq_{\text{may}} q$ car :

$$Tr(q) = \{\epsilon, a, b\}$$
$$Tr(p) = \{\epsilon, a\}$$

De même, l'exemple 1.8(b) montre $p \sqsubseteq_{\text{may}} q$ et $q \sqsubseteq_{\text{may}} p$ aussi.

Contrairement au préordre *May*, le préordre *Must* est défini en prenant en considération deux aspects comportementaux : le concept de convergence et le déterminisme. Avant de présenter le préordre *Must*, on présente les définitions de convergence et d'ensemble d'acceptance de Hennessy.

La notion de trace représente fidèlement le comportement des processus déterministes mais elles n'est pas suffisante pour les processus indéterministes. Les traces ne détectent pas non plus la divergence, à savoir si un processus s'engage ou non dans une séquence d'actions internes infinie.

Définition 1.2.15 (Convergence [CH93])**.** *Soit $\sigma \in Tr(p)$, la convergence $p \downarrow \sigma$ est définie de manière inductive comme suit :*

- *$p \downarrow \epsilon$ s'il n'y a pas séquence infinie $\langle p_i \rangle_{i \geqslant 0}$ telle que $p \xrightarrow{\tau} p_0$ et $p_i \xrightarrow{\tau} p_{i+1}$*
- *$p \downarrow a.\sigma'$ si $p \downarrow \epsilon$ et $p \stackrel{a}{\Rightarrow} p'$ implique $p' \downarrow \sigma'$.*

Le concept d'acceptance est défini de plusieurs manières en littérature [Hen88, Hen85, Led91b]. Il est lié fortement à la notion de refus ou d'« échec » en CSP. Nous retenons une autre définition de l'ensemble d'acceptance de p après une trace σ :

Définition 1.2.16 (Ensemble d'acceptance [Hen85])**.** *L'ensemble d'acceptance de p après σ est défini par :*

$$Acc_2(p, \sigma) \stackrel{def}{=} \{S(p') \mid p \stackrel{\sigma}{\Rightarrow} p' \wedge p' \stackrel{\tau}{\not\rightarrow}\}$$

C'est la deuxième définition d'ensemble d'acceptance d'Hennessy. L'ensemble d'acceptance représente l'ensemble d'ensembles d'actions qu'un processus peut faire après une trace le conduisant à un état stable.

Définition 1.2.17 (Préordre *Must* [NH83])**.**

$$p \sqsubseteq_{\text{must}} q \;\; si \; \forall \sigma \in L^*. p \downarrow \sigma \Rightarrow (q \downarrow \sigma \wedge Acc_2(q, \sigma) \subset\subset Acc_2(p, \sigma))$$

« L'ensemble d'acceptance de q est inclus dans l'ensemble d'acceptance de p » signifie que le processus q est plus déterministe que le processus initial p et que q doit faire tout ce que p doit faire.

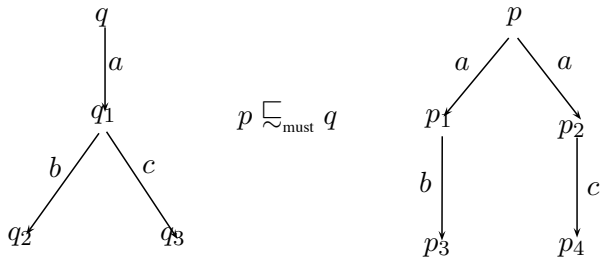


FIGURE 1.11 – Un exemple du préordre *Must*

L'exemple de la figure 1.11 montre que $p \sqsubseteq_{\approx \text{must}} q$, mais pas l'inverse car $Acc_2(p,a) \not\subset\subset Acc_2(q,a)$.

- $Acc_2(q,\epsilon) = \{\{a\}\} \subset\subset Acc_2(p,\epsilon) = \{\{a\}\}$
- $Acc_2(q,a) = \{\{b,c\}\} \subset\subset Acc_2(p,a) = \{\{b\},\{c\}\}$
- $Acc_2(q,ab) = \{\varnothing\} \subset\subset Acc_2(p,ab) = \{\varnothing\}$
- $Acc_2(q,ac) = \{\varnothing\} \subset\subset Acc_2(p,ac) = \{\varnothing\}$

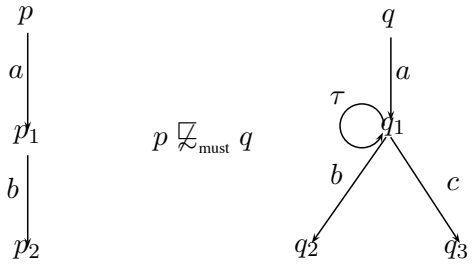


FIGURE 1.12 – Il n'existe pas de préordre *Must* entre p et q

Dans l'exemple 1.12, il n'y a pas de préordre *Must* entre q et p à cause de la divergence sur q_1.

Définition 1.2.18 (Préordre de Test [NH83])**.**

$$p \sqsubseteq_{\approx \text{T}} q \stackrel{def}{=} p \sqsubseteq_{\approx \text{may}} q \ et \ p \sqsubseteq_{\approx \text{must}} q$$

Le préordre de Test (testing preorder) est la synthèse du préordre *Must* et *May*. Cela implique l'égalité de traces entre les deux processus.

Définition 1.2.19 (Équivalence de Test)**.** *L'équivalence de Test* $\approx_t$ *est définie par :*

$$p \approx_t q \stackrel{def}{=} p \sqsubseteq_{\approx \text{must}} q \ et \ q \sqsubseteq_{\approx \text{must}} p$$

Ces relations de Test sont implantées par Cleaveland et al. [CH93]. Le préordre *Must* correspond à la relation de réduction, tout en prenant en compte le cas des processus divergents.

Proposition 1.2.6 ([Led91b]).

$$p \approx_t q \textit{ ssi } p \text{ red } q \textit{ et } q \text{ red } p$$

Les relations de préordre *Must* et de préordre de Test préservent les propriétés de sûreté, de refus et de vivacité [EF02].

1.2.7 Relations d'échec

La sémantique d'échec permet de décrire et de classifier les *blocages* que les processus rencontrent. En sémantique de traces en CSP, il n'est pas possible de décrire certains concepts tels que l'indéterminisme ou la distinction de blocages. Le modèle d'échec intègre les informations disponibles dans la sémantique de traces, en introduisant la notion de refus pour modéliser de tels concepts. Les relations d'échec proposées par Brookes, Hoare [BHR84], sont définies en CSP. La notion de *paire d'échecs* de CSP est proche de la notion de refus en LOTOS.

Définition 1.2.20 (Ensemble de paires d'échecs d'un processus)**.**

$$\textit{failures}(p) \stackrel{def}{=} \{(\sigma, X) \in A^* \times 2^A \mid \exists p'.\ p \stackrel{\sigma}{\Rightarrow} p' \wedge S(p') \cap X = \varnothing\}$$

En CSP, le couple (σ, X) désigne une paire d'échecs où σ est une trace représentant une séquence possible d'actions que le processus peut effectuer et X est l'ensemble d'actions que la machine ne peut pas faire.

Définition 1.2.21 (Préordre d'échec [BHR84])**.** *Le préordre d'échec $\sqsubseteq_{fail}$ est défini par :*

$$q \sqsubseteq_{fail} p \stackrel{def}{=} \textit{failures}(q) \subseteq \textit{failures}(p)$$

Le préordre d'échec raffine le préordre de trace et également le refus qui est défini comme suit : $\textit{refusals}(p) = \{X \mid (\varnothing, X) \in \textit{failures}(p)\}$

Le concept de refus permet de préciser la notion de déterminisme ce que la notion de trace dans CSP ne peut pas représenter. La définition de Brinksma [BS86] est inspirée de cette définition de refus. On peut trouver que la relation de réduction en LOTOS coïncide avec le préordre d'échec dans CSP.

En sémantique d'échec, deux processus sont équivalents au sens de l'échec si leurs ensembles d'échecs sont équivalents :

Définition 1.2.22 (Équivalence d'échec [BHR84])**.** *L'équivalence d'échec $\approx_{fail}$ est définie par :*

$$q \approx_{fail} p \stackrel{def}{=} \textit{failures}(q) = \textit{failures}(p)$$

Notons que l'équivalence de test coïncide avec l'équivalence d'échec [vG90].

1.3 Bilan

Dans ce chapitre, les raffinements en B, CSP, ou CCS sont étudiés. Quelques relations sont analysées afin de trouver des candidats pour la relation de raffinement. Nous retiendrons que la technique de raffinement cherchée consiste à ajouter des détails concrets et à réduire l'indéterminisme. Un modèle raffiné est plus précis (ajout de détails ou de fonctionnalités) et plus concret (l'abstraction est réduite, les fonctionnalités nécessaires sont préservées). Les relations *Must*, conf, ext,red conf* respectent ces contraintes. Elles permettent de s'orienter vers une démarche de vérification en *intension* des propriétés liées à l'ajout de détails et à la réduction de l'indéterminisme.

Le tableau 1.1 propose un résumé de ces relations selon les critères considérés.

	Réduction de l'indéterminisme	Réduction de blocages	Extension des traces	Transitivité
$\sqsubseteq_{\text{may}}$	∅	∅	⋆⋆⋆	⋆⋆⋆
conf	⋆⋆⋆	⋆⋆⋆	⋆	∅
ext	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆
red ($\sqsubseteq_{fail}$, $\sqsubseteq_{\text{must}}$)	⋆⋆⋆	⋆⋆⋆	∅	⋆⋆⋆
$\succsim$ ou simulation faible	∅	∅	⋆⋆⋆	⋆⋆⋆
conf*	⋆	∅	⋆	⋆⋆⋆

⋆	peut être supporté.
⋆⋆⋆	est garanti.
∅	n'est pas supporté.

TABLE 1.1 – Comparaison des relations envisagées selon quatre critères

Les relations de conformité (la relation d'extension en particulier) sont des candidats pour le raffinement. La relation d'extension est transitive. Ainsi, elle combine la réduction de l'indéterminisme et l'extension de traces. La relation d'extension est la seule qui correspond aux deux critères ciblés. En plus, on peut trouver que la relation d'extension ext satisfait la propriété de raffinement stipulée par Boiten(cf. propriété 1.1) :

$$R \text{ ext } A \Rightarrow \forall I.\ (I \text{ conf } R \Rightarrow I \text{ conf } A). \tag{1.2}$$

où I désigne une implantation et R est le raffinement d'une spécification abstraite A.

La relation ext est par conséquent utile dans le cas où elle est satisfaite (puisqu'elle garantit alors que le raffinement est correct), et peut être utilisée comme un avertissement dans le cas où elle ne l'est pas. Cependant, ces relations (conf, red et ext) s'avèrent difficiles à vérifier en pratique [LS00]. Si l'inclusion de traces et les relations de simulation sont vérifiées par des boîtes à outils telles que CADP [GMLS06], à notre connaissance, ni conf, ni ext ne sont actuellement outillées. Basées sur les méthodes de calcul des préordres *Must* et *May* de Cleaveland [CH93], nous proposons par la suite des méthodes permettant d'implanter la vérification de ces relations de conformité.

Chapitre 2

Machine d'états UML et raffinement

Sommaire

Le langage UML [SP99, OMG09] est largement diffusé et a été en général bien accueilli par le milieu industriel. Étant un langage de modélisation visuelle avec différents diagrammes, il est utilisé pour spécifier, construire et documenter les artefacts d'un système [OMG03]. Bien que la syntaxe d'UML soit précisément définie, sa sémantique n'est pas suffisante [Cra06]. Cette insuffisance concerne notamment les diagrammes dynamiques tels que les diagrammes de séquences et les machine d'états [vdB01]. Comme mentionné dans le chapitre précédent, UML est un des supports pour améliorer la qualité des systèmes critiques. Dans cette thèse, nous nous concentrons sur les machines d'états UML. Dans ce chapitre, nous abordons deux problèmes. Pourquoi sont-elles difficiles à construire ? Pourquoi y a t-il peu d'outils supportant la construction et la vérification de ces machines ? Nous verrons dans les premières parties les concepts de base des machines d'états et leur sémantique d'exécution. Nous ferons ensuite une synthèse des travaux traitant précisément de la vérification des machines d'états et plus généralement de ceux qui abordent la notion de raffinement dans l'approche objet.

2.1 Origine

En modélisant par des diagrammes d'états-transitions traditionnels, des automates par exemple, il est nécessaire de créer des nœuds distincts pour chaque combinaison valide des

paramètres qui définissent l'état. Cela peut conduire à un très grand nombre de nœuds et de transitions même pour modéliser un système simple. Cette complexité réduit la lisibilité et le traitement du diagramme d'états. Par exemple, pour modéliser un distributeur de billets ayant deux fonctionnalités simples, on a besoin d'un diagramme de dizaines d'états et de transitions. Cela est vraiment illisible si on souhaite étiqueter les états, les transitions ou mettre un commentaire. Avec les *statecharts* d'Harel [Har87], il est possible de modéliser des combinaisons valides de paramètres en un seul état, ce qui conduit à manipuler des modèles d'un niveau d'abstraction plus élevé. Une modélisation d'un distributeur de billets par *statechart* reflétant les mêmes caractéristiques conduit à un diagramme ne possédant que 5 états. Cela permet d'améliorer la lisibilité et les traitements du diagramme.

Les machines d'états d'UML ont été documentées dans la première version d'UML. Elles sont influencées par les statecharts d'Harel qui sont apparus dix ans auparavant. Une machine d'états UML est présentée comme un diagramme d'états-transitions organisé hiérarchiquement permettant de modéliser des états composites, des états concurrents, des activités et des historiques, etc.

Une autre variante des statecharts à laquelle Harel a lui-même contribué sont les Statecharts Rhapsody outillés dans IBM Rational Rhapsody [IBM09]. Ce dialecte a été créé après l'introduction d'UML 1.1. En fait, le dialecte Rhapsody est plus étroitement lié à UML que son ancêtre classique.

Les critiques des machines d'états portent principalement sur leur sémantique. L'article de Reggio en 1999 [RW99] recense une trentaine de problèmes de sémantique des machines d'états. Fecher et al ont rediscuté de ce problème en 2005 [FSKdR05] et ont recensé vingt neuf ambiguïtés de la norme 2.0. Au lieu d'aborder ce problème dans ce chapitre, nous étudions les concepts de base des machines d'états UML en leur choisissant une sémantique.

2.2 Concepts de base des machines d'états UML

Les concepts de base des machines d'états peuvent être rattachés à trois niveaux (c.f figure 2.1 [OMG09]) où chaque niveau dépend du niveau inférieur.

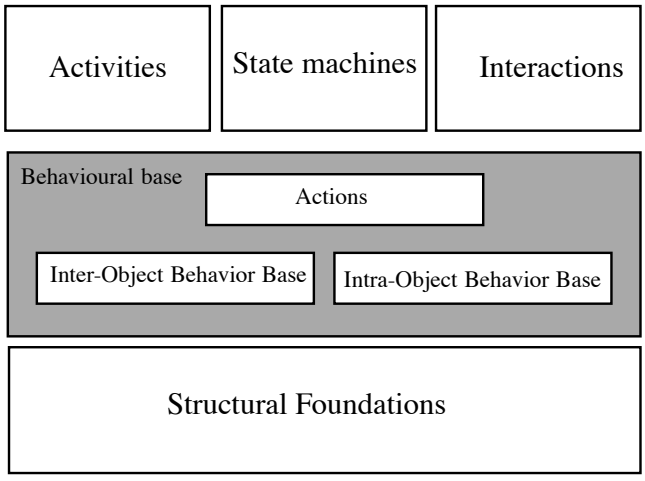


FIGURE 2.1 – Trois couches de sémantique d'UML [OMG09]

Le niveau le plus bas appelé niveau structurel comprend des concepts comme les valeurs, objets, liens, messages, etc. Le niveau intermédiaire est la base comportementale, qui contient des mécanismes pour les comportements individuels des objets et aussi la description d'un ensemble d'actions UML. Ce niveau comprend aussi *la base de comportement inter-objets* qui est responsable de la communication entre entités structurelles et *la base de comportement intra-objets* qui concerne les comportements d'une entité structurelle. Le niveau supérieur représente les différents formalismes de comportement en UML qui s'appuient sur les comportements de base : activités, machine d'états et interactions. Tous les trois utilisent les concept d'action pour exprimer le comportement. Les actions sont des unités fondamentales de comportement [Sel04] et sont exprimées en termes de structures comportementales de base.

2.2.1 États

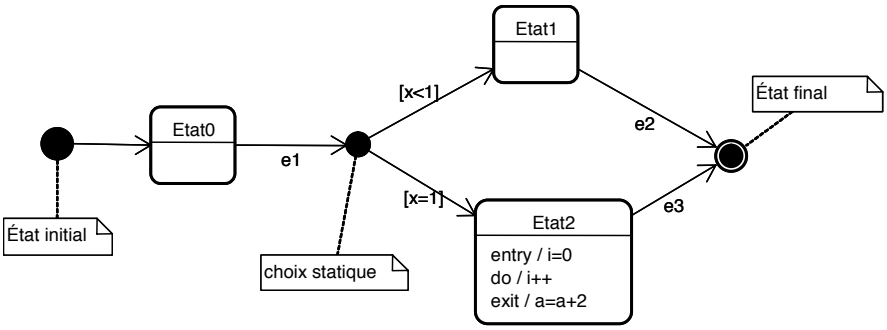


FIGURE 2.2 – États et pseudo-états

Le comportement d'un objet est décrit à l'aide d'états qui représentent les configurations de l'objet. Un état est défini par des valeurs d'attributs et par l'état d'autres objets dont il dépend. Un état peut être caractérisé par un aspect statique et un aspect dynamique. En effet, un état peut à la fois traduire le fait que les attributs de l'objet ont pris certaines valeurs (aspect statique), mais également que l'objet effectue un traitement ou des opérations de manière automatique (aspect dynamique). Un état représente une configuration durant le cycle de vie d'un objet pendant laquelle :

- il satisfait une certaine condition ;
- il exécute une certaine activité ;
- ou bien il attend un certain événement.

Il est possible de distinguer différents types d'états : pseudo-états, état composite, état concurrent.

Pseudo-états Il existe des types de pseudo-états : état initial, état final, choix, divergence (*fork*), jonction(*join*), historique.

- Une machine d'états dispose toujours d'un état initial unique qui est son point d'entrée. Il est visuellement représenté par un rond noir. Si la machine d'états possède des états composites, chacun doit avoir à son tour un état initial. L'état initial a la

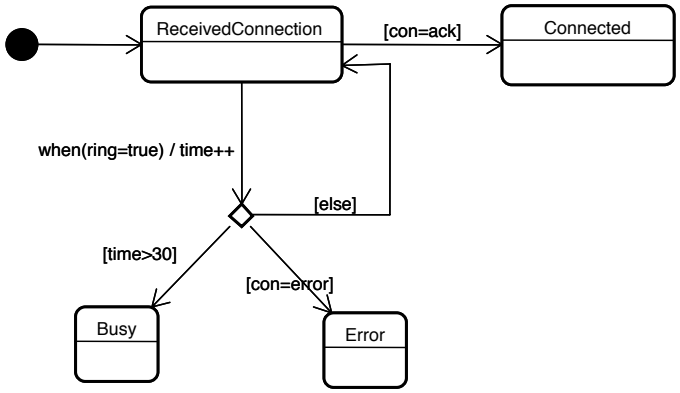


FIGURE 2.3 – Choix dynamique

particularité de ne pas comporter de transitions entrantes. La transition à partir de l'état initial n'est pas étiquetée, sauf si la transition crée l'objet.

- Au contraire, l'état final désigne un état stable dans lequel la machine n'évolue plus. Il présente la particularité de ne pas avoir de transition sortante. Une machine d'états ainsi qu'un état composite peuvent ne pas avoir d'état final.
- Choix : il est utilisé pour distinguer des états accessibles en fonction d'expressions conditionnelles. Par conséquent, les transitions sortantes d'un choix possèdent des gardes. On distingue deux types de choix :
  - Le choix statique : les conditions dans les gardes sont évaluées avant que la transition entrante du choix ne soit empruntée (cf. figure 2.2).
  - Le choix dynamique : l'évaluation des gardes se fait après l'exécution des actions portées par la transition entrante (cf. figure 2.3). Les pseudo-états de jonction et de choix introduisent de l'indéterminisme si plusieurs gardes sont vraies simultanément. Dans certains cas de modélisation, cet indéterminisme ne pose pas problème et peut être souhaité.
- Pseudo-état historique : Un automate n'a pas de mémoire. UML propose une notion d'historique offrant un mécanisme pour mémoriser le dernier sous-état atteint au sein d'un état composite. Deux mécanismes d'historique sont proposés :
  - Historique plat (*Shallow history*) : lorsqu'on atteint l'état historique, l'état actif sera le dernier état atteint en restant au même niveau hiérarchique que le pseudo-état historique.
  - Historique profond (*Deep history*) : l'état actif sera le dernier état atteint même si celui-ci est à un niveau hiérarchique inférieur à celui du pseudo-état historique.

Les transitions sortant des pseudo-états ne peuvent pas avoir de déclenchement, sauf celles issues du pseudo-état initial (page 573 dans [OMG09]).

État composite Un état composite est composé de sous-états. Les sous-états d'un état donné constituent un diagramme d'états à part entière. Tout événement envoyé et traité

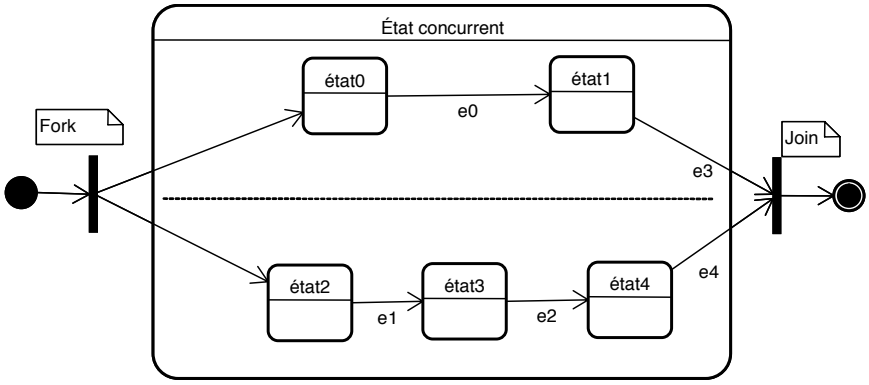


FIGURE 2.4 – État concurrent

par un état composite est envoyé et sous-traité par des sous-états. Les états composites peuvent disposer des mêmes propriétés que les états simples. S'il existe une transition entrante vers un état composite, la machine d'états associée à l'état composite doit avoir un état initial. Il existe deux types de pseudo-états utilisés dans l'état composite : point d'entrée et point de sortie (nouveau dans UML 2.1) (c.f figure 2.5).

État concurrent Un état concurrent est un état composite qui contient au moins deux régions séparées par des lignes de tirets. Dans chaque région, les composants sont exécutés parallèlement. Néanmoins, nous ne traitons pas ce type d'état dans cette thèse.

Activité Un état peut se composer d'activités. La modélisation d'activité porte sur la séquence et les conditions de la coordination des comportements de bas niveau [OMG03] qui correspondent à la déclaration d'une opération exécutable (pas nécessairement un appel de méthode). Étant un flot de contrôle d'actions dans un contexte particulier, une activité est décrite par une ou plusieurs opérations durant un certain temps dans un état particulier. On distingue trois types d'activité au sein d'un état :

- *entry* : cette activité est exécutée lorsque la machine atteint l'état et elle n'est pas interruptible.
- *doActivity* : cette activité est exécutée tant que la machine reste dans l'état. La différence principale entre l'activité *doActivity* et l'action est que l'activité peut être interrompue par un événement. L'activité peut donc être spécifiée par ailleurs et modélisée par un diagramme d'activités qui est un graphe de nœuds et de flux qui montre le flux de contrôle (et éventuellement des données) à travers les étapes d'un calcul [RJB05].
- *exit* : cette activité est exécutée lorsque l'on sort de l'état et elle n'est pas interruptible.

2.2.2 Transitions

Une machine d'états UML est un graphe orienté, parce que les états sont reliés par des connexions unidirectionnelles, appelées transitions. Une transition est composée de trois

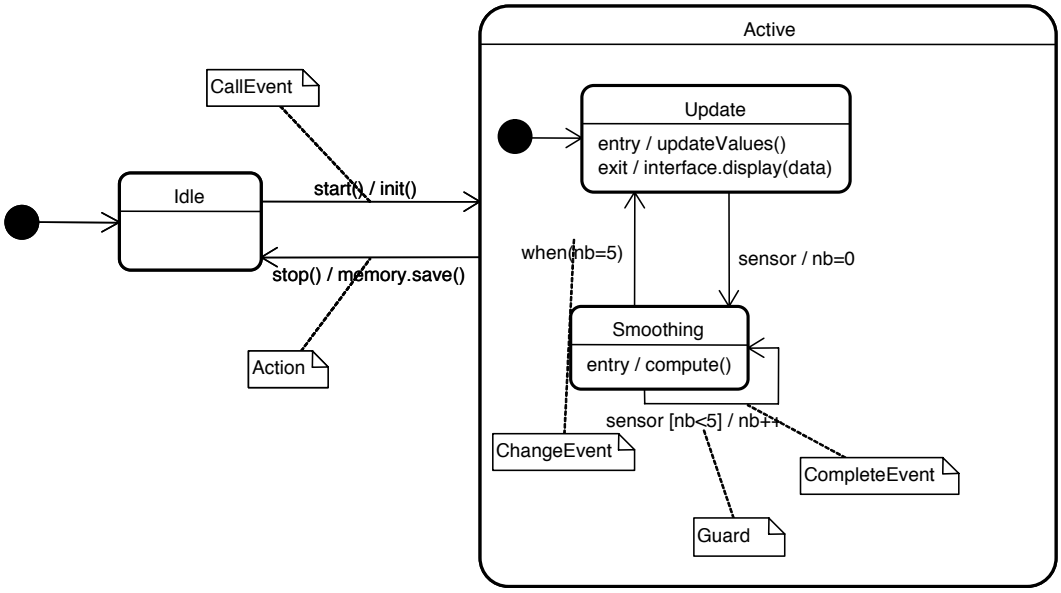


FIGURE 2.5 – États et état composite avec les événements.

éléments : trigger, garde, action, qui peuvent tous être facultatifs.

Trigger Un trigger est associé à un événement et son déclenchement permet d'activer la transition. L'événement est soit externe, soit interne. Les événements externes sont échangés entre objets. Les événements internes sont créés, émis et reçus au sein de l'objet.

i. **CallEvent** : modélise la réception d'un message synchrone (une requête) en invoquant un appel d'opération [OMG09]. Il conduit à l'exécution de l'opération et au changement d'état de la machine d'états.

ii. **AnyReceiveEvent** : apparu depuis la norme 2.0, il est noté par : *all. AnyReceiveEvent* déclenche la transition à la réception de tout message reçu s'il n'est pas trigger d'une autre transition sortante du même état source.

iii. **TimeEvent** : ce type d'événement désigne deux types d'événements temporisés
   - *after <expression>* qui spécifie une durée relative,
   - *at <expression>* qui spécifier une date absolue.

iv. **ChangeEvent** : il est défini par une expression booléenne qui est continuellement évaluée. Lorsqu'elle devient vraie, la transition est activée. Remarquons que ce type d'événement est différent de la garde. Une condition de garde n'est évaluée qu'une seule fois, lorsque l'événement associé est sollicité. Si elle est fausse, la transition n'est pas déclenchée et l'événement est ignoré.

v. **SignalEvent** : modélise la réception d'un message asynchrone. Comme **CallEvent**, il conduit à l'exécution d'opérations et au changement d'état de la machine d'états.

Garde Basée sur le concept « Constraint », la garde est une expression booléenne qui est évaluée lorsqu'un trigger se déclenche pour permettre d'activer une transition. La garde

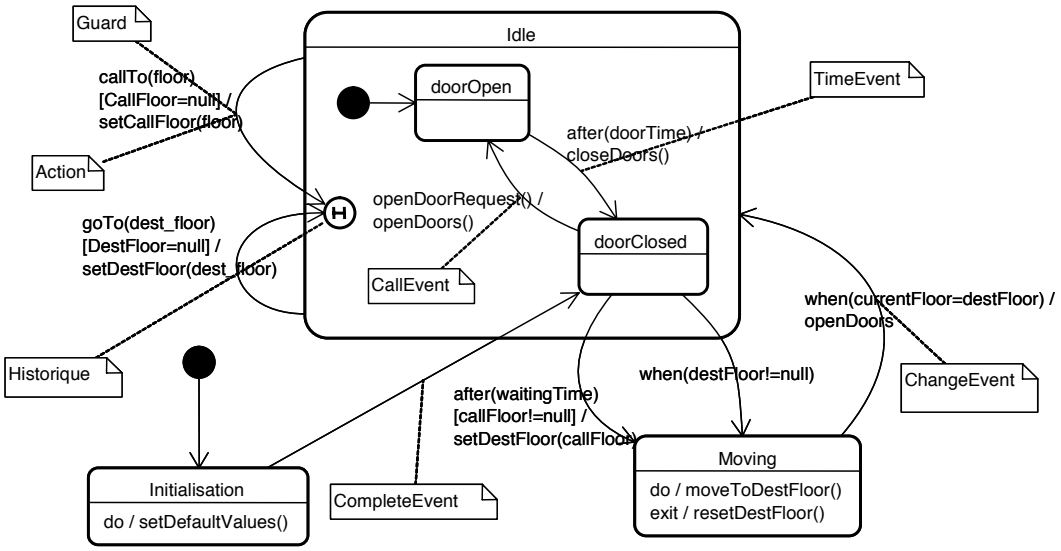


FIGURE 2.6 – Illustration d'un état composite avec ses transitions, gardes et événements

peut être exprimée comme une contrainte formulée dans le langage OCL. Par exemple la garde $[a > b]$, permet d'évaluer si a est supérieur à b.

Actions L'expression comportementale associée à une transition est définie par une ou plusieurs actions. Une action est un élément atomique exécutable non interruptible, comme par exemple la création ou la destruction d'autres objets ou l'envoi de signaux. En UML, l'ordre d'exécution des actions est séquentiel, comme pour les statecharts Rhapsody, à la différence des statecharts classiques où l'ordre est parallèle.

2.3 Sémantique d'exécution

Bien que les syntaxes abstraites et concrètes soient soigneusement définies dans la norme, UML n'a pas encore de sémantique formelle. Le terme sémantique se réfère à l'exécution des machines d'états UML. Cette sémantique se définit comme une mise en correspondance des concepts de modélisation avec leur exécution [OMG09]. Les critiques adressées aux machines d'états UML portent sur le manque d'une approche formelle pour la sémantique d'exécution, si bien que l'on trouve de nombreux travaux proposant des sémantiques formelles [LMM99b, DDd03, GLM01, vdB01]. Dans la norme, la description de sa sémantique est ambiguë, ce qui pose des problèmes pour les utilisateurs dès lors qu'ils souhaitent analyser, simuler ou valider les modèles élaborés. Par exemple, en machine d'états UML, est-ce que l'activité s'exécute en boucle, de façon autonome et interruptible comme un *thread* ? La norme 1.3 mentionnait que lorsqu'une activité dans un état est démarrée, elle s'exécute dans son propre thread. Mais dans la norme 2.2, le mécanisme d'exécution n'est pas abordé. Dans le cas positif, est-il possible d'associer une activité à un objet passif ? Il subsiste plusieurs problèmes d'incohérence, de carence dans la sémantique des machines d'états UML. Il est possible de voir dans cette ambigüité de sémantique, un moyen pour définir sa propre sémantique ce qui ferait la richesse d'UML

Crane [Cra06] a catégorisé les approches sémantiques en trois familles principales : les modèles mathématiques, les systèmes de réécriture et les approches de traduction (c.f la figure 2.7).

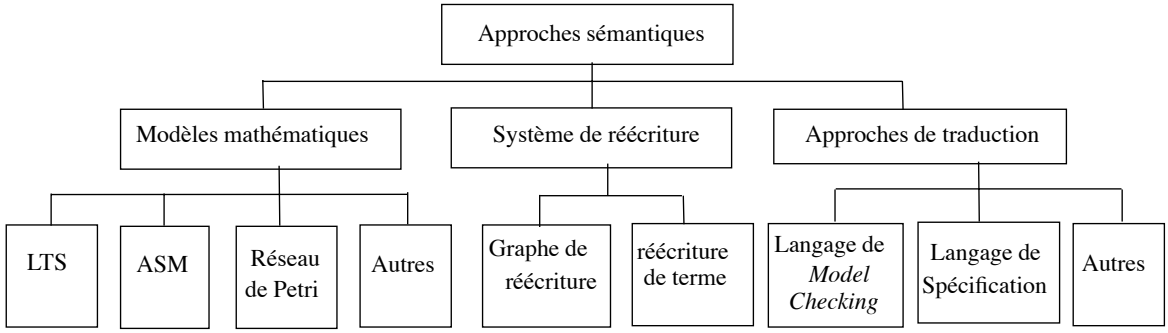


FIGURE 2.7 – Les approches sémantiques pour les machines d'états UML [Cra06]

2.3.1 Exécution jusqu'à terminaison (*Run to completion*)

Dans ce paragraphe, on présente le concept de *Run-To-Completion* (RTC) qui est au cœur des sémantiques des machines d'états UML. La sémantique de traitement d'événements dans la machine d'états UML est basée sur RTC. RTC est un passage entre deux configurations stables de la machine en explicitant les traitements à effectuer depuis la diffusion des événements jusqu'à l'accomplissement du dernier traitement avant d'atteindre la configuration cible. La machine est dite en *configuration stable* lorsque toutes les activités au sein de l'état qu'elle a atteint (sauf *doActivity* qui est interruptible) sont terminées. Par exemple dans la figure 2.8, la machine est en configuration stable *State1* et l'activité *entry/temp=0* est terminée. Lorsque l'événement e survient, l'activité associée à la transition sortante de *State1* est exécutée ($temp = 90$) et l'état cible est *State3*. Remarquons qu'il y a deux transitions dont le *trigger* est e, le choix de la transition franchie sera abordé au paragraphe suivant.

2.3.2 File d'événements et priorité des événements

L'environnement peut envoyer un ou plusieurs événements à la machine d'états à la même date. Ces événements sont rassemblés dans une file d'attente associée à la machine d'états. La machine d'états se trouve toujours dans un état actif, soit, elle exécute des activités associées à ses états actifs : soit elle est en attente d'un événement pour déclencher une transition. Toute transition sortant de l'état actif dont les triggers sont dans la file d'attente peut être déclenchable sous réserve que sa garde (si elle en possède une) soit vraie. Si plusieurs transitions sont déclenchables, il est nécessaire d'en sélectionner une. Plusieurs politiques de sélection sont possibles et sont laissées au choix des utilisateurs. Une fois qu'un événement est sélectionné, il est traité par le mécanisme de RTC (cf. paragraphe ci-dessus). Ces événements sont examinés dans l'état actif suivant. S'il n'y a aucune transition activable, l'ensemble des événements est alors ignoré. Dans le cas où la file est vide, la machine reste dans l'état atteint en attendant l'occurrence d'un événement pour changer

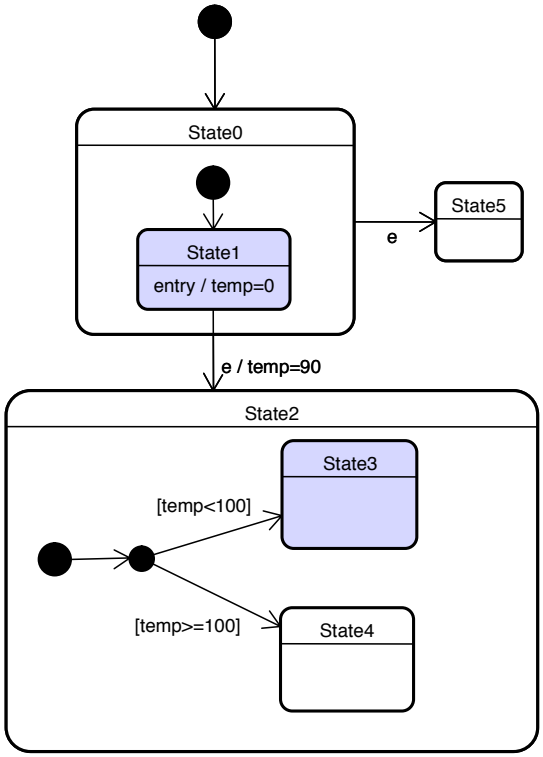


FIGURE 2.8 – Configuration stable - *State0* et *State1*

d'état. Notons qu'il existe un type d'événement *CompletionEvent* qui permet de marquer la fin d'une activité de type *doActivity* ; cet événement est toujours prioritaire quel que soit le cas de figure.

2.3.3 Indéterminisme, priorité des transitions et transitions en conflit

Après avoir sélectionné l'événement à traiter, on peut se retrouver dans une situation conflictuelle où plusieurs transitions sont franchissables :

- Les transitions sortent du même état (cf. figure 2.9 (a)). C'est un cas indéterministe qui doit être signalé au modélisateur car il peut s'agir d'une erreur. Si l'indéterminisme n'est pas levé, la transition franchie peut être sélectionnée aléatoirement ou selon des critères donnés par le modélisateur.

- Les transitions sortent d'états imbriqués (cf. figure 2.9 (b)). Dans ce cas, la priorité d'une transition provenant d'un sous-état est supérieure à celle d'une transition en conflit origine de l'un de ses états contenant [OMG09].

Le paragraphe suivant donne un aperçu des méthodes et outils existants pour aider à la vérification et au raffinement de machines d'états.

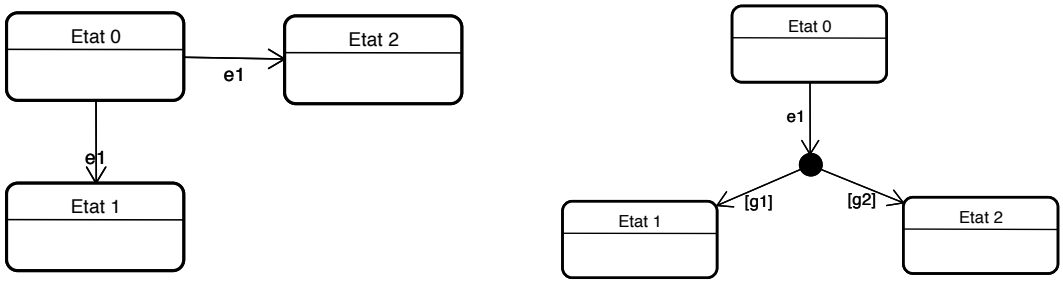


FIGURE 2.9 – (a) Exemple d'indéterminisme (b) Les gardes sont disjonctives

2.4 Raffinement et vérification dans l'approche objet

L'approche orientée objet est une méthodologie de conception modulaire, basée sur la construction de systèmes à base de composants interactifs appelés objets. Elle est née avec le langage de programmation Simula 67 [BDMN73], qui concerne la modélisation d'objets dans un objectif de simulation. En conséquence, l'approche objet offre une méthodologie intuitive pour la modélisation des systèmes. Quatre concepts sont au cœur de l'approche objet : abstraction, encapsulation, héritage et sous-typage (polymorphisme) [MA01]. De nombreux travaux portent sur les concepts d'héritage et de sous-typage car ils sont fondamentaux pour mettre en œuvre la construction [WZ88, LW94, Nie93]. Vérifier les relations d'héritage et de substituabilité (sous-typage) par analyse des vues dynamiques des composants reste problématique. Nous présentons dans une première partie les travaux portant sur la construction et la vérification des modèles dynamiques en se basant sur les concepts objets : héritage, substituabilité ou relations inter-modèles. Dans la deuxième partie, nous nous intéressons aux travaux portant sur la transformation des machines d'états en modèles fondés sur des langages formels permettant d'utiliser les approches de *model checking*.

