Raffinement de modèles comportementaux UML, vérification des relations d'implantation et d'extension sur les machines d'états

Thomas Lambolais, Anne-Lise Courbis, Hong-Viet Luong

Raffinement de modèles comportementaux UML, vérification des relations d'implantation et d'extension sur les machines d'états.

Thomas Lambolais, Anne-Lise Courbis, Hong-Viet Luong

Laboratoire LGI2P, École des Mines d'Alès
Site EERIE, Parc Scientifique Georges Besse
30 035 Nîmes cedex 1, France
{Thomas.Lambolais, Anne-Lise.Courbis, Hong-Viet.Luong}@ema.fr

Résumé Nous nous intéressons à la construction de spécifications comportementales UML selon une démarche de raffinements successifs visant à construire un modèle détaillé, proche d'un modèle de l'implantation, à partir d'un modèle abstrait initial. Nous cherchons à savoir définir et vérifier une relation de raffinement entre modèles. La relation de spécialisation proposée entre classes d'un modèle objet s'apparente à une relation de raffinement, malheureusement, cette relation n'est pas définie sur les machines d'états UML. La solution que nous proposons consiste à transformer les machines d'états en systèmes de transitions étiquetés (LTS) et à définir la notion de raffinement sur ce formalisme. Dans un premier temps, nous montrons comment mettre en oeuvre la relation grâce à l'implantation des relations d'extension et d'implantation sur les LTS, ce qui n'a pas été réalisé à un coût raisonnable jusqu'ici. Dans un second temps, nous abordons la traduction de machines d'états UML vers des LTS. Les résultats de vérification obtenus sur les systèmes de transitions peuvent être réinterprétés sur les machines d'états UML, afin de proposer des corrections si nécessaire.

1 Introduction

Les qualités attendues des systèmes critiques, telles que celles de fiabilité ou de disponibilité, exigent actuellement des efforts importants et coûteux en terme de conception et de test. Il est de plus en plus indispensable d'améliorer la productivité des techniques de construction et de vérification dans les phases descendantes du développement, de manière à réduire les coûts de validation *a posteriori*. À l'heure actuelle, les descriptions comportementales réalisées en langage UML sont devenues incontournables, mais elles ne répondent pas suffisamment à ces objectifs : d'une part les modèles de machines d'états UML s'avèrent délicats à construire, d'autre part les outils associés apportent peu de réponse en terme de vérification. Nous avons proposé des solutions [7,8] pour la construction incrémentale de machines d'états UML (que l'on désignera par UML SM), consistant pragmatiquement à suivre un cycle en spirale de propositions, évaluations et corrections.

En continuité de ce travail, nous nous intéressons au raffinement de machines d'états et en particulier aux relations permettant de le caractériser entre différentes versions comportementales d'un modèle. Nous adoptons comme caractérisation du raffinement le fait que toute implantation d'un modèle raffiné est aussi une implantation du modèle abstrait [1,2]. Dans l'approche orientée objet, le raffinement n'est pas une notion native, bien que, comme nous le verrons, la relation de spécialisation entre classes présente des similitudes avec une relation de raffinement. Cette problématique est néanmoins l'une des préoccupations actuelles et depuis quelques années des travaux s'intéressent à la

cohérence inter et intra-modèles objets [11]. Mais, peu de travaux à notre connaissance traitent de la construction incrémentale par raffinement de machines d'états UML. Notre approche se base sur les travaux théoriques des modèles comportementaux plus abstraits que sont les systèmes de transitions étiquetées (LTS). Nous proposons une définition du raffinement de LTS faisant appel aux deux relations d'extension et d'implantation préalablement définies sur les LTS. Nous proposons des techniques de vérification de ces relations [17,18] en nous basant sur la bisimulation de graphes transformés [3]. Le raffinement étant défini et outillé sur les LTS, il est nécessaire de transformer les machines d'états en LTS. Les LTS étant plus abstraits que les machines d'états, l'interprétation des résultats exige une certaine réserve que nous exprimerons dans la suite.

Ce papier est organisé de la manière suivante. Au paragraphe 2, nous posons la problématique sur un exemple qui sera repris en détails au paragraphe 6. Au paragraphe 3, nous présentons les différentes approches qui donnent une définition du raffinement, nous positionnons notre propre définition puis nous donnons les définitions formelles nécessaires à la compréhension de l'article. Dans le paragraphe 4, nous énonçons des théorèmes permettant d'appréhender notre méthode de calcul du raffinement. La transformation des machines d'états UML en LTS est abordée au paragraphe 5. Enfin, nous montrons au paragraphe 6, l'application des relations, leur mise en œuvre et programmation sur l'exemple préalablement présenté. Les conclusions et perspectives sont présentées au paragraphe 7.

2 Positionnement de la problématique à travers un exemple

Ce paragraphe illustre sur un cas concret simple mais représentatif, des problèmes posés par la transformation de modèles comportementaux selon une démarche de raffinement. L'exemple choisi sera repris au paragraphe 6.

Nous considérons un téléphone spécifié selon un point de vue utilisateur par une classe très simple, appelée par la suite $PhoneSpec$, dont l'interface sera composée de signaux émis par l'utilisateur (décrocher, numéroter, raccrocher, parler) et de ceux émis par le téléphone (ligne occupée, connexion interrompue, appel entrant, erreur sur appel).

