Synthèse de diagrammes d'états par classe à partir de diagrammes de séquence

Bontemps, Yves; Saval, Germain; Schobbens, Pierre-Yves; Heymans, Patrick

Published in:
Technique et Science Informatiques

Publication date:
2007

Document Version
Première version, également connu sous le nom de pré-print

Synthèse de diagrammes d'états par classe à partir de diagrammes de séquence

Yves Bontemps — Germain Saval
Pierre-Yves Schobbens — Patrick Heymans

Institut d'Informatique, FUNDP, Namur
rue Grandgagnage, 21
B5000 - Namur (Belgique)
{ybo,gsa,pys,phe}@info.fundp.ac.be

RÉSUMÉ. *Afin de modéliser le comportement d'un système distribué, les analystes utilisent deux types de langages : les diagrammes de séquence et les diagrammes d'états. Les premiers fournissent une vue de haut niveau des interactions entre les objets, alors que les seconds se concentrent sur le comportement local de chaque objet. Des algorithmes existent pour synthétiser des machines à états à partir des diagrammes de séquence, mais se limitent à des objets particuliers, en nombre fixe. De nombreux systèmes nécessitent en revanche un nombre d'objets arbitraire. Nous proposons d'adapter les diagrammes et l'algorithme pour traiter ces systèmes, en ajoutant simplement des quantificateurs. Nous donnons la syntaxe et la sémantique des deux langages étendus, puis nous prouvons la correction (faible) de notre algorithme. Comme pour les algorithmes existants qu'il étend, cette correction est faible car de nouveaux comportements peuvent être introduits.*

ABSTRACT. *To model the behavior of a distributed system, analysts often use two types of languages: Sequence Diagrams and State Diagrams. The former presents a bird's eye view on objects interactions, whereas the latter describes the complete local behavior of every object. Many algorithms translating scenarios to state machines have been devised. All these algorithms work at instance-level, i.e. for a fixed finite number of objects. Real-world object-oriented systems often contain arbitrarily many objects. Modeling languages and synthesis algorithms need to be adapted to this situation. We propose to add universal and existential quantifiers. After defining the syntax and semantics of the two extended languages, we extend also a state of the art algorithm by a novel instantiation step to cope with quantifiers. As the base algorithm, our correction is weak since it allows implied behaviors.*

MOTS-CLÉS : *diagramme de séquence, MSC, diagramme d'état, synthèse de classe.*

KEYWORDS: *sequence diagram, MSC, state diagram, class-level synthesis.*

RSTI - TSI – 26/2007. AFADL 2006, pages 797 à 817

1. Introduction

Un développement typique de système à objets distribués débute par l'écriture de scénarios, qui décrivent les comportements les plus importants. Ils sont progressivement enrichis, précisés et composés jusqu'à décrire tous les comportements du système. Une fois tous les scénarios composés, on obtient une description globale complète des interactions entre objets. Toutefois, cette spécification n'est pas directement implémentable. Le but final est d'obtenir une description séparée et exécutable du comportement de chaque objet, mais qui présente globalement tous les comportements décrits dans la spécification.

Dans l'esprit de l'architecture dirigée par modèles – *Model-Driven Architecture* (MDA) – lancée par l'*Object Management Group* (OMG), de nombreuses approches (Koskimies *et al.*, 1996 ; Leue *et al.*, 1998 ; Whittle *et al.*, 2000 ; Krüger *et al.*, 1999 ; Mansurov *et al.*, 1999 ; Uchitel *et al.*, 2001a ; Uchitel *et al.*, 2003 ; Uchitel *et al.*, 2004b) ont été proposées pour automatiser la traduction des scénarios vers des modèles exécutables. Le cadre de travail MDA décrit ceci comme une transformation d'un modèle indépendant de la plate-forme – *Platform Independent Model* (PIM) – vers un autre. Les transformations ultérieures vers un modèle spécifique à une plate-forme – *Platform Specific Model* (PSM) – puis vers du code exécutable n'entrent pas dans le cadre de cet article et sont assurées par les outils existants. Pour être utiles, les modèles utilisés initialement doivent rester proches de l'intuition des participants à la construction du système et permettre une élaboration incrémentale. Les scénarios (Jarke *et al.*, 1998), qui décrivent des cas d'utilisation du système, ont en pratique ces qualités (Weidenhaupt *et al.*, 1998) : ils permettent de plus un lien facile avec les tests, facilitent l'évolution, la communication entre participants, le développement agile. Les scénarios sont en général décrits avec des diagrammes de séquence UML (OMG, 2003) ou des diagrammes de séquence de messages – *Message Sequence Charts* (MSC) (ITU, 2004). Mais ces diagrammes fournissent « une histoire de tous les objets » alors que l'implantation d'un système distribué a besoin de « toutes les histoires possibles d'un objet » (Harel, 2001), c'est-à-dire une description exécutable pour chaque classe, par exemple sous forme de machines à états UML (OMG, 2003).