2.4.1 Raffinement dans l'approche objet

Comme nous l'avons mentionné dans le chapitre précédent, le concept de raffinement ne trouve pas ses origines dans l'approche orientée objet. Cependant le raffinement présente des similitudes avec la relation de spécialisation qui, elle, est une notion native de l'approche objet. Conformément au paradigme de la programmation par objets, les éléments du système sont considérés comme des objets qui communiquent entre eux et avec leur environnement via des interfaces. Un système est alors tout simplement vu comme un ensemble d'objets. Cela permet aux concepteurs de se concentrer sur les fonctions, les services et les interfaces des objets sans se préoccuper de leurs interactions possibles avec d'autres parties du système. En outre, cette approche facilite la maintenance de gros systèmes car *les services* peuvent facilement être introduits et mis à jour. Pour assister la *conception* et la *vérification* des modèles, des travaux s'intéressent depuis quelques années à la cohérence inter-modèles [LP05, OAK03] et intra-modèles [HKRS05]. L'incohérence intra-modèles s'explique selon [HKRS05] de deux façons principales : le concept multi-vue et le développement en plusieurs phases au cours desquelles chaque modèle est défini de façon plus détaillée. Encore une fois, l'imprécision de la sémantique d'UML est une source d'incohérence, tandis que la relation de raffinement préserve la cohérence inter-modèles

(appelée également la cohérence verticale).

Pour la cohérence inter-modèles, plusieurs auteurs abordent la cohérence entre les diagrammes de séquences et les machines d'états. Dans [SJM04], les auteurs s'intéressent à la préservation de cohérence des modèles comportementaux (diagramme de séquences et machine d'états de protocole) dans le cadre de la restructuration de modèles (refactoring). La cohérence inter-modèles est fondée sur la préservation des traces de deux modèles. Dans un autre article [SMSJ03], les auteurs transforment les diagrammes de classes, les diagrammes de séquences et les machines d'états dans la logique de description pour vérifier la cohérence entre les différentes versions de ces diagrammes. Engels et al [EHHS02] vérifient la cohérence entre les diagrammes de séquence et les machines d'états en utilisant des règles de méta-modélisation dynamique. Dans les travaux de Lam et al. [LP05], les diagrammes de séquences et les machines d'états sont traduits dans le langage de processus π-calcul. Le problème de vérification de la cohérence entre les diagrammes de séquences et les machines d'états se ramène à un problème de vérification de bisimulation ouverte (*open bisimulation*). Le problème est résolu en utilisant l'outil *Mobility Workbench*.

L'analyse de cohérence intra-modèles est traitée par plusieurs auteurs [SPTJ01, BMP+02, DAB02, GW03]. Dans [SPTJ01], pour préserver des propriétés comportementales, les auteurs proposent un ensemble des transformations basiques également connues sous le nom de restructuration de modèles. Ces transformations de modèles à base de règles et de patrons assurent la préservation de comportements et les propriétés sont définies par des règles de pré et post-condition exprimées en langage OCL. La préservation de comportement dans UML peut être traduite par le fait que les concepteurs pourront effectuer des améliorations de la conception sans générer des résultats inattendus sur les différents points de vue UML. Derrick et al. [DAB02] s'intéressent à la cohérence intra-modèles dans le cadre ODP (*Open Distributing Processing*). Chaque diagramme représente une description partielle du système appelée point de vue. La cohérence des points de vue de la spécification est basée sur deux concepts : le raffinement et la correspondance de points de vue.

Étant une des caractéristiques principales de l'approche objet, le polymorphisme de sous-typage peut être considéré comme un raffinement de types [WZ88]. La théorie de sous-typage et en particulier la propriété de la substituabilité ont été étudiées par Liskov et al [LW94] qui ont défini une relation de sous-typage fort. Le problème est que cette relation ne permet pas l'ajout de nouveaux comportements dans le sous-type même s'ils ne pertubent pas les comportements initiaux définis dans le super-type. Pour répondre à ce problème, Nierstrasz [Nie93] caractérise le sous-typage de la manière suivante : « une instance d'un sous-type peut toujours être utilisée dans les contextes dans lesquels les instances du supertype étaient utilisés ». Il est à noter que substituer un type par un sous-type préserve la sûreté du système (dans le sens où il n'y aura pas de blocages) et non pas la compatibilité comportementale totale. Dans cet article, Nierstrasz considère les objets actifs comme des processus offrant des services. Il définit un cadre formel basé sur la relation d'extension de Brinksma [BS86] permettant d'étudier la substituabilité des services et la satisfaction des requêtes.

Les travaux de Sourrouille [Sou99, Sou01] s'intéressent également à la substituabilité et à l'héritage dans les machines d'états. [Sou01] propose une définition de l'héritage de comportement fondée sur des propriétés de substituabilité et applicable aux langages à objets courants. Des contraintes sur les domaines de définition sont introduites dans l'héritage des états. L'héritage de transitions impose des contraintes sur les séquences d'opérations auto-

risées en intégrant l'héritage des gardes. Ceci permet de supprimer des erreurs potentielles en cas d'indéterminisme et est accompagné de relations basées sur l'inclusion de traces. On vérifie le lien entre l'héritage de comportement et les règles usuelles de raffinement des pré/post-conditions. L'auteur a réussi à proposer des règles d'héritage des états, des traces et des gardes. Pourtant, la transformation de machines d'états en LTS n'est pas exposée en détail.

L'intérêt à la substituabilité dans le cadre d'un développement incrémental en général et pour la spécification d'architectures composées de plusieurs composants sera présenté dans le chapitre concernant le raffinement d'architectures.

En ce qui concerne l'héritage qui est une autre propriété importante dans l'approche objets, il est légitime de se demander si la relation de spécialisation traduit la relation de raffinement. Si une classe est une spécialisation d'une autre classe, cela signifie que toute instance de la sous-classe peut se comporter comme une instance de sa super-classe. Cela est proche de la notion d'extension comportementale en LOTOS [BS86]. Cusack [Cus91b] s'intéresse à l'héritage dans l'algèbre de processus CSP en se basant sur le concept générique d'héritage dans l'approche par objets. L'auteur propose un formalisme proche de CSP en redéfinissant les relations d'extension et de réduction de LOTOS. Notons que l'auteur différencie l'héritage strict (dans lequel les propriétés existantes sont conservées) de l'héritage non strict (où les propriétés existantes peuvent être altérées). Puisqu'aucune hypothèse n'est formulée sur la nature des objets, les classes sont représentées comme des ensembles d'objets et l'héritage comme un sous-ensemble de la relation. Ce modèle insiste sur le fait que si une classe est un sous-type d'un autre type, alors il est nécessairement aussi une sous-classe. Ce travail est prolongé par Cusack dans [Cus91a] en considérant l'héritage en langage Z orienté objet.

Dans [PPGK03], les auteurs soulignent la différence sémantique entre des relations de raffinement/abstraction et de généralisation/spécialisation. Le raffinement est une relation entre une spécification et son implantation. Selon eux, il existe deux types de raffinement : le raffinement homogène (sur tout l'ensemble des instances des classes) et le raffinement hétérogène (sur un sous-ensemble des instances des classes). Malheureusement, ces travaux ne traitent pas de l'aspect dynamique des diagrammes UML. Dans [PR94], les auteurs s'intéressent au raffinement de type, mais au contraire des travaux cités précédemment, le raffinement coïncide avec la spécialisation. Une définition précise de spécialisation des automates a été définie mais la transformation en automate n'est pas complète.

Dans notre contexte, nous considérons que les machines d'états UML sont associées aux classes. Ils sont utilisés pour décrire les comportements attendus des objets d'une classe. Dans le contexte des objets, si une classe C_R est une spécialisation d'une classe C_A, ce que l'on notera $C_R \prec C_A$, cela signifie que toutes les instances de C_R doivent se comporter comme des instances de C_A. Énoncé de façon plus formelle, nous pourrions écrire :

$$C_R \prec C_A \Rightarrow \text{Instances}(C_R) \subseteq \text{Instances}(C_A). \tag{2.1}$$

Cette définition sur la spécialisation peut se trouver dans [Duc02]. On peut constater que les propriétés de raffinement et (2.1) sont similaires. Mais qu'en est-il du raffinement des machines d'états associées aux classes ? Quel type de relation peut-on définir pour le raffinement de machines d'états, et comment peut-on vérifier cette relation ?

2.4.2 Vérification des machines d'états

Travaux académiques

La plupart des travaux de vérification des machines d'états sont basés sur les techniques de *model checking* décrites dans le chapitre précédent. Il s'agit de vérifier si un modèle du système satisfait une spécification, souvent formulée en termes de logique temporelle dans divers langages. Lilius et Paltor [LP99b, LP99a] ont proposé un cadre pour formaliser la sémantique opérationnelle d'une machine d'états UML et l'utiliser comme base de génération de code, de simulation et d'outils de vérification. La formalisation se fait en deux étapes. Tout d'abord, la structure d'une machine d'états UML est traduite dans un système de réécriture en langage PROMELA [Hol97]. Dans la deuxième étape, la sémantique opérationnelle des machines d'états est définie. Cette formalisation permet de traiter tous les aspects des machines d'états UML 1.3. La vérification a été mise en œuvre dans l'outil de *model checking* SPIN [Hol97]. Ces travaux visent à vérifier également la cohérence entre diagrammes de collaboration et machines d'états.

L'outil SPIN est également utilisé pour vérifier des machines d'états UML dans les travaux de Latella et al [LMM99b]. Dans ce papier, une variante des structures Kripke est utilisée comme modèle intermédiaire pour le modèle sémantique. Elle est appelée automate hiérarchique étendu. Toutefois, la création et la manipulation de ces automates présentent des difficultés et rendent l'utilisation de cette approche ardue. Suite à ces travaux, Latella et al [LMM99a] ont proposé de traduire les machines d'états UML en langage PROMELA. Ils ont suivi une approche similaire à celle de Mikk et al [MLSH98]. En 2001, Latella avec Gnesi et Mieke [GLM01] ont proposé d'exprimer la sémantique opérationnelle en exploitant la logique temporelle arborescente, notamment par les LTS, et utilisé l'environnement JACK [BGL94] pour la vérifier.

De la même manière, Schäfer, Knapp et Merz [SKM01, KMR02, KM02] proposent des méthodes pour traduire les machines d'états en langage PROMELA et les diagrammes de collaboration en ensemble d'automates Büchi. L'outil SPIN sert à vérifier le modèle PROMELA par rapport aux automates de Büchi. Certains aspects temps réel sont également pris en compte en compilant la machine d'états temporisée en automate temporisé. La vérification est réalisée par l'outil UPPAAL [BLL+96].

L'article de Engels [EHKG02] aborde le problème de préservation de cohérence entre modèles UML-RT au niveau des architectures. Les auteurs définissent des règles de transformation d'interconnexions de modèles. Ils montrent que ces règles préservent ou non les propriétés d'absence de blocage et de consistance de protocoles. La formalisation utilisée est CSP.

Burmester et al [BGHS04] ont présenté une solution pour le développement incrémental et la vérification de modèles UML/RT utilisant l'outil FUJABA. Ils transforment les modèles UML/RT en modèles logiques de l'environnement HUppaal [DM01] pour permettre la vérification de modèle par *model checking*.

Les travaux cités ci-dessus couvrent un grand nombre d'aspects des machines d'états (parallélisme, historique...), à l'exception des travaux de Latella qui ne considèrent pas le concept d'historique.

Travaux industriels

Rose RT(Real-time) [IBM09] est un environnement de développement UML complet permettant de développer des stratégies itératives permettant de réduire des erreurs de conception. Les machines d'états dans Rose RT respectent la norme UML en particulier la sémantique de RTC. Il existe des fonctionnalités telles que la mise au point, l'animation, le test pour détecter les erreurs et les problèmes de conception afin de tester les modèles de conception en continu pendant le processus de développement.

Un autre produit sur le marché commercial est Telelogic Rhapsody (récemment devenu IBM Rational Rhapsody [Rha09]). C'est un outil pour le développement de modèles temps réel et embarqués. Basé sur UML et SysML, Rhapsody met en œuvre le paradigme MDT (*Model Driven Testing*). Il offre des fonctionnalités de simulation permettant de tester les modèles continuellement tout au long du processus de conception. Cela garantit que les comportements sont analysés pendant le processus de développement afin de réduire les erreurs le plus tôt possible. Pourtant, les *statecharts* de Rhapsody ont des points légèrement différents des machines d'états UML comme mentionné dans la partie précédente.

Bien que ces outils soient très puissants, ils ne sont pas appropriés pour le développement incrémental car ils ne permettent pas de comparer les versions différentes d'un même modèle durant le processus de développement.

2.5 Bilan

Dans ce chapitre, nous avons fait une présentation des machines d'états UML. Nous avons non seulement présenté des aspects nécessaires à la compréhension de ces machines, mais également donné un bilan des travaux portant sur le raffinement et la vérification de modèles comportementaux. Les critiques communes adressées à UML portent toujours sur sa sémantique. La norme 2.0 détaille plus la sémantique que la norme précédente, mais elle n'est toujours pas satisfaisante car elle reste ambiguë sur certains points. Nous avons vu au travers de l'état de l'art que ce problème est toujours résolu partiellement en associant aux machines d'états une sémantique formelle par l'intermédiaire de langages de processus ou de description e.g LTS, PROMELA, CSP...

De plus, les machines d'états sont conceptuellement difficiles à construire. Il est donc nécessaire d'aider les concepteurs à construire de manière incrémentale des machines et à les vérifier par comparaison à chaque étape. La plupart des méthodes et des outils que nous avons présentés sont appropriés pour vérifier *a posteriori* un ensemble de propriétés explicites telles que l'absence de blocage, la compatibilité au sens conformité. Mais ils ne permettent pas de comparer si l'incrément d'un modèle reste cohérent avec le modèle initial.

Dans nos travaux antérieurs [GL04], des relations de conformité sur des machines d'états et des machines de protocole ont été définies pour la construction incrémentale. Elles s'inspirent des travaux existants sur les LTS et présentent la particularité d'être définies sur les concepts même des machines d'états. Ces relations permettent de détecter des erreurs pendant les étapes de construction comme par exemple la présence de blocages. Bien que ces travaux ne traitent pas de tous les aspects des machines d'états tels que le pseudo-historique, les pseudo-états de choix et le parallélisme, ils proposent des perspectives pour les prendre en compte. Cependant la mise en œuvre d'un point de vue pratique de ces

relations n'est pas proposée car se pose le problème de la manipulation de traces (traces infinies et inclusion de traces).

L'étude des travaux existants nous a conduit à définir et mettre en œuvre une nouvelle approche. Parmi les approches sémantiques pour les machines d'états, nous avons choisi de raisonner sur les LTS car ce sont des modèles formels facilement manipulables fixant la sémantique d'exécution des machines d'états UML. En nous inspirant des travaux de Cleaveland et Hennessy [CH93] qui nous paraissent très intéressants sur les aspects de calculabilité des relations de test, nous proposons de définir et implanter des relations de raffinement sur les LTS.

Deuxième partie

Contributions

Chapitre 3

Calculabilité des relations de conformité

Ce chapitre est consacré à la présentation de l'implémentation de la vérification des relations de conformité.

Sommaire

Les relations de conformité définies sur des LTS nous apparaissent appropriées pour comparer des modèles élaborés lors de différentes étapes de construction incrémentale. Un travail antérieur [Gou06] a été mené afin de formaliser ces relations sur des machines d'états UML. Ce travail a conduit à la définition de trois relations de conformité sur les machines d'états UML servant à s'assurer de la cohérence des étapes de construction incrémentale. Cependant, aucune implémentation de ces relations n'est proposée. Nous avons par conséquent cherché à résoudre les problèmes du calcul des relations sur les LTS en considérant le fait que les LTS étaient candidats pour formaliser la sémantique des machines d'états. Si l'inclusion de traces et les relations de simulation sont vérifiées par des boîtes à outils telles que CADP [GMLS06], à notre connaissance, ni conf, ni ext ne sont actuellement implantées.

Ce chapitre concerne les définitions du cadre mathématique nécessaires à la démonstration de la calculabilité des relations de conformité. Dans la première partie, nous nous intéressons à l'implantation de la vérification des relations d'extension (ext) et de réduction (red). A partir de ces relations, nous proposons une implantation de la relation de conformité (conf). Puis nous présentons l'algorithme de calcul de la relation de conformité ainsi que sa complexité. Nous terminons ce chapitre en mettant en évidence l'intérêt de la vérification de conformité par rapport au test de conformité. La transformation des machines d'états UML en LTS n'est pas abordée dans ce chapitre et fait l'objet du chapitre suivant.

3.1 Implantation des relations d'extension (ext) et de réduction (red)

La similarité entre le préordre $\sqsubseteq_{\text{must}}$ et la relation de réduction red présentés dans l'état de l'art nous a conduit à nous inspirer de l'approche de Cleaveland [CH93] pour proposer une implantation de red et ext. Cette approche étant basée sur les notions de simulation et de graphe d'acceptance, nous donnons tout d'abord une définition de ces notions. Puis nous prouvons la calculabilité de ces relations.

3.1.1 Bisimulation

Pour servir à démontrer des théorèmes dans ce chapitre et le chapitre suivant, nous reformulons et généralisons la définition de la bisimulation donnée par Milner [Mil99] en nous inspirant des travaux de Cleaveland.

Définition 3.1.1 (Bisimulation [CC92, CH93])**.** *Soient* $\Pi \subseteq \mathcal{P} \times \mathcal{P}$ *et* $\Psi_1, \Psi_2 \subseteq \mathcal{P} \times Act$. *La relation binaire* $\mathcal{R}_{\langle \Pi, \Psi_1, \Psi_2 \rangle}$ *est une bisimulation si* $\mathcal{R} \subseteq \Pi$ *et* $p\mathcal{R}q$ *implique :*

i. $\langle p, a \rangle \in \Psi_1 \Rightarrow (p \xrightarrow{a} p' \Rightarrow \exists q'.\ q \xrightarrow{a} q' \wedge p'\mathcal{R}q')$

ii. $\langle q, a \rangle \in \Psi_2 \Rightarrow (q \xrightarrow{a} q' \Rightarrow \exists p'.\ p \xrightarrow{a} p' \wedge p'\mathcal{R}q')$

Lorsque $\Pi = \mathcal{P} \times \mathcal{P}$ et $\Psi_1, \Psi_2 = \mathcal{P} \times Act$, nous retrouvons la définition de bisimulation de Milner.

Définition 3.1.2 (Plus grande bisimulation [CH93])**.** $p \sim^{\Pi}_{\langle \Psi_1, \Psi_2 \rangle} q$ *s'il existe une bisimulation* $\mathcal{R}_{\langle \Pi, \Psi_1, \Psi_2 \rangle}$ *avec* $p\mathcal{R}q$.

On note $\succsim^{\Pi} \stackrel{def}{=} \sim^{\Pi}_{\langle \varnothing, \mathcal{P} \times Act \rangle}$ et $\precsim^{\Pi} \stackrel{def}{=} \sim^{\Pi}_{\langle \mathcal{P} \times Act, \varnothing \rangle}$.

Remarquons qu'on a $p \succsim^{\Pi} q$ si p simule q et si $p \Pi q$; on a $p \precsim^{\Pi} q$ si q simule p et si $p \Pi q$ (*et non pas si* $q \Pi p$).

3.1.2 Graphes d'acceptance

Un graphe d'acceptance associé au LTS p est un graphe déterministe obtenu à partir de p, où les nœuds sont étiquetés par leur ensemble d'acceptance.

Rappelons que nous avons donné deux définitions de l'ensemble d'acceptance (cf. définitions 1.2.11 [p 27] et 1.2.16 [p 29]), la différence étant que la deuxième définition ne tient

compte que des états stables. Pour l'analyse de conformité, on ne peut se restreindre à l'étude seule des états stables. Pour cette raison, nous retiendrons la première définition de l'ensemble d'acceptance (cf. définition 1.2.11).

Avant de définir la notion de graphe d'acceptance, nous donnons la définition de la fermeture d'un ensemble d'états :

Définition 3.1.3 (Fermeture [CH93]). *Soit* $Q \subseteq \mathcal{P}$, $Q^{\varepsilon} = \{p \mid \exists q \in Q.\ q \stackrel{\varepsilon}{\Longrightarrow} p\}$.

Nous définissons le graphe d'acceptance en s'inspirant des définitions de [CH93, KvB93].

Définition 3.1.4 (Graphe d'acceptance). *Étant donné un LTS* $\langle P, A, \rightarrow, p\rangle$, *où* $A = L \cup \{\tau\}$, *son graphe d'acceptance* $\mathcal{A}(p)$ *est le LTS déterministe* $\langle T, L, \rightarrow_T, t\rangle$ *tel que :*

i. T *est un ensemble de noms d'états.*

ii. $\rightarrow_T$ *est un ensemble de transitions de* $T \times L \times T$.

iii. $t \in T$ *est l'état initial.*

iv. Pour tout état u *de* T, *on associe les ensembles* $u.states \in 2^P$ *et* $u.acc \subseteq 2^L$ *tels que :*

- $u.acc = \{X \mid X = Out(q, \varepsilon) \wedge q \in u.states\}$;
- $t.states = \{\{p\}\}^{\varepsilon}$;
- *Pour tout* $t_1 \in T$, *pour tout* $X \in t_1.acc$, *pour tout* $x \in X$, *il existe un unique* $t_2 \in T$ *tel que* $t_1 \xrightarrow{x}_T t_2$ *avec* $t_2.states = (\cup_{s \in t_1.states} D(s, x))^{\varepsilon}$.

Cette définition est similaire aux graphes d'acceptance de [CH93] sans prendre en compte la notion de divergence.

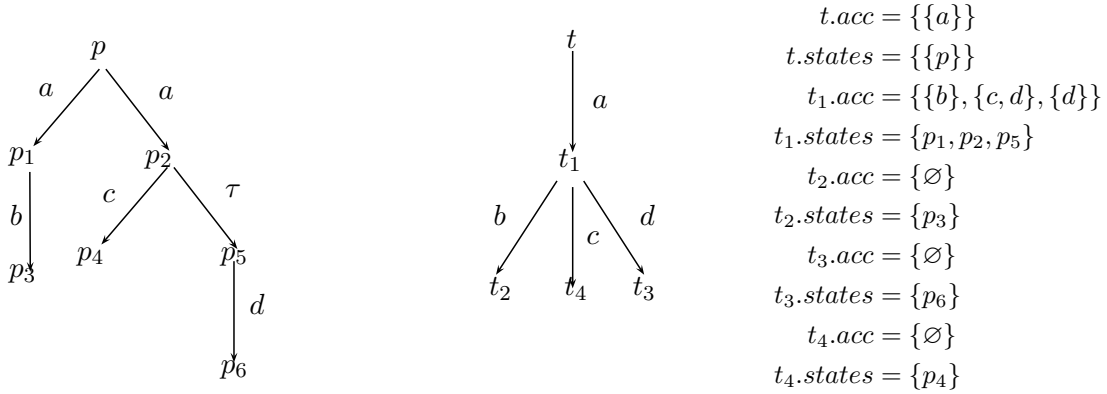


FIGURE 3.1 – LTS p et son graphe d'acceptance $\mathcal{A}(p)$

L'exemple de la figure 3.1 montre le LTS p avec son graphe d'acceptance. Chaque nœud du graphe contient deux informations : l'ensemble d'acceptance et les états associés du LTS origine. Par exemple, après la trace a, l'état du graphe d'acceptance contient :

- l'ensemble d'acceptance : $\{\{b\}, \{c, d\}, \{d\}\}$
- les états associés dans le LTS : $\{p_1, p_2, p_5\}$

3.1.3 Calculabilité des relations d'extension et de réduction

Soient p un LTS et $\mathcal{A}(p)$ son graphe d'acceptance, on a [CH93] :

$$Tr(\mathcal{A}(p)) = Tr(p). \tag{3.1}$$

Le théorème suivant permet de ramener le calcul des relations d'extension et de réduction à des simulations sur des graphes d'acceptance.

Théorème 3.1.1. *Soient p, q deux LTS et leur graphe d'acceptance :*
$\mathcal{A}(p) = \langle T, \mathcal{L}, \rightarrow_T, t\rangle, \mathcal{A}(q) = \langle U, \mathcal{L}, \rightarrow_U, u\rangle$. *Soit* $\Pi = \{\langle t, u\rangle \mid u.acc \subset\subset t.acc\}$. *On a :*

1. q red $p \Leftrightarrow t \succsim^{\Pi} u$
2. q ext $p \Leftrightarrow t \precsim^{\Pi} u$

Démonstration. On démontre (2). (1) est similaire et n'est pas présenté ici. Tout d'abord, on définit $\mathcal{R}_{\langle \Pi, \Psi_1, \varnothing\rangle} \subseteq \mathcal{T} \times \mathcal{T}$ la relation entre deux graphes d'acceptance comme suit :

$$t\mathcal{R}u \stackrel{def}{=} u.acc \subset\subset t.acc \ \wedge \forall a.\ t \xrightarrow{a}_T t' \Rightarrow \exists u'.\ u \xrightarrow{a}_U u' \wedge t'\mathcal{R}u'$$

$\Leftarrow$) On démontre $t \precsim^{\Pi} u \Rightarrow q$ ext p.

Premièrement, on constate que la simulation forte implique l'inclusion de traces :

$$\begin{aligned} t \precsim^{\Pi} u \Leftrightarrow & t\mathcal{R}u \\ \Rightarrow & \forall a.\ t \xrightarrow{a}_T t' \Rightarrow \exists u'.\ u \xrightarrow{a}_U u' \wedge t'\mathcal{R}u' \\ \Rightarrow & \forall \sigma.\ \sigma \in Tr(t) \Rightarrow \sigma \in Tr(u) \\ \Rightarrow & Tr(t) \subseteq Tr(u) \Leftrightarrow Tr(p) \subseteq Tr(q) \text{ (résultat de 3.1).} \end{aligned} \tag{3.2}$$

Deuxièmement, nous devons vérifier que l'inclusion des ensembles d'acceptance des états simulés des graphes d'acceptance implique l'inclusion des ensembles d'acceptance des LTS.

A partir de la définition des graphes d'acceptance t et u, après une trace $\sigma \in Tr(p)$, si $t \xrightarrow{\sigma}_T t_n$ et $u \xrightarrow{\sigma}_U u_n$, on a :

- $u_n.states = \{q' \mid q \stackrel{\sigma}{\Longrightarrow} q'\}$
- $u_n.acc = Acc_1(q, \sigma)$
- $t_n.states = \{p' \mid p \stackrel{\sigma}{\Longrightarrow} p'\}$
- $t_n.acc = Acc_1(p, \sigma)$.

Étant donné $t\mathcal{R}u$, en considérant le fait que t et u sont déterministes et le résultat $Tr(t) \subseteq Tr(u)$ (cf. résultat 3.2), on a :

$$\begin{aligned} & \forall \sigma \in Tr(t).\ t \xrightarrow{\sigma} t_n \Rightarrow \exists u_n.\ u \xrightarrow{\sigma} u_n \wedge\ u_n.acc \subset\subset t_n.acc \\ \Rightarrow & \forall \sigma \in Tr(p).\ Acc_1(q, \sigma) \subset\subset Acc_1(p, \sigma) \\ \Rightarrow & \forall \sigma \in Tr(p).\ Ref(q, \sigma) \subseteq Ref(p, \sigma) \ \text{ (Proposition 1.2.3 [p 27].)} \end{aligned} \tag{3.3}$$

Par conséquent, (3.2) et (3.3) impliquent q ext p.

⇒) A partir q ext p, on prouve la relation $t\mathcal{R}u$.

$$\begin{aligned}
& q \text{ ext } p \Rightarrow q \text{ conf } p \\
\Leftrightarrow & \forall \sigma \in Tr(p).\ Ref(q,\sigma) \subseteq Ref(p,\sigma) \\
\Leftrightarrow & \forall \sigma \in Tr(p).\ Acc_1(q,\sigma) \subset\subset Acc_1(p,\sigma) \quad \text{(Proposition 1.2.3 [p 27])} \\
\Rightarrow & \forall \sigma \in Tr(t).\ \forall t_n.\ \forall u_n.\ t \xrightarrow{\sigma}_T t_n \wedge u \xrightarrow{\sigma}_U u_n \Rightarrow\ u_n.acc \subset\subset t_n.acc \qquad (3.4)
\end{aligned}$$

Étant donné que t et u sont déterministes, comme q ext $p \Rightarrow Tr(p) \subseteq Tr(q) \Rightarrow Tr(t) \subseteq Tr(u)$ (cf. résultat 3.2), on a :

$(3.4) \Rightarrow \forall \sigma \in Tr(t).\ t \xrightarrow{\sigma}_T t_n \Rightarrow \exists u_n.\, u \xrightarrow{\sigma}_U u_n \wedge\ u_n.acc \subset\subset t_n.acc$

et $\Rightarrow \forall \sigma.a \in Tr(t).\ t \xrightarrow{\sigma.a}_T t_{n+1} \Rightarrow \exists u_{n+1}.\, u \xrightarrow{\sigma.a}_U u_{n+1} \wedge\ u_{n+1}.acc \subset\subset t_{n+1}.acc$

que l'on peut écrire :

$$\forall \sigma \in Tr(t).\ t \xrightarrow{\sigma}_T t_n \Rightarrow \exists u_n.\ u \xrightarrow{\sigma}_U u_n\ \wedge t_n \mathcal{S} u_n \qquad (3.5)$$

où $t\mathcal{S}u \stackrel{def}{=} u.acc \subset\subset t.acc \wedge \forall a.\ t \xrightarrow{a}_T t' \Rightarrow \exists u'.\ u \xrightarrow{a}_U u'$

La relation $\mathcal{S}$ est définie de la même façon que $\mathcal{R}$, sauf qu'elle n'est pas récursive. On peut vérifier que (3.5) implique $t\mathcal{R}u$:

A partir de (3.5), si $\sigma = \varepsilon$, avec t et u déterministes, on montre que $t\mathcal{S}u$.

Supposons que la trace σ conduise aux états $u_n,\ t_n$ tels que $u \xrightarrow{\sigma}_U u_n$ et $t \xrightarrow{\sigma}_T t_n$, on a : $t_n \mathcal{S} u_n$

Si $t_n \xrightarrow{a}_T t_{n+1}$, on a :

$t \xrightarrow{\sigma.a}_T t_{n+1}$, $\exists u_{n+1}, u \xrightarrow{\sigma.a}_U u_{n+1} \wedge t_{n+1}\mathcal{S}u_{n+1}(3.5)$.

Puisque u est déterministe, on a : $u \xrightarrow{\sigma}_U u_n$ et $u_n \xrightarrow{a}_U u_{n+1}$.

$\forall t_n, u_n,\ t_n \mathcal{S} u_n \Rightarrow (u_n.acc \subset\subset t_n.acc \wedge \forall a. t_n \xrightarrow{a}_T t_{n+1} \Rightarrow \exists u_{n+1}.\ u_n \xrightarrow{a}_U u_{n+1}) \wedge t_{n+1}\mathcal{S}u_{n+1}$.

Donc (3.4) et (3.5) impliquent $t\mathcal{R}u$.

Finalement, on a q ext $p \Leftrightarrow t \precsim^{\Pi} u$. □

3.2 Implantation de la relation de conformité conf

Pour implanter la relation de conformité conf, nous faisons appel à la fusion de graphes d'acceptance. La caractéristique de cette fusion est de produire un graphe d'acceptance dont les traces sont cycliques. Nous donnons tout d'abord une définition d'une trace cyclique, puis de l'opération de fusion. Enfin nous prouvons la calculabilité de la relation de conformité.

3.2.1 Trace cyclique

Définition 3.2.1 (Trace cyclique [KvB94]). *Soit un LTS $\langle P, A, \rightarrow, p\rangle$, une trace σ est cyclique si :*

p after $\sigma = \{p_i \in P$ tel que $p \stackrel{\epsilon}{\Longrightarrow} p_i\}$

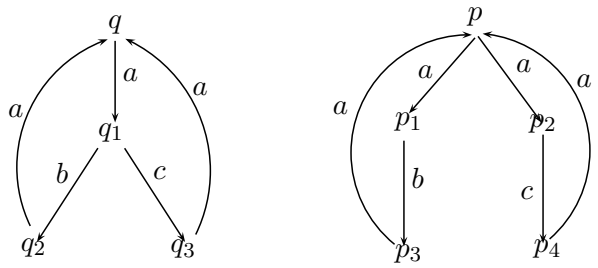


FIGURE 3.2 – LTS dont des traces sont cycliques

Dans la figure 3.2, les traces cycliques de q et p sont : $(aba)^*, (aca)^*$.

On note $Tr_c(p)$ l'ensemble de toutes les traces cycliques du LTS p.

Par rapport à la définition normale de trace, la trace cyclique permet de préserver le caractère récursif d'une spécification comme c'est fréquemment le cas en LOTOS.

Définition 3.2.2 (Extension cyclique de LTS [KvB93])**.** *Soient p et q deux LTS. q est une extension cyclique de p, que l'on notera q extc p, si*

i. q ext p *et*

ii. $\forall \sigma \in Tr_c(p)$, $\sigma \in Tr_c(q)$.

La relation d'extension cyclique peut alors servir de condition supplémentaire à la préservation de l'extension. De plus, cette relation permet de préserver le caractère récursif de la structure de la spécification. En outre, il est possible de définir une extension finie d'une spécification qui présente une boucle infinie.

Dans la figure 3.2, q extc p.

L'extension du graphe d'acceptance est définie de la manière suivante :

Définition 3.2.3 (Extension du graphe d'acceptance [KvB93])**.** *Soient p et q deux LTS et $\mathcal{A}(p), \mathcal{A}(q)$ leur graphe d'acceptance. $\mathcal{A}(q)$ est une extension de $\mathcal{A}(p)$, noté $\mathcal{A}(q)$ ext_g $\mathcal{A}(p)$, ssi*

i. $Tr(\mathcal{A}(p)) \subseteq Tr(\mathcal{A}(q))$ *et*

ii. $\forall \sigma \in Tr(\mathcal{A}(p)), Acc_1(q, \sigma) \subset\subset Acc_1(p, \sigma)$.

On peut définir l'extension cyclique du graphe d'acceptance de façon similaire à la définition de l'extension du graphe d'acceptance.

Définition 3.2.4 (Extension cyclique du graphe d'acceptance [KvB93])**.** *Soient p et q deux LTS et $\mathcal{A}(p), \mathcal{A}(q)$ leur graphe d'acceptance. $\mathcal{A}(q)$ est une extension cyclique de $\mathcal{A}(p)$, noté $\mathcal{A}(q)$ extc_g $\mathcal{A}(p)$, si*

i. $\mathcal{A}(q)$ ext_g $\mathcal{A}(p)$ *et*

ii. $\forall \sigma \in Tr_c(\mathcal{A}(p))$, $\sigma \in Tr_c(\mathcal{A}(q))$.

Définition 3.2.5 (Plus petite extension commune)**.** *Soient p, q et r trois LTS, tels que r ext p et r ext q.*

r est la plus petite extension commune de q et p si $\forall t$ tel que t ext $q \wedge t$ ext $p \Rightarrow t$ ext r

La définition de la plus petite extension commune de graphes d'acceptance est similaire à celle des LTS.

Définition 3.2.6 (Plus petite extension commune de graphes d'acceptance[KvB93])**.** *Soient $\mathcal{A}(p), \mathcal{A}(q)$ et $\mathcal{A}(r)$ trois graphes d'acceptance de trois LTS correspondants p, q, r, tels que $\mathcal{A}(r)$ ext_g $\mathcal{A}(p) \wedge \mathcal{A}(r)$ ext_g $\mathcal{A}(q)$.*

$\mathcal{A}(r)$ est la plus petite extension commune de $\mathcal{A}(q)$ et $\mathcal{A}(p)$ si pour toute extension cyclique commune de $\mathcal{A}(p), \mathcal{A}(q)$ est également l'extension cyclique commune de $\mathcal{A}(r)$.

3.2.2 Fusion de graphes d'acceptance

Comme introduite par Khendek [KvB93], l'opération *Merge* permet de fusionner des graphes d'acceptance afin d'obtenir un nouveau graphe d'acceptance. Elle est associative et commutative. Pour tout graphe d'acceptance fusionné, il existe un LTS associé. On note $Merge(p, q)$ le LTS de $Merge(\mathcal{A}(p), \mathcal{A}(q))$. Le graphe fusionné est toujours la plus petite extension cyclique commune des graphes initiaux [KvB93].

Dans ce paragraphe, nous présentons la définition de graphes d'acceptance fusionnés [KvB93], puis nous démontrons que la relation conf peut-être calculée à partir d'un graphe d'acceptance fusionné. Rappelons tout d'abord la définition d'un graphe d'acceptance introduite par Khendek [KvB93] :

Définition 3.2.7 (Merge [KvB93])**.** *Soient p et q deux LTS, et $\mathcal{A}(p), \mathcal{A}(q)$ leur graphe d'acceptance $\mathcal{A}(p) = \langle T, \mathcal{L}, \rightarrow_T, t\rangle$, $\mathcal{A}(q) = \langle U, \mathcal{L}, \rightarrow_U, u\rangle$.*

Le graphe fusionné des graphes d'acceptance

$$Merge(\mathcal{A}(p), \mathcal{A}(q)) = \langle V, \mathcal{L}, \rightarrow_V, \langle t, u\rangle\rangle$$

est défini comme suit :

i. $V = (T \times U) \cup T \cup U$

ii. Pour chaque état $v_i \in V$,
si $v_i = \langle t_j, u_k\rangle, v_i.acc = \{X_1 \cup X_2 \mid X_1 \in t_j.acc \wedge X_2 \in u_k.acc\}$,
si $v_i = t_i$, $v_i.acc = t_i.acc$,
si $v_i = u_i$, $v_i.acc = u_i.acc$.

iii. Pour chaque $\langle t_j, u_k\rangle \in V$,

- $\langle t_j, u_k\rangle \xrightarrow{a}_V \langle t_l, u_m\rangle$ *si* $t_j \xrightarrow{a}_T t_l \wedge u_k \xrightarrow{a}_U u_m$.
- $\langle t_j, u_k\rangle \xrightarrow{a}_V \langle t, u\rangle$ *si* $(t_j \xrightarrow{a}_T t \wedge u_k \not\xrightarrow{a}_U) \vee (t_j \not\xrightarrow{a}_T \wedge u_k \xrightarrow{a}_U u)$.
- $\langle t_j, u_k\rangle \xrightarrow{a}_V t_l$ *si* $t_j \xrightarrow{a}_T t_l,\ t_l \neq t \wedge u_k \not\xrightarrow{a}_U$.
- $\langle t_j, u_k\rangle \xrightarrow{a}_V u_m$ *si* $u_k \xrightarrow{a}_U u_m,\ u_m \neq u \wedge t_j \not\xrightarrow{a}_T$.

iv. On note $X = T, U$ et $x = t, u$. Pour chaque état $v_j \in X$, $v_j = x_j$,

- $x_j \xrightarrow{a}_V \langle t, u\rangle$ *si* $x_j \xrightarrow{a}_X x$,
- $x_j \xrightarrow{a}_V x_l$ *si* $x_j \xrightarrow{a}_X x_l,\ x_l \neq x$.

Les transitions sortant d'un nœud du graphe fusionné possèdent les labels du nœud (ou des nœuds) correspondant des graphes à fusionner. Par exemple, s'il existe une transition $t_j \xrightarrow{a}_T t_l$ dans le graphe d'acceptance t, mais qu'il n'en existe pas dans u, le graphe fusionné aura la transition $\langle t_j, u_k \rangle \xrightarrow{a}_V \langle t, u \rangle$. S'il existe une transition $t_j \xrightarrow{a}_T t_l$ dans le graphe d'acceptance t, et une dans u de la forme $u_k \xrightarrow{a}_U u_m$, alors le graphe fusionné aura la transition $\langle t_j, u_k \rangle \xrightarrow{a}_V \langle t_l, u_m \rangle$. Pour préserver des traces cycliques, les transitions arrivant à l'état initial de t ou u, sont remplacées par des transitions correspondantes qui atteignent l'état initial $\langle t, u \rangle$ de $Merge(t, u)$.