Nous abordons le raffinement selon deux approches. Une première approche conduit à réaliser le modèle d'une implantation possible du téléphone. Il est alors nécessaire d'identifier des actions ou des signaux qui étaient restés abstraits et de les reformuler en tenant compte du niveau de détail nécessaire pour satisfaire l'implantation. Le modèle d'une implantation possible de $PhoneSpec$ doit par exemple prendre en compte que le téléphone est connecté à un réseau et que l'appel entrant est en fait un signal venant du réseau qui sera ensuite transmis à l'utilisateur, ou encore, que le téléphone doit pouvoir détecter des numéros mal formés. Cela conduit le concepteur à définir une nouvelle classe, appelée $Phone$ dont l'interface sera identique à celle de $PhoneSpec$, avec en plus, l'ensemble des signaux transmis entre le téléphone et le réseau.

Une seconde approche de raffinement conduit à spécifier un nouveau téléphone dont les fonctionnalités seraient étendues. Supposons que l'on souhaite spécifier un téléphone gérant les double appels. Le concepteur définit une nouvelle classe, appelée $DoubleCall$, dont l'interface comprend les signaux préalablement cités pour la classe $PhoneSpec$ et des signaux spécifiques à la nouvelle fonctionnalité : arrivée d'un second appel, acceptation ou refus de l'utilisateur pour interrompre le premier appel et prendre le second.

À ce stade, trois classes ont été définies en termes d'interfaces et de machines d'états. Nous ne détaillons pas pour le moment les machines d'états car cela n'est pas nécessaire pour exposer la problématique. Se pose alors le problème de la cohérence entre ces classes (figure 1 ; pour visualiser le détail des machines d'états, se reporter aux figures 2 et 3). Dans la mesure où on souhaite que la classe *Phone* soit une réalisation de la classe *PhoneSpec*, quelle est la relation permettant de comparer leur machine d'états ? Comment la vérifier ? Les mêmes questions se posent si l'on souhaite que *DoubleCall* soit une extension de la classe *PhoneSpec*.

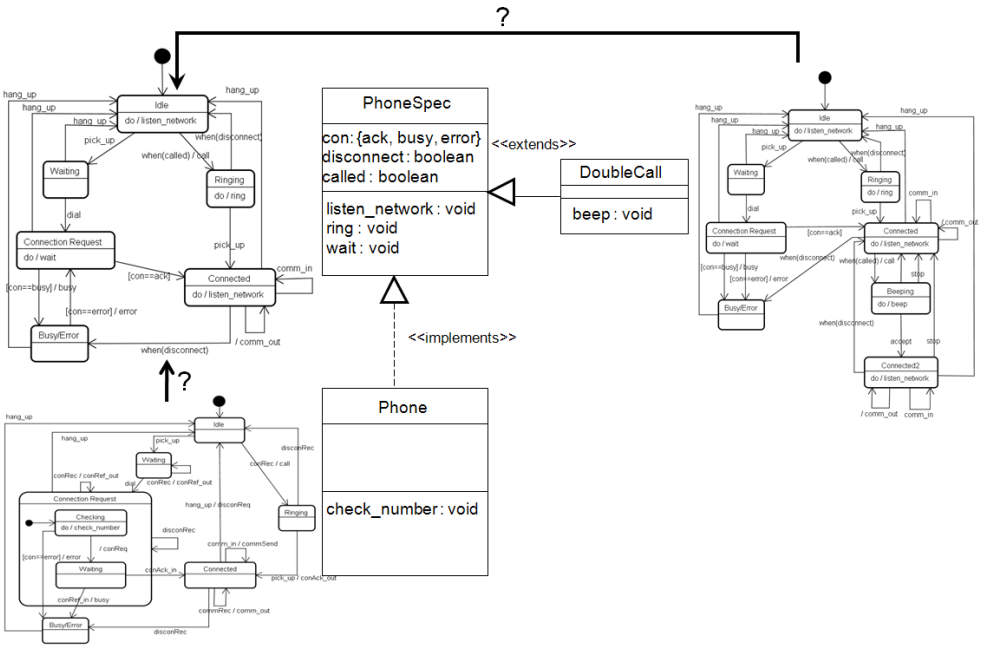

FIGURE 1. Relations entre les classes *Phone*, *PhoneSpec* et *DoubleCall*

3 Raffinement

Ci-dessous, nous examinons les notions générales de raffinement de modèles, d'abord dans le cadre d'UML, ensuite sur les modèles comportementaux.

3.1 Raffinement et cohérence de modèles UML

Les travaux relatifs au raffinement de modèles UML se trouvent dans le cadre plus général des études de cohérence entre modèles UML. Dans [21], le raffinement est vu généralement comme la relation existant entre un modèle spécifié à un certain niveau de détail et une spécification enrichie, comme par exemple entre un modèle d'analyse et un modèle de conception. [11,10] donne une vision plus précise, qui peut s'exprimer sur les diagrammes de collaboration en termes de propriétés relatives aux classes et opérations raffinées.

Deux types de raffinement sont envisagés. Le premier que l'on appellera « *transformationnel* » consiste à ajouter des éléments de modélisation afin d'obtenir dans l'étape finale un modèle de la réalisation du système. Le second point de vue, que l'on appellera « *additionnel* », consiste à étendre les fonctionnalités d'un système existant afin de définir les spécifications d'un nouveau système. Notons que ces deux points de vue ne sont pas exclusifs, et qu'une démarche de raffinement peut les combiner.

Ainsi, [22] propose une définition du raffinement sur les machines d'états, en le rapprochant du problème de *refactoring* de modèles. Le processus de raffinement consiste à ajouter à chaque étape des détails plus concrets. La préservation des propriétés est ici étudiée à l'aide de logique de description.