De nombreux algorithmes de synthèse de machines à états à partir de scénarios ont été proposés durant la dernière décennie. Nous nous limitons aux principaux d'entre eux, car des comparaisons plus complètes sont disponibles dans (Hélouët *et al.*, 2001 ; Amyot *et al.*, 2003 ; Liang *et al.*, 2006), et nous les présentons par ordre croissant de complexité des dialectes traités. Les premières approches (Whittle *et al.*, 2000 ; Leue *et al.*, 1998) ne traitaient que les diagrammes de base. Yamanaka *et al.* (1996) ont initié l'approche par projection que nous suivons ici. Koskimies *et al.* (1996) utilisent un algorithme d'apprentissage proche de Biermann-Krishnaswamy (Biermann *et al.*, 1976) qui génère une machine à états minimale à partir de traces d'exécution. Whittle *et al.* (2000) ajoutent des conditions aux messages individuels. Les scénarios peuvent être considérés comme des exemples d'exécution qu'une machine à états synthétisée devrait reproduire. Différentes compositions de diagrammes de base ont ensuite été traitées. Krüger *et al.* (1999) proposent de spécifier les sé-

quences possibles de MSC par une pré- et une post-condition, utilisées pour calculer les collages possibles. Leur traduction préserve la structure dans le Statechart résultant. La façon la plus classique de composer les diagrammes de séquence sont les diagrammes de haut niveau (HMSC) (ITU, 2000), qui ont une notation graphique mais que nous notons textuellement dans la suite. En synthèse, ils sont utilisés par (Mansurov *et al.*, 1999 ; Leue *et al.*, 1998 ; Uchitel *et al.*, 2001a ; Uchitel *et al.*, 2003). Ziadi *et al.* (2004) définissent une structure algébrique similaire pour les scénarios, qu'ils transposent sur les machines à états. Nous suivons et étendons cette approche. Les Use Case Maps (ITU, 2003 ; Buhr, 1998) sont utilisés par Martínez (2005), Bordeleau *et al.* (2000). Les diagrammes de séquence actifs – *Live Sequence Charts* (LSC) (Harel *et al.*, 2002) – constituent une autre approche de la combinaison des scénarios. Les diagrammes s'y tiennent prêt à s'exécuter dès que leur condition est satisfaite, y compris en parallélisme avec eux-mêmes. Nous montrons dans (Bontemps *et al.*, 2004) que la synthèse est dans ce cas doublement exponentielle, pour une sémantique où le système doit éviter les conflits entre MSCs. En pratique, des hypothèses simplificatrices permettent souvent une synthèse plus efficace (Harel *et al.*, 2005). Notre approche se limite à la composition séquentielle forte et la communication synchrone. Khendek *et al.* (1998), Mansurov *et al.* (1999), Abdalla *et al.* (1999) et Engels (2001) étudient le cas asynchrone pour générer des automates SDL.

Toutes ces approches ne traitent que des instances spécifiques. La plupart des applications nécessitent des classes dont le nombre d'instances n'est pas connu à l'avance. Marelly *et al.* (2002) a le premier proposé une extension des LSC de ce type. Dans (Bontemps *et al.*, 2003), nous l'avons appliquée au problème classique du système de contrôle aérien CTAS de la NASA (NASA AMES, 2006). Cet exemple a également été traité avec un nombre fixe d'instances par Whittle *et al.* (2002), qui cite comme un important problème ouvert le traitement général que nous proposons ici. Uchitel *et al.* (2004a) contournent ce problème en utilisant une architecture de composants. Chaque composant peut être intégré en plusieurs copies dans des architectures différentes.