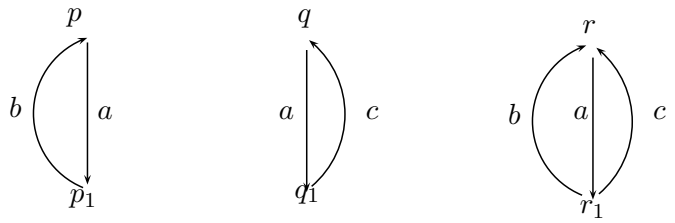


FIGURE 3.3 – Deux processus p et q et $r = Merge(p, q)$

L'opération *Merge* a quelques propriétés intéressantes.

Proposition 3.2.1. *Soient p, q deux LTS.*

i. $Merge(p, q) = Merge(q, p)$

ii. $Merge(p, q)$ extc $p \wedge Merge(p, q)$ extc q.

Le graphe fusionné est toujours la plus petite extension commune cyclique des graphes initiaux. En outre, dans le cas de traces cycliques, le nouveau LTS obtenu par la fusion offre le choix entre les comportements des deux spécifications initiales de manière récursive.

Dans la figure 3.3, $r = Merge(p, q)$ est la plus petite extension cyclique de p et q.

3.2.3 Calculabilité de la relation de conformité conf

Ce paragraphe donne la démonstration de la calculabilité de la conformité entre les LTS q et p au travers de la relation de réduction calculée entre le graphe d'acceptance de q et le graphe d'acceptance fusionné de p et q.

Théorème 3.2.1. *Soient p et q deux LTS.* q conf $p \Leftrightarrow q$ red $Merge(p, q)$.

Démonstration. $\Rightarrow$)q conf $p \Rightarrow q$ red $Merge(p, q)$

– $Merge(p, q)$ ext $q \Rightarrow Tr(q) \subseteq Tr(Merge(p, q))$

– Suite à la définition du graphe fusionné :

$$\begin{aligned} &\forall \sigma \in Tr(q) \wedge \sigma \notin Tr(p).\ Acc_1(Merge(p, q), \sigma) = Acc_1(q, \sigma) \\ &\Rightarrow \forall \sigma \in Tr(q) \wedge \sigma \notin Tr(p).\ Acc_1(q, \sigma) \subset\subset Acc_1(Merge(p, q), \sigma) \end{aligned} \tag{3.6}$$

En se référant à la définition de conformité 1.2.10 et à la proposition 1.2.3 [p 27], on a :

q conf $p \Rightarrow \forall \sigma \in Tr(q) \cap Tr(p).\ Acc_1(q, \sigma) \subset\subset Acc_1(p, \sigma)$.

A partir de la définition de l'inclusion d'ensemble d'ensembles 1.2.5 [p 27], on peut récrire cette dernière formule comme suit :

$$\begin{aligned}
&\forall \sigma \in Tr(q) \cap Tr(p). \forall X_2 \in Acc_1(q,\sigma).\ \exists X' \in Acc_1(p,\sigma).\ X' \subseteq X_2 \\
\Rightarrow &\forall X_2 \in Acc_1(q,\sigma).\ \exists X' \in Acc_1(p,\sigma).\ X' \cup X_2 \subseteq X_2
\end{aligned}$$

De plus, en se référant à la définition 3.2.7 :

$$\forall \sigma \in Tr(q) \cap Tr(p).$$
$$Acc_1(Merge(p,q)) = \{X_1 \cup X_2 \mid X_1 \in Acc_1(p,\sigma) \wedge X_2 \in Acc_1(q,\sigma)\}$$

on a :

$$\begin{aligned}
&\forall X_2 \in Acc_1(q,\sigma).\ \exists X \in Acc_1(Merge(p,q),\sigma), X = X' \cup X_2.\ X \subseteq X_2 \\
\Rightarrow &Acc_1(q,\sigma) \subset\subset Acc_1(Merge(p,q),\sigma) \qquad (3.7)
\end{aligned}$$

Avec (3.6) et (3.7), on a : $\forall \sigma \in Tr(q).\ Acc_1(q,\sigma) \subset\subset Acc_1(Merge(p,q),\sigma)$, c'est-à-dire q conf $Merge(p,q)$. Donc, avec la condition $Tr(q) \subseteq Tr(Merge(p,q))$, on a :

q conf $p \Rightarrow q$ red $Merge(p,q)$.

$\Leftarrow$) q red $Merge(p,q) \Rightarrow q$ conf p

Comme conf = red $\circ$ ext, [Led91a], où $\circ$ est la composition entre deux relations binaires, on a donc :

q red $Merge(p,q) \wedge Merge(p,q)$ ext $p \Rightarrow q$ conf p.

□

En associant ce théorème au théorème 3.1.1, on en déduit le corollaire suivant :

Corollaire 3.2.1. *Soient p et q deux LTS. q* conf *$p \Leftrightarrow Merge(\mathcal{A}(p),\mathcal{A}(q)) \gtrsim^{\Pi} \mathcal{A}(q)$.*

Ce théorème et son corollaire nous permettent d'implanter la relation de conformité entre des LTS à états finis.

3.2.4 Discussion sur d'autres relations de fusion

Nous présentons dans cette partie des travaux portant sur la fusion de spécifications et nous justifions le choix que nous avons fait sur l'opérateur *Merge*.

Étant données deux spécifications q et p, on souhaite avoir une composition s à partir de q et p qui soit une extension des deux spécifications. Plusieurs auteurs ont proposé des opérateurs de composition satisfaisant cette propriété.

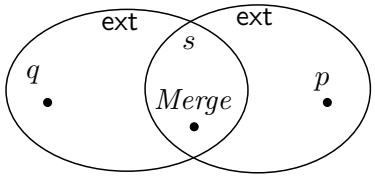


FIGURE 3.4 – Vue ensembliste des extensions de q et p et leur fusion

L'opérateur $\oplus$ proposé par Ichikawa [IYK90] sur LOTOS, appelé opérateur de fusion de spécifications, s'exprime par des règles d'inférence. Cette approche est limitée aux spécifications comportementales sans action interne. Ichikawa a montré que $q \oplus p$ est toujours la plus petite extension commune de q et p.

Steen et al [SBD96] ont proposé un opérateur de jointure $\bowtie$ préservant l'extension sans garantir que le résultat soit la plus petite extension commune. Cet opérateur présente l'inconvénient de ne pas garantir la réduction de l'indéterminisme par la composition.

Comme mentionné dans ce paragraphe, Khendek et al [KvB93] ont proposé une solution pour combiner des spécifications en une spécification unique qui étend les deux spécifications initiales. Cette spécification n'est pas la plus petite extension. Dans l'exemple de la figure 3.5, étant une extension de q et p, r est plus petite que m. r ne préserve pas les traces cycliques de q et p.

Parmi ces opérateurs de composition, nous avons choisi l'opérateur de fusion de Khendek. L'intérêt ici est que non seulement Khendek propose un moyen de calcul de graphe fusionné, mais son algorithme est également polynomial. De plus, la preuve ci-dessus démontre que les deux clauses q conf p et q red $Merge(q,p)$ sont équivalentes. Avec la proposition de Brinksma [BS86] conf=red $\circ$ ext, vérifier la conformité se ramène à vérifier que q est une réduction de s, $s \in S$ où S désigne l'ensemble des extensions communes de q et p (cf. figure 3.4).

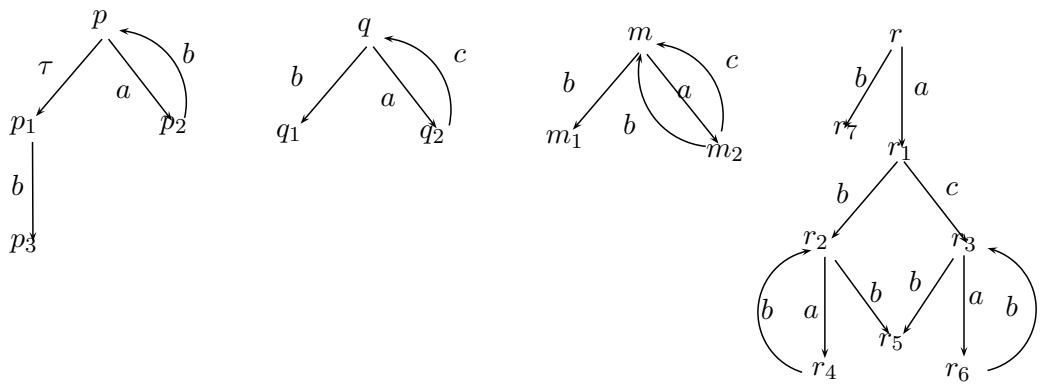


FIGURE 3.5 – $m = Merge(q,p)$ et m ext r

3.3 Algorithme de calcul et évaluation de la complexité

3.3.1 Algorithme de calcul de relation conf

Dans ce paragraphe, nous présentons l'algorithme de calcul de la relation conf que nous avons implanté en JAVA. Ceux de red, et ext sont similaires et ne sont pas présentés. L'algorithme de calcul de la relation conf entre deux LTS met en œuvre le corollaire 3.2.1. Tout d'abord, on doit construire les graphes d'acceptance des deux LTS. L'algorithme de construction de graphe d'acceptance nommé *buildAGraph* (cf. algorithme 1) est une variante de l'algorithme de construction des *AGraph* proposé dans [CH93]. Les informations de convergence ainsi que la construction d'autres types de graphe d'acceptance tels que

les *SAGraph*, *WAGraph* ne sont pas prises en compte. Une fois les graphes d'acceptance construits, ils sont fusionnés (fonction *Merge*).

Le calcul de la relation conf fait appel à la vérification de la simulation forte (fonction *simulate*). La fonction $\langle b, sp\rangle = simulate(ag_2, ag_1)$ retourne deux valeurs : le booléen b indiquant si le graphe d'acceptance ag_2 simule ag_1, et sp le vecteur de paires d'états simulés. On désigne par $first(sp[i])$ le premier élément de la paire de rang i dans sp et $second(sp[i])$ le second. Le premier élément fait référence à un nœud du graphe ag_1, et le second, à un nœud du graphe ag_2.

Pour calculer la simulation forte, il existe plusieurs algorithmes proposés par Fernandez et al [FM92], Bloom et Paige [BP95], ou Li Tan et Cleaveland [TC01]. Tous ces algorithmes sont polynomiaux.

Algorithm 1 *boolean* =conf $(LTS\ lts_2, LTS\ lts_1)$

$AG\ ag_1 = buildAGraph(lts_1)$;
$AG\ ag_2 = buildAGraph(lts_2)$;
$AG\ ag_3 = Merge(lts_1, lts_2)$;
$\langle b, sp\rangle = simulate(ag_3, ag_2)$;
si non b **alors**
  result = **false** ;
sinon
  $i = 0$;
  $n = length(sp)$;
  tantque $(i \leq n-1) \wedge (second(sp[i]).acc \subset\not\subset first(sp[i]).acc)$ **faire**
    $i = i + 1$;
  fin tantque
  result = $second(sp[i]).acc \subset\subset first(sp[i]).acc$;
finsi
retourne result ;

Si la relation de simulation est vérifiée, on vérifie qu'il y a l'inclusion des ensembles d'acceptance pour tous les nœuds simulés.

3.3.2 Évaluation de la complexité

Comme Cleaveland et Hennessy l'ont mentionné dans [CH93], le problème de la construction du graphe d'acceptance *buildAGraph* est un problème de déterminisation d'automates. Ce problème est PSPACE-complet avec une complexité temporelle qui est en général exponentielle tandis que les autres fonctions de l'algorithme de conf sont polynomiales. Par exemple, si n désigne le nombre d'états et m est le nombre de transitions, la complexité de la vérification de la simulation forte [FM92] est $O(nm)$ et celle de la fusion de graphes d'acceptance [KvB93] est $O(nm^2)$. D'après [Les95], la complexité temporelle de la déterminisation d'automates dans le pire des cas est $O(2^n m(n^3 m + 2^n n))$. Cleaveland et Hennessy ont montré que dans la pratique, les graphes transformés ont moins d'états que les systèmes de transitions originaux et généralement les transitions internes sont retirées après la transformation. Donc, en pratique, on peut considérer que le temps de calcul de la transformation est acceptable. Un autre argument va dans ce sens : les LTS sont générés à partir

nb d'états	nb de trans	(s)
69	121	47*10E-6
103	208	0,188
202	432	0,594
243	556	1,828
1518	3962	5,953
2026	3366	17,769
10 617	33 696	379

TABLE 3.1 – Test de performance du calcul de la relation d'extension

nb d'états	nb de trans	(s)
69	121	56*10E-6
103	208	0,203
202	432	0,656
243	556	1,922
1518	3962	5,969
2026	3366	17,285
10 617	33 696	379

TABLE 3.2 – Test de performance de l'algorithme conf

de machines d'états UML élaborées manuellement et par conséquent, on peut penser que le nombre d'états ne dépassera pas des centaines. Les tableaux 3.1 et 3.2 montrent respectivement la performance des programmes de calcul de l'extension et de la conformité sur un PC Pentium 3GHz de 2Go de mémoire.

3.3.3 Amélioration des algorithmes

Dans le cas des LTS dont le traitement pose un problème de mémoire, il est possible de réduire l'espace d'analyse en réduisant la taille des systèmes de transitions. Nous proposons une méthode d'analyse de conformité basée non plus sur les LTS mais sur leur minimisation en équivalence observationnelle. La validité de cette méthode est justifiée par les deux propositions suivantes.

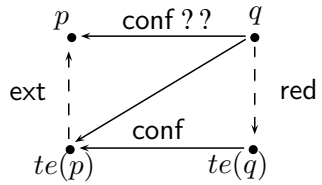


FIGURE 3.6 – Illustration de la proposition 3.3.1

Proposition 3.3.1. *$te(q)$ conf $te(p) \Leftrightarrow q$ conf p, où $te(p)$ désigne un LTS respectant l'équivalence de test de p.*

Démonstration. On sait que [Led91b] :
$\approx_t = \mathsf{red} \cap \mathsf{red}^{-1} = \mathsf{ext} \cap \mathsf{ext}^{-1}$
On en déduit (cf. figure 3.6) :

$$te(q)\ \mathsf{red}\ q,\ q\ \mathsf{red}\ te(q),$$
$$q\ \mathsf{ext}\ te(q),\ te(q)\ \mathsf{ext}\ q$$

De même pour p et $te(p)$.
Par conséquent :

$$q\ \mathsf{red}\ te(q) \wedge te(q)\ \mathsf{conf}\ \ te(p) \Rightarrow q\ \mathsf{conf}\ \ te(p)$$
$$\Leftrightarrow q\ \mathsf{conf}\ te(p) \wedge te(p)\ \mathsf{ext}\ \ p \Rightarrow q\ \mathsf{conf}\ p$$

□

On appelle $min(p)$ le LTS obtenu par la minimisation de p par équivalence observationnelle. Il existe de nombreux algorithmes de minimisation tels que la compression de τ-confluence [GvdP00] qui a une complexité en temps linéaire ou la compression de τ [Mat05] qui est implémentée par la fonction *bcg_min* de CADP (*aldebaran ?omin*, *-omin*). Démontrons que la conformité peut être vérifiée sur des LTS minimisés.

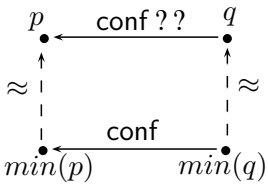


FIGURE 3.7 – Relations entre q, p et $min(p)$, $min(q)$

Proposition 3.3.2. *$min(q)$ conf $min(p) \Leftrightarrow q$ conf p*

Démonstration. En se référant au treillis de l'équivalence des relations de van Glabbeek [vG90] (cf. chapitre 1), nous constatons qu'il existe un chemin de la sémantique de bisimulation à la sémantique d'échec. Si deux systèmes sont équivalents quant à la bisimulation, alors ils sont équivalents en sémantique d'échec telle que définie par Hoare [Hoa78] qui elle-même coïncide avec l'équivalence de test de Hennessy et De Nicola [NH83].

On a donc :

$$min(q) \approx q \Rightarrow min(q) \approx_t q$$
$$\Rightarrow min(q)\ \mathsf{conf}\ min(p) \Leftrightarrow q\ \mathsf{conf}\ p$$

□

Sans	minimisation		Avec	minimisation	
nb d'états	nb de trans	(s)	nb d'états	nb de trans	(s)
1318	3962	6,078	152	449	0,125
1924	5830	11,434	225	679	0,391
10 617	33 696	379	48	130	0,016
23 887	77 644	1 522	93	266	0,047
130 047	434 156	66 265	413	1280	0,922
214 975	720 844	pb de mémoire	657	2069	2,625

TABLE 3.3 – Comparaison de performance de l'algorithme ext avec et sans minimisation

Nous avons donc démontré (cf. figure 3.7) que la conformité peut être vérifiée sur les LTS minimisés.

On peut s'attendre à ce que la minimisation conduise à des LTS dont le nombre de nœuds ne pose pas de problème. A titre d'illustration, nous avons fait une minimisation de LTS pour lesquels le calcul de conformité ne pouvait se faire.

Ce tableau montre que par la minimisation, on obtient un gain d'espace d'états et de temps de calcul considérable. Notons que la minimisation n'est pas proportionnelle au nombre d'états.

3.4 Vérification de conformité versus test de conformité

Il nous paraît important de mettre en évidence l'intérêt de la vérification de conformité par rapport au test de conformité. Initialement, l'objectif des deux approches est similaire : étant donnée une spécification, on souhaite déterminer si le système conçu pour répondre à cette spécification se comporte conformément à ce qui est attendu, et si ce n'est pas le cas, il faut déterminer ce qui doit être corrigé. Le test en général et le test de conformité en particulier offrent une technique fiable pour mesurer et améliorer la confiance que l'on peut avoir de la qualité d'un système. Le test de conformité [Tre99] est basé sur une génération de tests qui sont définis à partir de la spécification du système. Tout d'abord, l'ensemble de tests est nécessairement fini et ne peut être exhaustif. Par conséquent, si le système passe tous les tests avec succès, on ne peut pas garantir qu'il est sans erreur. En revanche, si le système échoue à un test, on peut utiliser ce test pour diagnostiquer la source de l'erreur. Dans l'application à UML, plusieurs travaux ont essayé d'adapter ces techniques de génération de tests de conformité aux machines d'états en définissant des relations de conformité adaptées à UML [TB03, JJ05].

A la différence du test de conformité, la vérification de conformité ne repose pas sur la génération de tests mais sur l'analyse comparative d'un modèle du système avec sa spécification. Cette analyse peut être exhaustive. Dans l'approche de vérification, la méthodologie assure que, lorsqu'un système répond à sa spécification, le système passe avec succès toute séquence de test générée à partir de la spécification. Toutefois, en cas d'échec les algorithmes classiques de calcul de ces relations ne génèrent pas d'informations de diagnostic facilement interprétables. Un traitement *a posteriori* peut être envisagé pour générer des informations aidant au diagnostic [CC92]. Une analyse des traces et des informations telles

que les ensembles d'acceptance et de divergence sert également à déterminer l'origine de l'erreur car elle permet d'identifier les actions qu'une machine doit faire et qui ne sont pas faites par l'autre. L'inconvénient de cette approche est que sa complexité est exponentielle.

3.5 Bilan

Dans ce chapitre, nous avons présenté l'implantation des relations de réduction et d'extension sur des LTS. Nous pouvons calculer ces relations en utilisant la relation de simulation de Milner appliquée aux graphes d'acceptance. Grâce à la fusion des graphes d'acceptance, la vérification de la relation de conformité est effectuée de manière similaire à celle de la relation de réduction.

La vérification nous procure des avantages par rapport à la méthode de test pour vérifier une réalisation par rapport à sa spécification. Cela est utilisable dans plusieurs cas et particulièrement dans la vérification des machines d'états qui sera présentée dans le chapitre suivant. Dans cas où les systèmes de transitions sont de taille trop importante, nous proposons d'utiliser la réduction observationnelle des deux systèmes à vérifier. La vérification sur des LTS réduits permet d'avoir un temps de calcul raisonnable. L'exploitation de ces relations dans le cadre de la construction incrémentale de machines d'états est présentée dans le chapitre suivant.

Chapitre 4

Exploitation des relations de conformité

Sommaire

Nous avons démontré théoriquement dans le chapitre précédent comment vérifier les relations de conformité. Mais comment exploiter ces relations pour aider les modélisateurs à élaborer des machines d'états ? Il importe de montrer que nos propositions ne sont pas uniquement théoriques mais peuvent être mises en œuvre dans le cadre d'une démarche incrémentale. Les relations étant basées sur une analyse des LTS, la première tâche consiste à transformer les machines d'états en LTS. Nous explicitons le principe de transformation et l'outil développé pour son automatisation. Nous présentons ensuite IDCM (*Incremental Development of Conforming Models*), le prototype Java que nous avons développé pour analyser les modèles. Puis nous montrons comment mettre en œuvre les relations de conformité dans le processus incrémental à travers d'une étude de cas portant sur la modélisation comportementale d'un téléphone.

4.1 Transformation de machines d'états UML en LTS

La transformation de machines d'états UML en LTS implique d'associer une sémantique précise aux concepts UML. En conséquence, nous présentons les concepts des ma-

chines d'états UML que nous traitons et leur représentation en LTS. Ensuite, nous donnons des règles de transformation en identifiant des patrons de base et leur correspondance en LTS. Avant de présenter la transformation, nous introduisons le principe de masquage qui est associé aux signaux et méthodes. En fonction de ce principe, nous allons voir que la transformation de la machine d'états en LTS conduira à des transitions internes τ ou à des transitions étiquetées.

4.1.1 Principe de masquage

Le masquage est un mécanisme utilisé dans les algèbres de processus. Il consiste à cacher des actions de l'interface d'un processus. Un processus dont les actions sont masquées ne peut plus se synchroniser sur ces actions. Cela signifie que les actions masquées deviennent anonymes : elles sont remplacées par l'action silencieuse τ (ou i).

L'approche que nous avons choisie pour implanter les relations de conformité est fondée sur la notion de refus : on analyse si une machine se bloque lorsqu'elle est sollicitée par l'environnement, qui lui, est supposé valide. C'est-à-dire que les services offerts par l'environnement (ou requis par la machine) sont toujours disponibles au moment où ils doivent l'être. L'environnement ne peut se bloquer sur une action qu'il est censée offrir. Il est nécessaire de mettre en évidence les interactions entre l'environnement et le processus ou la machine analysée. Les interactions à analyser sont les signaux de l'interface fournie susceptibles de faire l'objet de refus. Puisque l'environnement est supposé valide, les signaux de l'interface requise sont masqués. Les événements implicites (*when* et *after*) et les activités internes (*do activity*), tant qu'elles ne sont pas décrites à leur tour par une machine d'états, sont également supposées valides. Elles sont donc masquées.

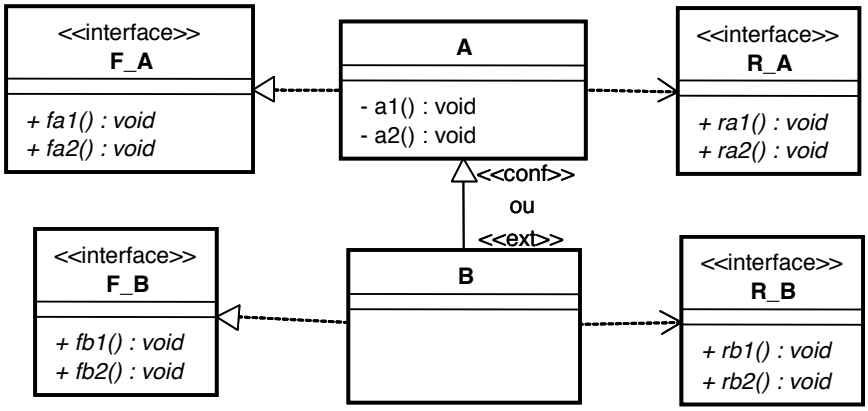


FIGURE 4.1 – Deux classes A, B et leur interface

Les méthodes et signaux des interfaces fournies ne sont pas masquées. Les méthodes privées sont masquées. On peut se trouver dans la situation où la machine à analyser propose des méthodes ou signaux supplémentaires par rapport à la machine de référence.

– Si ces méthodes ou signaux sont publiques ; il s'agit alors d'une interface supplémentaire (cf. figure 4.1). Les *CallEvents* et *SignalEvents* correspondants sont alors visibles. Les relations de comparaison envisagées sont celles d'extension. Il se peut toutefois que la réduction soit vérifiée, ce qui signifiera alors que les nouvelles mé-

thodes ou signaux publiques ne sont jamais accessibles, c'est-à-dire qu'ils sont déclarées dans l'interface sans jamais être utilisés dans la machine d'états.

– Si ces méthodes sont privées ; l'interface n'est pas enrichie. Les relations de comparaison envisagées peuvent être celles de réduction comme d'extension. La réduction s'assurera que l'interface n'a pas été enrichie, mais aussi que les machines sont conformes. L'extension correspondra quant à elle à un calcul de conformité associé à une identité des interfaces.

Dans tous les cas, on cache tous les signaux des interfaces requises du modèle de référence et du modèle à analyser ainsi que les méthodes privées ($ra1, ra2, rb1, rb2, a1, a2$ cf. figure 4.1).

4.1.2 Les règles de transformation

Nous ne prenons en considération qu'un sous-ensemble des concepts de machines d'états UML que nous présentons ci-après.

États Nous considérons seulement une partie des états définis dans le module *BehaviorStateMachines* de la norme UML 2.0. Nous ne prenons pas en compte les états spécifiques du module *ProtocolStateMachines*.

Les états considérés sont les suivants :

– états initiaux, finaux et états simples. La transition sortante de l'état initial est modélisée par une transition interne (cf. cas 0 figure 4.2). Nous détaillerons la modélisation des autres états en fonction du type de leurs transitions sortantes.

– états composites avec une seule région : ils sont mis à plat et les transitions entrantes et sortantes de ces états sont distribuées sur chaque sous-état. Nous ne traitons pas les pseudo-états tels que les états historiques.

Triggers Ils définissent les conditions de changement d'état et sont donc représentés par des transitions LTS. Nous ne considérons pas les transitions avec plusieurs déclencheurs. Nous ne considérons que les types d'événements suivants : *CallEvent, SignalEvent, TimeEvent, ChangeEvent*, et *CompleteEvent*.

– **CallEvent, SignalEvent** : l'étiquette de la transition est déterminée en fonction de la visibilité des méthodes ou des signaux de ce type d'événement. L'étiquette est le nom de la méthode ou du signal s'ils sont visibles et τ dans le cas contraire. Notons que nous ne prenons pas en compte les paramètres des méthodes (cas 1, 2 figure 4.2).

– **TimeEvent** : comme les LTS n'ont pas d'aspect temporisé, n'importe quel événement temporisé (*after, at,...*) des machines d'états sera représenté par une action interne τ (cas 4,5 figure 4.2).

– **ChangeEvent** : nous ne traitons que les événements de changement impliquant des attributs privés. Ils sont cachés et sont représentés par des actions internes τ (cas 4, 5 figure 4.2).

– **CompleteEvent** : ils représentent la fin d'une activité. Ils sont représentés par une action interne τ (cas 3, 6, 8 figure 4.2).

Gardes Pour ce qui est du traitement des gardes, la correspondance n'est pas triviale et il a été nécessaire de définir des règles selon les configurations possibles d'un état. Les gardes représentent des prédicats qui ne peuvent pas être évalués en LTS. Par conséquent, elles sont modélisées par des transitions internes τ. Étant donné que nous n'analysons pas l'ensemble de toutes les gardes des transitions sortantes de l'état, on ne peut savoir si toutes les conditions possibles sont exprimées. Il peut donc y avoir un blocage que nous représentons par une transition interne conduisant à un état puits (cf. cas 6(b) et 8(b) figure 4.2). Si le modélisateur souhaite indiquer que toutes les conditions sont exprimées, il devra utiliser la garde *else* pour la transition par défaut (cf. cas 6(a) et 8(a) figure 4.2). Notons que lorsque la garde est associée à un événement (cf. cas 7, 9 figure 4.2), on ne modélise pas l'état puits, parce que la garde sera évaluée chaque fois que l'événement se produira. Par conséquent, on modélise par des transtions internes le fait que l'on puisse sortir de l'état (ce qui correspond à la garde vraie) ou le fait que l'on reste dans le même état (ce qui correspond à la garde fausse).

Actions Étant donné que les actions se traduisent par des méthodes de l'interface requise ou des méthodes privées, elles sont marquées par des actions internes τ (cas 2, 3, 5, 8, 9 figure 4.2, en remplaçant *action1* par τ).

Activités Comme mentionné dans le chapitre relatif à la sémantique des machines d'états, les activités représentent un comportement associé à un état. Elles peuvent être effectuées en entrant dans l'état, en quittant l'état ou pendant la durée de vie de l'état. Elles sont considérées comme une séquence d'actions atomiques, en boucle infinie ou non, qui peut être interrompue par un trigger. Étant donné que les activités se traduisent par des méthodes internes ou des méthodes de l'interface requise, conformément au principe de masquage, elles sont cachées et sont représentées par des actions internes LTS τ. Tout état auquel une activité est associée sera représenté par un état avec une boucle de transition interne (cf. figure 4.3).

On distingue deux types d'activité dans le cadre des transformations en LTS.

- Le premier type, stéréotypé *NoExit* (cf. figure 4.3(a)) représente une activité en boucle infinie interruptible par l'occurrence d'un événement. Dans ce cas, on ne greffe que les transitions sortantes qui sont susceptibles d'interrompre l'activité (**CallEvent** cas 1, 2, 7, 9 et **ChangeEvent** ou **TimeEvent** 4, 5). Si d'autres types de transition sont détectés, un message d'erreur est adressé au modélisateur pour signaler l'incohérence.
- Dans le deuxième cas qui correspond à une activité avec terminaison (**CompleteEvent** cas 3, 6, 8), on greffe les transitions correspondant au **CompleteEvent** sur l'état *greffe3* (cf. figure 4.3(b)) et celles de l'interruption sur l'état *greffe2* (cf. figure 4.3(b)).

Le schéma de synthèse de traduction des concepts des machines d'états UML est donné dans la figure 4.1. Remarquons qu'il n'y a pas de correspondance bijective entre les états UML et ceux du LTS.

Notons que la transformation est compositionnelle : l'application des règles permet de traiter plusieurs transitions sortantes d'un même état. Nous donnons des illustrations de ces combinaisons dans l'annexe B.

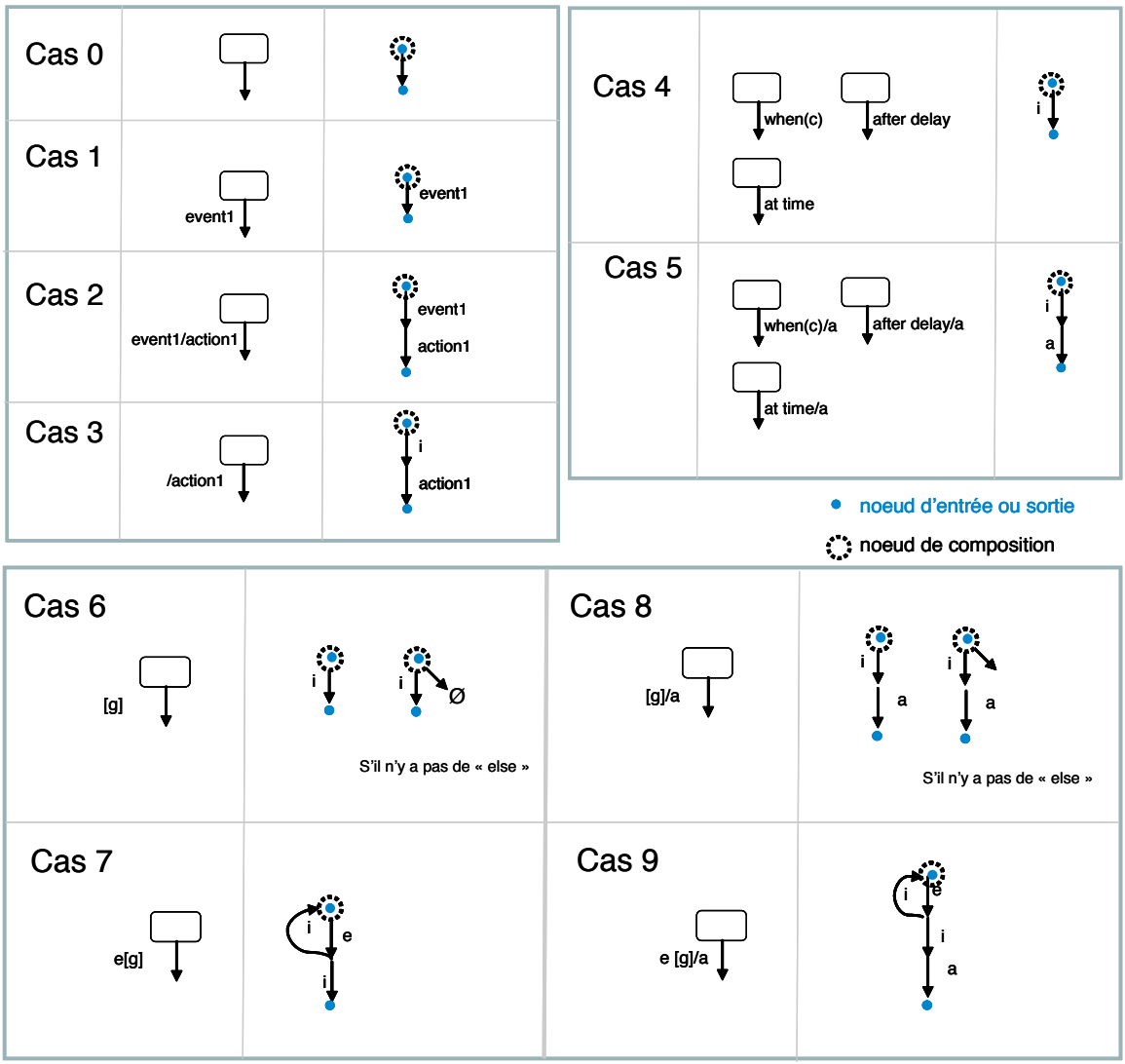


FIGURE 4.2 – Cas de transformation d'UML en LTS

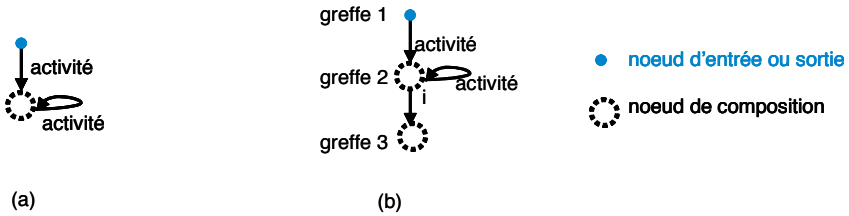


FIGURE 4.3 – Modélisation d'une activité (a) sans terminaison, (b) avec terminaison

Machines d'états	LTS	Machines d'états	LTS
action	τ	entry et exit	τ
Call ou SignalEvent	action ou τ	état composite	LTS à plat
Time, Change, CompleteEvent	τ	historique	non traité
gardes	τ	pseudo-état de choix	non traité
activité	τ		

TABLE 4.1 – Correspondance de concepts

4.1.3 Mise en œuvre des règles pour la transformation automatique

Il est nécessaire d'avoir un outil de traduction automatique des machines d'états UML en LTS en appliquant les règles que nous avons proposées. Avant d'implanter un outil automatique, nous avons étudié plusieurs approches :

- Utiliser un *atelier de transformation* (ATL, Kermeta [JK06]). Nous avons étudié la mise en œuvre sous ATL et nous avons été confrontés à des problèmes techniques (en particulier dans l'écriture de Helper) rendant impossible l'automatisation de la transformation.
- Développer un *parseur* pour analyser la syntaxe en utilisant des bibliothèques spécialisées (SAX ou DOM). Étant donnée la nature des règles et des combinaisons (cf. annexe B), nous avons réalisé un prototype en utilisant la bibliothèque DOM pour automatiser la transformation. Le fichier généré qui décrit le LTS est au format .aut qui est le format textuel utilisé dans l'environnement CADP. Nous pouvons ainsi utiliser les outils de vérification de minimisation et de visualisation de CADP.

4.2 IDCM – Prototype Java

Le prototype JAVA IDCM (*Incemental Development of Conforming Models*) que nous avons développé permet de calculer plusieurs types de relations :

- Relations de préordre : extension ext, réduction red, simulation forte, *Must*, préordre de Test, confrestr, red*.
- Relations d'équivalence : bisimulation, équivalence de test, équivalence de conformité.
- Relation de conformité conf.

Il contient 20 classes et 4214 lignes de code. Nous ne présentons ici que les classes principales relatives à l'implantation des LTS et des graphes d'acceptance (cf. figure 4.4) :

- La classe *LTS* implante un LTS comme un ensemble d'états et un ensemble de transitions. Les états sont des instances de la classe *State*. Les transitions sont des instances de la classe *Transition*. Comme nous l'avons démontré dans le chapitre précédent et dans [LLC08a], les relations red et ext font appel à la relation de simulation (fonction *isStrongSimulatedBy*).
- La classe *Agraph* implante le graphe d'acceptance associé à un LTS. Elle est elle-même un LTS dont les états sont de type *AState* défini comme une sous-classe de *State*.

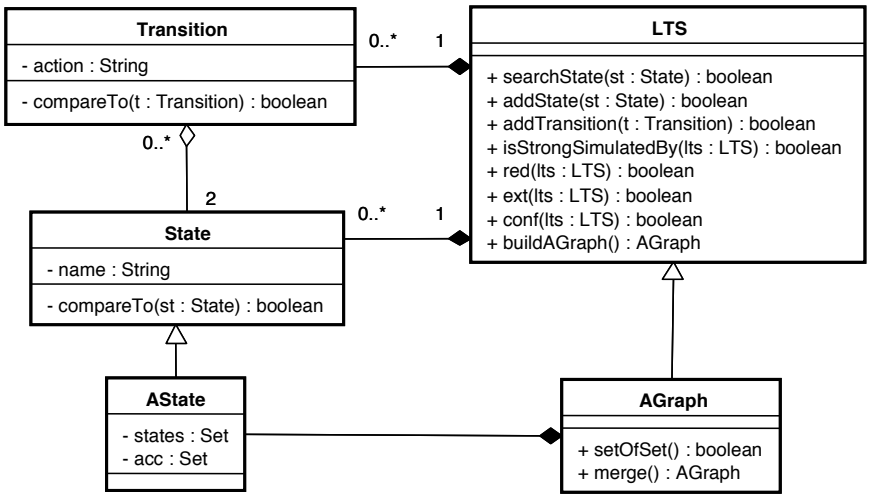


FIGURE 4.4 – Diagramme de classe du prototype IDCM

L'attribut *states* de la classe *AState* définit la liste des états associés dans le LTS. Cet attribut permet d'établir la correspondance entre les nœuds du graphe d'acceptance et ceux du LTS. Un autre attribut fondamental associé à tout nœud de type *AState* est *acc*, son ensemble d'acceptance défini comme un ensemble d'ensembles d'actions.

Conformément à l'algorithme 1 [p 61] dans le chapitre 3, la fonction *conf* est mise en œuvre en trois étapes :

- construction des graphes d'acceptance associés au LTS de référence (la spécification) et au LTS à analyser (l'implantation),
- fusion des graphes d'acceptance,
- enfin, calcul de la relation de réduction entre le graphe d'acceptance du LTS de référence et le graphe d'acceptance de la fusion.

Les paragraphes suivants montrent les résultats obtenus par le prototype JAVA sur une étude de cas.

4.3 Étude de cas et analyses

L'étude de cas porte sur la modélisation d'un téléphone en trois étapes :

- Modélisation de la spécification d'un téléphone simple *PhoneSpec*,
- Modélisation d'une implantation *Phone*,
- Modélisation de la spécification d'un téléphone à double appel *DoubleCallSpec*.