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). Ceci est en accord avec les définitions précédentes, mais à l'inverse de [22] nous ne souhaitons pas exprimer explicitement les propriétés à préserver. Nous préférons nous orienter vers une démarche de vérification en intension des propriétés liées à l'ajout de détails et à la réduction de l'indéterminisme.

3.2 Relations existantes : vers une relation de raffinement sur des modèles comportementaux

Dans cette partie, nous donnons une présentation informelle et une comparaison des principaux préordres définis sur les LTS [20], susceptibles de convenir en tant que relation de raffinement. Dans un premier temps nous considérerons les systèmes de transitions étiquetées et ensuite les machines d'états UML. Nous cherchons une relation asymétrique et transitive assurant deux propriétés : l'ajout de détails et la réduction de l'indéterminisme.

Relations de raffinement sur les LTS. Les systèmes de transitions étiquetées seront présentés formellement dans la suite, la compréhension intuitive utile pour le moment est qu'il s'agit de machines interagissant avec leur environnement au moyen d'*actions*. Ce sont des descriptions simples et abstraites, ne précisant ni si les actions sont en entrée ou en sortie (il n'y a pas de distinction événement / action), 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 ainsi, souvent, 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.

Inclusion de traces. Dans le cas des LTS, ajouter des détails consiste à définir de nouvelles traces et éventuellement de nouvelles actions. Une trace est une suite d'actions observables s'exécutant à partir de l'état initial. L'ajout de traces répond au premier critère souhaité sur le raffinement, à savoir l'ajout de détails . Cependant, la description étendue peut être moins déterministe que l'originale, dans le sens où elle pourra refuser des fonctionnalités définies comme nécessaires dans la spécification.

Relations de simulation. Pour les mêmes raisons, les relations de simulation proposées par Milner [20] ne sont pas adéquates pour représenter le raffinement, puisqu'un modèle simule une spécification dès qu'il *peut* faire ce que la spécification définit. Nous voulons définir un raffinement qui *doit* accepter ce que la spécification doit accepter.

Relations de conformité. La relation $conf$ a été proposée comme une relation d'implantation pour vérifier si un modèle d'une implantation respecte sa spécification [14,23]. C'est une formalisation de la relation de conformité définie pour le test. Une implantation I est conforme à sa spécification S si elle doit faire tout ce que S doit faire. La relation $conf$ capture 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.

Les relations d'extension et de réduction [14,23] sont des relations de conformité combinées à des extensions ou des réductions de traces. Les deux sont transitives. Ainsi, la relation d'extension combine la réduction de l'indéterminisme et l'extension de traces. Cependant, ces relations ($conf$, red et ext) s'avèrent difficiles à vérifier en pratique [13]. Des résultats intéressants ont été trouvés sur les relations similaires que sont les préordres de possibilité et de nécessité définis par Hennessy [9] et implantés par Cleaveland [3]. Le préordre de nécessité correspond à la relation de réduction, tout en prenant en compte le cas des processus divergents (c'est-à-dire les processus pouvant entrer dans des séquences infinies d'actions internes). Le tableau 1 propose un résumé de ces relations selon les critères considérés.

	Réduction de l'indéterminisme	Extension des traces
inclusion de traces (ou préordre de possibilité)	$\varnothing$	$\star\star\star$
relation de simulation	$\varnothing$	$\star\star\star$
$conf$	$\star\star\star$	$\star$
red	$\star\star\star$	$\varnothing$
ext	$\star\star\star$	$\star\star\star$
préordre de nécessité	$\star\star\star$	$\varnothing$

$\star$	peut être supporté.
$\star\star\star$	est garanti.
$\varnothing$	n'est pas supporté.

TABLE 1. Comparaison des relations envisagées selon deux critères.

La relation d'extension est donc la seule qui présente les deux critères recherchés. Examinons plus en détails si elle répond à la définition d'une relation de raffinement. Guy Leduc a établi que $conf$ et ext [14] sont telles que :

$$R\ ext\ A \Rightarrow \forall P.\ (P\ conf\ R \Rightarrow P\ conf\ A). \tag{1}$$

En considérant que $conf$ est une relation d'implantation, cette dernière propriété établit que ext est une relation de raffinement, mais que ce n'est pas la plus large. Elle est plus forte que la relation de raffinement $\sqsupseteq$ caractérisée par :

$$R\ \sqsupseteq A \Leftrightarrow \forall P.\ (P\ imp\ R \Rightarrow P\ imp\ A). \tag{2}$$

où $R \sqsupseteq A$ signifie que R raffine A et $P\ imp\ R$ signifie que P est une implantation de R. Ceci rejoint les formulations données par exemple dans [2], en accord avec la définition de raffinement en langage Z ou B [1].

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. Si l'inclusion de traces et les relations de simulation sont vérifiées par des boîtes à outils telles que CADP [6], à notre connaissance, ni $conf$, ni ext ne sont actuellement outillées.