Dans la section 2, nous présentons d'abord la sémantique de traces (2.1) partagée par nos extensions des diagrammes d'états (2.2) et de séquence (2.3). Après ces définitions, nous pouvons énoncer le problème de la synthèse (3.1) qui est insoluble, et nous nous restreignons donc au calcul de la meilleure approximation (Uchitel *et al.*, 2001a). Notre algorithme (3.2) résout ce problème en temps linéaire. Nous illustrons son application sur des extraits du problème CTAS (Whittle *et al.*, 2002 ; Bontemps *et al.*, 2003) dans la section 4.

2. Modèles

Nous présentons tout d'abord le domaine sémantique orienté-objet que nos deux formalismes partagent. Ceci nous permet ensuite de présenter la syntaxe et la sémantique des deux langages que nous proposons. Leur proximité sémantique facilitera leur traduction.

2.1. *Systèmes d'objets distribués*

Ici, nous voulons traiter d'un seul coup toute population d'objets, alors que les algorithmes connus traitent une population donnée. Nous modélisons ceci par la donnée d'un diagramme de classes : un graphe fini $\mathbb{C}$ muni d'un ordre partiel $A \sqsubseteq B$ lu « A hérite de B », où chaque classe est un ensemble fini d'attributs et de méthodes (messages synchrones) croissant le long de l'héritage. Il peut également contenir des associations entre classes.

Une population $\mathbb{P}$ est donnée par un ensemble fini d'objets $\mathbb{O}$ munis d'une valuation initiale de leurs attributs et d'une classe principale.[1]

$\mathbb{C}$ est un ensemble de noms de classes
$\mathbb{O}$ est un ensemble de noms d'objet
$\mathbb{A}$ttr est un ensemble de noms d'attribut
Γ est un ensemble de noms de messages
$\mathbb{V} = \{\top, \bot\} \cup \mathbb{O} \cup 2^{\mathbb{O}}$, $\mathbb{V}\text{al} = \mathbb{A}\text{ttr} \hookrightarrow \mathbb{V}$
$\mathbb{P} \subset \mathbb{O} \hookrightarrow \mathbb{C} \times \mathbb{V}\text{al}$

$$
\begin{array}{rcl}
o(\in \mathit{ObjExp}) & ::= & \texttt{nil} \mid x(\in \mathbb{A}\text{ttr}) \\
b(\in \mathit{BoolExp}) & ::= & \texttt{true} \mid \texttt{false} \mid \neg b_1 \mid b_1 \wedge b_2 \mid e_1 = e_2 \\
 & & \mid x(\in \mathbb{A}\text{ttr}) \\
e(\in \mathit{Exp}) & ::= & b \mid o \\
a(\in \mathit{Assign}) & ::= & x(\in \mathbb{A}\text{ttr})\ \texttt{:=}\ e
\end{array}
$$

$\mathcal{A} = \mathbb{O}.\mathit{Assign}$, $\mathcal{M} = \mathbb{O} \times \Gamma \times \mathbb{O} \times \mathbb{V}$
$\Sigma = \mathcal{A} \cup \mathcal{M}$

Figure 1. *Système d'objets (domaines)*

Les objets communiquent par rendez-vous, c'est-à-dire par envoi synchrone de messages $\mathcal{M}$. Un message est de la forme (s, m, r, v) : l'objet s envoie le message $m \in \Gamma$ avec la valeur de paramètre v à l'objet r. Nous définissons $\Sigma_r(o) = \{(s, m, o, v) \in \mathcal{M}\}$ les messages recevables par o, et $\Sigma(o) = \{(o, m, r, v) \in \mathcal{M}\}$ ceux que o peut envoyer.

Les diagrammes de séquence permettent habituellement aussi de noter des communications asynchrones. Nous ne les traitons pas dans cet article pour ne pas alourdir davantage la notation.