Après avoir présenté les machines d'états élaborées à chaque étape, nous montrons quelles erreurs peuvent être mises en évidence grâce au calcul des relations de conformité.

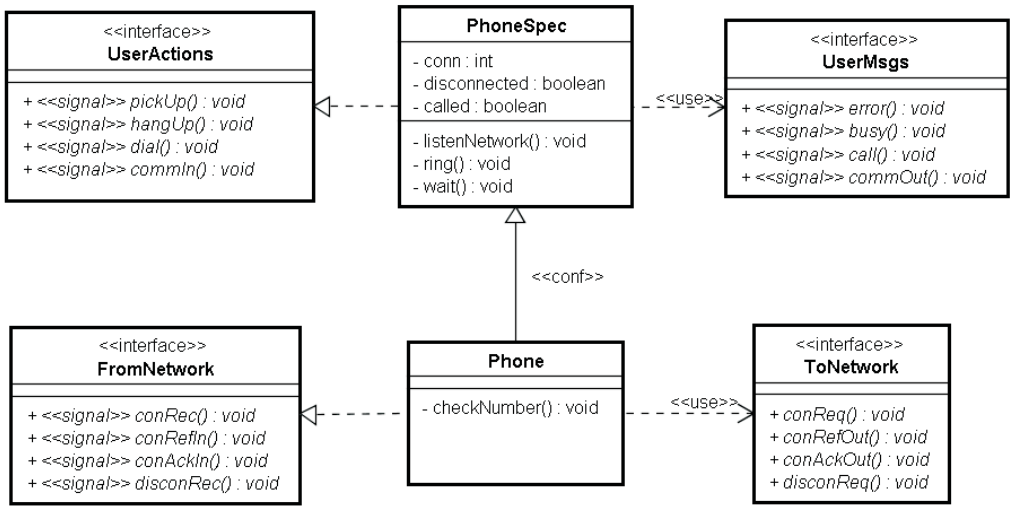


FIGURE 4.5 – Classes *PhoneSpec* et *Phone*, avec leurs interfaces fournies et requises

4.3.1 Étude de cas d'un téléphone

Étape 1 : Modélisation d'une spécification de téléphone simple

On définit la classe *PhoneSpec* par rapport au point de vue de l'utilisateur. L'interface fournie *UserActions* de *PhoneSpec* (cf. figure 4.5) est composée de signaux envoyés par l'utilisateur : *hangUp*, *pickUp*, *dial* et *commIn* (communication entrante). L'interface requise (classe *UserMsgs*) est composée de signaux envoyés par le téléphone à l'utilisateur : *error*, *busy*, *call* et *commOut*. Nous avons défini toutes les interactions sous forme de signaux et ne détaillons pas les informations échangées comme nous aurions pu le faire avec des méthodes.

Le comportement de *PhoneSpec* est spécifié par la machine d'états $SM_{PhoneSpec}$ représentée dans la figure 4.6. Elle décrit deux situations :

- L'utilisateur fait un appel (partie gauche de $SM_{PhoneSpec}$). Les actions possibles sont alors : décrocher (*pickUp*) puis numéroter (*dial*). Si la ligne est libre et le correspondant décroche, la machine passe dans l'état *Connected* pendant lequel les correspondants peuvent communiquer. Dans le cas contraire (erreur ou ligne occupée), l'utilisateur ne peut que raccrocher (*hangUp*).
- L'utilisateur est appelé (partie droite de $SM_{PhoneSpec}$). S'il décroche, la machine passe dans l'état *Connected* comme précédemment. Sinon, la machine reviendra dans l'état initial *Idle* lorsque le correspondant arrêtera son appel.

Notons que cette modélisation se focalise sur les actions possibles de l'utilisateur et sur les signaux émis par le téléphone à l'utilisateur (par exemple : tonalité d'erreur *error*, ou de ligne occupée *busy*). En revanche, elle fait abstraction des activités du téléphone liées au réseau en les modélisant par des méthodes privées (*do/wait*, *do/listenNetwork*, *do/ring*, *do/checkNetwork*). Le résultat de ces activités est modélisé par des variables internes *conn*, *called*, *disconnected*.

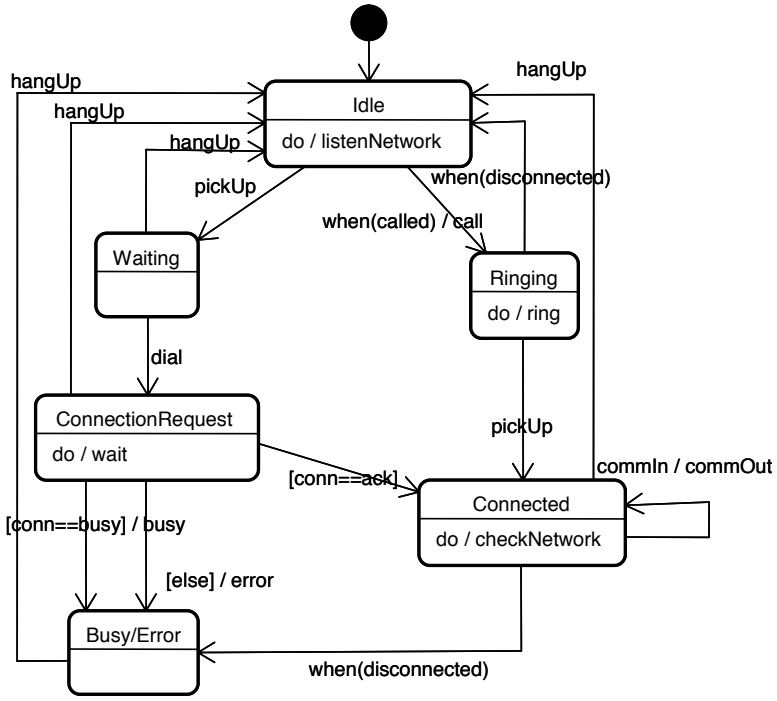


FIGURE 4.6 – Machine d'états de *PhoneSpec*

Étape 2 : Modélisation d'une implantation de téléphone simple

On souhaite modéliser une implantation du téléphone qui soit conforme à la spécification *PhoneSpec*. On définit la classe *Phone* qui hérite de la classe *PhoneSpec*. Elle sera cependant plus précise et les activités liées au réseau dont on a fait abstraction dans la spécification devront être explicitées. Ceci est réalisé par la définition d'une interface requise qui regroupe l'ensemble des signaux émis par le téléphone vers le réseau (*ToNetwork*) et une interface fournie qui regroupe les signaux émis par le réseau vers le téléphone (*FromNetwork*).

Le comportement de cette classe est décrit par la machine d'états SM_{Phone} (figure 4.7). Elle diffère de la classe *Phone* par le fait qu'elle fait référence aux signaux émis entre le téléphone et le réseau. Par exemple, lorsque le correspondant raccroche, ce n'est plus la variable interne *disconnected* qui est utilisée pour décrire la fin de la communication (cf. transition *when(disconnected)*, figure 4.6) en sortie des états *Connected* et *Ringing* mais un signal émis par le réseau (cf. transition *disconRec*, figure 4.7).

Étape 3 : Modélisation d'un téléphone avec double appel

Dans cette partie, notre objectif est d'étendre les fonctionnalités de la spécification du téléphone simple afin de modéliser un téléphone acceptant un second appel pendant que l'utilisateur est en ligne. Pour cela, nous définissons une nouvelle classe nommée *DoubleCall* (cf. figure 4.8) qui est censée être une spécialisation de *PhoneSpec*. On définit une nouvelle interface fournie par la classe *DoubleCall* définissant de nouveaux signaux émis par l'utilisateur afin de modélier s'il accepte (*accept*), rejette ou arrête (*stop*) le deuxième appel.

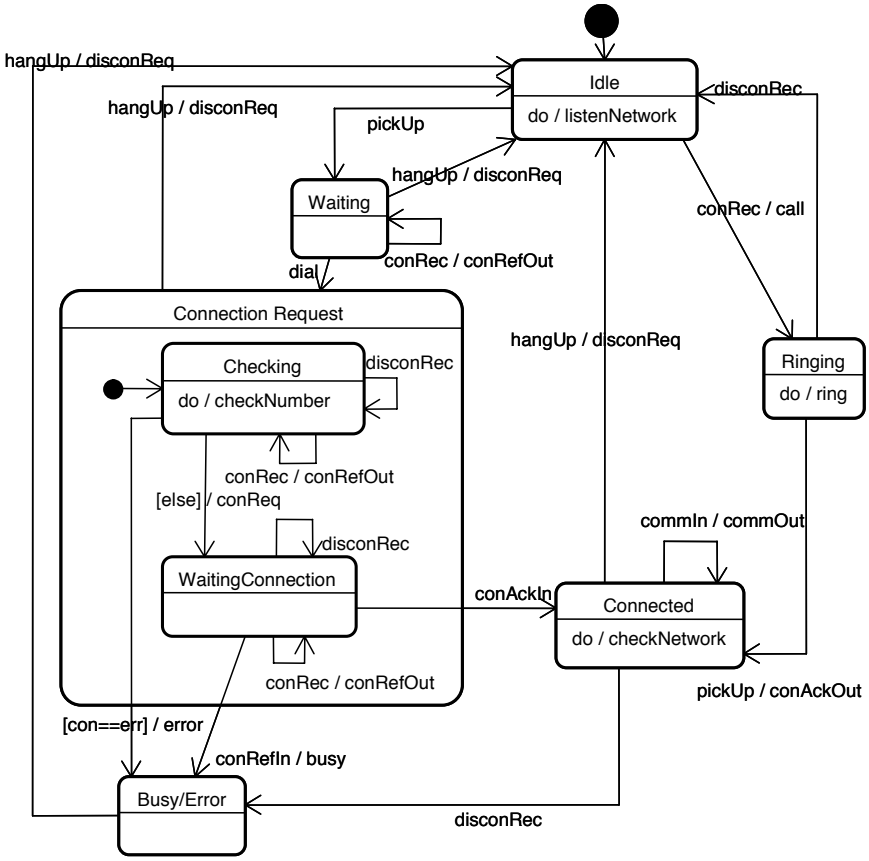


FIGURE 4.7 – Machine d'états de *Phone*

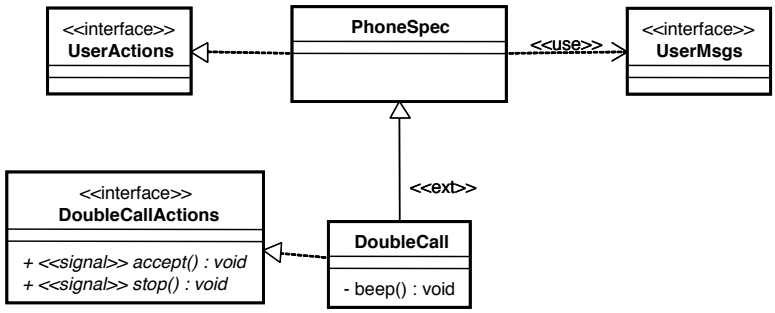


FIGURE 4.8 – Classe de spécification de double-appel, avec l'interface fournie et requise

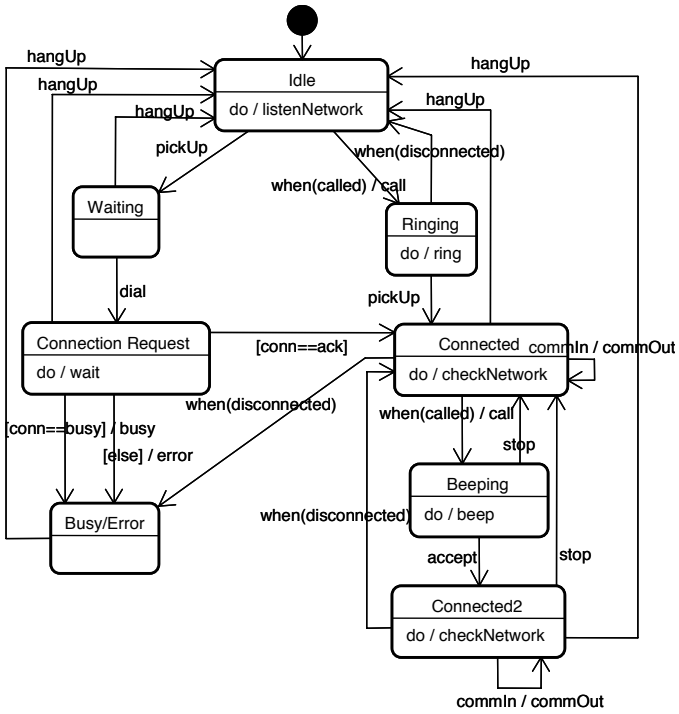


FIGURE 4.9 – Machine d'états de *DoubleCallSpec*

La figure 4.9 représente la machine d'états de cette nouvelle spécification, appelée $SM_{DoubleCallSpec}$. La différence avec la première spécification se trouve sur l'état *Connected* : une transition *when*(*called*)/*call* a été ajoutée pour modéliser le deuxième appel. Le nouvel état *Beeping* représente la situation où l'utilisateur décide d'accepter le deuxième appel ou de le refuser. S'il accepte, la machine se trouve dans l'état *Connected2*, sinon elle revient à l'état *Connected* par la transition *stop*. Lorsque le second appel prend fin (transitions *when*(*disconneted* ou *stop* en sortie de l'état *Connected2*), la machine revient dans l'état *Connected*.

4.3.2 Analyse et correction des modèles

La vérification des relations d'extension et de conformité entre deux machines d'états repose sur trois étapes (cf. figure 4.10) :

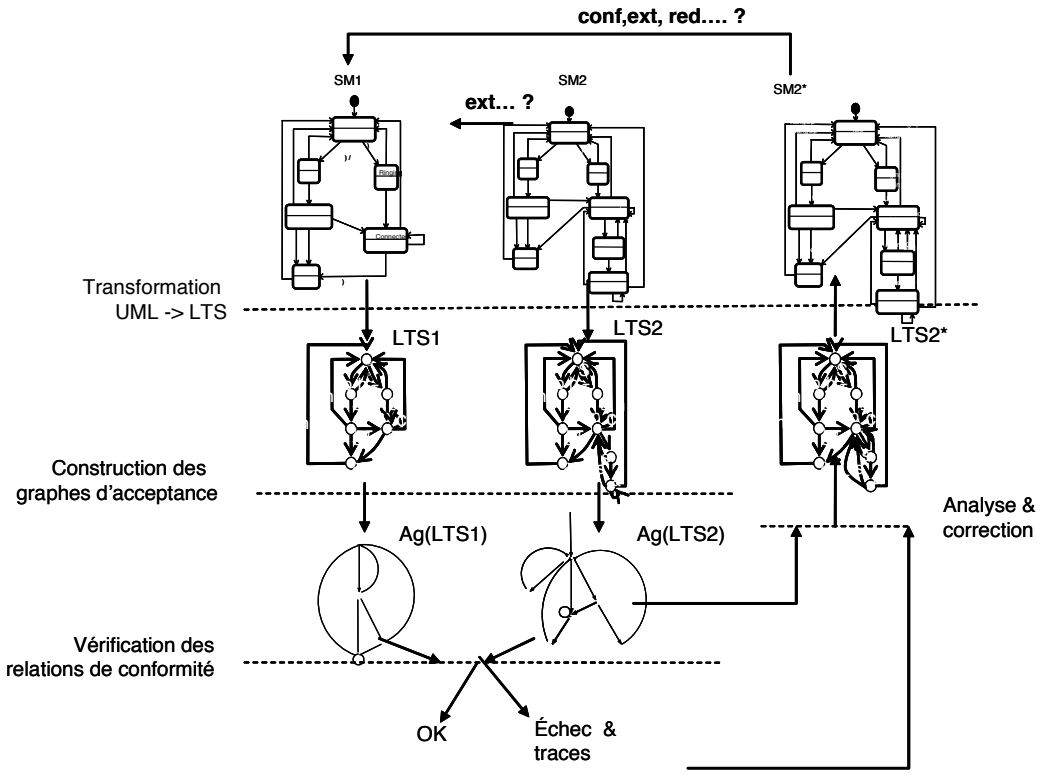


FIGURE 4.10 – La démarche de calcul des relations de conformité

- leur transformation en LTS ;
- la construction automatique des graphes d'acceptance associés aux LTS ;
- la vérification de la relation d'extension entre les graphes d'acceptance basée sur une relation de simulation et l'inclusion des ensembles d'acceptance.

La vérification de la relation de conformité inclut une étape supplémentaire de fusion des graphes d'acceptance [LLC08a, LLC08b].

Générations des LTS

La génération des LTS à partir des machines d'états s'effectue automatiquement par l'outil que nous avons développé conformément aux règles de transformation présentées dans la section 4.1. Trois LTS sont générés respectivement à partir des trois machines d'états *PhoneSpec*, *Phone*, *DoubleCallSpec*. Comme mentionné dans le chapitre précédent, nous pouvons minimiser ces LTS par l'outil CADP pour faciliter les calculs. La vérification de relation d'extension entre des LTS minimisés sera alors calculée.

Conformément aux règles de visibilité, les signaux des interfaces liées au réseau seront masqués et transformés en transitions internes τ dans le LTS. L'environnement BCG de CADP permet de générer l'image à partir de fichiers au format .aut. Dans ce format, les transitions internes sont marquées par i.

Après minimisation, les LTS de *PhoneSpec* et *Phone* sont identiques (cf. figure 4.12).

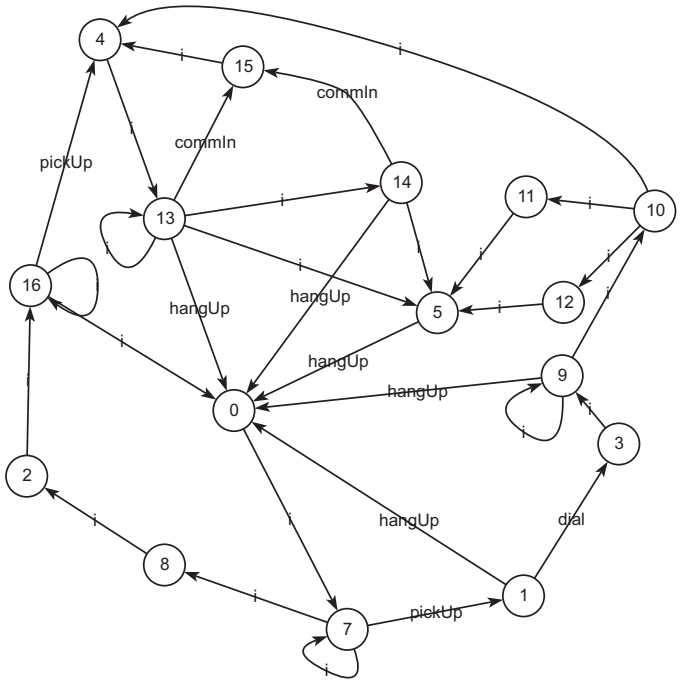


FIGURE 4.11 – Le LTS de *PhoneSpec* non minimisé

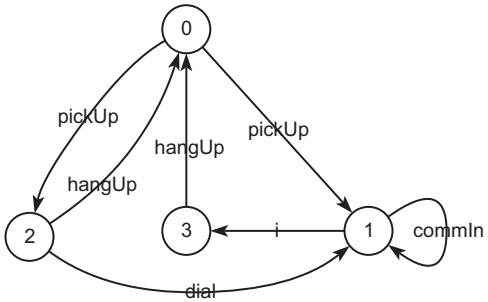
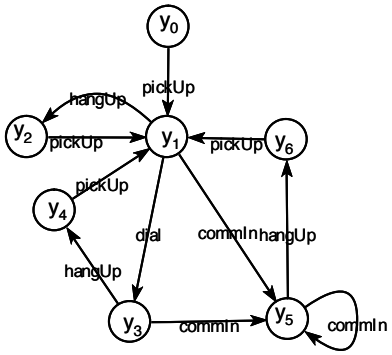


FIGURE 4.12 – Le LTS de *PhoneSpec* et *Phone* minimisé au sens de l'équivalence observationnelle

Analyse des classes *PhoneSpec* et *Phone* par la relation de conformité

Nous nous intéressons dans ce paragraphe à la relation entre la machine d'états de *PhoneSpec* représentant la spécification de haut niveau et celle de *Phone* modélisant une implantation possible.



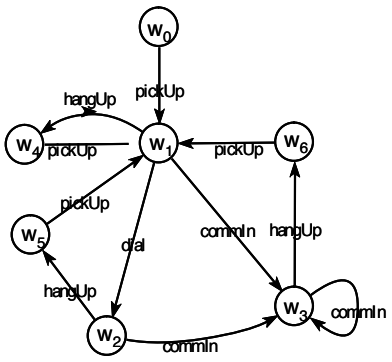
État	États associés de LTS_{Phone}	Ensemble d'acceptance
y_0	0, 2, 25, 8, 9	$\{\{pickUp\}\}$
y_1	1, 3, 4, 21, 20, 26	$\{\{commIn, hangUp\}, \{dial, hangUp\}, \{hangUp\}\}$
y_2	0, 2, 22, 25, 8, 24, 9, 10	$\{\{pickUp\}\}$
y_3	3, 4, 6, 7, 11, 12, 13, 14, 17, 16, 18, 21, 20	$\{\{commIn, hangUp\}, \{hangUp\}\}$
y_4	0, 19, 2, 22, 25, 8, 24, 9, 15	$\{\{pickUp\}\}$
y_5	3, 4, 6, 7, 11, 12, 13, 14, 17, 16, 18, 21, 20	$\{\{commIn, hangUp\}, \{hangUp\}\}$
y_6	0, 2, 22, 25, 8, 24, 9, 24, 9, 15	$\{\{pickUp\}\}$

FIGURE 4.13 – $Graphe\mathcal{A}(LTS_{Phone})$ et tableau de correspondance entre états associés et ensemble d'acceptance

Après avoir transformé les machines d'états en LTS et construit leur graphe d'acceptance $\mathcal{A}(LTS_{PhoneSpec})$ et $\mathcal{A}(LTS_{Phone})$, on construit le graphe fusionné $\mathcal{A}(Merge(LTS_{PhoneSpec}, LTS_{Phone}))$. La vérification de conformité consiste en deux étapes :

i. d'une part la vérification de la relation de simulation entre le graphe d'acceptance de LTS_{Phone} (cf. figure 4.13) et le graphe fusionné (cf. figure 4.14) ;

ii. et d'autre part, la vérification de l'inclusion des ensembles d'acceptance des états simulés.



État	Ensemble d'acceptance
w_0	$\{\{pickUp\}\}$
w_1	$\{\{dial, hangUp\}, \{hangUp\}, \{commIn, hangUp\}, \{dial, commIn, hangUp\}\}$
w_2	$\{\{hangUp\}, \{commIn, hangUp\}\}$
w_3	$\{\{hangUp\}, \{commIn, hangUp\}\}$
w_4	$\{\{pickUp\}\}$
w_5	$\{\{pickUp\}\}$
w_6	$\{\{pickUp\}\}$

FIGURE 4.14 – Graphe $\mathcal{A}(Merge(LTS_{PhoneSpec}, LTS_{Phone}))$ et tableau de correspondance entre états associés et ensemble d'acceptance

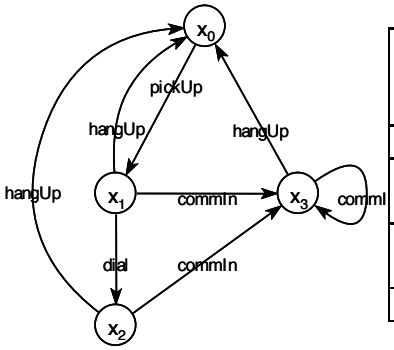
Le résultat de la première étape est positif et s'exprime par l'ensemble de paires d'états simulés suivant : $\{(y_0, w_0), (y_1, w_1), (y_2, w_4), (y_3, w_2), (y_4, w_5), (y_5, w_3), (y_6, w_6)\}$.

La seconde étape consiste à vérifier que l'ensemble d'acceptance de chaque nœud de $\mathcal{A}(LTS_{Phone})$ est inclus dans l'ensemble d'acceptance du nœud qui le simule dans $\mathcal{A}(Merge($ $LTS_{PhoneSpec}$, $LTS_{Phone}))$. Par exemple $w_3 \gtrsim y_5$ et $y_5.acc \subset\subset w_3.acc$. Cette propriété est vérifiée par l'analyse des ensembles d'acceptance donnés dans les tableaux des figures 4.13 et 4.14.

La relation de conformité est vérifiée : le comportement de *Phone* est donc conforme à sa spécification *PhoneSpec*.

Analyse des classes *PhoneSpec* et *DoubleCallSpec* par la relation d'extension

Nous nous intéressons maintenant à la relation existant entre la classe *PhoneSpec*, représentant un téléphone simple appel, et son extension en classe *DoubleCallSpec*, représentant un téléphone double appel.



État	États associés de $LTS_{PhoneSpec}$	Ensemble d'acceptance
x_0	0, 16, 2, 7, 8	$\{\{pickUp\}\}$
x_1	1, 4, 5, 13, 14	$\{\{hangUp, commIn\}, \{dial, hangUp\}, \{hangUp\}\}$
x_2	3, 4, 5, 9, 10, 11, 12, 13, 14	$\{\{hangUp, commIn\}, \{hangUp\}\}$
x_3	4, 5, 13, 14, 15	$\{\{hangUp, commIn\}, \{hangUp\}\}$

FIGURE 4.15 – Graphe $\mathcal{A}(LTS_{PhoneSpec})$ et tableau de correspondance entre états associés et ensemble d'acceptance

Après avoir transformé la machine d'états *DoubleCallSpec* en LTS, on construit son graphe d'acceptance $\mathcal{A}(LTS_{DoubleCallSpec})$. La vérification de la relation d'extension consiste en deux étapes : la vérification de la relation de simulation entre les graphes d'acceptance et la vérification de l'inclusion des ensembles d'acceptance des nœuds simulés. Le résultat de la première étape est positif et s'exprime par l'ensemble des paires d'états suivant : $(x_3, z_2), (x_2, z_6), (x_1, z_1), (x_0, z_0)$.

En revanche, l'inclusion des ensembles d'acceptance n'est pas respectée. Il n'y a donc pas extension. L'échec est mis en évidence par l'outil sur le couple (x_3, z_2) après les traces *pickUp ; commIn* ou *pickUp ;dial ;commIn* car l'ensemble $\{stop, accept\}$ n'est pas inclue dans l'ensemble d'acceptance $\{\{hangUp, commIn\}, \{hangUp\}\}$. Il apparaît qu'après la trace *pickUp ; commIn*, $SM_{DoubleCallSpec}$ peut refuser l'action *hangUp* alors que $SM_{PhoneSpec}$ ne peut pas la refuser. Lors d'un nouvel appel entrant, l'utilisateur est conduit à ne pas pouvoir raccrocher ni communiquer.

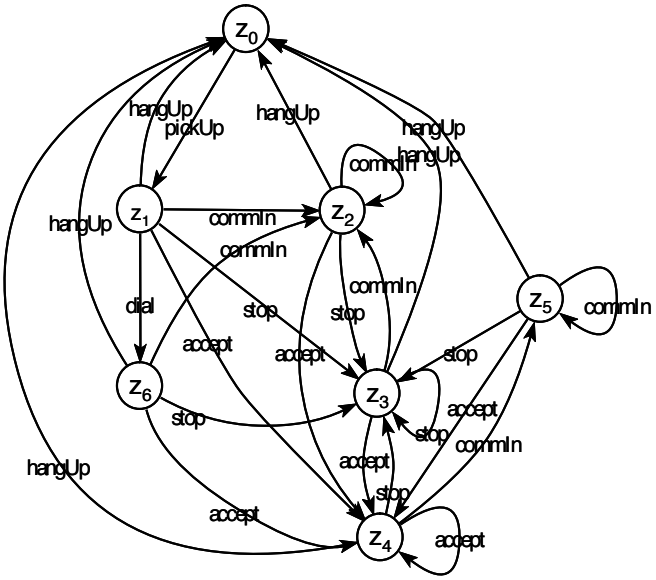


FIGURE 4.16 – Graphe $\mathcal{A}(LTS_{DoubleCallSpec})$

État	États associés $LTS_{DoubleCallSpec}$	Ensemble d'acceptance
z_0	0, 2, 23, 9, 10	$\{\{pickUp\}\}$
z_1	17, 16, 1, 19, 4, 5, 7, 15	$\{\{dial, hangUp\}, \{accept, stop\}, \{accept, stop, commIn, hangUp\}, \{hangUp\}\}$
z_2	17, 16, 19, 18, 4, 5, 7, 15	$\{\{accept, stop, commIn, hangUp\}, \{accept, stop\}, \{hangUp\}\}$
z_3	17, 16, 19, 4, 5, 7, 15	$\{\{accept, stop, commIn, hangUp\}, \{accept, stop\}, \{hangUp\}\}$
z_4	17, 16, 19, 21, 4, 20, 5, 7, 8, 15	$\{\{stop, accept\}, \{hangUp\}, \{stop, accept, commIn, hangUp\}\}$
z_5	17, 16, 19, 18, 21, 4, 20, 5, 22, 7, 8, 15	$\{\{stop, accept\}, \{hangUp\}, \{stop, accept, commIn, hangUp\}\}$
z_6	17, 16, 19, 3, 4, 5, 7, 11, 12, 13, 14, 15	$\{\{stop, accept\}, \{hangUp\}, \{stop, accept, commIn, hangUp\}\}$

TABLE 4.2 – Tableau de correspondance entre états associés et ensemble d'acceptance

Cette analyse est possible car l'outil met automatiquement en évidence les actions que la machine raffinée peut refuser en correspondance avec celles que la machine initiale doit accepter. Une correction possible consiste à ajouter une transition de type **TimeEvent** (*after*(10s)) sortant de l'état *Beeping* vers l'état *Connected* (c.f figure 4.17). Avec cette modification, la relation d'extension entre les deux machines est vérifiée.

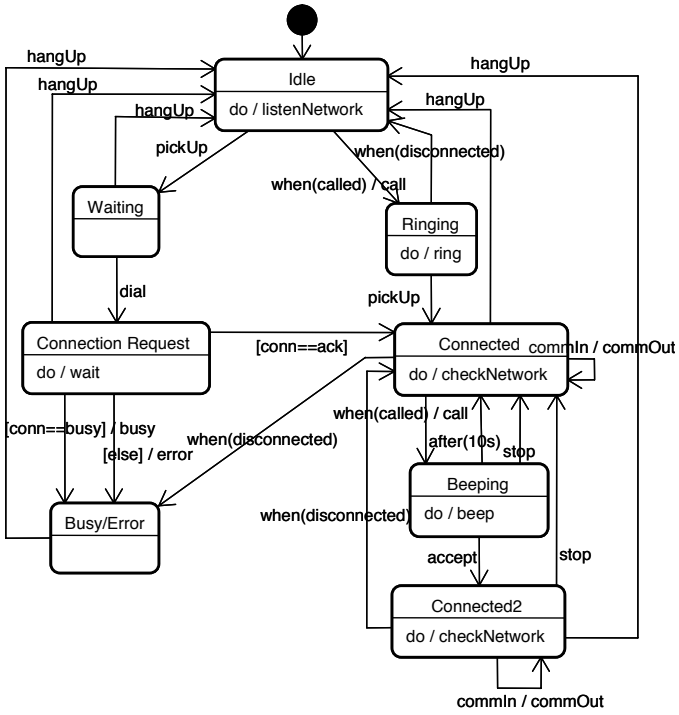


FIGURE 4.17 – $SM^*_{DoubleCallSpec}$, extension de $SM_{PhoneSpec}$

4.3.3 Bilan de l'étude de cas

Grâce à l'outil de vérification que nous avons développé, aucune erreur, au sens de la conformité des LTS, n'est détectée sur la machine d'états *Phone*. En revanche, nous avons pu détecter et corriger des erreurs sur la machine d'états de la classe *DoubleCallSpec*. Les causes d'échec sont mises en évidence de façon explicite par la donnée d'une (ou de plusieurs) trace(s) défaillante(s) ainsi que par la liste des actions oubliées qui ont conduit au non respect de l'inclusion des ensembles d'acceptance. Les causes d'échec peuvent donc être analysées aisément en termes d'actions, ce qui facilite la correction des machines d'états.

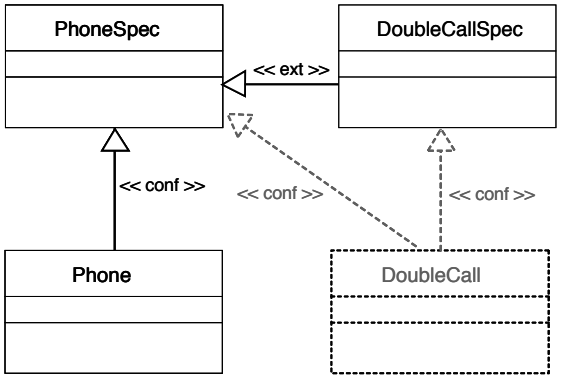


FIGURE 4.18 – Relations entre les spécifications *PhoneSpec* et *DoubleCallSpec* et leur implantation

L'étape suivante de modélisation consisterait à définir *DoubleCall*, la classe d'implantation de *DoubleCallSpec* et à vérifier sa conformité. Nous avons vu dans le chapitre 1 que la relation d'extension satisfait la propriété de raffinement : si *DoubleCall* est conforme à sa spécification, elle est aussi conforme à la spécification initiale *PhoneSpec* du téléphone. Ces résultats sont résumés dans la figure 4.18.

4.3.4 Interprétation des résultats

L'analyse de conformité étant basée sur les LTS, on peut légitimement se demander comment interpréter les résultats sur les machines d'états. Au sens de la conformité entre LTS (absence de refus définitifs), les machines sont conformes sans considérer les données. Il peut néanmoins subsister des blocages dus aux données (blocages provisoires ou conditionnels) que nous ne sommes pas en mesure de détecter. Il est donc nécessaire d'interpréter avec prudence les résultats obtenus sur les LTS associés aux machines d'états. L'exemple de la figure 4.19 illustre notre propos. La machine SM_2 ne peut pas accepter d après la trace ac car la garde $[x > 5]$ est contradictoire avec la garde $[else]$ qui la précède dans le chemin d'exécution, alors que SM_1 accepte toujours d. Étant donné que dans les LTS on fait abstraction des données, on modélise qu'après la trace ac, le LTS associé à SM_2 pourra toujours accepter d à un moment donné. Par conséquent, la relation de conformité entre les deux LTS associés à ces deux machines d'états est vérifiée.

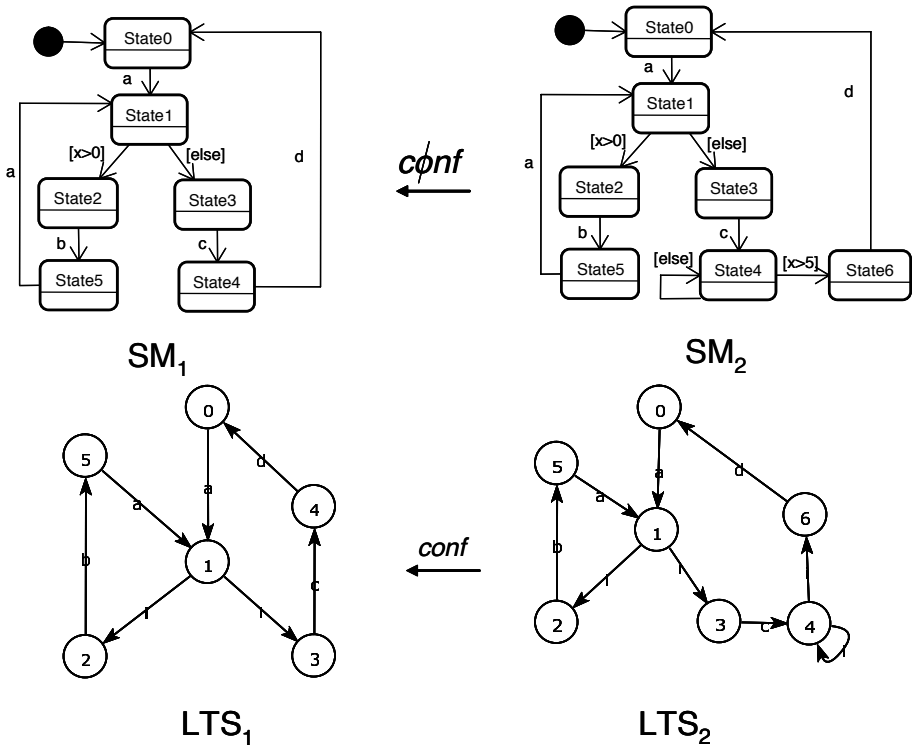


FIGURE 4.19 – Non conformité entre machines d'états alors qu'il y a conformité entre leur LTS associé

Des cas similaires de non conformité sur des machines d'états, alors qu'il y aurait

conformité sur les LTS associés peuvent être cités :

- événement **ChangeEvent** inconsistant, par exemple *when(false)* ou irréalisable, par exemple *when(altitude>100000)* alors que le domaine de valeurs d'*altitude* est [0..10000].

- événement temporel irréalisable, par exemple *after(3 ans)*.

De même, on peut citer des cas de non conformité de LTS, alors qu'il y aurait conformité sur les machines d'états. Ce sont des cas particuliers où les traces impliquant la non conformité sur des LTS ont été éliminées sur la machine d'états car elles correspondent à des traces irréalisables. L'exemple de la figure 4.20 illustre ce cas : le **ChangeEvent** *when(x<0)* de la machine SM_1 a été supprimé car le concepteur sait par exemple que le domaine de valeur de x est positif. Par conséquent, SM_2 est conforme à SM_1. En revanche, les LTS ne le sont pas.

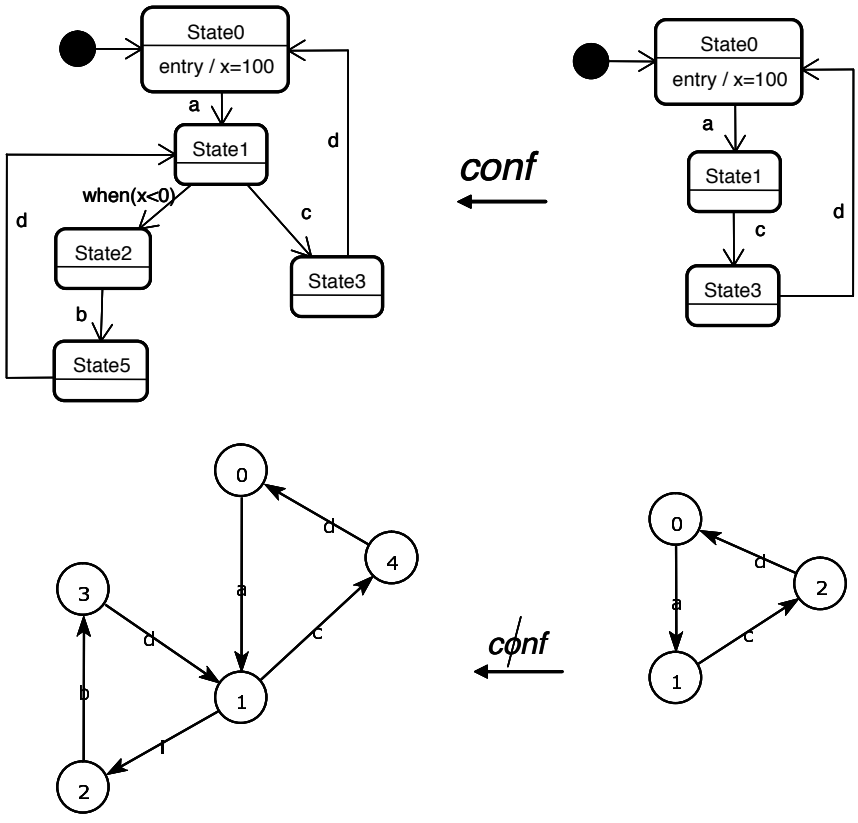


FIGURE 4.20 – Non conformité entre LTS alors qu'il y a conformité entre les machines d'états correspondantes

Ces exemples montrent qu'il faut interpréter les résultats obtenus sur des LTS avec prudence. Notons que les cas mentionnés sont très particuliers et une analyse de données préliminaires permettrait de les identifier.