Relations de raffinement sur les machines d'états UML. Considérons qu'une machine d'états UML décrit les comportements attendus des objets d'une classe. Si une classe C_R est une spécialisation d'une classe C_A, ce que l'on peut écrire $C_R \,\mathrm{extends}\, C_A$, cela signifie que toute instance de C_R peut se comporter comme une instance C_A : toute instance de C_R est aussi une instance de C_A. Plus formellement, nous pouvons écrire :

$$C_R \,\mathrm{extends}\, C_A \Rightarrow \mathrm{Instances}(C_R) \subseteq \mathrm{Instances}(C_A). \tag{3}$$

Une définition plus précise peut se trouver dans [5]. Les propriétés (1) et (3) sont similaires. Ceci signifie que la relation de spécialisation sur les classes est plus forte qu'une relation de raffinement. Mais qu'en est-il du raffinement des machines d'états associés aux classes ? Quel type de relation peut-on définir pour établir qu'une machine M_R raffine une machine M_A, et comment peut-on vérifier cette relation ? Comme les machines d'états UML peuvent être vues comme des extensions des systèmes de transitions étiquetées, notre point de vue est de considérer les LTS comme des interprétations abstraites des machines d'états. Nous ne définissons pas pour l'instant de relation détaillée sur les machines d'états (comme nous l'avons fait pour la conformité dans [7]), mais nous élaborons une traduction des machines d'états UML vers les LTS. L'interprétation abstraite des résultats obtenus nous permet de dire que si la relation n'est pas satisfaite sur les LTS, elle ne l'est pas sur les machines d'états.

3.3 Définitions formelles

Nous adoptons les notations suivantes pour les notions usuelles de LTS [19,20], et les définitions liées à la conformité [16,23,4].

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 1 (Systèmes de transitions étiquetées [20]). *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 $\rightarrow \subseteq P \times A \times P$, et 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. Nous définissons :

$$
\begin{aligned}
p \xrightarrow{a} p' &=_{def} (p, a, p') \in \rightarrow_P \\
p \xrightarrow{a_1 \cdots a_n} p' &=_{def} \exists p_0, \ldots, p_n.\ p = p_0 \xrightarrow{a_1} \ldots \xrightarrow{a_n} p_n = p' \\
p \xrightarrow{a_1 \cdots a_n} &=_{def} \exists p'.\ p \xrightarrow{a_1 \cdots a_n} p' \\
p \stackrel{\varepsilon}{\Longrightarrow} p' &=_{def} p = p' \text{ ou } p \xrightarrow{\tau \cdots \tau} p' \\
p \stackrel{a}{\Longrightarrow} p' &=_{def} \exists p_1, p_2.\ p \stackrel{\varepsilon}{\Longrightarrow} p_1 \xrightarrow{a} p_2 \stackrel{\varepsilon}{\Longrightarrow} p' \\
p \stackrel{a_1 \cdots a_n}{\Longrightarrow} p' &=_{def} \exists p_0, \ldots, p_n.\ p = p_0 \stackrel{a_1}{\Longrightarrow} \ldots \stackrel{a_n}{\Longrightarrow} p_n = p' \\
p \stackrel{\sigma}{\Longrightarrow} &=_{def} \exists p'.\ p \stackrel{\sigma}{\Longrightarrow} p' \\
\text{Traces (observables) : } Tr(p) &=_{def} \{\sigma \in L^* \mid p \stackrel{\sigma}{\Longrightarrow}\} \\
p \text{ after } \sigma &=_{def} \{p' \mid p \stackrel{\sigma}{\Longrightarrow} p'\} \\
D(p, a) &=_{def} \{p' \mid p \xrightarrow{a} p'\} \\
Out(p) &=_{def} \{a \in L \mid p \xrightarrow{a}\} \\
Out(p, \sigma) &=_{def} \bigcup_{p' \in p \text{ after } \sigma} Out(p')
\end{aligned}
$$

Une trace est une exécution partielle observable ; c'est une séquence d'actions visibles possible à partir de l'état initial. $Out(p, \sigma)$ est l'ensemble des actions observables de p après la trace σ. Nous retenons comme définition de l'ensemble d'acceptance de p après une trace σ :

Définition 2 (Ensemble d'acceptance [15]). $Acc(p, \sigma) = \{X \mid \exists p' \in p \textit{ after } \sigma.\ X = Out(p', \varepsilon)\}$

Cette notion sera utilisée au paragraphe suivant pour construire des graphes d'acceptance. Un ensemble d'acceptance représente les différents ensembles possibles d'actions nécessaires après une trace. L'inclusion d'ensembles d'acceptance définie par Cleaveland [3] nous permet de savoir si un processus est plus déterministe qu'un autre.

Définition 3 (Inclusion d'ensemble d'ensembles [3]). *Soient* $A, B \subseteq 2^{\mathcal{L}}$. $A \subset\subset B$ *si* $\forall S \in A.\ \exists S' \in B.\ S' \subseteq S$.

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

Définition 4 (Conformité [15]). q *conf* p *si* $\forall \sigma \in Tr(p).\ Acc(q, \sigma) \subset\subset Acc(p, \sigma)$.

L'extension et la réduction sont définies comme des extension ou réduction de traces préservant la conformité.

Définition 5 (Réduction [15]). q *red* p *si* $Tr(q) \subseteq Tr(p)$ *et* q *conf* p.

Définition 6 (Extension [15]). q *ext* p *si* $Tr(p) \subseteq Tr(q)$ *et* q *conf* p.

La relation *conf* n'est pas transitive. Les relations *red* et *ext* sont réflexives et transitives.

4 Calculabilité des relations de conformité

4.1 Bisimulation

Nous reformulons et généralisons la définition de la bisimulation donnée par Milner [20] en s'inspirant des travaux de Cleaveland [3].