Une exécution est une suite (finie ou infinie) de messages. Notre sémantique est donc entrelacée : deux messages ne sont jamais transmis exactement au même moment. Le *comportement* potentiel de tous les objets o de même classe principale est représenté par une *stratégie* (non déterministe à information partielle) $f_o : \Sigma_r(o)^* \to 2^{\Sigma(o)}$. La stratégie détermine les actions possibles de o à partir du passé tel que vu par

1. Nous divergeons ici du modèle UML, où un objet n'a pas de classe principale, pour suivre celui des langages de programmation à objets.

l'objet o. On étend la notation sur les histoires du système Σ^* en retirant de cette histoire les messages qui ne sont pas à destination de o. En revanche, lors d'une exécution γ, le comportement réel $\gamma|_{\Sigma(o')}$ d'un autre objet o' de même classe principale peut être différent. Les *résultats* possibles $Out(f_o)$ d'une stratégie f_o sont les exécutions γ où o exécute sa stratégie : $\forall e \in \Sigma(o) : \forall u, \gamma' : ue\gamma' = \gamma \Rightarrow e \in f_o(u)$. Une stratégie pour un ensemble d'objets O est simplement une famille de stratégies individuelles $(f_o)_{o \in O}$.

Les systèmes ne sont pas constitués d'un seul objet mais de plusieurs objets coopérant. Le comportement global du système émerge de l'interaction entre ces objets. Nous avons besoin d'un moyen de les composer pour former le système global. D'abord nous permettons aux stratégies d'être définies pour des ensembles d'objets (soit $A \subset \mathbb{O}$, $f_A : \Sigma^* \to 2^{\Sigma(A)}$). Ensuite nous fournissons un opérateur pour composer les stratégies.

Définition 1 (composition des stratégies). *Soient A and B deux ensembles d'objets (non nécessairement disjoints), avec des stratégies $f_A : \Sigma^* \to \Sigma(A)$ et $f_B : \Sigma^* \to \Sigma(B)$. La composition de f_A et f_B est la stratégie $f_A + f_B : \Sigma^* \to 2^{\Sigma(A) \cup \Sigma(B)}$, définie comme*

$$\begin{array}{rcl} (f_A + f_B)(w) & = & f_A(w) \cap (\Sigma(A) \setminus \Sigma(B)) \\ & \cup & f_B(w) \cap (\Sigma(B) \setminus \Sigma(A)) \\ & \cup & (f_A(w) \cap f_B(w)) \cap (\Sigma(A) \cap \Sigma(B)). \end{array}$$

Les deux stratégies doivent s'accorder sur le moment où les événements visibles de A et B doivent être déclenchés. Cela correspond au modèle de communication par rendez-vous que l'on désire.

Définition 2 (système d'objets distribués (DOS)). *Un* système d'objets distribués (DOS) *σ pour une population p est une fonction associant tout objet o dans $dom(p)$ à une stratégie $\sigma(o) : \Sigma^* \to 2^{\Sigma(o)}$ telle que :*

– $\sigma(o)$ est locale à o, et

– $\forall o' \in dom(p) : p(o) = p(o') \Rightarrow \sigma(o)$ est isomorphe à $\sigma(o')$, c'est-à-dire le comportement de deux objets de la même classe doit être identique modulo un renommage. Il peut ne pas dépendre de l'identité réelle de l'objet.

Nous dénotons par $DOS(p)$ l'ensemble de tous les systèmes d'objets distribués pour une population p. Les résultats d'un tel système d'objets distribués σ sont simplement les résultats de la composition de tous les objets de p : $Out(\sigma) = Out(\sum_{o \in dom(p)} \sigma(o))$.

2.2. *Machines à états*

Le diagramme de classes (OMG, 2003) de la figure 2 définit les différentes classes d'un système d'alarme, et donne les attributs de chaque classe. Les associations sont

dirigées ; leurs rôles définissent également des attributs, avec les cardinalités indiquant leur type. Remarquez que comme annoncé il y a une infinité de populations possibles, avec plusieurs `Sensor`, `Alarm`, `Guard` et `Input`. Le comportement de *tous* les objets de `Alarm` est décrit par la machine à états de la figure 3.

Nos machines à états sont inspirées des Statecharts de Harel (Harel, 1986) : elles ajoutent la concurrence, la hiérarchie *et* la quantification aux machines à états habituelles. Nous les appelons *Quantified Hierarchical Concurrent State Machines* ou QHCSM. Nous évitons d'appeler ces machines des Statecharts car bien qu'elles en soient fortement inspirées, elles n'en possèdent pas toutes les caractéristiques, y ajoutent d'autres constructions et n'en suivent pas la sémantique.