4.4 Bilan

Dans ce chapitre, nous avons montré comment exploiter les relations de conformité afin d'aider les modélisateurs à construire des modèles corrects. Afin de pouvoir comparer des machines d'états UML, nous avons systématisé leur traduction en LTS. La traduction des machines d'états en LTS est effectuée par un prototype JAVA en utilisant la bibliothèque DOM de JAVA [Sun10]. Pour l'instant, nous avons considéré un sous-ensemble des machines d'états. Les règles doivent être étendues afin de prendre en compte l'ensemble des concepts des machines d'états UML. Grâce à notre outil IDCM qui permet d'automatiser la vérification des relations de conformité, nous pouvons identifier les cas d'erreur mis en évidence et guider le concepteur pour corriger les modèles UML. La mise en œuvre de cette démarche est illustrée sur un cas d'étude de téléphone. Ceci permet de donner des évaluations intermédiaires, en phase de développement et de mettre en évidence des risques de non conformité.

Du fait de la différence du niveau d'abstraction entre LTS et machines d'états, nous ne pouvons que mettre en garde le concepteur sur des risques de non conformité. Il serait nécessaire d'utiliser un outil complémentaire pour faire une analyse de données et de conditions logiques ou temporelles qui sont irréalisables ou incohérentes afin de détecter sur des machines d'états des blocages (états puits) ou plus généralement des incohérences.

Nous avons donné une illustration de la mise en œuvre des relations de conformité ext, conf à travers une étude de cas dans laquelle ext est considéré comme une relation de raffinement et conf comme une relation d'implantation.

Dans le chapitre suivant, nous allons définir un cadre formel de mise en œuvre de ces relations dans le contexte d'une construction incrémentale.

Chapitre 5

Vers la formalisation d'un cadre de construction incrémentale

Sommaire

Dans le chapitre précédent, nous avons montré que la relation ext est une relation adéquate pour le raffinement car elle présente les critères recherchés : réduction de blocages, réduction de l'indéterminisme, extension de traces et transitivité. De plus, ext répond également à la propriété de raffinement énoncée par Leduc [Led91a] :

$$q \ \mathsf{raf}\ p \Rightarrow (\{r \mid r \ \mathsf{imp}\ q\} \subseteq \{r \mid r \ \mathsf{imp}\ p\}) \tag{5.1}$$

où raf est une relation de raffinement et imp est une relation d'implantation. La relation ext satisfait cette propriété dans le cas où la relation de conformité conf est choisie comme relation d'implantation. Cette propriété établit que ext est une relation de raffinement, mais ce n'est pas la plus large dans le sens où il existe des relations de raffinement qui ne sont pas des extensions.

Une première perspective consiste à trouver une caractérisation de la plus grande relation de raffinement ainsi qu'une implantation de la vérification de cette relation. Ensuite, dans ce chapitre, nous proposons de formaliser la démarche de construction incrémentale des modèles assurant la cohérence entre différentes versions de spécification et leur implantation. De ce fait, on propose un support de vérification de modèles adapté aussi bien

aux méthodes conventionnelles de développement qu'aux méthodes non-linéaires comme le prototypage rapide ou *l'extreme programming* .

5.1 Vers le raffinement : la relation confrestr

Relation d'implantation

D'abord, nous faisons un rappel de la notion de relations d'implantation. Dans le processus de développement, une *spécification* est une description de haut niveau du comportement souhaité d'un système [BB87] qui a pour objectif de constituer une référence à implanter ce système. En faisant référence à d'autres travaux, nous pouvons trouver plusieurs définitions de la relations d'implantation. Par exemple, pour des spécifications basées sur la logique [CM89], q est une implantation de p si q implique p, ou pour des automates E/S, un modèle est considéré comme implantation s'il existe une transformation adéquate de l'implantation à sa spécification [Mer90]. Nous retiendrons la définition selon laquelle une implantation est considérée valide si elle satisfait un ensemble de propriétés définies telles que la réduction des blocages, la réduction de l'indéterminisme... Cette relation doit être réflexive car une spécification est une implantation valide d'elle-même.

Relation de raffinement

Comme mentionné dans l'état de l'art, la propriété (5.1) stipule que toute implantation d'un modèle raffiné est également une implantation valide du modèle initial. Leduc appelle cette propriété la propriété de cohérence [Led92]. Par la propriété de réflexivité d'une relation d'implantation, un raffinement q est lui-même son implantation. Par la propriété de raffinement (5.1), q est également une implantation de sa spécification p. La relation de raffinement est donc également une relation d'implantation. Mais l'implication inverse n'est pas garantie. En notant imp une relation d'implantation, la propriété de raffinement (5.1) peut être représentée comme : raf $\circ$ imp $\subseteq$ imp. Un de nos objectifs dans ce chapitre est alors d'identifier la relation de raffinement la plus grande satisfaisant la propriété (5.1).

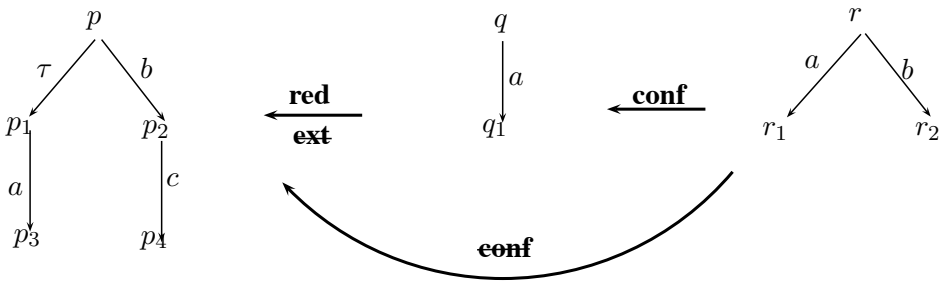


FIGURE 5.1 – red n'est pas une relation de raffinement

Relation red Nous pouvons penser en premier lieu à la relation de réduction red (le préordre d'échec) qui a été choisie comme relation de raffinement en CSP. red réduit l'indéterminisme, les blocages et possède la propriété de transitivité nécessaire pour le développement de spécifications. Étudions dans notre contexte si red traduit une relation de

raffinement. Si on se réfère à l'exemple de la figure 5.1,q conserve bien le comportement principal de p (ce que p doit faire) mais il est aussi une réduction de p (les traces de q sont incluses dans celles de p). On propose r comme implantation de q (cf. figure 5.1), r étant conforme à q. Pourtant r n'est pas conforme à p, car après la trace b, r refuse c. Cet exemple montre que red n'est pas une relation adéquate pour le raffinement.

De plus, l'exemple de la figure 5.2 illustre qu'il peut exister une implantation commune à deux modèles sans qu'il y ait de relation de réduction ou d'extension entre ces modèles. En effet, q est conforme à p sans en être une réduction ou une extension (car les traces de

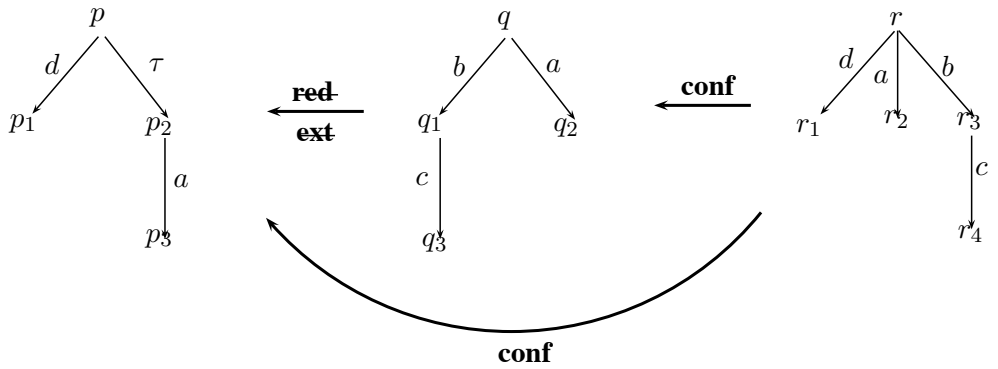


FIGURE 5.2 – Exemple de raffinement q raf p

q ne sont pas inclues dans celles de p et vice-versa). r est pourtant une implantation valide de q et de p.

Relation confrestr

Intuitivement, on peut penser que la relation de raffinement est plus large que ext et ne couvre pas entièrement red du point de vue ensembliste. Dans le cadre du développement formel de spécifications LOTOS, Leduc a défini la relation de conformité restrictive pour les étapes de conception intermédiaires (cf. définition 5.1.1) et en donne une caractérisation (cf. proposition 5.1.1).

Définition 5.1.1 (confrestr [Led91b]). q confrestr $p \Leftrightarrow (\{r \mid r \text{ conf } q\} \subseteq \{r \mid r \text{ conf } p\})$.

Proposition 5.1.1 (confrestr [Led91b]). *Soient p, q deux LTS, q* confrestr *p ssi*

i. q conf p,

ii. $\forall \sigma \in Tr(p) - Tr(q)$, *on a* $L \in Ref(p, \sigma)$.

L est l'ensemble des actions visibles de p et q. $L \in Ref(p, \sigma)$ peut être interprété par le fait que p refuse toutes les actions possibles, après la trace σ. Cela signifie qu'on a atteint un état terminal.

Grâce à cette proposition, nous pouvons évaluer la relation confrestr entre q et p de l'exemple de la figure 5.2 :

- q conf p,
- $Tr(p) - Tr(q) = \{a, d\} - \{bc, a\} = \{d\}$ et p after $d = stop$.

La relation confrestr entre q et p est vérifiée.

Relation de raffinement par réduction red^*, ct

L'exemple de la figure 5.1 montre que red n'est pas une relation de raffinement, car $\mathsf{red} \not\subset \mathsf{confrestr}$. red ne peut donc être choisie comme relation pour le développement incrémental selon les critères que nous avons explicités. Par conséquent, nous proposons une nouvelle relation satisfaisant à la fois les propriétés de la relation de réduction et celles de raffinement.

Définition 5.1.2. *Soient p et q deux LTS, q red^* $p \stackrel{def}{=} q$ $\mathsf{confrestr}$ $p \cap q$ red p.*

La notation red^* ne signifie pas que la relation est une fermeture transitive comme conf^*.

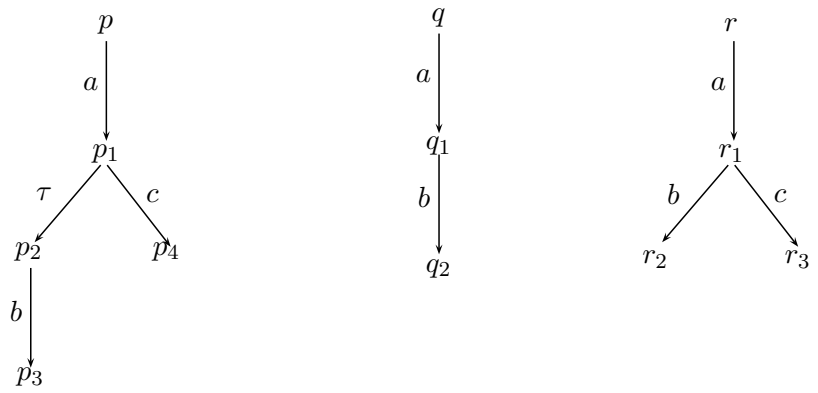


FIGURE 5.3 – q red^* p

L'exemple de la figure 5.3 montre que q red^* p. De ce fait, toute implémentation r de q est également une implémentation de p. De plus $Tr(p) - Tr(q) = \{ac\}$ et p after $ac = stop$.

Définition 5.1.3. $\mathsf{ct} \stackrel{def}{=} \mathsf{red} \cap \mathsf{ext}$

ct est la relation de conformité mais avec en plus, une autre propriété : l'égalité de traces. Cette relation est similaire au préordre de test mais elle n'a pas la contrainte d'implication de convergence. Il est donc évident que $\mathsf{ct} \subseteq \mathsf{ext}$, $\mathsf{ct} \subseteq \mathsf{red}^*$ et $\mathsf{ct} \subseteq \mathsf{confrestr}$.

La proposition (5.1.1) soulève le problème de calcul des traces $Tr(p)$ et $Tr(q)$. Comment implante-t-on la vérification de $\mathsf{confrestr}$ en prenant en compte les traces de deux LTS ? À notre connaissance, aucune solution d'implantation de vérification de ces relations n'a été proposée. Nous nous attachons à résoudre ce problème au paragraphe suivant.

5.2 Implantation des relations $\mathsf{confrestr}$ et red^*

5.2.1 Implantation de la relation $\mathsf{confrestr}$

Avant de présenter la méthode permettant de vérifier la relation $\mathsf{confrestr}$, nous introduisons quelques notations. Étant donnés p, q deux LTS, t, u leur graphe d'acceptance respectif, v le graphe d'acceptance fusionné de t et u, on note :

$$v = Merge(t, u) = Merge(\mathcal{A}(p), \mathcal{A}(q)) = \mathcal{A}(Merge(p, q))$$

On rappelle que dans la définition de la bisimulation 3.1.1 [p 52], Ψ_2 fait référence à :

$$\langle q, a\rangle \in \Psi_2 \Rightarrow (q \xrightarrow{a} q' \Rightarrow \exists p'.\ p \xrightarrow{a} p' \wedge p'\mathcal{R}q')$$

où $\mathcal{R}$ est la relation de simulation.
On note $\Psi_2^p(q)$ la fonction qui identifie les états de p simulant l'état q :

$$\Psi_2^p(q) = \{p \mid q \xrightarrow{a} q' \Rightarrow \exists p'.\ p \xrightarrow{a} p' \wedge p'\mathcal{R}q'\}$$

On note $\mathcal{T} : V \to \mathcal{P}(T)$ la fonction identifiant les états correspondant à t dans v :

$$\mathcal{T}(v) = \{t \in T \mid \exists u,\ Merge(u,t) = v\}$$

Lemme 5.2.1. *Soient p, q deux LTS, t, u leur graphe d'acceptance respectif et v le graphe d'acceptance fusionné. Si q* conf *p, on a :*

$$\forall \sigma \in Tr(u).\ (u \xrightarrow{\sigma} u' \Rightarrow u' \in \Psi_2^v(u))$$

Démonstration. q conf $p \Rightarrow v \gtrsim u$ (Théorème 3.2.1 [p 58]). On a donc :

$$\begin{aligned} u \xrightarrow{a} u' \wedge u' \in \Psi_2^v(u) &\Rightarrow \exists v'(v \xrightarrow{a} v' \wedge v'\mathcal{R}u') \\ \Rightarrow \forall \sigma \in Tr(u).\ (u \xrightarrow{\sigma} u' &\Rightarrow u' \in \Psi_2^v(u)) \end{aligned}$$

□

Théorème 5.2.1. *Soient p, q deux LTS, t, u leur graphe d'acceptance et v le graphe d'acceptance fusionné. $Merge(t,u) = \langle V, \mathcal{L}, \rightarrow_V, v\rangle$. q* confrestr *$p \Leftrightarrow$*

- $v \gtrsim^{\Pi} u$;
- $\forall v' \in (V - \Psi_2^v(u)), \forall p' \in (\mathcal{T}(v')).states,\ p' = stop.$

L'interprétation de ce théorème est la suivante : après avoir calculé la relation de conformité des LTS q et p par l'analyse de leur graphe d'acceptance respectif u et t, si la relation de conformité est vérifiée, nous identifions l'ensemble des états $\Psi_2^v(u)$ du graphe fusionné v simulant les états du graphe u. Ensuite, nous considérons le complément relatif de cet ensemble d'états avec l'ensemble d'états du graphe d'acceptance fusionné $(V - \Psi_2^v(u))$. Ceci permet d'identifier les états correspondant de v dans le LTS t par la fonction $\mathcal{T}(v')$. Puis, nous vérifions tous les états dans le LTS origine p qui sont dans l'ensemble d'états associés $(\mathcal{T}(v')).states$. Si ces états sont tous des états finaux $(p' = stop)$, la relation de conformité restrictive entre q et p est vérifiée.

Pour faciliter la lecture de la démonstration, on peut se référer à la figure 5.4 qui synthétise les fonctions définies sur des LTS et leur graphe d'acceptance.

Démonstration. ($\Leftarrow$) En se référant à la proposition de Leduc [Led91b], on a :

i. $v \gtrsim^{\Pi} u \Rightarrow q$ red $Merge(p,q) \Rightarrow q$ conf p

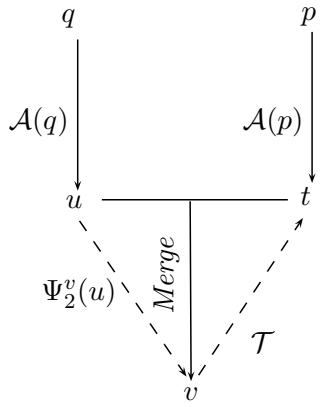


FIGURE 5.4 – Principales fonctions pour la démonstration du calcul de confrestr

ii. On doit prouver : $\forall \sigma \in Tr(p) - Tr(q).\, L \in Ref(p, \sigma)$

Soit $v' \in (V - \Psi_2^v(u))$, $p' \in (\mathcal{T}(v')).states,\ p' = stop.$

$\Rightarrow \forall \sigma \in (Tr(v) - Tr(u)),\ v \xrightarrow{\sigma}_V v' \wedge p' \in (\mathcal{T}(v')).states,\ p' = stop.$

$\Rightarrow \forall \sigma \in ((Tr(u) \cup Tr(t)) - Tr(u)),\ v \xrightarrow{\sigma}_V v' \wedge \forall p' \in (\mathcal{T}(v')).states,\ p' = stop.$

Comme $\mathcal{T}(v) = \{t \in T \mid \exists u,\ Merge(u,t) = v\}$, on peut reformuler :

$$\Rightarrow \forall \sigma \in (Tr(t) - Tr(u)),\ v \xrightarrow{\sigma}_V v' \wedge p' \in (\{t' \in T \mid \exists u',\ Merge(u', t') = v'\}).states,\ p' = stop.$$

Comme u et t sont déterministes, par la construction du graphe fusionné, on a :

$$v \xrightarrow{\sigma}_V v' \Rightarrow t \xrightarrow{\sigma}_T t', \exists p', p \xRightarrow{\sigma} p'$$
$$\Rightarrow \forall \sigma \in (Tr(t) - Tr(u)).v \xrightarrow{\sigma}_V v' \wedge p \xRightarrow{\sigma} p' \Rightarrow p' = stop$$

Comme $Tr(t) = Tr(p)$ et $Tr(u) = Tr(q)$, on a :

$$\forall \sigma \in Tr(p) - Tr(q), \text{on a } p \xRightarrow{\sigma} p',\ p' = stop$$
$$\Rightarrow \forall \sigma \in Tr(p) - Tr(q), \text{on a } L \in Ref(p, \sigma)$$

($\Rightarrow$) À partir de la définition de confrestr

i. q conf $p \Rightarrow q$ red $Merge(p, q) \Rightarrow v \precsim^{\Pi} u$

ii. $\forall \sigma \in Tr(p) - Tr(q),\ L \in Ref(p, \sigma)$

$$\Rightarrow \forall \sigma \in Tr(p) - Tr(q), p \xRightarrow{\sigma} p' \Rightarrow p' = stop$$
$$\Rightarrow \forall \sigma \in (Tr(q) \cup Tr(p)) - Tr(q), \forall p'.\ p \xRightarrow{\sigma} p'.\ p' = stop$$

Car $p' \in t'.states$ tel que $t \xRightarrow{\sigma} t'$ et $t' = \mathcal{T}(v')$ on a :

$$\Rightarrow \forall p' \in (\mathcal{T}(v')).states \wedge\ \forall \sigma \in (Tr(u) \cup Tr(t)) - Tr(u).\, v \xrightarrow{\sigma}_V v'.\ p' = stop.$$
$$\Rightarrow \forall p' \in (\mathcal{T}(v')).states \wedge \forall \sigma \in Tr(v) - Tr(u).v \xrightarrow{\sigma}_V v'.\ p' = stop. \quad (5.2)$$

Comme $Tr(u) \subseteq Tr(v)$, en utilisant le lemme 5.2.1, on a :

$$\Rightarrow \forall \sigma' \in Tr(u).\ (u \xrightarrow{\sigma'}_U u' \wedge u' \in \Psi_2^v(u))$$

Par la construction de graphe d'acceptance fusionné $u \xrightarrow{\sigma'}_U u' \Rightarrow v \xrightarrow{\sigma'}_V v'$

$$\begin{aligned} &\Rightarrow \forall \sigma' \in Tr(v).(v \xrightarrow{\sigma'}_V v' \wedge u' \in \Psi_2^v(u)) \\ &\Rightarrow \forall \sigma \in (Tr(v) - Tr(u)).(v \xrightarrow{\sigma}_V v' \wedge v' \in (V - \Psi_2^v(u))) \end{aligned} \tag{5.3}$$

En combinant (5.2) et (5.3), on a :

$$\begin{aligned} &\forall p' \in (\mathcal{T}(v')).states \wedge v' \in (V - \Psi_2^v(u)) \wedge \forall \sigma \in (Tr(v) - Tr(u)).\ v \xrightarrow{\sigma}_V v',\ p' = stop. \\ \Rightarrow &\forall v' \in (V - \Psi_2^v(u)).\forall p' \in (\mathcal{T}(v')).states,\ p' = stop. \end{aligned}$$

□

Nous savons que la relation de conformité conf garantit la réduction de l'indéterminisme et des blocages, et que confrestr est un préordre plus fort que conf. L'exemple 4 de la figure 5.5 montre que confrestr conserve la fonctionnalité principale (action a) de p, elle enlève l'action optionnelle conduisant à un blocage (l'action b). La relation confrestr préserve non seulement toutes les propriétés de conf, mais en plus, elle supprime les fonctionnalités optionnelles incomplètes risquant de conduire à des blocages.

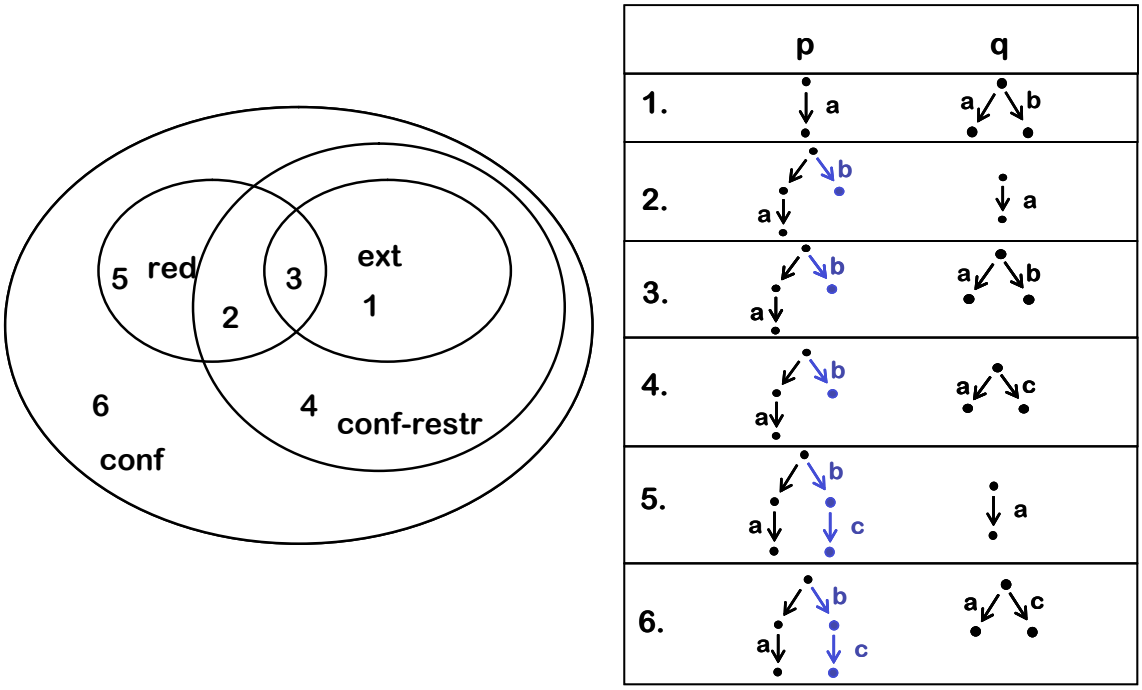


FIGURE 5.5 – Représentation des relations de conformité du point de vue ensembliste [Led92, LCL09]

Les relations de conformité pour lesquelles nous avons développé des algorithmes de calcul sont synthétisées dans la figure 5.5 selon le point de vue ensembliste. Des exemples illustrent les relations et leur intersection. La relation confrestr couvre la relation d'extension, une partie de relation de réduction red et confrestr est incluse dans conf.

La vérification des relations confrestr est implantée dans l'outil IDCM. La complexité de calcul confrestr est la même que celle de conf (PSPACE-complet).

5.2.2 Implantation de relation de raffinement par réduction red*

Dans le paragraphe précédent, nous avons présenté l'intérêt des relations d'extension ext, de conformité restrictive confrestr pour le raffinement. La relation de conformité est présentée comme la relation d'implantation. La relation de réduction red qui est similaire au préordre d'échec préserve les propriétés de vivacité (*liveness*) et de sûreté (*safety*).

Avec la définition ci-dessus et grâce à la proposition 5.1.1, on a la proposition suivante :

Proposition 5.2.1. *q red* p ssi*

- *q red p,*
- *$\forall \sigma \in Tr(p) - Tr(q)$, on a $L \in Ref(p, \sigma)$.*

Du point de vue ensembliste, la relation red* est incluse dans la relation de réduction red et satisfait également la conformité restrictive. Afin de calculer la relation red*, nous pouvons utiliser les théorèmes 3.1.1 [p 54] et 5.2.1 [p 91]. Mais nous proposons une autre méthode permettant de calculer red* directement sans construire de graphe fusionné.

Théorème 5.2.2. *Soient p et q deux LTS et t, u leur graphe d'acceptance.*

- *$t \succsim^{\Pi}_{\sim} u$*
- *$\forall t' \in (T - \Psi_2^t(u)) \wedge \forall p' \in t'.states,\ p' = stop$*

$\Leftrightarrow q$ red* p

La preuve est similaire à la démonstration du théorème 5.2.1.

Comme dans le théorème de calcul de confrestr 5.2.1, on peut mettre en œuvre la vérification de red* en vérifiant la relation de simulation forte entre graphes d'acceptance et l'inclusion des ensembles d'acceptance. Nous vérifions également que tout état du graphe d'acceptance t qui n'est pas simulé par un état de u correspond à un état final du LTS p.

Dans la représentation ensembliste des relations de conformité (cf. fig 5.5), la relation red* correspond à l'union des régions 2 et 3.

La vérification de la relation de raffinement par réduction est implantée dans l'outil IDCM.

5.3 Cadre pour la construction incrémentale

5.3.1 Stratégies de raffinement

Notre objectif ici est d'outiller formellement la démarche incrémentale pour la construction de machines d'états UML intégrant le processus de vérification. Les spécifications et implantations considérées seront donc des machines d'états UML associées à des classes. Comme mentionné dans les paragraphes précédents, le raffinement est considéré comme

une relation liant des spécifications définies à différentes étapes de modélisation. Le premier point de vue du raffinement est lié à l'ajout de nouvelles fonctionnalités en vue d'enrichir des spécifications successives. Nous appelons cela le raffinement horizontal. Un autre point de vue du raffinement consiste à ajouter des détails dans les modèles successifs du système sans ajout de fonctionnalités de façon à obtenir un modèle concret proche d'un modèle d'implantation. Nous appelons cela le raffinement vertical.

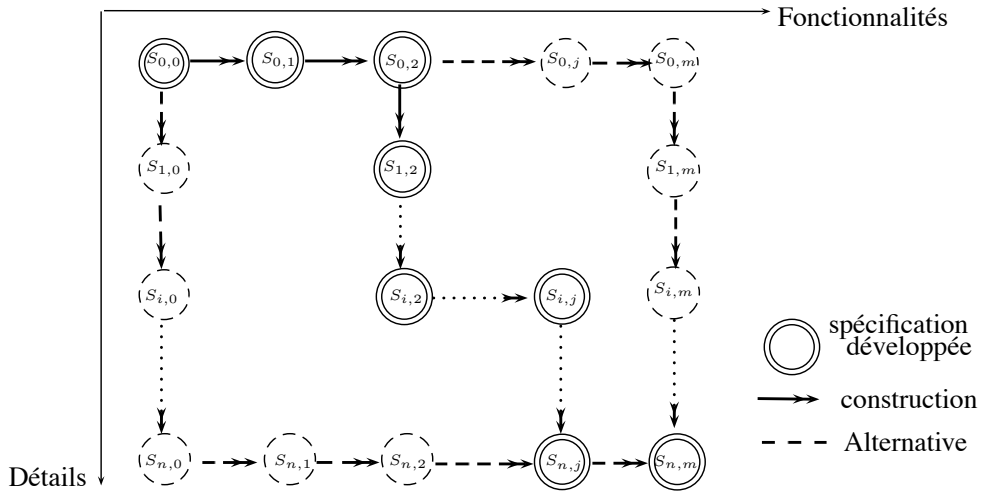


FIGURE 5.6 – Cadre conceptuel de développement incrémental de modèles

Nous pouvons représenter de manière conceptuelle (cf. figure 5.6) le développement incrémental d'une spécification initiale notée $s_{0,0}$, conduisant à son modèle d'implantation final, noté $s_{n,m}$ en passant par des spécifications intermédiaires notées $s_{i,j}$. L'axe vertical représente les niveaux d'abstraction du plus abstrait au plus concret. Les modèles les plus concrets ($s_{n,i}$) désignent des spécifications suffisamment détaillées pour être traduites automatiquement en un programme exécutable. On les appelle modèles d'implantation. On distingue deux stratégies globales de raffinement :

Stratégie h-v (horizontal-vertical) consiste à développer par étapes successives la spécification initiale. Ce n'est que lorsque la version complète des spécifications sera obtenue que la phase de raffinement vertical débutera. Cette stratégie suit les démarches traditionnelles de conception de logiciels où il faut attendre d'avoir une spécification finale pour réaliser l'implantation. Cette stratégie suit le chemin $s_{0,0}; s_{0,1}; \ldots; s_{0,m}\ ; s_{1,m}; \ldots; s_{n,m};$ (cf. figure 5.6). Localement, c'est-à-dire à toute étape, cette stratégie satisfait la propriété de raffinement (5.1) stipulant que toute implantation du modèle raffiné r est également une implantation valide du modèle initial p.

Stratégie v-h (vertical-horizontal) consiste à détailler par étapes successives la spécification initiale afin d'obtenir un modèle d'implantation du système. Ce modèle sera à son tour enrichi par l'ajout de fonctionnalités afin d'obtenir la version finale du modèle d'implantation. Cette stratégie suit le chemin $s_{0,0}; s_{1,0}; \ldots; s_{n,0}; s_{n,1}; \ldots; s_{n,m}$ (cf. figure 5.6). Cette stratégie permet d'obtenir une première version du logiciel pour répondre rapidement à des exigences du client ou encore pour se positionner sur un marché sans avoir

à attendre le développement de la version finale. Elle permet d'être réactif à des besoins d'utilisateurs, sans investissement trop important en temps ou en moyen. La démarche v-h fait le choix d'exploiter comme version délibérément partielle ou provisoire une première version de la réalisation. Cette démarche s'inspire des méthodes de développement agile et peut se révéler très productive industriellement.

Stratégie mixte En pratique, la stratégie de développement est mixte et consiste à développer de façon itérative des raffinements horizontaux et verticaux. Elle suit, par exemple, le chemin $s_{0,0}; s_{0,1}; s_{0,2}; s_{1,2}; \ldots s_{i,2}; \ldots; s_{i,j}; \ldots; s_{n,j}; \ldots s_{n,m}$ (cf. figure 5.6).

5.3.2 Formalisation des stratégies

Dans cette partie, nous nous intéressons à la formalisation des stratégies de raffinement à un niveau local constitué d'une étape de raffinement vertical et d'une étape de raffinement horizontal. Ensuite, nous verrons comment étendre la formalisation de ces stratégies en plusieurs étapes successives dans l'objectif de les rendre équivalentes.

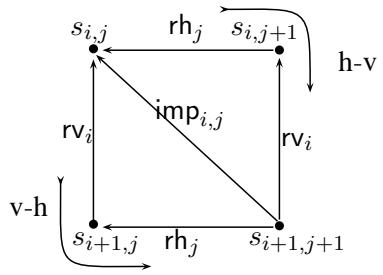


FIGURE 5.7 – Diagramme commutatif du raffinement h-v et v-h

On considère rh_j une relation de raffinement local horizontal, rv_i une relation de raffinement local vertical et $s_{i,j}$ la spécification obtenue après i étapes de raffinement vertical et j étapes de raffinement horizontal. Si on considère la spécification $s_{i,j+1}$ (resp. $s_{i+1,j}$) après une étape de raffinement horizontal (resp. vertical), on a donc : $s_{i,j+1}\ \mathsf{rh}_j\ s_{i,j}$ (resp. $s_{i+1,j}\ \mathsf{rv}_i\ s_{i,j}$) et $\mathsf{imp}_{i,j}$ une relation d'implantation locale entre $s_{i+1,j+1}\ \mathsf{imp}_{i,j}\ s_{i,j}$. La récapitulation est présentée par la définition suivante :

Définition 5.3.1 (Carré de développement local)**.** *Un carré de développement des spécifications $SR_{i,j}$ est un couple $\langle S_{i,j}, R_{i,j}\rangle$ dans lequel $S_{i,j}$ est un ensemble des spécifications :*

$$S_{i,j} \stackrel{def}{=} \{s_{i,j},\ s_{i,j+1},\ s_{i+1,j}\ s_{i+1,j+1}\}$$

et $R_{i,j}$ un ensemble des relations de vérifications (préordres) $R_{i,j} \stackrel{def}{=} \{\mathsf{rh}_j, \mathsf{rv}_i, \mathsf{imp}_{i,j}\}$ vérifiant :

i. rh_j *une relation de raffinement horizontal :* $s_{i,j+1}\ \mathsf{rh}_j\ s_{i,j}$ *et* $s_{i+1,j+1}\ \mathsf{rh}_j\ s_{i+1,j}$ *;*

ii. rv_i *une relation de raffinement vertical :* $s_{i+1,j}\ \mathsf{rv}_i\ s_{i,j}$ *et* $s_{i+1,j+1}\ \mathsf{rv}_i\ s_{i,j+1}$ *;*

iii. $\mathsf{imp}_{i,j}$ *une relation d'implantation locale :* $s_{i+1,j+1}\ \mathsf{imp}_{i,j}\ s_{i,j}$*.*

Dans ce cadre, nous nous inspirons de la théorie des catégories [Pie91] pour formaliser la démarche incrémentale. La théorie des catégories est une théorie générale qui étudie des structures mathématiques de type ensembliste et les fonctions qui les caractérisent. Un concept fondamental de la théorie des catégories [Pie91] est le diagramme commutatif (cf. figure 5.7) dans lequel les chemins de développement sont équivalent.

$$\mathsf{rv}_i \circ \mathsf{rh}_j = \mathsf{rh}_j \circ \mathsf{rv}_i$$

On ne peut appliquer au sens strict cette théorie, car les foncteurs dans la théorie des catégories sont des transformations tandis que dans notre cadre de développement ils sont considérés comme des relations. De plus, on réduit la complexité de notre problème en supposant que les relations de raffinement horizontal (resp. vertical) dans le carré sont les mêmes.

Au niveau global, si on souhaite étendre la démarche incrémentale à n niveaux de raffinement vertical et m niveaux de raffinement horizontal ($m, n > 0$), on peut utiliser des relations différentes de raffinement horizontal rh_j, et de raffinement vertical rv_i (cf. figure 5.8). Nous pouvons représenter de manière conceptuelle (cf. figure 5.6) le développement incrémental que nous avons présenté par la définition suivante :

Définition 5.3.2 (Cadre de construction incrémentale)**.** *Soit $m > 0, n > 0$ et pour tout $0 \leq i \leq n, 0 \leq j \leq m$, $SR_{i,j} = \langle S_{i,j}, R_{i,j} \rangle$ un carré de développement local. Un cadre de construction incrémentale FIC est un couple $\langle S_F, R_F \rangle$ dans lequel S_F est un ensemble de spécifications et R_F est un ensemble de relations de vérification défini par :*

$$S_F \stackrel{def}{=} \bigcup_{i,j} S_{i,j} \qquad R_F \stackrel{def}{=} \bigcup_{i,j} R_{i,j} \cup \{\mathsf{rh}, \mathsf{rv}, \mathsf{imp}\}$$

i. rh *une relation de raffinement horizontal global :* $s_{i,m}\ \mathsf{rh}\ s_{i,0}$ *et* $\forall\ \mathsf{rh}_j, \mathsf{rh}_j \subseteq \mathsf{rh}$ *;*

ii. rv *une relation de raffinement vertical global :* $s_{n,j}\ \mathsf{rv}\ s_{0,j}$ *et* $\forall\ \mathsf{rv}_i, \mathsf{rv}_i \subseteq \mathsf{rv}$ *;*

iii. imp *une relation d'implantation globale :* $s_{n,m}\ \mathsf{imp}\ s_{0,0}$ *et* $\forall\ \mathsf{imp}_{i,j}, \mathsf{imp}_{i,j} \subseteq \mathsf{imp}$.

iv. Soit $\sigma_i \in [0..n] \times [0..m]$, avec $\sigma_0 = (0,0)$ et $\sigma_{n+m} = (n,m)$, et, pour tout i : si $\sigma_{i-1} = (a,b)$, et $\sigma_i = (c,d)$, alors $c = a$ et $d = b+1$ ou $c = a+1$ et $d = b$. Une stratégie *ST est définie par : $ST \stackrel{def}{=} s_{\sigma_0}; s_{\sigma_1}; \cdots; s_{\sigma_{n+m}}$.*

Notons que imp n'est pas nécessairement un préordre.

5.3.3 Exigences locales pour des relations de raffinement

Quelle que soit la stratégie choisie, il est nécessaire de vérifier les modèles élaborés à chaque étape. Nous souhaitons montrer que ces stratégies sont équivalentes dans le sens où elles assurent que le modèle final obtenu est un bon modèle : il est toujours valide (conforme à sa spécification initiale) et implémente éventuellement des fonctionnalités supplémentaires. De plus, pour les processus de raffinement (horizontal et vertical), la garantie de la cohérence entre les modèles est une des propriétés très importantes. rh_j et rv_i doivent être transitives. La réflexivité est nécessaire car une spécification est une implantation (ou un raffinement) valide d'elle-même. Ces relations peuvent être antisymétriques. Donc, les relations rh_j et rv_i doivent être au moins des préordres.

Contraintes : Pour les relations de raffinement local $X = \mathsf{rh}_j, \mathsf{rv}_i$, on peut caractériser les relations de raffinement local par les conditions suivantes :

i. $X \subseteq \mathsf{imp}$ (chaque relation de raffinement local est également une relation d'implantation),

ii. X est une relation de raffinement : $q\ X\ p \Rightarrow (\{r \mid\ r\ \mathsf{imp}\ q\} \subseteq \{\ r \mid r\ \mathsf{imp}\ p\})$,

iii. Équivalence locale : $\mathsf{rv}_i \circ \mathsf{rh}_j = \mathsf{rh}_j \circ \mathsf{rv}_i = \mathsf{imp}_{i,j}$. Cette condition assure que la spécification $s_{i+1,j+1}$ est une implantation valide de $s_{i,j}$.

iv. $\mathsf{rh}_j \subseteq \succsim_{Tr}$, où $\succsim_{Tr}$ est l'extension de traces ;

v. $\mathsf{rv}_i \subseteq \precsim_{Tr}$, où $\precsim_{Tr}$ est la réduction de traces.