Définition 7 (Bisimulation [3]). *Soient $\Pi \subseteq \mathcal{P} \times \mathcal{P}$ et $\Psi_1, \Psi_2 \subseteq \mathcal{P} \times Act$. La relation $\mathcal{R}$ $\langle \Pi, \Psi_1, \Psi_2 \rangle$ est une bisimulation si $\mathcal{R} \subseteq \Pi$ et $p\mathcal{R}q$ implique :*

1. $\langle p, a \rangle \in \Psi_1 \Rightarrow (p \xrightarrow{a} p' \Rightarrow \exists q'.\ q \xrightarrow{a} q' \wedge p'\mathcal{R}q')$
2. $\langle q, a \rangle \in \Psi_2 \Rightarrow (q \xrightarrow{a} q' \Rightarrow \exists p'.\ p \xrightarrow{a} p' \wedge p'\mathcal{R}q')$

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

Définition 8 (Plus grande bisimulation [3]).
$p \sim^{\Pi}_{\langle \Psi_1, \Psi_2 \rangle} q$ *s'il existe une bisimulation $\mathcal{R}$ $\langle \Pi, \Psi_1, \Psi_2 \rangle$ avec $p\mathcal{R}q$.*

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

4.2 Graphes d'acceptance

Un graphe d'acceptance d'un LTS p est le graphe déterministe obtenu à partir de p, où les nœuds sont étiquetés par les ensembles d'acceptances précédemment définis. La fermeture d'un ensemble d'états est définie comme suit :

Définition 9 (Fermeture [3]). *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 [3,12].

Définition 10 (Graphe d'acceptance). *Étant donné un LTS $\langle P, A, \rightarrow, p \rangle$, $A = L \cup \{\tau\}$, son graphe d'acceptance $\mathcal{A}(p)$ est le LTS déterministe $\langle T, L, \rightarrow_T, t \rangle$ tel que :*

1. *T est un ensemble de noms d'états.*
2. *$\rightarrow_T$ est un ensemble de transitions de $T \times L \times T$.*
3. *$t \in T$ est l'état initial.*
4. *Pour tout état u de T, on associe les ensembles $u.states \in 2^P$ et $u.acc \subseteq 2^L$ tels que :*
   - *$u.acc = \{X \mid X = Out(q, \varepsilon) \wedge q \in u\}$;*
   - *$t.states = (\{p\})^{\varepsilon}$;*
   - *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 [3] sans prendre en compte la notion de divergence. Nous avons démontré [17,18] deux théorèmes permettant de calculer les relations d'extension, de réduction et de conformité.

4.3 Calculabilité des relations d'extension et de réduction

Soient p un LTS et $\mathcal{A}(p)$ son graphe d'acceptance, on a $Tr(\mathcal{A}(p)) = Tr(p)$ [3]. Le théorème suivant (démontré dans [17]) permet de ramener le calcul des relations d'extension et de réduction à des simulations sur les graphes d'acceptance.

Théorème 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* $p \Leftrightarrow t \precsim^{\Pi} u$

4.4 Calculabilité de la relation de conformité

Telle qu'introduite par Khendek [12], l'opération $Merge$ permet de fusionner deux 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 écrit $Merge(p, q)$ le LTS de $Merge(\mathcal{A}(p), \mathcal{A}(q))$. Le graphe fusionné est toujours la plus petite extension cyclique commune aux deux graphes initiaux [12].

Nous avons démontré dans [17] la relation suivante :

Théorème 2. *Soient p et q deux LTS, q conf $p \Leftrightarrow q$ red $Merge(p, q)$.*

Ainsi que :

Corollaire 1. *Soient p et q deux LTS, q conf $p \Leftrightarrow Merge(\mathcal{A}(p), \mathcal{A}(q)) \succsim^{\Pi} \mathcal{A}(q)$.*

Ce théorème et son corollaire nous permettent de mettre en œuvre la relation de conformité entre LTS. L'outil que nous avons développé suit strictement ce résultat : il calcule les deux graphes d'acceptance, les fusionne, et puis vérifie si la réduction est satisfaite. Son utilisation est montrée au paragraphe 6.

5 Interprétation de machines d'états UML en LTS

Les machines d'états sont plus riches que les LTS : en particulier, la distinction entre les concepts d'événement, d'action et d'activité n'existe pas dans le formalisme des LTS, ainsi que les notions de garde et de hiérarchie. Il a donc été nécessaire dans un premier temps d'établir une correspondance entre les concepts des deux formalismes (cf. tableau 2).

Les événements, actions et activités UML peuvent se traduire systématiquement par des actions LTS. Nous distinguons cependant les événements explicites (appel de méthode et réception de signal) qui se traduiront par des actions visibles, et les événements implicites (événement de terminaison à la fin d'une activité) qui se traduiront par une action interne. Les événements de changement et les événements temporisés seront également traduits par l'action interne.

En ce qui concerne les actions, il est nécessaire de définir en UML quelles sont les actions observables. Pour cela, on peut avoir recours à l'utilisation de *tags* UML, tel que nous l'avons manipulé dans [7]. Une action observable se traduira par une action du LTS, une action non observable par une action interne.

Machines d'états	LTS	Machines d'états	LTS
action observable	action	action non observable	τ
call ou change event	action	*time, change, complete event*	τ
activité visible	partie du LTS	activité interne	τ
hiérarchie	LTS à plat	entry	τ
gardes	plusieurs schémas. Indéterminisme. Voir tab. 3	exit	τ
historique	non traité	pseudo-état de choix	non traité

TABLE 2. Correspondance de concepts

Une activité UML est représentée par une action LTS en boucle. Par abus de notation, l'action porte le même nom que l'activité mais elle représente en fait un pas non interruptible de l'exécution de l'activité. La fin de l'activité est représentée par une action interne permettant de sortir de l'état d'exécution de l'activité. L'interruption de l'activité par un événement est représentée par une action LTS portant la même étiquette que l'événement.