Comme mentionné au début de ce paragraphe, le développement horizontal est le fait de définir des incréments de fonctionnalité dans les modèles successifs. Ceci correspond à l'extension conforme des traces. Pour le développement vertical, le raffinement consiste à ajouter des détails dans les modèles successifs du système pour obtenir un modèle concret proche d'un modèle d'implantation. L'ajout de détails est un remplacement d'une action par d'autres pouvant être nouvelles actions. Lorsqu'on compare deux modèles consécutifs, on va masquer ces actions. Le raffinement vertical est donc une réduction de traces.

5.3.4 Exigences globales pour l'équivalence de stratégies

Nous souhaitons que les stratégies non seulement soient localement équivalentes mais également soient globalement équivalents. Ceci est formulé par l'objectif suivant.

Définition 5.3.3 (Équivalence de stratégies)**.** *Dans un FIC, deux stratégies h-v et v-h sont équivalentes si :* $\mathsf{rv} \circ \mathsf{rh} = \mathsf{rh} \circ \mathsf{rv} = \mathsf{imp}$, $\mathsf{rh} = \mathsf{rh}_{m-1} \circ \cdots \circ \mathsf{rh}_0$, *et* $\mathsf{rv} = \mathsf{rv}_{n-1} \circ \cdots \circ \mathsf{rv}_0$.

Cette définition stipule qu'à partir d'une spécification initiale $s_{i,j}$, le même résultat de qualité sera obtenu quelles que soient les stratégies v-h et h-v.

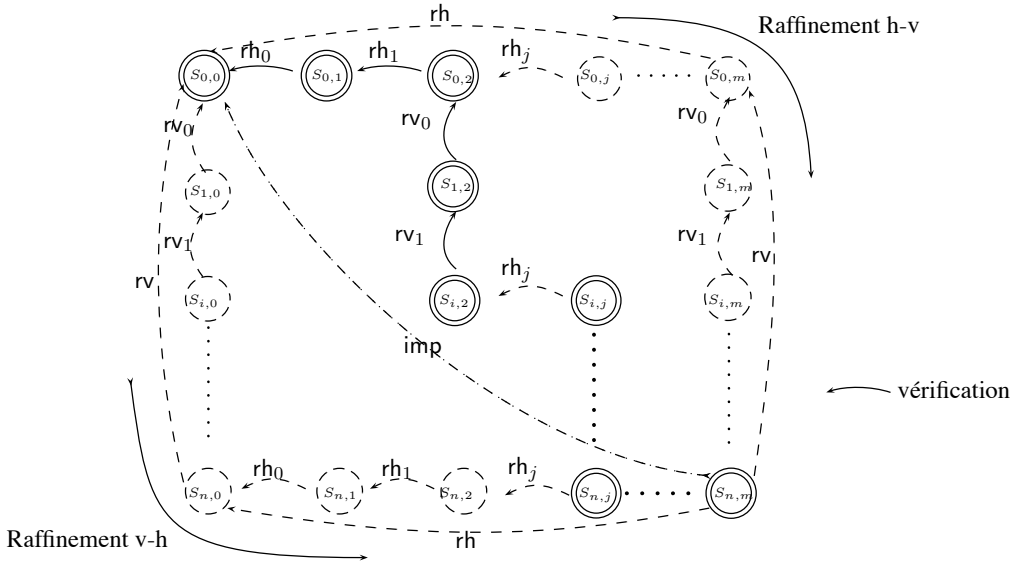


FIGURE 5.8 – Deux approches, h-v et v-h, et l'approche mixte

La proposition suivante permet de fixer les contraintes pour atteindre l'objectif d'équivalences de stratégies h-v et v-h :

Proposition 5.3.1. *Pour un cadre de développement FIC* $\stackrel{def}{=} \langle S_F, R_F \rangle$*, les stratégies v-h and h-v sont équivalentes, si :*

i. $\mathsf{rv}_{i+1} \circ \mathsf{rv}_i = \mathsf{rv}$ *et* $\mathsf{rh}_{j+1} \circ \mathsf{rh}_j = \mathsf{rh} \quad \forall i, j, 0 \leq i < n-1, 0 \leq j < m-1$

ii. $\mathsf{imp}_{i+1,j+1} \circ \mathsf{imp}_{i,j} = \mathsf{imp} \quad \forall i, j, 0 \leq i < n-1, 0 \leq j < m-1$

iii. $\mathsf{rv}_i \circ \mathsf{rh}_j = \mathsf{rh}_j \circ \mathsf{rv}_i = \mathsf{imp}_{i,j} \quad \forall i, j, 0 \leq i < n, 0 \leq j < m.$

Démonstration. L'objectif est de vérifier que la spécification finale $s_{n,m}$ est valide (conforme) à sa spécification initiale. On doit donc prouver :

$$\mathsf{rv} \circ \mathsf{rh} = \mathsf{rh} \circ \mathsf{rv} = \mathsf{imp}$$

et

$$\begin{aligned} \mathsf{rh} &= \mathsf{rh}_{m-1} \circ \cdots \circ \mathsf{rh}_0 \\ \mathsf{rv} &= \mathsf{rv}_{n-1} \circ \cdots \circ \mathsf{rv}_0 \end{aligned}$$

Sans perte de généralité, supposons que :

$$\begin{aligned} X_j \circ X_{j-1} &= X \\ X_{j+1} \circ X_j &= X \end{aligned}$$

où $X = \mathsf{rh}, \mathsf{rv}$. Par composition des équations ci-dessus, on obtient :

$$\begin{aligned} X_{j+1} \circ X_j \circ X_j \circ X_{j-1} &= X \circ X \\ X_{j+1} \circ X_j \circ X_{j-1} &= X \text{ (car } X_j \text{ et } X \text{ sont transitives)} \end{aligned}$$

Pour tout i, j, k, l, $0 \leq j < k \leq m$ and $0 \leq i < l \leq n$

$$\begin{aligned} \mathsf{rh}_k \circ \cdots \circ \mathsf{rh}_{j+1} \circ \mathsf{rh}_j &= \mathsf{rh} \\ \mathsf{rv}_l \circ \cdots \circ \mathsf{rv}_{i+1} \circ \mathsf{rv}_l &= \mathsf{rv} \end{aligned}$$

On a :

$$\begin{aligned} \mathsf{rv} \circ \mathsf{rh} &= \mathsf{rv}_{i+1} \circ \mathsf{rv}_i \circ \mathsf{rh}_{j+1} \circ \mathsf{rh}_j \\ &= \mathsf{rv}_{i+1} \circ \mathsf{rh}_{j+1} \circ \mathsf{rv}_i \circ \mathsf{rh}_j \\ &= \mathsf{imp}_{i+1,j+1} \circ \mathsf{imp}_{i,j} \\ &= \mathsf{imp} \end{aligned}$$

Donc : $\mathsf{rv} \circ \mathsf{rh} = \mathsf{imp}$

La preuve que $\mathsf{rh} \circ \mathsf{rv} = \mathsf{imp}$ est similaire.

On a donc : $\mathsf{rv} \circ \mathsf{rh} = \mathsf{rh} \circ \mathsf{rv} = \mathsf{imp}$ □

Cette proposition traduit le fait que les stratégies globales sont équivalentes s'il existe une équivalence locale des stratégies (condition (*iii*)) et si l'enchaînement des raffinements locaux de même nature (vertical ou horizontal) conduit à un raffinement global (conditions *i* et *ii*) (cf. figure 5.8). Elle montre l'équivalence des stratégies h-v et v-h. À partir de cette proposition, on peut déduire que l'approche mixte composée de plusieurs sous-stratégies h-v et v-h est équivalente aux stratégies globales v-h et h-v. La stratégie mixte permettrait de rationaliser les étapes du cycle de développement en spirale.

5.3.5 Mise en œuvre des relations de raffinement

Dans cette partie, nous sélectionnons des relations de raffinement (vertical et horizontal, local et global) assurant l'équivalence des stratégies de développement incrémental.

En ce qui concerne la relation de raffinement (horizontal et vertical) local, les trois relations $\mathsf{confrestr}, \mathsf{ext}$ et red^* sont candidates (cf. propriété 5.1). Nous devons vérifier si les propriétés d'équivalence de stratégies sont satisfaites pour toutes les combinaisons possibles. Les propositions suivantes montrent quels raffinements locaux peuvent être combinés pour assurer le raffinement global confrestr.

Proposition 5.3.2. *Soit A un préordre, pour toute relation $X \subseteq A$*

$$A \circ X \;=\; X \circ A \;=\; A$$

La démonstration est dans l'annexe A. Cette proposition permet d'impliquer les propositions suivantes :

Corollaire 5.3.1. *Pour* $\mathsf{ext}, \mathsf{red}^*, \mathsf{confrestr}$

i. $\mathsf{confrestr} \circ \mathsf{ext} = \mathsf{ext} \circ \mathsf{confrestr} = \mathsf{confrestr}$

ii. $\mathsf{confrestr} \circ \mathsf{red}^* = \mathsf{red}^* \circ \mathsf{confrestr} = \mathsf{confrestr}$

De plus, on peut démontrer que (cf. annexe A) :

Proposition 5.3.3. $\mathsf{red}^* \circ \mathsf{ext} = \mathsf{ext} \circ \mathsf{red}^* = \mathsf{confrestr}$

Corollaire 5.3.2. *Pour* $\mathsf{ct}, \mathsf{ext}, \mathsf{red}^*, \mathsf{confrestr}$

i. $\mathsf{ext} \circ \mathsf{ct} = \mathsf{ct} \circ \mathsf{ext} = \mathsf{ext}$

ii. $\mathsf{red}^* \circ \mathsf{ct} = \mathsf{ct} \circ \mathsf{red}^* = \mathsf{red}^*$

iii. $\mathsf{confrestr} \circ \mathsf{ct} = \mathsf{ct} \circ \mathsf{confrestr} = \mathsf{confrestr}$

Raffinement horizontal		Raffinement vertical		Implantation	
Global (rh)	Local (rh_j)	Global (rv)	Local (rv_i)	Global (imp)	Local (imp_{ij})
ext	ext	red^* ct confrestr	red^* ct $\mathsf{red}^*, \mathsf{ct}$	confrestr	confrestr
ext	ext	ct	ct	ext	ext, ct

TABLE 5.1 – Combinaisons des relations de raffinement assurant l'équivalence de stratégies

Le tableau 5.1 identifie pour chaque relation de raffinement horizontal global, les combinaisons locales possibles ainsi que les relations de raffinement vertical garantissant l'équivalence de stratégies. Dans tous les cas, la relation ext est considérée comme relation de raffinement horizontal global et local. Il y a deux relations d'implantation globale confrestr et ext.

Si la relation confrestr est considérée comme relation d'implantation globale, la relation d'implantation locale et la relation de raffinement vertical global seront confrestr, et la relation de raffinement vertical local seront red^* ou ct. Si on choisit confrestr comme relation de raffinement vertical global, les relations de raffinement vertical local seront red^* ou ct.

Il existe un cas que la relation ext est la relation d'implantation globale. Dans ce cas, ext ou ct sont choisies comme relations d'implantation locale. red* ou confrestr ne sont pas choisies car : red* $\circ$ ext $\not\subseteq$ ext. Par la contrainte (ii) de la proposition 5.3.1, si on utilise ct comme relation d'implantation au niveau i, j, il est obligatoirement d'utiliser ext au niveau $i+1, j+1$ et $i-1, j-1$.

La proposition 5.3.3 montre également une similarité entre les relations de conformité (red $\circ$ ext = conf) et les relations de conformité restrictive (red* $\circ$ ext = confrestr). Notons que, pour la relation de conformité, la composition inverse n'est pas vérifiée, ie. conf $\neq$ ext $\circ$ red, et de même red $\circ$ conf $\neq$ conf.

5.3.6 Exemple de développement par la stratégie mixte

Nous illustrons sur un simple exemple une stratégie de développement mixte, composée séquentiellement d'une extension, suivie d'une réduction et une extension. Les machines sont présentées dans la figure 5.9.

La première machine est Machine Indéterministe. Cette machine contient les deux fonctions principales (livraison de boissons et d'annulation/remboursement). Elle s'apparente à une spécification déclarative puisque les deux fonctions sont offertes d'une manière non-déterministe. La machine est exhaustive : elle offre l'annulation du service ou la distribution de boissons. En outre, il est nécessaire de payer pour obtenir une boisson, au moins un, mais il n'est pas déterminé le montant exigé.

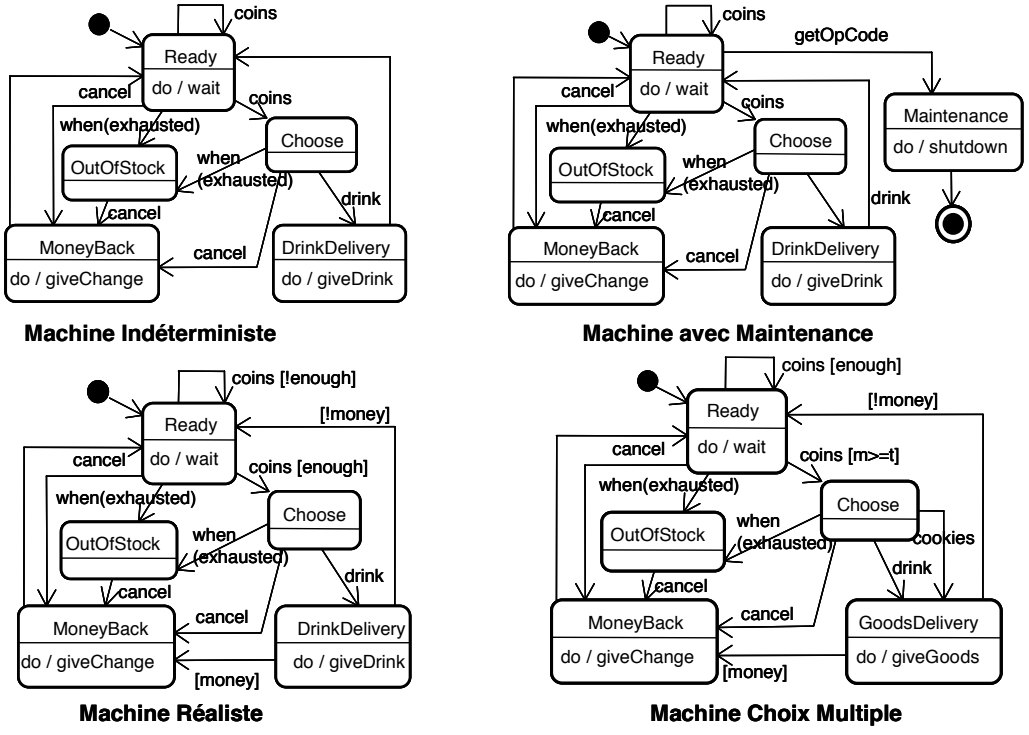


FIGURE 5.9 – Machines d'états simples d'un distributeur automatique

La deuxième machine (Machine avec Maintenance) est une extension de la machine Machine Indéterministe, offrant aussi la possibilité d'arrêter la machine quand elle est dans

son état initial pour faire une maintenance. La relation ext vérifiant entre ces deux machines assure que la Machine avec Maintenance peut encore être utilisé exactement comme Machine Indéterministe.

La troisième machine (Machine Réaliste) est un raffinement vertical (dans ce cas une réduction) de Machine avec Maintenance. Tout d'abord, l'indéterminisme est réduit : les gardes de l'événement coins sont ajoutées. Deuxièmement, elle peut rendre la monnaie, après avoir livré la boisson. Troisièmement, la fonction de maintenance est supprimée. Cette dernière réduction est possible car après l'événement getOpcode aucun autre n'est possible.

La dernière machine (Machine Choix Multiple) est une extension de la Machine Réaliste. Elle offre une seconde option de livraison. Là encore, elle peut être utilisée exactement comme le Machine Réaliste.

On peut en déduire que Machine Choix Multiple est un raffinement (relation confrestr vérifiée) de Machine avec Maintenance, mais aussi une implantation de Machine Indéterministe.

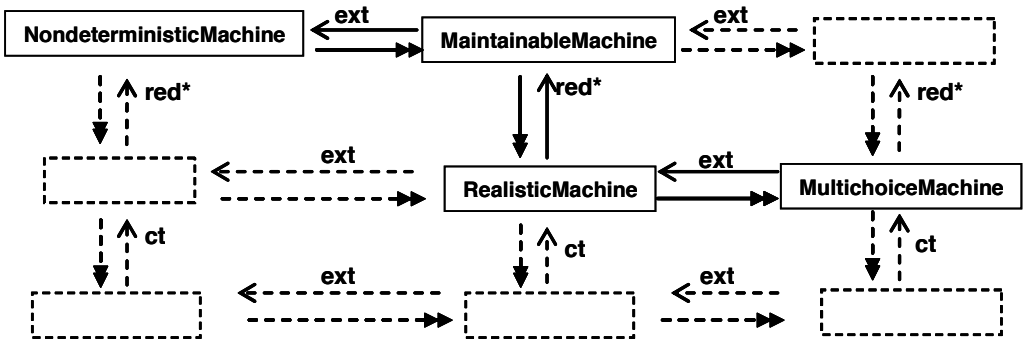


FIGURE 5.10 – Illustration d'une approche mixte

La figure 5.10 présente également les liens pour des machines qui auraient être construites en suivant d'autres stratégies. Elle montre que nous aurions pu utiliser la relation ct à certaines étapes de raffinement verticale, à condition que nous l'utilisions aussi pour tous les autres raffinements verticaux de même niveau.

5.4 Bilan

Des relations de comparaison de modèles ont été définies dans la théorie des processus. Elles peuvent être utilisées pour comparer un modèle d'implantation avec un modèle de spécification, afin de vérifier si les exigences sont remplies ou non. On a montré que les relations d'extension ext, de conformité restrictive confrestr et de réduction red* et ct sont adéquates pour les démarches de raffinement que nous proposons. L'implantation de ces relations a été également présentée.

La relation de conformité restrictive a été présentée comme la plus grande relation de raffinement. De plus, nous avons proposé la construction incrémentale selon deux aspects : la stratégie h-v visant à obtenir une implantation à la fin d'une chaîne de spécifications et la stratégie v-h visant à obtenir une chaîne d'implantations à partir d'une spécification

initiale. La stratégie h-v présente l'avantage d'offrir des supports pour le développement de spécifications de systèmes complexes qui nécessitent plusieurs étapes intermédiaires, soit parce que le système est complexe et on ne peut faire un modèle en une seule étape, soit parce que l'ensemble des fonctionnalités n'est pas encore déterminé. La stratégie v-h présente l'avantage de proposer une première version d'implantation le plus tôt possible à partir d'une spécification partielle. Cette implantation pourra évoluer vers d'autres implantations par l'ajout de fonctionnalités. Pour profiter des avantages des deux approches, on propose l'approche mixte qui est plus pragmatique. Cette approche consiste à développer de façon itérative des raffinements horizontaux et verticaux.

En choisissant les relations red^* et ct comme relation de raffinement vertical et les relations ext comme relations de raffinement horizontal, nous disposons de relations fondamentales pour la construction incrémentale quelles que soient les stratégies de raffinement choisies par le modélisateur.

Dans notre cadre de développement, nous considérons que le raffinement horizontal s'applique uniquement à l'extension de fonctionnalités. Dans une perspective proche, nous pourrions considérer la réduction de traces (red^*) comme raffinement horizontal, ce qui traduit la suppression de traces en préservant la validité du modèle obtenu. Ce type de raffinement peut paraître antagoniste dans le cas d'un développement incrémental. Il est néanmoins utile pour supprimer des fonctionnalités optionnelles. Cette situation se rencontre lors d'un changement imprévu d'exigences du client ou lorsque l'on veut diminuer le coût de développement et implémenter uniquement les fonctionnalités principales. La relation de réduction assure que la suppression de certaines fonctionnalités ne dégrade pas les fonctionnalités principales.

Chapitre 6

Raffinement d'architecture

Sommaire

Dans les chapitres précédents, nous nous sommes intéressés aux démarches de construction de machines d'états UML par raffinement qui consiste à définir des spécifications de plus en plus précises ou complètes. Nous avons abordé la problématique pour un seul modèle interagissant avec l'environnement. Or dans la pratique, un système complexe est décrit par plusieurs composants interagissants. La construction d'architectures par assemblage de composants, la construction incrémentale et la vérification d'architectures font l'objet de nombreuses études mais restent des sujets délicats.

Dans ce chapitre, nous souhaitons appliquer la démarche incrémentale que nous avons élaborée pour un composant à une architecture. Ce chapitre se veut prospectif : nous posons les problèmes induits par le raffinement d'architecture, nous donnons des pistes de solutions et mettons en évidence des travaux futurs à mener.

6.1 Motivations et problématique

L'accroissement de la complexité des systèmes et la nécessité de diminuer le temps et les coûts de développement posent de réels défis pour les concepteurs. De surcroît, cela entraîne la non maîtrise d'erreurs de conception dont les conséquences peuvent être importantes. Pour répondre à cette problématique, le développement à base de composants est proposé

comme une démarche prometteuse dans lequel on construit un système par assemblage de plusieurs composants dont la validation est assurée dans des étapes amont. L'intérêt de cette approche est, comme dans l'approche objets, de bénéficier de composants réutilisables. D'autre part, cette approche répond à la facilité de la maintenance du système [Crn01]. Une fois qu'un système a été déployé, il est possible qu'à un certain moment, l'un des composants soit mis à jour, tandis que d'autres composants restent inchangés. Lorsque la spécification du composant est étendue par ajout de fonctionnalités, cette transformation devrait s'assurer que le composant étendu fonctionnera correctement dans l'environnement existant.

Dans les chapitres précédents, nous avons présenté le raffinement consistant à définir plusieurs modèles successifs d'un composant. De la même manière, lorsqu'un système est complexe, on est amené à spécifier son comportement sous forme d'un assemblage de composants inter-communicants. Le raffinement dans ce cas consiste à ajouter de nouveaux composants ou enrichir un composant existant et assurer que l'architecture obtenue est conforme à l'architecture de référence.

Dans les langages de processus, la notion d'assemblage est traitée par plusieurs travaux. En LOTOS ou CCS, il existe des opérateurs de composition permettant de définir la sémantique de processus inter-communicants. Dans la norme UML, bien qu'il existe des notations pour définir des composants [ICG+04, OMG09] et une communication via des connecteurs, la sémantique associée à l'architecture n'est pas clairement définie.

Une fois la sémantique définie, les problèmes posés pour la construction incrémentale d'architectures sont les suivantes :

- Comment calculer le comportement d'un assemblage de composants dont les comportements sont spécifiés par des machines d'états ?
- Comment garantir qu'un assemblage est conforme à une spécification exprimée par une machine d'états ?
- Comment comparer des assemblages ayant même structure (en remplaçant un composant par un autre composant ou un assemblage conforme), ou de structures différentes (en ajoutant ou enlevant des composants) ?

Nous illustrons les problèmes posés par la construction UML d'assemblage de composants au travers d'un exemple proposé par Milner [Mil89], consistant à modéliser et valider un atelier d'assemblage. Nous présentons tout d'abord des différentes architectures obtenues par construction incrémentale. Puis, nous analysons leur conformité en définissant une sémantique du comportement d'un assemblage de composants.

6.2 Étude de cas et représentation UML de l'architecture

Ce paragraphe reprend l'exemple de l'atelier traité par [Mil89] afin de mettre en évidence les problèmes liés à la construction de différentes architectures dans le paragraphe suivant.

Énoncé du problème

> « Nous souhaitons modéliser une ligne de production simple. Nous supposons que deux personnes (appelées *Jobber*) partagent l'utilisation d'un ou plusieurs

outils (*Tool*), pour fabriquer des objets à partir de composants simples. Chaque objet est constitué en assemblant un manche dans un socle. On appellera *travail* (*job*) une paire constituée d'un socle et d'un manche. Les *jobs* arrivent en séquence sur un convoyeur et les objets assemblés repartent sur un autre convoyeur. Certains *jobs* qualifiés de difficiles nécessitent l'utilisation d'un outil et d'autres, qualifiés de faciles, n'en nécessitent pas. Un *Jobber* ne peut traiter qu'un seul *job* à la fois. »

En suivant la démarche incrémentale, nous réalisons la modélisation de l'atelier en quatre étapes.

6.2.1 Modélisation de la spécification initiale

Dans la première étape, nous réalisons une spécification simplifiée (*SingleJob*) qui répond de façon incomplète à l'énoncé : un seul travail peut être réalisé à la fois et on ne se préoccupe pas de la façon dont il est traité. On fait donc abstraction de la présence d'ouvriers et d'outils. L'interface de la classe *SingleJob* et sa machine d'états sont présentées dans la figure 6.1. L'interface fournie *JobReceipt* est composée du signal *inp(job)* envoyé par l'environnement. L'interface requise *JobDeposit* est composée du signal *outp(job)* envoyé par le *SingleJob* à l'environnement. La machine d'états (cf. figure 6.1 (b)) de *SingleJob* décrit son comportement. Après avoir reçu le *job*, la *SingleJob* réalise la tâche par l'exécution de la méthode *perform(job)*. Ensuite, il transmet le *job* sur sa sortie.

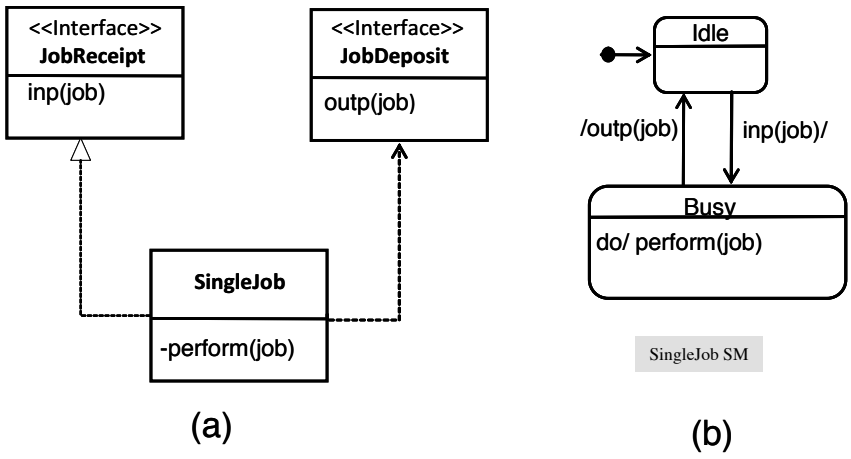


FIGURE 6.1 – Diagramme de classes et machine d'états de *SingleJob*

Pour répondre à la spécification initiale, nous créons l'architecture A_0 constituée de deux composants *SingleJob* mis en parallèle sans synchronisation (cf. figure 6.2). Cette architecture possède les mêmes interfaces requise et fournie que *SingleJob*.

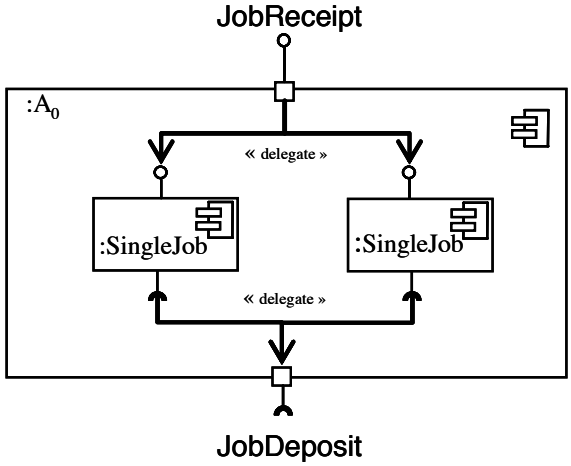


FIGURE 6.2 – La première spécification – A_0

Dans la figure 6.2 apparaît un nouveau type de connecteur défini en UML 2.0 : la délégation. Un connecteur de *délégation* lie l'interface de l'assemblage à l'interface d'un ou plusieurs composants internes. Il permet de faire transiter des signaux dans les différents niveaux de l'assemblage. Il est possible d'associer au connecteur des contrats (de type : *Behavior*[0..*]) définissant des règles de propagation des signaux. Dans notre cas, on associe au connecteur de *JobReceipt* un contrat qui propagera le signal *inp(job)* sur un seul des *SingleJob* : le premier disponible.

Cette architecture constitue la spécification initiale qui sera enrichie selon l'approche incrémentale.

6.2.2 Modélisation d'un assemblage de deux ouvriers partageant un outil

Dans la seconde étape, nous nous intéressons à une spécification A_1 plus détaillée dans laquelle deux ouvriers réalisant chacun la même tâche que *SingleJob* partagent un outil. Cela nous conduit à définir une classe active *Jobber* modélisant le comportement d'un ouvrier et une classe passive *Tool* modélisant des conditions d'utilisation d'un outil (ressource partagée).

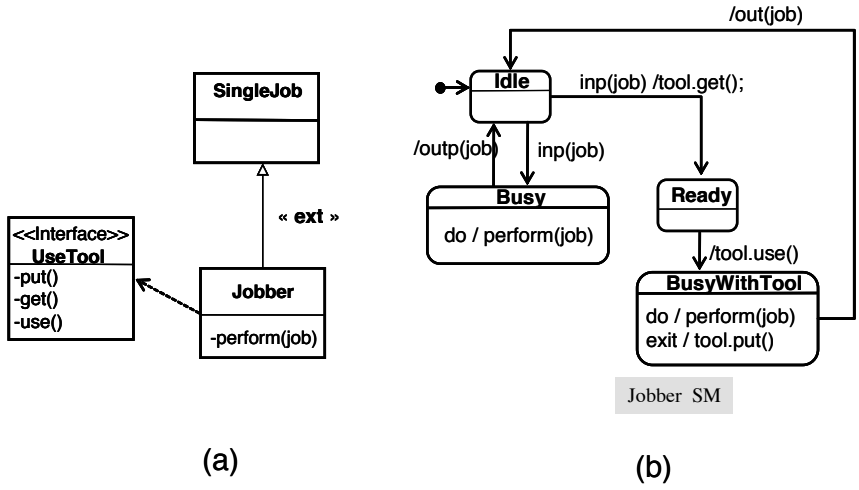


FIGURE 6.3 – Diagramme de classe et machine d'états de *Jobber*

La classe *Jobber* (cf. figure 6.3 (a)) est de type *SingleJob* et par conséquent hérite de ses interfaces et de sa méthode privée. Elle possède une interface requise supplémentaire (*UseTool*). La machine d'états (cf. figure 6.3 (b)) de *Jobber* décrit son comportement. Après avoir reçu le *job*, le *Jobber* réalise la tâche par l'exécution de la méthode *perform(job)*. Dans certains cas (partie droite), il prend un outil pour réaliser la tâche et le repose lorsque la tâche est terminée. Le choix de l'utilisation de l'outil est indéterministe. Il pourrait être rendu déterministe par ajout d'une condition portant sur le niveau de difficulté du *job*. Ce niveau de détail n'est pas important à ce stade de modélisation. Indépendamment de l'utilisation de l'outil, lorsque le *job* est terminé, *Jobber* émet le signal *outp(job)* qui se propagera sur l'interface requise *JobDeposit* de l'assemblage A_1 grâce aux liens de délégation.

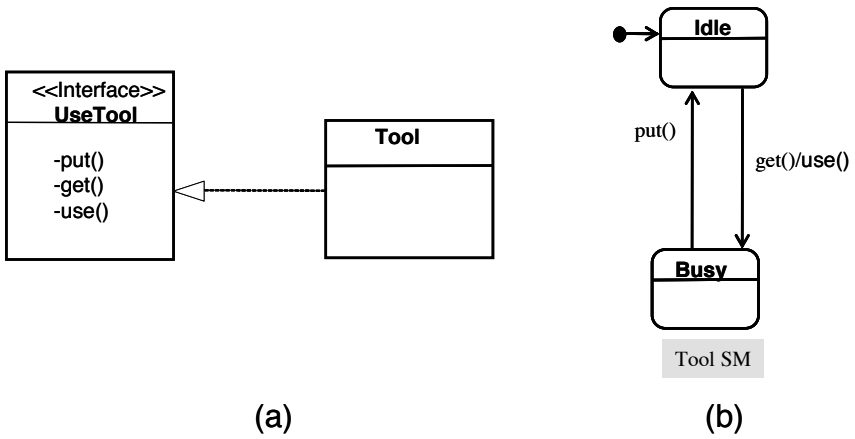


FIGURE 6.4 – Diagramme de classe et machine d'états de *Tool*

La classe passive *Tool* (cf. figure 6.4 (a)) est caractérisée par l'interface fournie *UseTool* composée de signaux relatifs à l'utilisation de l'outil : *put, get* et *use*. Sa machine d'états (cf. figure 6.4 (b)) met en évidence que l'outil ne peut être utilisé que lorsqu'il est disponible et qu'il doit être reposé après son utilisation.

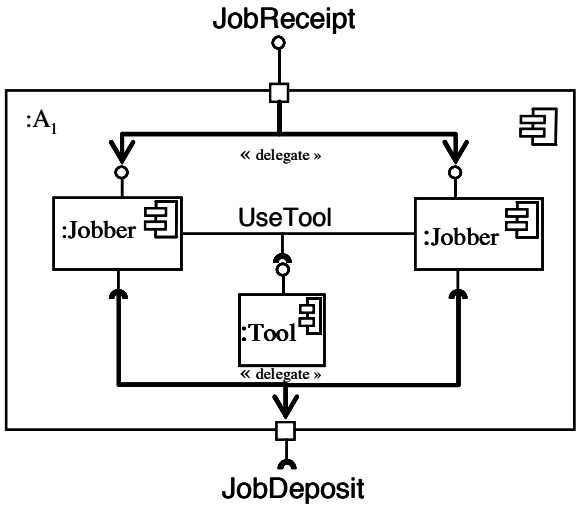


FIGURE 6.5 – Assemblage A_1 : deux instances de *Jobber* partageant un outil *Tool*

La figure 6.5 présente l'assemblage de deux *Jobber* travaillant en parallèle sans synchronisation en partageant le même outil. Dans cette figure apparaît un nouveau type de connecteur défini en UML 2.0 : l'assemblage qui est une liaison entre une interface fournie de *Tool* et les interfaces requises des deux *Jobber*. L'assemblage A_1 constitue un incrément de la spécification initiale A_0 par définition de nouveaux composants. Le problème posé est de vérifier s'il en est conforme.

6.2.3 Modélisation d'un assemblage de deux ouvriers partageant deux outils

Dans cette étape, on étend l'architecture par l'ajout d'un composant modélisant un outil supplémentaire de type *Tool*. Cet assemblage A_2 comprend deux instances de *Jobber* partageant deux outils *Tool*

Le connecteur d'assemblage permet d'associer les interfaces *UseTool* de chaque composant. Le choix de l'outil est effectué par le connecteur. On suppose dans un premier temps que ce choix est indéterministe. Il pourra être rendu déterministe dans une phase avale de modélisation en fonction de la difficulté de la tâche. Remarquons que les deux outils ne sont pas liés aux interfaces de l'assemblage par le connecteur de délégation. L'assemblage A_2 constitue un incrément de l'assemblage A_1 par ajout d'un composant. Le problème posé est de vérifier s'il en est conforme.

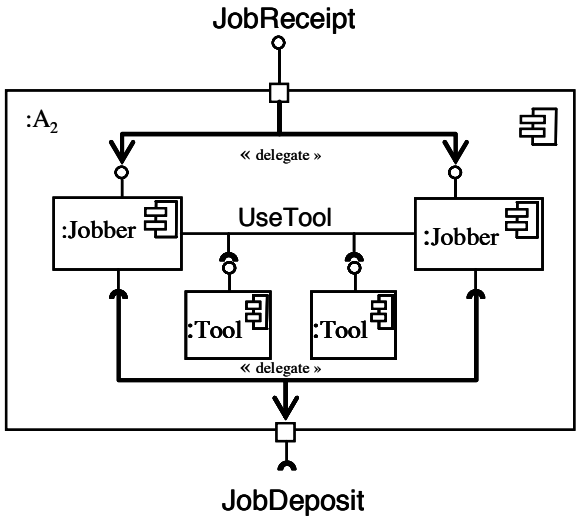


FIGURE 6.6 – Assemblage A_2 : deux instances de *Jobber* partageant deux outils *Tool*

6.2.4 Modélisation d'un assemblage en substituant un composant

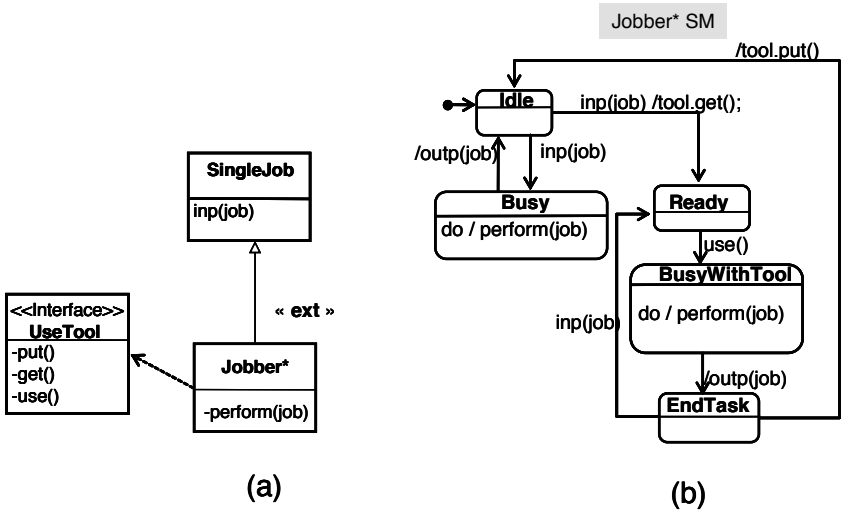


FIGURE 6.7 – Diagramme de classes et machine d'états de *Jobber**

Dans la quatrième étape, on reste dans le contexte de deux ouvriers partageant deux outils mais on introduit un nouveau type d'ouvrier *Jobber** que l'on pense plus qualifié. La figure 6.7 présente la classe de *Jobber** ainsi que sa machine d'états. Cette classe diffère très peu de *Jobber* : elle possède les mêmes interfaces et la même méthode privée. La seule différence se situe dans la partie droite de la machine d'états : *Jobber** ne repose pas l'outil si une nouvelle tâche arrive (cf. nouvelle transition *inp(job)* de l'état *EndTask* à l'état *Ready*). Le nouvel assemblage A_3 est structurellement identique à l'assemblage A_2 ; il diffère uniquement par la substitution d'un composant *Jobber* par le nouveau composant

*Jobber**. Le problème posé est de vérifier si cette substitution assure la conformité de la nouvelle architecture.

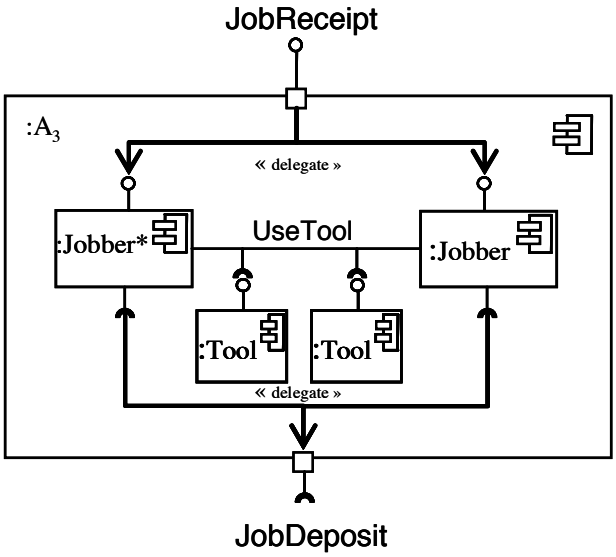


FIGURE 6.8 – Assemblage A_3 : deux instances *Jobber* et *Jobber** partageant deux outils *Tool*

6.3 Analyse architecturale

6.3.1 Sémantique du comportement d'un assemblage

Nous distinguons les composants simples dont le comportement est défini par une machine d'états UML et les composants composites définis par un ensemble de composants inter-agissants, ces composants pouvant être simples ou à leur tour composites. Le comportement d'un composant composite n'est donc pas défini explicitement et doit être déduit du comportement de ses composants. La sémantique associée au comportement d'un composant est basée sur des LTS. Des opérateurs de composition de LTS ont été définis, ce qui nous permet d'associer une sémantique à un ensemble de LTS inter-agissants de façon parallèle ou par un mécanisme de synchronisation. Il nous faut donc définir le comportement d'une architecture en termes de compositions de LTS que nous exprimerons en langage LOTOS.