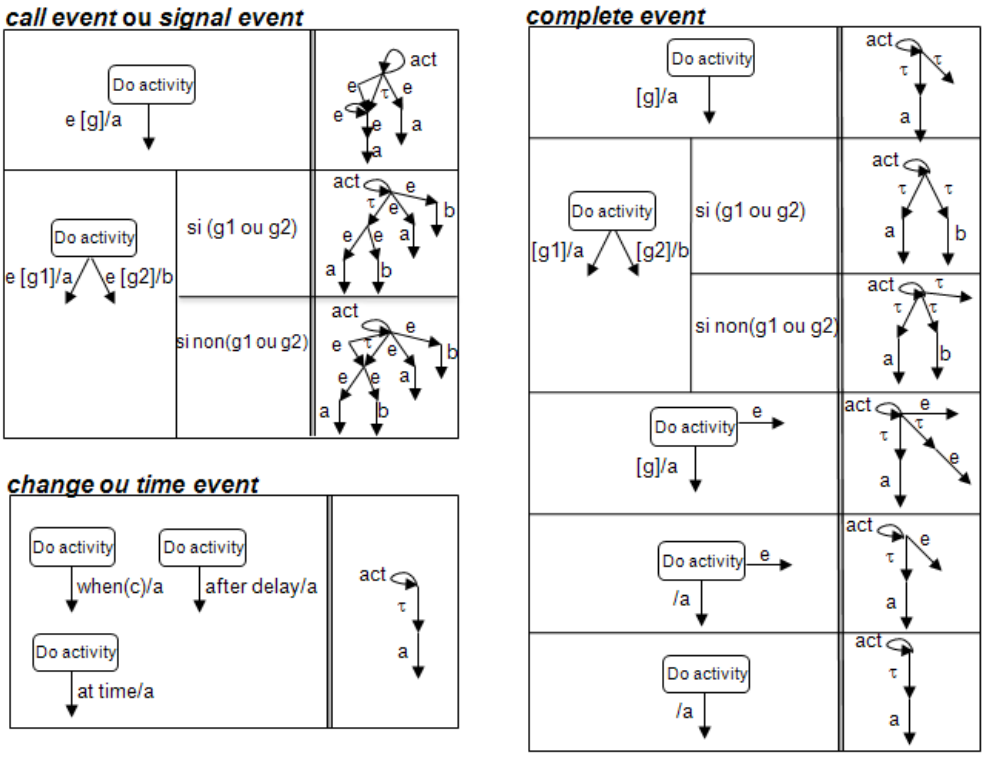

TABLE 3. Règles de transformation des machines d'états en LTS, dans le cas d'états comportant une activité

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 (cf. tableau 3). La difficulté vient du fait que la garde induit la possibilité d'un blocage ou d'un

refus. Par exemple, la transition UML « $\xrightarrow{\mathrm{e[g]/a}}$ » doit se traduire par un LTS qui pourra refuser d'exécuter l'action a après l'action e, ce qui correspond au cas où la garde est fausse. Le tableau 3 donne uniquement les règles correspondant à des états comportant une activité. Dans le cas où aucune activité n'est présente, les schémas LTS se simplifient. Notons qu'il n'y a pas de correspondance bijective entre états de la machine UML et ceux du LTS.

La satisfaction des relations de conformité sur les LTS garantit l'absence de refus définitif. D'autres propriétés plus précises des machines d'états (telles qu'un refus provisoire ou conditionnel) ne sont pas détectées. La détection d'une erreur sur les LTS correspond ainsi à un risque d'erreur sur les machines d'états.

6 Outil de vérification et analyse du cas d'étude

Cette partie montre les résultats obtenus grâce à l'outil de vérification que nous avons développé. Nous donnons tout d'abord les étapes principales des algorithmes de calcul des relations de conformité et d'extension. Après avoir présenté les machines d'états des classes exposées au paragraphe 2, nous montrons quelles erreurs peuvent être mises en évidence grâce au calcul de ces deux relations.

6.1 Étapes de vérification des relations de conformité et d'extension

La vérification de la relation d'extension entre deux machines d'états UML repose sur trois étapes :

- 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 bisimulation et l'inclusion des ensembles d'acceptance.

La vérification de la relation de conformité inclut une étape supplémentaire de fusion des graphes d'acceptance. Nous ne détaillerons pas dans la suite les résultats de chacune de ces étapes ; pour plus de détails, se reporter à [18,17].

6.2 Analyse des classes $PhoneSpec$ et $Phone$ par la relation d'implantation

Nous nous intéressons dans ce paragraphe à la relation entre la classe $PhoneSpec$ représentant la spécification de haut niveau et la classe $Phone$ modélisant une implantation possible.

$SM_{PhoneSpec}$ (figure 2.a) représente la machine d'états de la classe $PhoneSpec$. Elle décrit deux possibilités :

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

SM_{Phone} (figure 2.b) représente la machine d'états de la classe $Phone$. 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 une variable interne qui est utilisée pour décrire la fin de communication (cf. transition *when(disconnect)*, figure 2.a) en sortie des états *Connected* et *Ringing* mais un signal émis par le réseau (cf. transition *disconRec*, figure 2.b). Notons que ces signaux, non visibles par l'utilisateur, seront masqués et transfomés en transitions internes τ dans le LTS.

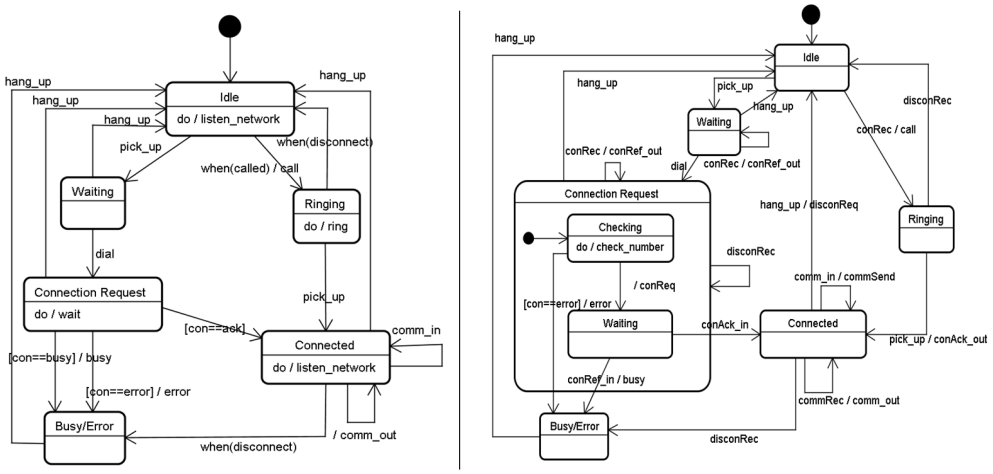

FIGURE 2. (a) machine d'états de *PhoneSpec* (b) machine d'états de *Phone*

Se pose donc la question de la conformité de ces deux machines. L'analyse des deux LTS par l'outil de vérification indique qu'il y a conformité. Au sens de la conformité entre LTS (absence de refus définitifs), les machines sont conformes. Il peut néanmoins subsister des erreurs plus fines spécifiques aux machines d'états UML (blocages provisoires ou conditionnels).

6.3 Analyse des classes $PhoneSpec$ et $DoubleCall$ 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 $DoubleCall$, représentant un téléphone double appel. La machine d'états $SM_{DoubleCall}$ (figure 3) est très proche de celle du téléphone simple. Elle diffère au niveau de l'état *Connected* : une transition sortante, étiquetée *when(called)/call* est ajoutée modélisant l'arrivée du second appel. Plusieurs scénarios peuvent alors se produire : l'utilisateur accepte l'appel (transition *accept* en sortie de l'état *Beeping*), la machine passe alors dans l'état *Connected2* ; l'utilisateur refuse l'appel (transition *stop*) et revient en communication avec son correspondant initial ; le second correspondant met fin à la communication (transition de fin d'activité *beep*).

Se pose donc la question de la vérification de la relation d'extension. L'analyse des deux machines par l'outil de vérification révèle qu'il n'y a pas extension. L'échec est mis en évidence après les traces *pick_up ; dial* ou *call ; pick_up*. Il apparaît que $SM_{DoubleCall}$

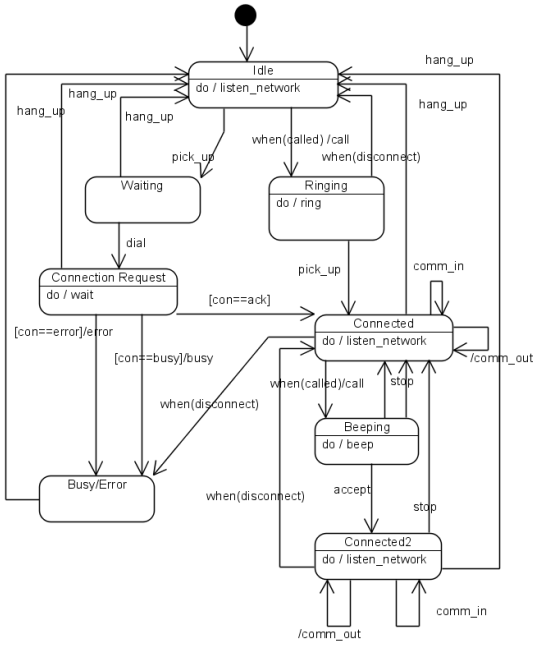

FIGURE 3. Machine d'états de $DoubleCall$

peut refuser à la fois l'action *hang_up* et les actions *comm_in*, *comm_out* alors que SM_{Phone} peut refuser chacune de ces actions mais jamais les trois en même temps. Lors d'un nouvel appel entrant, l'utilisateur est conduit à ne pas pouvoir raccrocher ni communiquer.

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 (figure 4) consiste à décomposer la transition *when(called)/call* en deux transitions distinctes dont l'état intermédiaire accepterait les actions *comm_in* et *comm_out*. Avec cette modification, la relation d'extension entre les deux machines est vérifiée.

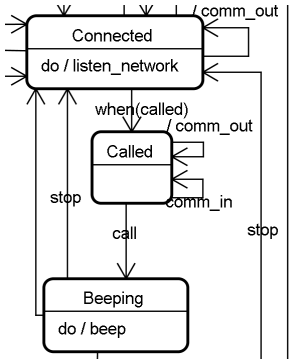

FIGURE 4. Correction à apporter à $SM_{DoubleCall}$ pour qu'il y ait relation d'extension

6.4 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 machines d'états $Phone$. En revanche, nous avons donc pu détecter et corriger des erreurs sur la machine d'états de la classe $DoubleCall$. 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 terme d'actions, ce qui facilite la correction des machines d'états.