Nous utiliserons l'opérateur de parallélisme sans synchronisation ||| pour composer des LTS associés à des composants qui ne sont pas en interaction. En revanche, lorsque les composants sont en interaction via une interface I, on utilisera l'opérateur $|[I]|$ qui est l'opérateur de synchronisation sur l'interface I. Remarquons que le connecteur de délégation entre un composant et l'architecture n'a pas à être représenté explicitement par un opérateur LOTOS. Il suffira que les portes de l'architecture et du composant portent les mêmes noms.

Nous distinguons deux vues architecturales :

– La vue externe est une vision boîte noire de l'architecture dans laquelle les signaux de

communication inter-composants sont masqués. Nous utiliserons l'opérateur LOTOS *hide* pour masquer les signaux.

– La vue interne est une vision boîte blanche de l'architecture dans laquelle aucun signal de communication inter-composants n'est masqué.

Comme nous l'avons fait pour l'analyse de composant simple, nous masquons les signaux non contrôlables par l'environnement, pour comparer deux architectures. Par conséquent, nous faisons l'analyse sur la vue externe en masquant l'interface requise de l'architecture. Nous explicitons dans les paragraphes suivants la sémantique LOTOS associée aux architectures élaborées pour l'étude de cas. Nous donnons ensuite les résultats obtenus sur la comparaison des architectures.

6.3.2 Comparaison des comportements

La relation d'extension comportementale a été définie entre deux composants, assurant que le composant étendu offre au moins les mêmes services que le composant initial. On souhaite l'appliquer à la comparaison d'architectures constituées de plusieurs composants. L'étude de cas met en évidence plusieurs cas de figure :

– comparaison d'architectures dont les composants sont de types différents (A_1 comparée à A_0) ;

– comparaison d'architectures par ajout d'un composant (A_2 comparée à A_1) ;

– comparaison d'architectures par substitution d'un composant étendu (A_3 comparée à A_2).

Une solution évidente de comparaison d'architectures est de construire le comportement de chaque architecture en LTS et de les comparer selon les méthodes présentées précédemment. Une solution judicieuse serait de déduire les propriétés de l'architecture à partir de ses composants sans avoir à recalculer le comportement de l'architecture. Pour chaque cas de figure, nous présentons les solutions envisageables.

Comparaison d'architectures dont les composants sont de types différents

Dans ce paragraphe, on s'intéresse à l'analyse de l'architecture A_1 par rapport à A_0. Pour cela, nous générons les LTS traduisant les comportements des deux architectures à partir de leur spécification LOTOS.

Pour A_0, la formulation LOTOS tient compte des caractéristiques suivantes :

– les composants *SingleJob* ne sont pas synchronisés ;

– des connecteurs de délégation existent entre les interfaces de l'architecture et ces composants ;

– comme nous l'avons fait pour un composant simple, l'interface requise de l'architecture est masquée.

L'architecture de A_0 peut être formulée :

$$A_0 = \mathit{hide}\ I_D\ \mathit{in}\ (SJ[I_D, I_R]\ |||\ SJ[I_D, I_R])$$

où I_D désigne l'interface *JobDeposit* et I_R désigne l'interface *JobReceipt*, $|||$ en LOTOS est la composition parallèle sans synchronisation . *hide I in* désigne le masquage des signaux de l'interface I.

Pour A_1, la formulation LOTOS tient compte des caractéristiques suivantes :

- les composants *Jobber* et *Tool* sont synchronisés par un connecteur d'assemblage sur l'interface *UseTool* ;
- des connecteurs de délégation existent entre les interfaces de l'architecture A_1 et celles de ses composants *Jobber* ;
- comme nous l'avons fait pour un composant simple, l'interface requise de l'architecture est masquée.

Pour comparer A_1 par rapport à A_0, il faut masquer les interfaces de synchronisation des composants. Donc, on cache l'interface *UseTool*. Dénotant T l'outil et J le *Jobber*, cette architecture peut être formulée :

$$A_1 = hide(I_D \cup I_T)\ in\ ((J[I_D, I_R]\ |||\ J[I_D, I_R])\ |[I_T]|\ T[I_T])$$

où I_T désigne l'interface *UseTool* et $|[I_T]|$ désigne en LOTOS la composition parallèle avec synchronisation sur les signaux de l'interface I_T.

A partir des formulations des deux architectures en LOTOS, nous générons de façon automatique (sous CADP) les LTS traduisant leur comportement. Par notre outil IDCM, la relation d'extension est vérifiée entre les deux architectures. De plus, il est à noter que le LTS de *Jobber* est une extension de celui de *SingleJob*. Mais cette condition n'est pas suffisante pour assurer l'extension des architectures. En effet, si on modélise un *Jobber* qui ne repose pas l'outil, son comportement est une extension de *SingleJob*, car on fait abstraction de l'utilisation de l'outil en masquant l'interface *UseTool*. En revanche, l'architecture avec ce *Jobber* n'est pas une extension de A_0 car on détecte un blocage sur la deuxième tâche entrante.

Comparaison d'assemblages de structure différente

Dans cette étape, on s'intéresse à l'analyse de l'architecture A_2 par rapport à A_1. La différence est qu'un deuxième outil est ajouté. Les deux outils *Tool* sont en parallèle sans synchronisation, mais en revanche ils sont synchronisés avec les deux composants *Jobber*.

L'architecture A_2 peut être formulée :

$$A_2 = hide(I_D \cup I_T)\ in\ ((J[I_D, I_R]\ |||\ J[I_D, I_R])\ |[I_T]|(T[I_T]\ |||\ T[I_T]))$$

En suivant la même démarche que précédemment, on obtient le nouveau LTS associé à A_2 (cf. figure 6.6). L'extension entre les architectures est vérifiée.

Comme précédemment, avec des *Jobber* ne reposant pas l'outil, on détecte un blocage dans l'architecture car un outil ne peut être pris sans avoir été reposé.

Comparaison d'assemblages dans le cas de substitution d'un composant

Dans cette partie, nous souhaitons vérifier l'extension de l'architecture A_3 en substituant dans A_2 un des composants *Jobber* par un composant $Jobber^*$ qui est une extension de *Jobber*.

Cette architecture peut être formulée :

$$A_3 = hide(I_D \cup I_T)\ in\ ((J^*[I_D, I_R]\ |||\ J[I_D, I_R])|[I_T]|(T[I_T]\ |||\ T[I_T]))$$

L'extension entre les deux architectures n'est pas vérifiée. Une analyse des traces montre qu'il peut y avoir un blocage après l'arrivée de la seconde tâche car *Jobber** sollicite l'outil par l'action *use*, mais l'outil refuse cette action.

6.3.3 Bilan de l'étude de cas et propositions

Les analyses des architectures de l'étude de cas ont été basées sur la comparaison de leur comportement explicite. Nous n'avons pas d'autres solutions à proposer lorsque les architectures sont de structures différentes comme c'est le cas pour les architectures A_2, A_1 et A_0 qui n'ont pas le même nombre de composants. Lorsque les architectures sont de même structure, nous avons vu que la substitution d'un composant par un autre composant conforme, n'assure en rien l'extension de la nouvelle architecture. Pour ce cas précis, on se demande s'il existe une condition sur les composants et leur interaction qui permettrait d'assurer l'extension des architectures sans avoir à recalculer leur comportement, car dans le cas d'architecture complexe ce calcul est coûteux.

L'analyse de l'architecture A_3 montre que la relation d'extension ext entre les composants n'est pas suffisante pour assurer l'extension architecturale. Les travaux de Leduc [Led92] présentent la relation cext qui répond à notre problématique. Nous présentons les définitions des relations cred, cext et nous illustrons comment les mettre en œuvre dans le cas de la substitution d'un composant.

Définition 6.3.1 (cred [BS86]). cred *est la plus faible pré-congruence plus forte que* red.
C'est-à-dire $\forall q, p$ *on a* q cred $p \Leftrightarrow$ *pour tout contexte* $C[.], C[q]$ red $C[p]$

Définition 6.3.2 (cext [BS86]). cext *est la plus faible pré-congruence plus forte que* ext.
C'est-à-dire $\forall q, p$ *on a* q cext $p \Leftrightarrow$ *pour tout contexte* $C[.], C[q]$ ext $C[p]$

Proposition 6.3.1 (cred [Led92]). q cred p *ssi* q red p *et* stable $(p) \Rightarrow$stable (q)

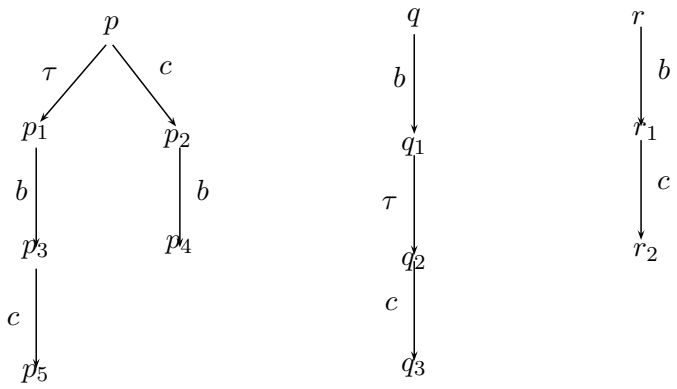


FIGURE 6.9 – $q \not\text{cred}\ p$ et r cred p et r cred q

L'exemple de la figure 6.10 montre que q red p mais $q \not\text{cred}\ p$. Pourtant r cred p et r cred q.

Proposition 6.3.2 (cext [BS86]). q cext p *ssi* q ext $p \wedge q$ cred p.

cext implique l'égalité de traces car :

$q \text{ cext } p \Rightarrow q \text{ ext } p \wedge q \text{ red } p \Rightarrow Tr(q) = Tr(p)$.

L'exemple de la figure 6.10 montre que $q \not\text{cext } p$ et $r \text{ cext } p$

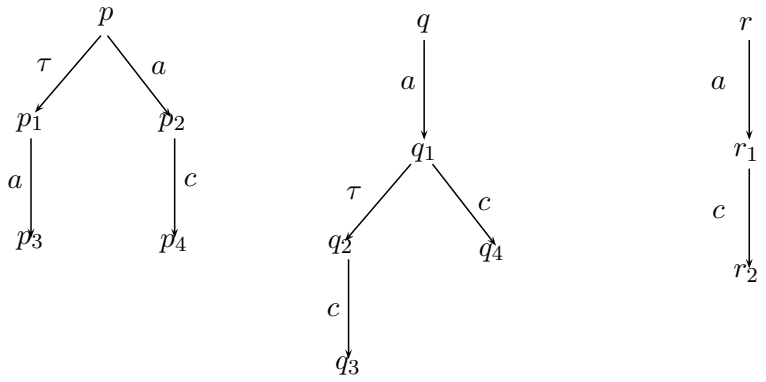


FIGURE 6.10 – $q \not\text{cext } p$ et $r \text{ cext } p$

Notons que la relation cext est définie pour tout contexte. Pour un contexte $C[]$ prédéfini, $C[q]$ ext $C[p]$ ne garantit pas q cext p.

La notion de contexte est définie dans le préliminaire (cf. définition [p 8]). Elle nous permet de désigner l'ensemble des composants d'une architecture. Appliqué à l'architecture A_3 et A_2, on définit $C[.] = \mathit{hide}(I_D \cup I_T)\ \mathit{in}\ ([.]\ |||\ J)|[I_T]|(T\ |||\ T))$. On a :

$$A_3 = \mathit{hide}(I_D \cup I_T)\ \mathit{in}\ (J^*\ |||\ J)|[I_T]|(T\ |||\ T)$$
$$A_2 = \mathit{hide}(I_D \cup I_T)\ \mathit{in}\ (J\ |||\ J)|[I_T]|(T\ |||\ T)$$
$$J^* \text{ cext } J \Rightarrow A_3 \text{ ext } A_2$$

Les deux relations cext et cred peuvent être vérifiées par notre outil IDCM. On vérifie ainsi que $J^* \not\text{cext } J$ car $Tr(J^*) \neq Tr(J)$, mais on ne peut rien déduire sur l'extension des architectures. On pourrait avoir A_3 ext A_2 sans avoir l'égalité de traces des composants ($Tr(J^*) = Tr(J)$).

En revanche, si on définit un *SuperJobber* tel que SJ^* cext J, on peut garantir que la nouvelle architecture sera une extension de A_2. Prenons l'exemple d'un *SuperJobber* réalisant un contrôle qualité après avoir effectué la tâche à l'aide de l'outil. La machine d'états correspondante est donnée dans la figure 6.11. Après avoir effectué la tâche, *SuperJobber* vérifie si elle est bien faite (activité *verify(Job)*. Si c'est le cas, il pose l'outil, si non il recommence la tâche (activité *perform(Job)*)

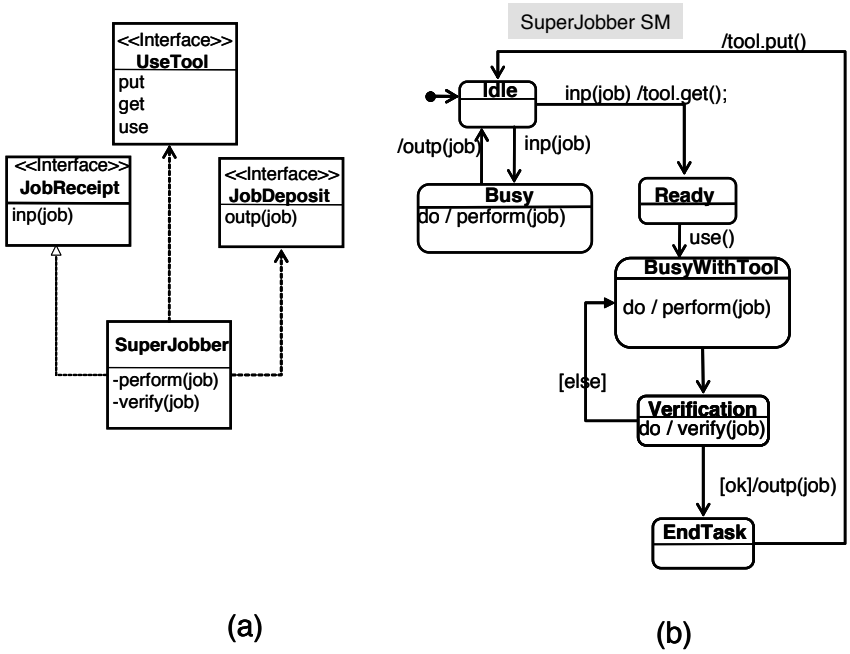


FIGURE 6.11 – Diagramme de classe et machine d'états de *SuperJobber*

Par l'outil IDCM, on vérifie bien que SJ^* ext J. Par conséquent, une architecture composée d'un *SuperJobber* et d'un *Jobber* partageant l'utilisation de deux outils *Tool* est une extension de l'architecture A_2.

6.4 Travaux connexes

Il existe de nombreux travaux portant sur la problématique de l'extension d'architecture. Ichikawa et al [IYK90] ont proposé notamment une condition pour l'extension d'architectures dont les composants sont également fondés sur les LTS.

Proposition 6.4.1 (Ichikawa [IYK90]). $q|[G]|r$ ext $p|[G]|r$ *si* q ext $p \wedge L(q) \cap L(r) \subseteq G$

Cette proposition permet l'extension entre architectures si les composants de la nouvelle architecture satisfont la condition $L(q) \cap L(r) \subseteq G$. Cela signifie que les portes du nouveau composant q communiquant avec celles du composant r reste les mêmes que celles de p qui étaient en communication avec r.

L'application de cette proposition dans le cadre de notre étude pose des problèmes. Nous avons fait le choix d'analyser les architectures par rapport à leur vue externe en masquant les signaux de communication inter-composants. Avec ce masquage, la proposition d'Ichikawa ne peut être appliquée car elle est appropriée à l'analyse de la vue interne, mais cela ne garantit pas la conformité des vues externes.

Il y a un certain nombre de travaux tels que [EHKG02, EKG02, MRR03] portant sur l'analyse d'architectures UML. Engels, Küster, Groenwegen [EKG02] s'intéressent à la cohérence de modèles d'architectures constitués de sous-modèles UML-RT par une démarche proche de la nôtre. La différence est que chaque architecture est composée de *capsules*

interagissant par un connecteur auquel est associé un comportement représenté explicitement par une machine d'états de protocole. La sémantique comportementale des capsules, des connecteurs et de l'architecture est traduite en CSP. La cohérence des architectures est vérifiée en termes de raffinements d'échec défini en CSP et est calculée par l'outil FDR [Eur05]. La limite de ces travaux est qu'il est nécessaire de recalculer le comportement d'une architecture dans laquelle on substitue un composant par un autre. Dans un autre article [EHKG02], Engels et al. proposent une extension de leurs travaux en définissant des règles de couplage de machines d'états UML-RT assurant l'absence de blocages et la consistance de protocoles. L'intérêt de ce travail est de construire de façon incrémentale une architecture tous les composants ont un comportement prédéfini qui ne peut évoluer. Ces travaux n'apportent pas de réponse à notre problématique initiale qui vise à faire évoluer l'architecture et le comportement des composants.

Moisan et al [MRR03] présentent un cadre d'assemblage de composants déterministes pour construire une application. Leur approche est fondée sur la substituabilité comportementale en utilisant des formalismes LFSM (Labelled Finite Deterministic State Machines) et la logique temporelle. Ces travaux sont intéressants car ils apportent une aide à la construction d'architectures, mais ils ne répondent pas à la problématique de la construction incrémentale d'architectures.

6.5 Bilan

Dans ce chapitre, nous avons présenté un cadre de développement d'architectures à base de composants. Les intérêts de ce cadre de modélisation de systèmes sont multiples : la construction progressive d'architectures dans un contexte de spécification incrémentale de systèmes complexes et la substitution d'un composant dans un contexte d'évolution d'architectures pour des raisons de mise à jour ou de maintenance d'un système. De plus, cette approche vise à la réutilisation de composants existants et validés. L'exemple de Milner portant sur la modélisation d'un atelier dans lequel les ouvriers partagent des outils communs a été exploité pour illustrer la problématique de la vérification d'architectures construites par une approche incrémentale.

Nous distinguons deux vues différentes : la vue interne de l'architecture dans laquelle les interactions des composants ne sont pas masquées à l'observateur et la vue externe dans laquelle elles sont masquées. Nous avons étudié deux propositions de construction incrémentale, l'une adaptée à la vue interne et l'autre, à la vue externe. Ces propositions permettent d'exprimer une relation d'extension entre architectures sous des conditions restrictives dont la plus importante est que le composant étendu doit avoir les mêmes traces que le composant substitué. Nous préconisons d'utiliser l'extension de vues externes pour mettre en œuvre au niveau architectural le principe de raffinement exposé dans le chapitre précédent. Une perspective de ces travaux est de trouver une autre condition moins contraignante assurant l'extension architecturale dans le cas de la substitution d'un composant. Reste un problème pour lequel nous n'apportons pas de solutions satisfaisantes qui est celui de l'analyse de la conformité d'une architecture dans laquelle on ajoute de nouveaux composants.

Conclusion

Ce travail a porté sur la définition d'un cadre théorique et pragmatique pour le développement incrémental de machines d'états UML. Bien que la norme UML soit devenue un standard de fait, peu de méthodes d'aide à la construction ont été proposées et peu d'outils offrent des moyens d'évaluation de machines d'états. Si les automates/systèmes de transitions, voire les *statecharts* de Harel sont maîtrisés, ce n'est pas le cas des machines d'états UML. Nous nous intéressons au développement incrémental car il repose sur une technique naturelle pour construire des modèles complexes de façon progressive : on suppose que la spécification peut être incomplète et non déterministe ; elle va subir des transformations par ajouts ou suppressions de fonctionnalités, de détails, de façon à obtenir, en fin du processus de construction, un modèle d'implantation. Un raffinement, une modification ou une correction d'une spécification risque d'introduire à son tour de nouvelles erreurs. La difficulté est de s'assurer que chaque nouveau modèle produit reste cohérent avec les modèles produits lors des étapes antérieures. Nous avons choisi de nous intéresser à l'analyse de cohérence des machines d'états selon un point de vue observationnel qui correspond au point de vue boîte noire : l'utilisateur de la machine d'états ignore les détails de son implantation. L'analyse s'opère donc uniquement par rapport aux actions de sollicitation de la machine. L'évaluation nécessitant un modèle formel de niveau d'abstraction en adéquation avec le point de vue observationnel, nous avons choisi d'associer aux machines d'états une sémantique en termes de systèmes de transitions étiquetées (LTS) et d'effectuer leur comparaison selon ce point de vue sémantique. Nous synthétisons dans un premier temps les résultats que nous avons obtenus. Puis, nous identifions les limites de notre travail et les problèmes qui restent résolus ouvrant ainsi des perspectives à ce travail de recherche.

Résultats

Plusieurs relations de conformité ont été définies sur des systèmes de transitions étiquetées dans le cadre du test de conformité. Parmi elles, trois relations (conformité, réduction et extension) nous sont apparues intéressantes pour aider les concepteurs lors de la construction incrémentale de modèles comportementaux. Les relations de conformité garantissent que les implantations sont au moins aussi déterministes que leur spécification. Elles assurent qu'un modèle d'implantation ne peut pas refuser ce qui est exigé dans la spécification. La relation de réduction assure que le modèle raffiné est conforme au modèle initial sans offrir plus de fonctionnalités, et la relation d'extension permet aux concepteurs d'ajouter de nouvelles fonctionnalités. De par leur spécificité, nous constatons que les relations de réduction et d'extension répondent à nos objectifs pour aider à la construction de machines d'états UML pendant les phases de spécification et de conception des systèmes.

Cependant, elles n'ont été préalablement définies que d'un point de vue mathématique : aucun algorithme de calcul n'a été proposé et elles ne sont pas implantées dans des outils tels que CADP ou TorX. Notre premier résultat a été de proposer deux théorèmes permettant la mise en œuvre des relations red, ext et conf. Ayant démontré la calculabilité de ces relations, nous avons proposé des algorithmes d'implantation de relations de conformité plus spécifiques telles que confrestr, cext, et cred. Cela nous a conduit à développer un prototype JAVA IDCM dans lequel toutes les relations de conformité pré-citées ont été implantées pour comparer des LTS. La complexité de ces algorithmes est PSPACE-Complet du fait de la construction des graphes d'acceptance et non de leur analyse.

Nous avons ensuite mis en application ces résultats pour résoudre le problème initial de comparaison de machines d'états UML. Pour cela, nous avons associé une sémantique LTS aux machines d'états sous forme de patrons unitaires identifiant des configurations types d'états/transitions. Le caractère compositionnel des patrons a facilité l'automatisation de la transformation et nous a conduit à développer un outil JAVA de transformation de machines d'états en LTS dans un format permettant leur traitement dans l'environnement CADP et également l'analyse par IDCM. L'analyse des LTS consiste à mentionner si la relation souhaitée est vérifiée et le cas échéant à donner la séquence d'événements qui conduit à la détection d'une incohérence. Bien que définie sur les LTS, la séquence peut être interprétée par le concepteur sur la machine d'états défaillante. Nous avons illustré la nature des résultats obtenus sur une étude de cas de modélisation d'un téléphone.

La pertinence des relations étant mise en évidente, nous avons formalisé un cadre de construction incrémentale permettant d'apporter un support aux modélisateurs. L'originalité de notre approche réside dans la distinction de deux familles de relations de raffinements : le raffinement horizontal qui assure que le nouveau modèle, même enrichi avec de nouvelles fonctionnalités, reste cohérent avec le modèle initial ; le raffinement vertical qui assure que l'ajout de détails, le changement de niveau d'abstraction et la réduction de l'indéterminisme conduisent à un modèle cohérent. Ceci nous permet de proposer une notion de "stratégie" de construction. Nous caractérisons l'ensemble des stratégies possibles de construction de machines d'états garantissant que le modèle d'implantation est cohérent par rapport à la spécification initiale. Ces stratégies imposent certes des contraintes sur les relations de raffinement à utiliser pour vérifier les modèles durant le processus incrémental mais elles garantissent la construction de modèles valides, quel que soit le choix de stratégie que fera le modélisateur.

La dernière partie de cette thèse est prospective et s'attache à identifier, au travers d'un exemple, les problèmes liés à la construction incrémentale d'architectures. Nous appelons *architecture* un assemblage de composants spécifiés selon une vue comportementale par une machine d'états. Dans le cas où les architectures ne sont pas comparables structurellement (par exemple, lorsque le nombre de composants et les types de composants diffèrent), nous préconisons de construire explicitement le comportement des architectures sous forme de LTS pour se ramener au problème de la comparaison de modèles simples. Dans le cas où les architectures diffèrent par la substitution d'un composant par un autre, nous définissons des conditions sous lesquelles la relation de conformité entre les composants substitués assure la conformité de la nouvelle architecture.

Limitations

Les résultats que nous avons obtenus pour comparer des machines d'états sont encourageants mais possèdent certaines limites. Nous identifions deux problèmes théoriques majeurs : la complexité des algorithmes de vérification des relations et la légitimité de l'interprétation des résultats issus d'une analyse de LTS sur des machines d'états.

Bien qu'il soit suffisant de calculer la conformité sur des LTS minimisés (au sens d'équivalence observationnelle), les algorithmes de vérification des relations de conformité sont de complexité exponentielle. L'origine en est la transformation d'automates non-déterministes en automates déterministes, problème dont la réduction de complexité est un défi depuis une trentaine d'années. Cependant, ce problème est à nuancer, car dans le cadre de la construction de machines d'états, les temps de calcul et la place mémoire sont raisonnables. En effet, on peut traiter des LTS de l'ordre de la centaine de milliers d'états (pouvant se réduire observationnellement à l'ordre du millier d'états), ce qui correspondrait dans le pire des cas à une machine de l'ordre du millier d'états. On peut supposer que, même en utilisant des mécanismes de hiérarchisation et de parallélisation, les machines d'états construites manuellement n'atteindront pas cette limite.

Le deuxième problème est celui du niveau d'abstraction retenu pour la traduction vers les LTS. En effet, il serait souhaitable de pouvoir transposer les résultats obtenus sur les LTS aux machines d'états : si les LTS sont conformes, les machines d'états le sont ; si les LTS ne sont pas conformes, les machines d'états ne le sont pas. Comme nous l'avons montré dans le chapitre 4, ce raisonnement n'est pas possible. D'une part, la notion de conformité n'a été définie que sur les LTS et non directement sur les machines d'états. D'autre part, les LTS font abstraction des données, conditions, gardes, et du temps réel. Ces informations ne sont pas prises en compte dans les calculs de refus. Nous ne pouvons que mettre en garde le concepteur sur des risques de non conformité des machines d'états si l'analyse des LTS montre une non conformité. Si la conformité est satisfaite, on ne peut pas conclure formellement qu'il n'y a pas d'erreurs sur les machines d'états.

Enfin, nous identifions des limites au niveau pratique : nous n'avons pas traité les pseudos-états et le parallélisme. Il faudrait donc étudier quelle sémantique LTS associer à ces éléments de modélisation et aux mécanismes spécifiques qu'ils induisent

Perspectives

Les améliorations qui peuvent être apportées à ce travail concernent différents aspects : la complexité des algorithmes de calcul des relations, le niveau d'abstraction de la transformation et des relations de conformité, le traitement des architectures et la prise en compte de modèles autres que les machines d'états.

Dans un futur proche, afin de diffuser largement notre travail, nous pouvons réaliser des plugins dans TopCASED [FGC+06] qui est un environnement logiciel dédié principalement à la réalisation de systèmes embarqués critiques. Les plugins intégreront non seulement nos algorithmes de vérification, mais également des outils de manipulation de LTS (minimisation observationnelle, composition parallèle, visualisation des LTS,...). Ceci permet de mettre en valeur notre travail, de créer un outil autonome qui est capable de fournir un ensemble de fonctionnalités nécessaires pour aider les concepteurs dans la construction et l'évaluation de spécifications sans dépendre d'autres outils tels que CADP.

Le calcul des relations de conformité étant PSPACE-complet, il serait souhaitable de réduire sa complexité. Des pistes sont envisageables en s'appuyant sur l'algorithme de raffinement de partitions [PT87] ou les algorithmes en temps linéaires développés pour vérifier les relations de simulation [TC01, FM92], bisimulation forte [FM91] et bisimulation arborescente [GV90]. En plus, l'algorithme de minimisation observationnelle est également basé sur l'algorithme de raffinement de partitions. Une combinaison de ces algorithmes est envisageable afin de créer un nouvel algorithme de complexité temporelle moins importante.

Concernant le niveau d'abstraction de la transformation en LTS et des relations de conformité, on peut considérer les éléments de réflexion suivants :

- prendre en compte certaines informations sur les types de données à partir, par exemple, des diagrammes de classes et de contraintes OCL, de manière à produire des LTS plus détaillés. Ceci pourrait se faire à l'image de la transformation de processus valués, telle qu'elle est réalisée dans la traduction de *full* LOTOS en LTS, intégrant une description des types de données par des types abstraits algébriques.
- distinguer les événements d'entrée et de sortie. Ceci revient à considérer par exemple un formalisme tel que celui des IOLTS (Input Output LTS) ou les IOSTS (Input Output Symbolic Transition Systems). Sur les IOLTS, une autre notion de conformité a été reformulée [Tre99], basée non pas sur la notion de refus mais sur ce qu'une machine doit fournir après une séquence d'entrées donnée.
- intégrer dans la définition des relations de conformité la notion de séquence interdite : dans la définition de conformité que nous avons considérée, un produit est conforme à sa spécification s'il doit faire ce que la spécification impose, mais rien n'indique ce qu'il ne devrait pas faire. Il s'agirait par exemple de proposer au spécifieur la possibilité de déclarer un ensemble de séquences interdites, puis de s'assurer automatiquement que les extensions proposées ne les contiennent pas.

Pour ce qui est de la vérification des architectures, nous sommes arrivés à définir des conditions assurant la conformité dans le contexte de substitution d'un composant, mais elles restent contraignantes. Il faut donc poursuivre les travaux, d'autant plus que dans le cas de l'ajout d'un nouveau composant, aucune solution satisfaisante n'a été trouvée.

Si la démarche incrémentale part d'une spécification partielle dans laquelle seules quelques fonctionnalités sont prises en compte, le problème d'identifier les fonctionnalités essentielles ou supplémentaires reste à étudier. On peut identifier les fonctionnalités principales en se basant sur des exigences fonctionnelles dans le cahier de charges. Une autre perspective est de chercher comment minimiser la spécification initiale et partielle en respectant la conformité pour obtenir une spécification minimale qui par conséquent modéliserait les fonctionnalités essentielles.

Notre cadre de construction est actuellement monotone (on ne peut qu'ajouter des fonctionnalités) et manque de flexibilité pour le développement. Dans une perspective proche, nous pourrions considérer la réduction de fonctionnalités (red^*) comme raffinement horizontal. La relation de réduction assure que la suppression de certaines fonctionnalités ne dégrade pas les fonctionnalités principales. Cela est utile pour supprimer des fonctionnalités optionnelles lors d'un changement imprévu d'exigences du client ou lorsque l'on veut diminuer le coût de développement et implanter uniquement les fonctionnalités principales.

Enfin, nous pensons qu'il est nécessaire de pouvoir confronter les machines d'états avec d'autres modèles tels que les machines d'états protocole, les diagrammes de séquences et les

diagrammes d'activités. Les diagrammes de séquences sont des modèles largement utilisés en début de conception des systèmes. Ce couplage fournira à des concepteurs des moyens pour évaluer la cohérence de différents types de modèles comportementaux. De surcroît, on pourrait envisager la définition de patrons de construction préservant la conformité ainsi que des opérateurs de composition de ces patrons.

Bibliographie

[Abr96] Jean-Raymond Abrial, *The B-Book : Assigning programs to meanings*, Cambridge University Press, 1996. [p 16]

[ACGW94] M. Ainsworth, A. H. Cruickshank, L. J. Groves, and P. J. L. Wallis, *Viewpoint specification and z*, Information and Software Technology **36** (1994), no. 1, 43—51. [p 16]

[AHR02] Myla Archer, Constance Heitmeyer, and Elvinia Riccobene, *Proving invariants of I/O automata with TAME*, Automated Software Engineering **9** (2002), no. 3, 201–232. [p 14]

[AL88] M. Abadi and L. Lamport, *The existence of refinement mappings*, Logic in Computer Science, 1988. LICS '88., Proceedings of the Third Annual Symposium on, 1988, pp. 165–175. [p 16]

[ALRL04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, *Basic concepts and taxonomy of dependable and secure computing*, IEEE Transactions on Dependable and Secure Computing **1** (2004), no. 1, 11–33. [p 13]

[Bac78] Ralph-Johan Back, *On the correctness of refinement steps in program development*, Ph.D. thesis, Department of Computer Science, University of Helsinki, 1978. [p 16]

[BB87] Tommaso Bolognesi and Ed Brinksma, *Introduction to the ISO specification language LOTOS*, Comput. Netw. ISDN Syst. **14** (1987), no. 1, 25–59. [p 88]

[BB03] Eerke Boiten and Marius Bujorianu, *Exploring UML refinement through unification*, Critical Systems Development with UML - Proceedings of the UML'03 workshop, LNCS, 2003, p. 47—62. [p 20]

[BC04] Yves Bertot and Pierre Casteran, *Interactive theorem proving and program development*, Texts in Theoretical Computer Science. An EATCS Series, SpringerVerlag, 2004. [p 16]

[BDM98] Patrick Behm, Pierre Desforges, and Jean-Marc Meynadier, *MÉTÉOR : An industrial success in formal development*, B'98 : Recent Advances in the Development and Use of the B Method, 1998, p. 26. [p 3]

[BDMN73] Graham Birtwistle, Ole-Johan Dahl, BJ ?rn Myhrhaug, and Kristen Nygaard, *SIMULA begin*, Petrocelli/Charter, 1973. [p 42]

[BFLW09] J. Bicarregui, J. Fitzgerald, P. Larsen, and J. Woodcock, *Industrial practice in formal methods : A review*, FM 2009 : Formal Methods, 2009, pp. 810–813. [p 14]

[BGHS04] Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling, *Incremental design and formal verification with UML/RT in the FUJABA Real-Time tool suite*, SVERTS 2004, 2004. [p 45]

[BGL94] Amar Bouali, Stefania Gnesi, and Salvatore Larosa, *The integration project for the JACK environment.*, Tech. report, CWI (Centre for Mathematics and Computer Science), 1994. [p 45]

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe, *A theory of communicating sequential processes*, J. ACM **31** (1984), no. 3, 560–599. [p 20, 31]

[BLL+96] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi, *UPPAAL — a tool suite for automatic verification of real-time systems*, Hybrid Systems III, 1996, pp. 232–243. [p 45]

[BMP+02] Jean-Paul Bodeveix, Thierry Millan, Christian Percebois, Christophe Le Camus, Pierre Bazex, Louis Feraud, and Ralph Sobek, *Extending OCL for verifying UML models consistency*, Model Engineering, Concepts and Tools, 2002, p. 75. [p 43]

[BP95] Bard Bloom and Robert Paige, *Transformational design and implementation of a new efficient solution to the ready simulation problem*, Sci. Comput. Program. **24** (1995), no. 3, 189–220. [p 61]

[BS86] E. Brinksma and G. Scollo, *Formal notions of implementation and conformance in LOTOS*, Tech. Report INF-86-13, Twente University of Technology, Department of Informatics, Enschede, Netherlands, December 1986. [p 4, 18, 26, 27, 28, 31, 43, 44, 60, 115]

[CC92] Ufuk Celikkan and Rance Cleaveland, *Computing diagnostic test for incorrect processes*, Proceedings of the IFIP TC6/WG6.1 Twelth International Symposium on Protocol Specification, Testing and Verification XII, North-Holland Publishing Co., 1992, pp. 263–277. [p 52, 64]

[CH93] Rance Cleaveland and Matthew Hennessy, *Testing equivalence as a bisimulation equivalence*, Formal Aspects of Computing **5** (1993), 1–20. [p 28, 29, 30, 32, 47, 52, 53, 54, 60, 61]

[CM89] K. M Chandy and J. Misra, *Parallel program design*, Addison-Wesley Pub. Co., 1989. [p 88]

[Cra06] Michelle L Crane, *On the syntax and semantics of state machines*, 44. [p 33, 40]

[Crn01] Ivica Crnkovic, *Component-based software engineering - new challenges in software development*, Software Focus **2** (2001), no. 4, 127–133. [p 106]

[CS90] Rance Cleaveland and Bernhard Steffen, *A preorder for partial process specifications*, CONCUR '90 : Proceedings on Theories of concurrency : unification and extension (New York, NY, USA), Springer-Verlag New York, Inc., 1990, p. 141—151. [p 20]

[Cus91a] Elspeth Cusack, *Inheritance in object oriented z*, ECOOP'91 European Conference on Object-Oriented Programming, 1991, p. 167. [p 44]

[Cus91b] ———, *Refinement, conformance and inheritance*, Formal Aspects of Computing **3** (1991), no. 2, 129–141. [p 44]

[DAB02] John Derrick, David Akehurst, and Eerke Boiten, *A framework for UML consistency*, Kuzniarz et al.[19] (2002), 30–45. [p 43]

[DB01] John Derrick and Eerke Boiten, *Refinement in z and object-Z : foundations and advanced applications*, Springer-Verlag, 2001. [p 16]

[dBBKM83] J. de Bakker, J. Bergstra, J. Klop, and J. Meyer, *Linear time and branching time semantics for recursion with merge*, Automata, Languages and Programming, 1983, pp. 39–51. [p 22]

[DDd03] Alain David, Johann Deneux, and Julien d'Orso, *A formal semantics for UML statecharts*, Tech. report, 2003. [p 39]

[Dij76] E. W Dijkstra, *A discipline of programming*, Prentice Hall PTR Upper Saddle River, NJ, USA, 1976. [p 16]

[DM01] A. David and M. O Möller, *From HUppaal to uppaal : A translation from hierarchical timed automata to flat timed automata.* [p 45]

[Duc02] Roland Ducournau, *Real world as an argument for covariant specialization in programming and modeling*, 2002, pp. 3–13. [p 44]

[EF02] Rik Eshuis and Maarten M. Fokkinga, *Comparing refinements for failure and bisimulation semantics*, Fundamenta Informaticae **52** (2002), no. 4, 297–321. [p 31]

[EH86] E. Allen Emerson and Joseph Y. Halpern, *"Some times" and "not never" revisited : on branching versus linear time temporal logic*, J. ACM **33** (1986), no. 1, 151–178. [p 22]

[EHHS02] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer, *Testing the consistency of dynamic UML diagrams*, IDPT, 2002. [p 43]

[EHKG02] G. Engels, R. Heckel, J. M. Kuster, and L. Groenewegen, *Consistency-preserving model evolution through transformations*, UML (2002), 212–226. [p 45, 117, 118]

[EKG02] Gregor Engels, Jochen M. Küuster, and Luuk Groenwegen, *Consistent interaction of software components*, Journal Integrated Design and Process Science **6** (2002), no. 4, 2–22. [p 117]

[Eur05] Formal Systems Europe, *Failures-Divergence refinement*, Tech. report, 2005. [p 118]

[FGC+06] Patrick Farail, Pierre Gaufillet, Agusti Canals, Christophe Le Camus, David Sciamma, Pierre Michel, Xavier Crégut, and Marc Pantel, *The TOPCASED project : a toolkit in open source for critical aeronautic systems design*, EMBEDDED REAL TIME SOFTWARE (Toulouse), vol. 781, 2006, pp. 54–59. [p 121]

[FM91] Jean-Claude Fernandez and Laurent Mournier, *A tool set for deciding behavioral equivalences*, Proceedings of the 2nd International Conference on Concurrency Theory, Springer-Verlag, 1991, pp. 23–42. [p 122]

[FM92] Jean Fernandez and Laurent Mounier, *"On the fly" verification of behavioural equivalences and preorders*, Computer Aided Verification, 1992, pp. 181–191. [p 61, 122]

[FSKdR05] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem-Paul de Roever, *29 new unclarities in the semantics of UML 2.0 state machines*, Formal Methods and Software Engineering, 2005, pp. 52–65. [p 34]

[GFL05] Frédéric Gervais, Marc Frappier, and Régine Laleau, *Vous avez dit raffinement ?*, Tech. Report CEDRIC 829, 2005. [p 16]

[GG91] Paul Gochet and Pacal Gribomont, *Logique tome 1 : méthodes pour l'informatique fondamentale*, Logique, vol. 1, Hermès, 1991. [p 12]

[GL94] Orna Grumberg and David E. Long, *Model checking and modular verification*, ACM Trans. Program. Lang. Syst. **16** (1994), no. 3, 843–871. [p 24]

[GL04] Olivier Gout and Thomas Lambolais, *Construction incrémentale de modèles comportementaux UML*, AFADL04, 2004, p. 29—42. [p 46]

[GLM01] Stefania Gnesi, Diego Latella, and Mieke Massink, *Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking*, Journal of Logic and Algebraic Programming **51** (2001), no. 1, 43–75. [p 39, 45]

[GMLS06] Hubert Garavel, Radu Mateescu, Frédéric Lang, and Wendelin Serwe, *CADP 2006 : A toolbox for the construction and analysis of distributed processes*, Computer Aided Verification, 2006, pp. 158–163. [p 32, 51]

[Gou06] Olivier Gout, *Développement incrémental de spécifications orientées objets*, Ph.D. thesis, école de mines d'Alès, université de Montpellier 2, 2006. [p 2, 51]

[GV90] Jan Groote and Frits Vaandrager, *An efficient algorithm for branching bisimulation and stuttering equivalence*, Automata, Languages and Programming, 1990, pp. 626–638. [p 122]

[GvdP00] Jan Groote and Jaco van de Pol, *State space reduction using partial ?-Confluence*, Mathematical Foundations of Computer Science 2000, 2000, pp. 383–393. [p 63]

[GW03] Hassan Gomaa and Duminda Wijesekera, *Consistency in Multiple-View UML models : A case study*, Workshop on Consistency Problems in UML-based Software Development II, 2003, p. 1. [p 43]

[Hal90] Anthony Hall, *Seven myths of formal methods*, IEEE Softw. **7** (1990), no. 5, 11–19. [p 14]

[Hal98] Nicolas Halbwachs, *Synchronous programming of reactive systems*, Computer Aided Verification, 1998, pp. 1–16. [p 1]

[Har87] David Harel, *Statecharts : A visual formalism for complex systems*, Sci. Comput. Program. **8** (1987), no. 3, 231–274. [p 34]

[Hen85] M. Hennessy, *Acceptance trees*, Journal of the ACM (JACM) **32** (1985), no. 4, 896–928. [p 27, 29]

[Hen88] Matthew Hennessy, *Algebraic theory of processes*, MIT Press, 1988. [p 29]

[HHK95] M.R. Henzinger, T.A. Henzinger, and P.W. Kopke, *Computing simulations on finite and infinite graphs*, Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on, 1995, pp. 453–462. [p 26]

[HKR02] T. A. Henzinger, O. Kupferman, and S. K. Rajamani, *Fair simulation*, Information and Computation **173** (2002), no. 1, 64–81. [p 24]

[HKRS05] Zbigniew Huzar, Ludwik Kuzniarz, Gianna Reggio, and Jean Louis Sourrouille, *Consistency problems in UML-Based software development*, UML Modeling Languages and Applications, Springer Berlin / Heidelberg, 2005, pp. 1–12. [p 42]

[Hoa69] C. A. R. Hoare, *An axiomatic basis for computer programming*, Communications of the ACM **12** (1969), no. 10, 576–580. [p 12]

[Hoa78] Charles Antony Richard Hoare, *Communicating sequential processes*, Commun. ACM **21** (1978), no. 8, 666–677. [p 12, 20, 63]

[Hol97] G.J. Holzmann, *The model checker SPIN*, Software Engineering, IEEE Transactions on **23** (1997), no. 5, 279–295. [p 45]

[HP85] David Harel and Amir Pnueli, *On the development of reactive systems*, Logics and models of concurrent systems, Springer-Verlag New York, Inc., 1985, pp. 477–498. [p 1]

[IBM09] IBM, *IBM rational rose technical developer*, http ://www-01.ibm.com/software/awdtools/developer/technical/, 2009. [p 34, 46]

[ICG+04] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, and Jaime Rodrigo Oviedo Silva, *Documenting component and connector views with uml 2.0*, Proceedings of the Third International Conference on the Unified Modeling Language-« UML», vol. 44, 2004, p. 23–49. [p 106]

[IYK90] Haruhisa Ichikawa, Kenjiroh Yamanaka, and June Kato, *Incremental specification in LOTOS*, Proceedings of the IFIP WG6.1 Tenth International Symposium on Protocol Specification, Testing and Verification X, North-Holland Publishing Co., 1990, pp. 183–196. [p 60, 117]

[JJ05] Claude Jard and Thierry Jéron, *TGV : theory, principles and algorithms*, International Journal on Software Tools for Technology Transfer (STTT) **7** (2005), no. 4, 297–315. [p 64]

[JK06] Frédéric Jouault and Ivan Kurtev, *Transforming models with ATL*, Satellite Events at the MoDELS 2005 Conference, 2006, pp. 128–138. [p 72]

[Jon70] John Chris Jones, *Design methods : Seeds of human futures*, 1st ed., John Wiley & Sons Ltd, 1970. [p 1]

[KM02] Alexander Knapp and Stephan Merz, *Model checking and code generation for UML state machines and collaborations*, 59—64. [p 45]

[KMR02] Alexander Knapp, Stephan Merz, and Christopher Rauh, *Model checking timed UML state machines and collaborations*, Formal Techniques in Real-Time and Fault-Tolerant Systems, 2002, pp. 395–414. [p 45]

[KvB93] Ferhat Khendek and Gregor von Bochmann, *Merging behavior specifications*, Formal Methods in System Design **6** (1993), no. 3, 259—293. [p 53, 56, 57, 60, 61]

[KvB94] ———, *Incremental construction approach for distributed system specifications*, Proceedings of the IFIP TC6/WG6.1 Sixth International Conference

on Formal Description Techniques, VI, North-Holland Publishing Co., 1994, pp. 87–102. [p 55]

[LAB+95] J. C Laprie, J. Arlat, J. P Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J. C Fabre, H. Guillermain, M. Kaâniche, and K. Kanoun, *Guide de la sûreté de fonctionnement*, cépaduès ed., Toulouse, 1995. [p 13]

[Lam01] Axel Van Lamsweerde, *Goal-Oriented requirements engineering : A guided tour*, Requirements Engineering, IEEE International Conference on (Los Alamitos, CA, USA), vol. 0, IEEE Computer Society, 2001, p. 0249. [p 13]

[LCL09] Thomas Lambolais, Anne-Lise Courbis, and Hong-Viet Luong, *Raffinement de modèles comportementaux UML, vérification des relations d'implantation et d'extension sur les machines d'états*, AFADL'09, 2009. [p 93]

[Led91a] Guy Leduc, *Conformance relation, associated equivalence, and minimum canonical tester in LOTOS*, PSTV XI. North-Holland (1991), 249–264. [p 26, 27, 59, 87]

[Led91b] ———, *On the role of implementation relations in the design of distributed systems using LOTOS*, Ph.D. thesis, Université de Liège, Faculté des sciences appliquées, 1991. [p 27, 28, 29, 31, 63, 89, 91]

[Led92] ———, *A framework based on implementation relations for implementing LOTOS specifications*, Computer Networks and ISDN Systems **25** (1992), no. 1, 23—41. [p 88, 93, 115]

[Les95] T. Leslie, *Efficient approaches to subset construction*, Ph.D. thesis, University of Waterloo, 1995. [p 61]

[LG05] Thomas Lambolais and Olivier Gout, *Ingénierie du logiciel : allons-nos vers des systèmes plus fiables ?*, Revue de l'Électricité et de l'Électronique et des Technologies de l'Information et de la Communication (2005). [p 15]

[Lio96] J. L Lions, *Ariane 5 flight 501 failure*, Report by the Inquiry Board, Paris **19** (1996). [p 1]

[LLC08a] Hong-Viet Luong, Thomas Lambolais, and Anne-Lise Courbis, *Implementation of extension and reduction relations for incremental development of behavioural models*, Tech. Report RR-006-2008, EMA, May 2008. [p 72, 78]

[LLC08b] ———, *Implementation of the conformance relation for incremental development of behavioural models*, MoDELS 2008 (Krzysztof Czarnecki, ed.), LNCS, vol. 5301/2009, Springer, October 2008, pp. 356–370. [p 78]

[LMM99a] D. Latella, I. Majzik, and M. Massink, *Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker*, Formal Aspects of Computing **11** (1999), no. 6, 637–664. [p 45]

[LMM99b] Diego Latella, Istvan Majzik, and Mieke Massink, *Towards a formal operational semantics of UML statechart diagrams*, Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS), Kluwer, B.V., 1999, p. 465. [p 39, 45]

[LP99a] Lilius and Paltor, *Formalising UML state machines for model checking*, UML 99 — The Unified Modeling Language, 1999. [p 45]

[LP99b] J. Lilius and I. P Paltor, *The semantics of UML state machines*, Turku Centre for Computer Science, 1999. [p 45]

[LP05] Vitus Lam and Julian Padget, *Consistency checking of sequence diagrams and statechart diagrams using the \Pi-Calculus*, 2005, pp. 347–365. [p 42, 43]

[LS00] François Laroussinie and Philippe Schnoebelen, *The state explosion problem from trace to bisimulation equivalence*, 2000, pp. 192–207. [p 28, 32]

[LW94] Barbara H. Liskov and Jeannette M. Wing, *A behavioral notion of subtyping*, ACM Trans. Program. Lang. Syst. **16** (1994), no. 6, 1811–1841. [p 42, 43]

[MA01] John C. Mitchell and Krzysztof Apt, *Concepts in programming languages*, Cambridge University Press, 2001. [p 42]

[Mat05] Radu Mateescu, *On the fly state space reductions for weak equivalences*, Proceedings of the 10th international workshop on Formal methods for industrial critical systems (Lisbon, Portugal), ACM, 2005, pp. 80–89. [p 63]

[Mer90] Michael Merritt, *Completeness theorems for automata*, Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness, 1990, pp. 544–560. [p 88]

[Mil71] Robin Milner, *An algebraic definition of simulation between programs*, Tech. report, Stanford University, 1971. [p 12, 24]

[Mil89] ———, *Communication and concurrency*, Prentice-Hall, Inc., 1989. [p 20, 22, 24, 25, 106]

[Mil99] ———, *Communicating and mobile systems : the Pi-Calculus*, 1st ed., Cambridge University Press, June 1999. [p 21, 25, 52]

[MLSH98] E. Mikk, Y. Lakhnech, M. Siegel, and G.J. Holzmann, *Implementing statecharts in PROMELA/SPIN*, Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on, 1998, pp. 90–101. [p 45]

[MRR03] S. Moisan, A. Ressouche, and J. P. Rigault, *A behavioral model of component frameworks*, Tech. Report 5065, 2003. [p 117, 118]

[NH83] Rocco De Nicola and Matthew Hennessy, *Testing equivalence for processes*, Proceedings of the 10th Colloquium on Automata, Languages and Programming, Springer-Verlag, 1983, pp. 548–560. [p 20, 28, 29, 30, 63]

[Nie93] Oscar Nierstrasz, *Regular types for active objects*, SIGPLAN Not. **28** (1993), no. 10, 1–15. [p 42, 43]

[OAK03] Mitsutaka Okazaki, Toshiaki Aoki, and Takuya Katayama, *Formalizing sequence diagrams and state machines using concurrent regular expression*, Proc. Int. Workshop on Scenarios and State Machines : Models, Algorithms, and Tools, 2003, pp. 74–79. [p 42]

[OMG03] OMG, *UML 2.0 infrastructure specification. object management group*, 2003. [p 33, 37]

[OMG09] ———, *Unified modeling language, superstructure, version 2.2*, 2009. [p 2, 33, 34, 36, 38, 39, 41, 106]

[Par81] David Park, *Concurrency and automata on infinite sequences*, Theoretical Computer Science, 1981, pp. 167–183. [p 24]

[Pie91] Benjamin C. Pierce, *Basic category theory for computer scientists*, MIT Press, 1991. [p 97]

[PPGK03] Claudia Pons, Gabriela Perez, Roxana Giandini, and Ralf-D Kutsche, *Understanding refinement and specialization in the UML*, International Workshop on MAnaging SPEcialization, 2003. [p 44]

[PR94] Barbara Paech and Bernhard Rumpe, *A new concept of refinement used for behaviour modelling with automata*, FME '94 : Industrial Benefit of Formal Methods, 1994, pp. 154–174. [p 44]

[PT87] Robert Paige and Robert E. Tarjan, *Three partition refinement algorithms*, SIAM J. Comput. **16** (1987), no. 6, 973–989. [p 122]

[QS82] Jean-Pierre Queille and Joseph Sifakis, *Specification and verification of concurrent systems in CESAR*, International Symposium on Programming, 1982, pp. 337–351. [p 12]

[Rha09] Rhapsody, *Rational rhapsody : Model-Driven development for SE SD*, http ://www.telelogic.com/products/rhapsody/index.cfm, 2009. [p 46]

[RJB05] James Rumbaugh, Ivar Jacobson, and Grady Booch (eds.), *The unified modeling language reference manual*, Addison-Wesley Longman Ltd., 2005. [p 37]

[Rus94] J. Rushby, *Critical system properties : Survey and taxonomy*, Reliability Engineering and System Safety **43** (1994), no. 2, 189–220. [p 1]

[RW99] Gianna Reggio and R.J Wieringa, *Thirty one problems in the semantics of UML 1.3 dynamics*, Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA-99)—Workshop" Rigorous Modelling and Analysis of the UML : Challenges and Limitations (1999). [p 34]

[SBD96] Maarten Steen, Howard Bowman, and John Derrick, *Composition of LOTOS specifications*, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV, Chapman Hall, Ltd., 1996, pp. 87–102. [p 60]

[SD01] Graeme Smith and John Derrick, *Specification, refinement and verification of concurrent Systems—An integration of Object-Z and CSP*, Formal Methods in System Design **18** (2001), no. 3, 249–284. [p 16]

[Sel04] Bran V. Selic, *On the semantic foundations of standard UML 2.0*, Formal Methods for the Design of Real-Time Systems (M. Bernardo and F. Corradini, eds.), vol. 3185, Springer, 2004, pp. 181–199. [p 35]

[Ser99] Giovanna DI MARZO Serugendo, *Stepwise refinement of formal specifications based on logical formulae : from CO-OPN/2 specifications to java programs*, Ph.D. thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 1999. [p 16]

[SJ95] Yellamraju Srinivas and Richard Jüllig, *Specware : Formal support for composing software*, Mathematics of Program Construction, vol. 947/1995, 1995, pp. 399–422. [p 16]

[SJM04] Ragnhild Van Der Straeten, Viviane Jonckers, and Tom Mens, *Supporting model refactorings through behaviour inheritance consistencies*, 2004 - The Unified Modelling Language, 2004, pp. 305–319. [p 43]

[SKM01] Timm Schäfer, Alexander Knapp, and Stephan Merz, *Model checking UML state machines and collaborations*, Electronic Notes in Theoretical Computer Science **55** (2001), no. 3, 357–369. [p 45]

[SMSJ03] Ragnhild Van Der Straeten, Tom Mens, Jocelyn Simmonds, and Viviane Jonckers, *Using description logic to maintain consistency between UML models*, 2003, pp. 326–340. [p 43]

[Sou99] Jean-Louis Sourrouille, *UML behavior : Inheritance and implementation in current Object-Oriented languages*, UML99 — The Unified Modeling Language, 1999, p. 748. [p 43]

[Sou01] ———, *Héritage et substituabilité de comportement*, Approches Formelles dans L'Assistance au Développement Logiciel, 2001, pp. 9–22. [p 43]

[SP99] Perdita Stevens and Rob Pooley, *Using UML : software engineering with objects and components - revised edition*, Addison Wesley, November 1999. [p 33]

[SPTJ01] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel, *Refactoring UML models*, UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools, 2001, pp. 134–148. [p 43]

[Sun10] Sun, *DocumentBuilder (Java 2 platform SE 5.0)*, http ://java.sun.com/j2se/1.5.0/docs/api/javax/xml/parsers/DocumentBuilder.html, 2010. [p 86]

[TB03] Jan Tretmans and Ed Brinksma, *TorX : Automated model-based testing*, December 2003, pp. 31—43. [p 64]

[TC01] Li Tan and Rance Cleaveland, *Simulation revisited*, Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer-Verlag, 2001, pp. 480–495. [p 61, 122]

[Tre99] Jan Tretmans, *Testing concurrent systems : A formal approach*, vol. 1664, Springer-Verlag Berlin Heidelberg, 1999, pp. 46–65. [p 26, 27, 28, 64, 122]

[vdB00] Michael von der Beeck, *Behaviour specifications : Equivalence and refinement*, 2000. [p 20]

[vdB01] ———, *Formalization of UML-Statecharts*, "UML" 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools, 2001, pp. 406–421. [p 33, 39]

[vG90] Rob J. van Glabbeek, *The linear Time-Branching time spectrum (Extended abstract)*, Proceedings of the Theories of Concurrency : Unification and Extension, Springer-Verlag, 1990, pp. 278–297. [p 22, 23, 31, 63]

[vGW96] Rob J. van Glabbeek and W. Peter Weijland, *Branching time and abstraction in bisimulation semantics*, J. ACM **43** (1996), no. 3, 555–600. [p 24]

[Wir71] Niklaus Wirth, *Program development by stepwise refinement*, Communications of the ACM **14** (1971), no. 4, 221–227. [p 16]

[WZ88] Peter Wegner and Stanley Zdonik, *Inheritance as an incremental modification mechanism or what like is and isn't like*, ECOOP '88 European Conference on Object-Oriented Programming (1988), 55–77. [p 42, 43]

Annexe A

Algorithmes et démonstrations

Dans cette annexe, nous présentons quelques démonstration et l'algorithme de calcul de confrestr pour renforcer les explications dans le chapitre 5

A.1 Preuves des propositions du chapitre 5

Proposition. *Soit A un préordre, pour toute relation $X \subseteq A$*

$$A \circ X = X \circ A = A$$

Démonstration. Si on note Id la relation d'identité qui est la plus petite relation réflexive et $Id \subseteq X$, nous avons :

i.

$$\begin{aligned} & Id \subseteq X \subseteq A \\ \Rightarrow\ & A \subseteq A \circ X \subseteq A \circ A \\ \Rightarrow\ & A \subseteq A \circ X \subseteq A \\ \Rightarrow\ & A \circ X = A \end{aligned}$$

ii. On a :

$$\begin{aligned} & Id\ \subseteq X \subseteq A \\ \Rightarrow\ & A \subseteq X \circ A \subseteq A \circ A \\ \Rightarrow\ & A \subseteq X \circ A \subseteq A \\ \Rightarrow\ & A = X \circ A \end{aligned}$$

$$\Rightarrow A \circ X = X \circ A = A$$

□

Proposition. $\mathsf{red}^* \circ \mathsf{ext} = \mathsf{ext} \circ \mathsf{red}^* = \mathsf{confrestr}$

Démonstration. A partir de la définition de $\mathsf{red}^* = (\mathsf{confrestr} \cap \mathsf{red})$

i.

$$\begin{aligned} \mathsf{red}^* \circ \mathsf{ext} &= (\mathsf{confrestr} \cap \mathsf{red}) \circ \mathsf{ext} \\ &= (\mathsf{confrestr} \circ \mathsf{ext}) \cap (\mathsf{red} \circ \mathsf{ext}) \\ &= \mathsf{confrestr} \cap \mathsf{conf} \\ &= \mathsf{confrestr} \end{aligned}$$

ii.

$$\begin{aligned} \mathsf{ext} \circ \mathsf{red}^* &= \mathsf{ext} \circ (\mathsf{confrestr} \cap \mathsf{red}) \\ &= (\mathsf{ext} \circ \mathsf{confrestr}) \cap (\mathsf{ext} \circ \mathsf{red}) \\ &= \mathsf{confrestr} \cap \mathsf{conf}^* \ ^{1} \\ &= \mathsf{confrestr} \end{aligned}$$

□

A.2 Algorithme de confrestr

Algorithm 2 *boolean* =confrestr $(LTS\ q, LTS\ p)$

```
1:  AG ag1 = buildAGraph(p) ;
2:  AG ag2 = buildAGraph(q) ;
3:  AG ag3 = Merge(p, q) ;
4:  ⟨b, sp⟩ = simulate(ag3, ag2) ;
5:  si not b alors
6:      b = false ;
7:  sinon
8:      i = 0 ;
9:      n = length(sp) ;
10:     tantque (i ≤ n − 1) ∧ (second(sp[i]).acc ⊂⊄ first(sp[i]).acc) faire
11:         i = i + 1 ;
12:     fin tantque
13:     b = second(sp[i]).acc ⊂⊂ first(sp[i]).acc ;
14: finsi
15: si b alors
16:     tantque (p′ ∈ {p′ ∈ P|p′.states ∈ (V − second(sp[i]))} ∧ b) faire
17:         si p ≠ stop alors
18:             b = false ;
19:         finsi
20:     fin tantque
21: finsi
22: retourne b ;
```

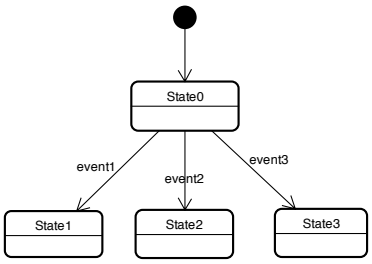
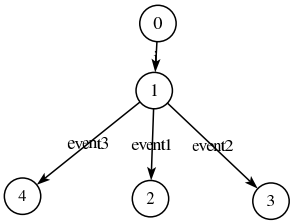
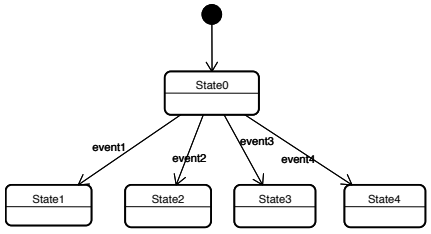
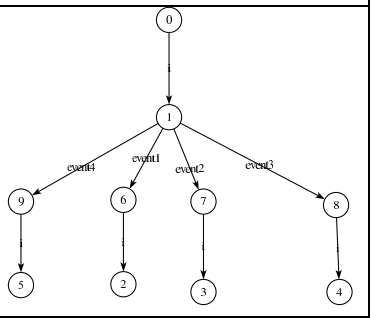
1. conf*=ext ∘ red

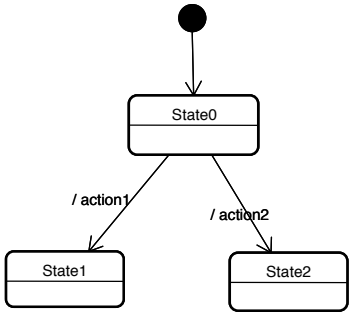
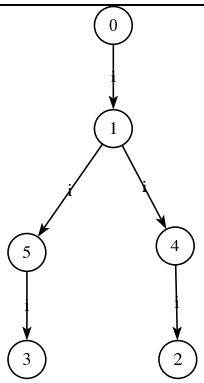
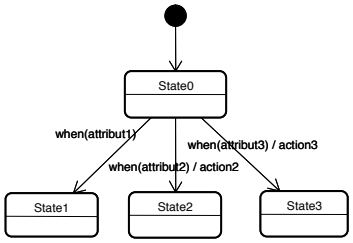
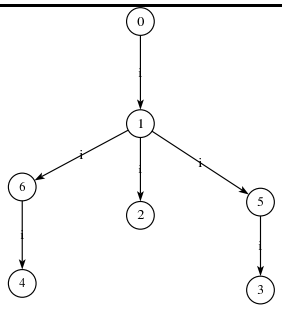
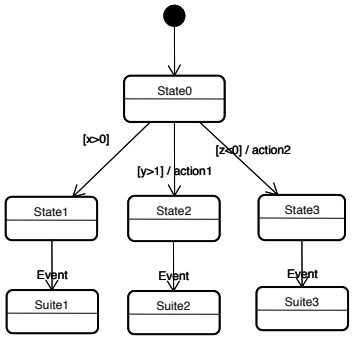
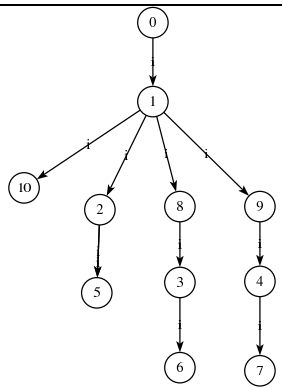
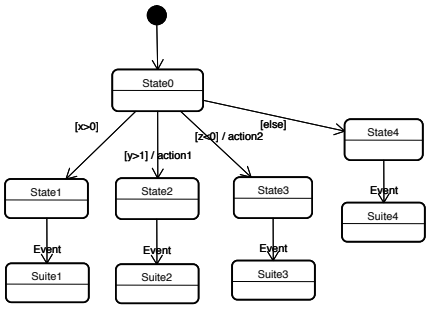
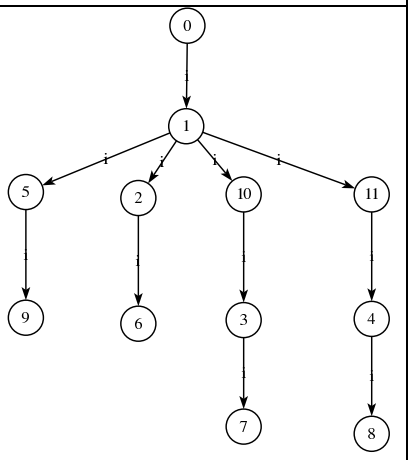
La fonction $\langle b, sp\rangle = simulate(ag_3, ag_2)$ retourne deux valeurs : le booléen b indiquant si le graphe d'acceptance ag_3 simule le premier ag_2, et sp est le vecteur de pairs d'états simulés. On désigne par $first(sp[i])$ le premier élément de la pair de rang i dans sp et $second(sp[i])$ le second. Le premier élément fait référence à un nœud du graphe ag_2, et le second, un nœud du graphe ag_3 V est l'ensemble d'états de graphe d'acceptance fusionné ag_3. La deuxième boucle cherche des états dans le premier LTS dont ces états ayant des états associés qui ne sont pas dans l'ensemble $second(sp[i])$

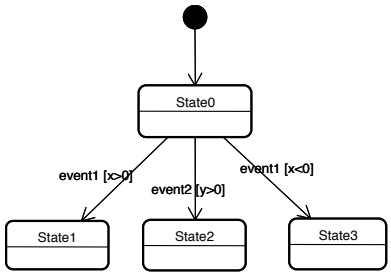
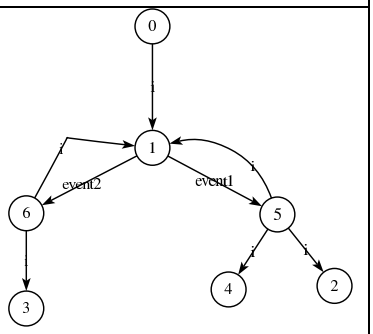
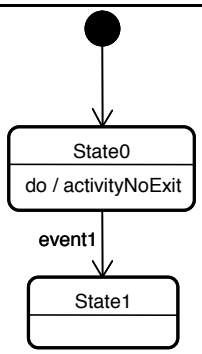
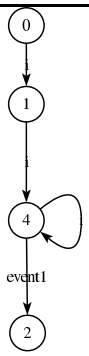
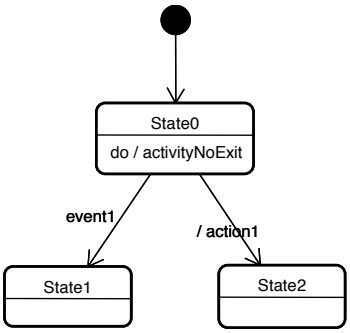
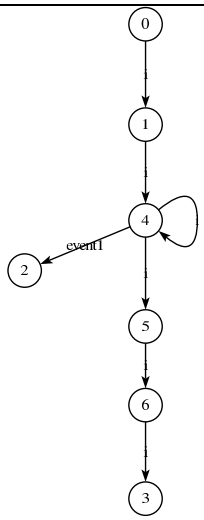
Annexe B

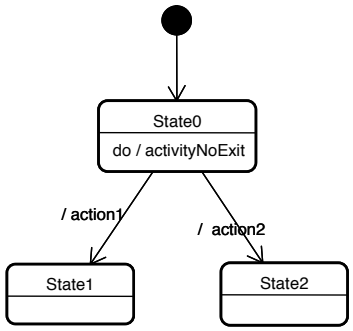
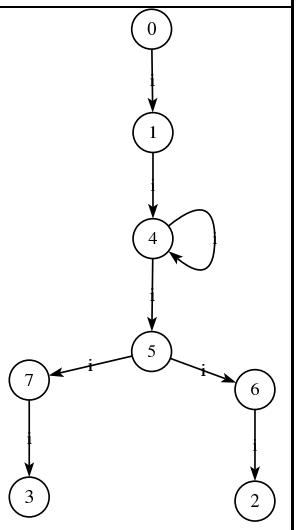
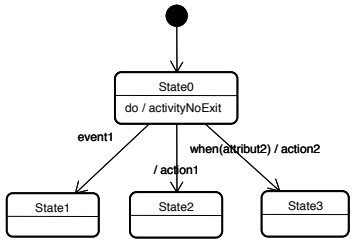
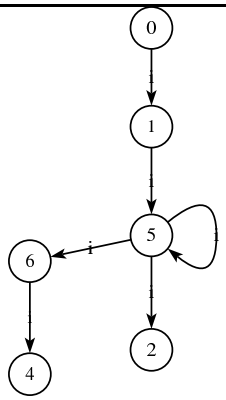
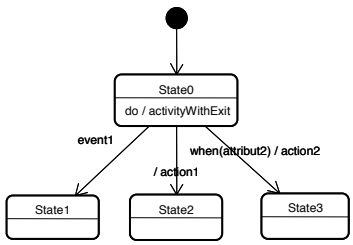
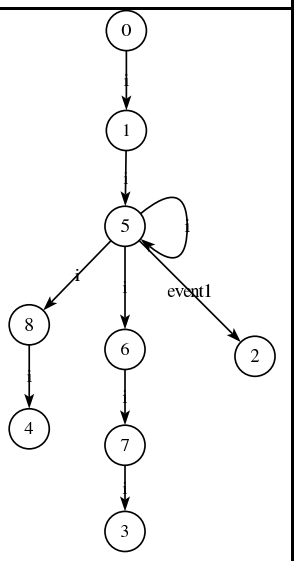
Patterns de transformation d'UML en LTS

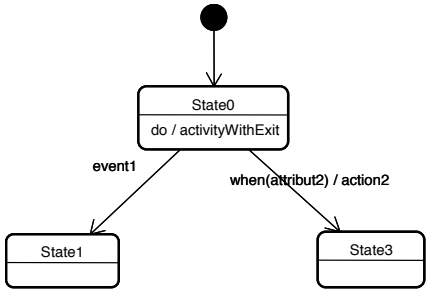
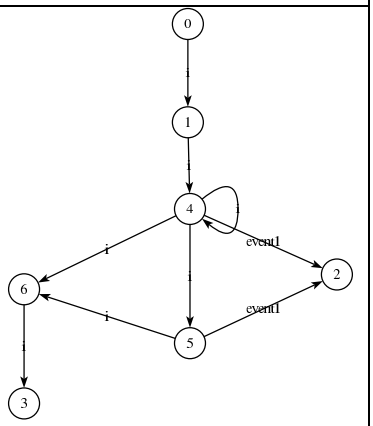
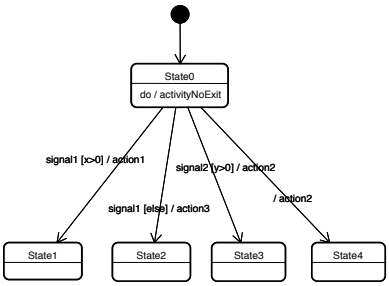
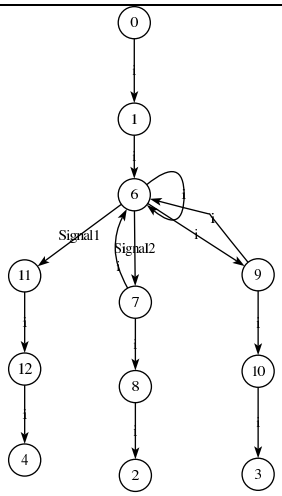
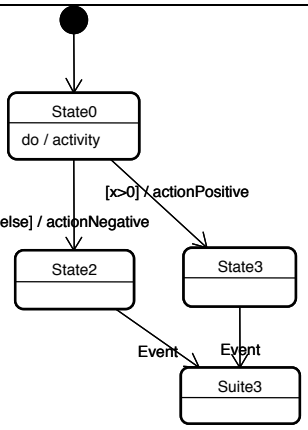
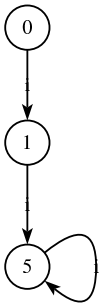
Dans ce chapitre nous présentons des cas de transformation de machines selon les règles présentées dans le chapitre 4 et leurs cas combinatoires.

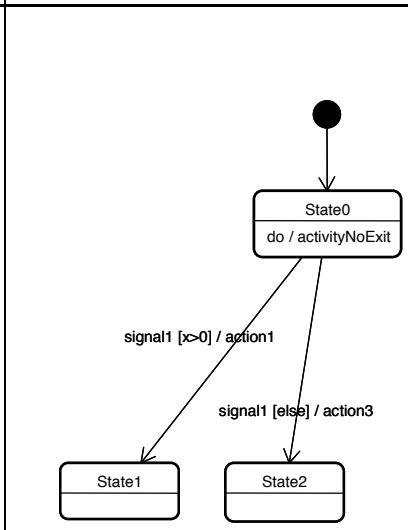
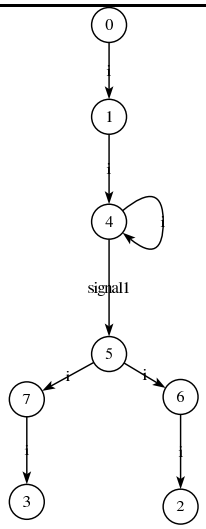
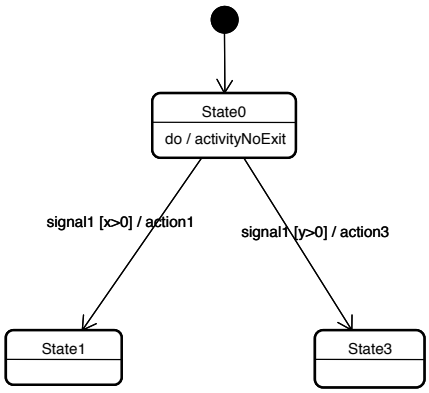
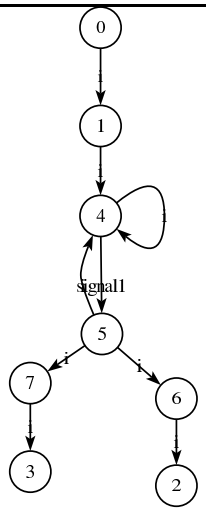
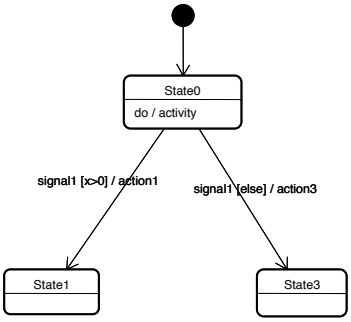
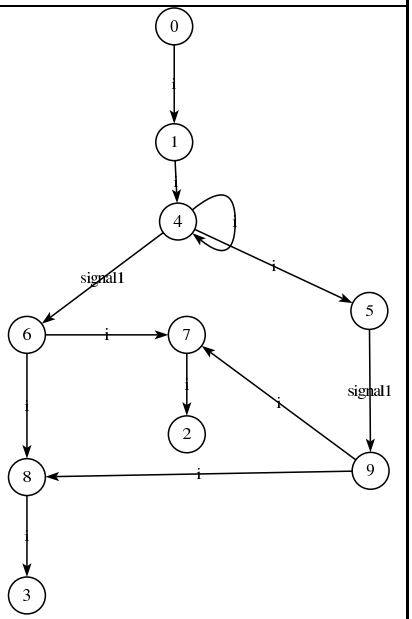
Cas	UML	LTS
1		
2		

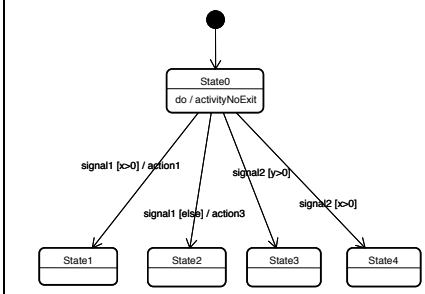
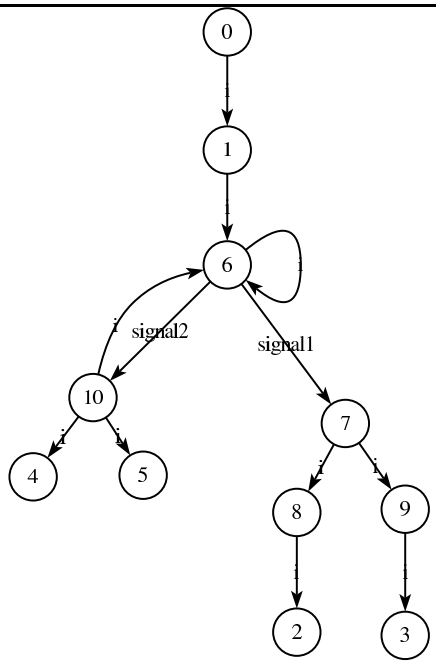
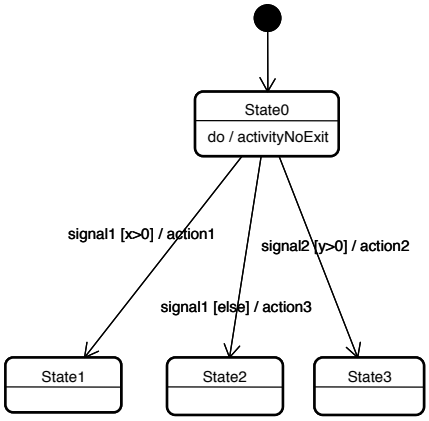
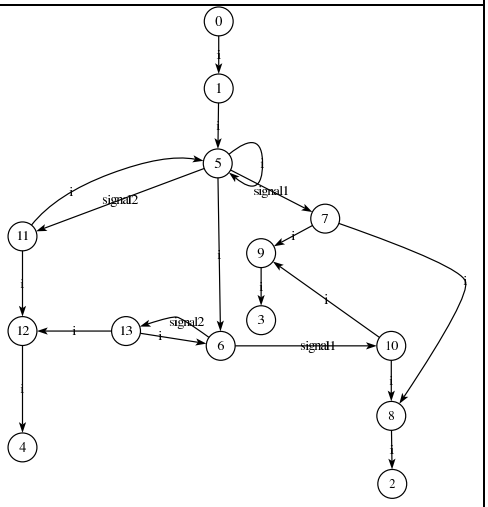
Cas	UML	LTS
3		
4-5		
6-8		
avec Else		

Cas	UML	LTS
7		
10		
11		

Cas	UML	LTS
12		
13		
14		

Cas	UML	LTS
15		
16		
17		

Cas	UML	LTS
18		
19		
20		

Cas	UML	LTS
21		
22		
23