7 Conclusion et perspectives

Dans ce papier, nous avons étudié le raffinement de machines d'états UML selon deux aspects, qualifiés de transformationnel et additionnel. La relation d'extension définie sur les LTS couvre ces deux aspects. Pour la dernière étape du raffinement, correspondant à la définition du modèle de l'implantation, la relation plus faible de conformité peut être utilisée. Nous avons proposé des algorithmes pour la vérification de ces relations. Afin de pouvoir comparer des machines d'états UML, nous avons systématisé leur traduction vers les LTS. La mise en œuvre de cette démarche est illustrée sur un cas d'étude. Une erreur est ainsi mise en évidence sur les modèles développés.

La relation de raffinement retenue n'étant pas la plus grande, elle peut cependant ne pas être satisfaite alors qu'il existe un lien plus faible de raffinement. Un première perspective consiste à trouver une réalisation de la plus grande relation de raffinement définie comme suit :

$$R \; refines \; A \Leftrightarrow \forall P.\, (P \; conf \; R \Rightarrow P \; conf \; A). \tag{4}$$

Cette relation a déjà été caractérisée dans [15] mais n'a jamais été programmée.

La seconde perspective consiste à prendre en compte davantage de caractéristiques des machines d'états UML, telles que le parallélisme, les pseudo-états de choix ou d'historique, ainsi qu'une architecture de plusieurs machines communicantes.

Références

1. Jean-Raymond Abrial. *The B-Book : Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. Eerke Boiten and Marius Bujorianu. Exploring UML refinement through unification. In *Critical Systems Development with UML - Proceedings of the UML'03 workshop*, Lecture Notes in Computer Science, pages 47–62. Technische Universität München, 2003.
3. Rance Cleaveland and Matthew Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing*, 5 :1–20, 1993.
4. Rance Cleaveland and Bernhard Steffen. A preorder for partial process specifications. In *CONCUR '90 : Proceedings on Theories of concurrency : unification and extension*, page 141—151, New York, NY, USA, 1990. Springer-Verlag New York, Inc.
5. Roland Ducournau. "Real world" as an argument for covariant specialization in programming and modeling. In *Advances in Object-Oriented Information Systems*, pages 3–13, 2002.
6. Hubert Garavel, Radu Mateescu, Frédéric Lang, and Wendelin Serwe. *CADP 2006 : A Toolbox for the Construction and Analysis of Distributed Processes*, pages 158–163. 2007.

7. Olivier Gout. *Développement incrémental de spécifications orientées objets*. PhD thesis, école de mines d'Alès, université de Montpellier 2, 2006.
8. Olivier Gout and Thomas Lambolais. Construction incrémentale de modèles comportementaux UML. In *Approches Formelles Dans L'Assistance Au Développement Des Logiciels*, pages 29–42, 2004.
9. Matthew Hennessy. *Algebraic theory of processes*. MIT Press, 1988.
10. Bogumila Hnatkowska, Zbigniew Huzar, and Lech Tuzinkiewicz. On understanding of refinement relationship. In *Third International Workshop, Consistency Problems in UML-based Software Development III–Understanding and Usage of Dependency Relationships*, pages 11–22, 2004.
11. Zbigniew Huzar, Ludwik Kuzniarz, Gianna Reggio, and Jean Louis Sourrouille. *Consistency Problems in UML-Based Software Development*, pages 1–12. Springer Berlin / Heidelberg, 2005.
12. Ferhat Khendek and Gregor von Bochmann. Merging behavior specifications. *Formal Methods in System Design*, 6 :259–293, 1993.
13. François Laroussinie and Philippe Schnoebelen. The state explosion problem from trace to bisimulation equivalence. In *Foundations of Software Science and Computation Structures*, pages 192–207, 2000.
14. Guy Leduc. Conformance relation, associated equivalence, and minimum canonical tester in LOTOS. *PSTV XI. North-Holland*, pages 249–264, 1991.
15. Guy Leduc. *On the role of implementation relations in the design of distributed systems using LOTOS*. PhD thesis, Université de Liège, Faculté des sciences appliquées, 1991.
16. Guy Leduc. A framework based on implementation relations for implementing LOTOS specifications. *Computer Networks and ISDN Systems*, 25 :23–41, 1992.
17. Hong-Viet Luong, Thomas Lambolais, and Anne-Lise Courbis. Implementation of extension and reduction relations for incremental development of behavioural models. Technical report, EMA, May 2008.
18. Hong-Viet Luong, Thomas Lambolais, and Anne-Lise Courbis. Implementation of the conformance relation for incremental development of behavioural models. In K. Czarnecki et al, editor, *MoDELS 2008*, volume 5301, page 356–370, October 2008.
19. Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.
20. Robin Milner. *Communicating and Mobile Systems : the Pi-Calculus*. Cambridge University Press, 1st edition, 1999.
21. OMG. *Unified Modeling Language Superstructure 2.0*. 2007.
22. Ragnhild Van Der Straeten. Formalizing behaviour preserving dependencies in UML. *Third International Workshop, Consistency Problems in UML-based Software Development III–Understanding and Usage of Dependency Relationships*, pages 71–82, 2004.
23. Jan Tretmans. Testing concurrent systems : A formal approach. In *CONCUR 99 Concurrency Theory*, page 779, 1999.