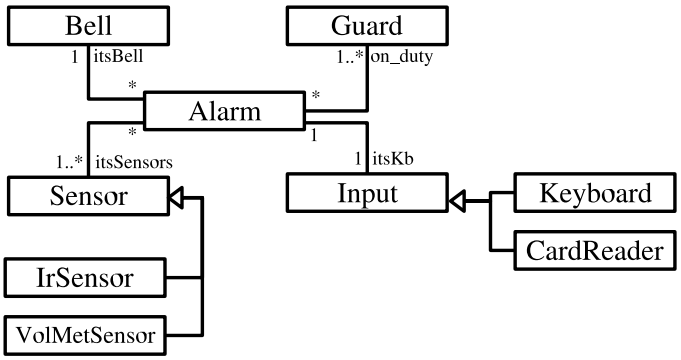


Figure 2. *Diagramme de classes du système d'alarme*

Un modèle de machines à états est donc une relation finie associant une machine à états à toutes les classes ($\mathbb{C} \rightarrow$ QHCSM). Pour une population donnée, cette machine à états sera *instanciée*, c'est-à-dire copiée autant de fois qu'il y a d'instances de la classe en question et ses étiquettes de transition seront renommées avec le nom de son possesseur. La sémantique d'un modèle de machines à états m sera donnée par $[\![\cdot]\!] : (\mathbb{C} \rightarrow \text{QHCSM}) \rightarrow \mathbb{P} \rightarrow DOS$.

Un QHCSM est constitué d'états, représentés par une boîte arrondie. Les états sont liés par des flèches appelées *transitions*, qui sont étiquetées par des éléments de la forme « événement-garde-action » ($e[g]/a$), qui sont tous optionnels. Un événement ne peut se référer qu'à des attributs de cet objet ou à « self » qui dénote l'objet lui-même.

Les transitions qui franchissent les barrières des états sont interdites dans les QHCSM, ce qui permet de préserver la compositionnalité de la sémantique (von der Beeck, 2002). Ces transitions sont simplement décomposées en deux transitions. La première va vers une porte d'exception d'entrée, représentées par un cercle barré d'une croix et positionnée sur la frontière de l'état (voir figure 3). La seconde part de la porte correspondante, dite de sortie, dans l'appelant.

Définition 3 (QHCSM). *Un QHCSM est défini comme un n-uplet*

$M = \langle Q, I, F, Exc, A, \delta, \kappa, \eta, h \rangle$, *où*

Q est un ensemble fini d'états, $I \subseteq Q$ sont les états initiaux et $F \subseteq Q$ les états finaux,

$Exc = Exc_i \cup Exc_o$ est un ensemble fini où Exc_i et Exc_o sont respectivement des portes d'entrée et de sortie de machine,

$A \subseteq \mathbb{A}ttr$ est un ensemble d'attributs,

$\delta \subseteq Q \times \mathcal{M} \times Exp(A) \times \mathcal{A} \times Q \cup Exc_o$ est une relation de transition,

$\kappa \subseteq Q \times Q \cup Exc_o$ est une relation de transition de « complétion »,

$\eta : Exc_i \rightarrow Q \cup Exc_o$ associe les portes d'entrée aux portes de sortie de machine,

$h : Q \rightarrow 2^{QHCSM \cup (\langle \forall x{:}C{:}\phi \rangle QHCSM) \cup (\langle \exists x{:}C{:}\phi \rangle QHCSM)}$ est une fonction représentant la hiérarchie des états, associant un état à l'ensemble des machines à états qu'il contient et qui s'exécutent concurremment à l'intérieur. Ces machines doivent utiliser Exc_i comme ensemble de portes de sortie.

Pour définir la sémantique opérationnelle d'un QHCSM, nous avons besoin du concept de configuration. Une configuration est de la forme (q, c), où q est l'état courant d'un QHCSM et c associe chaque état à la configuration de ses sous-états, ce qui donne :

$$config(M) = (q, config(h(Q))),$$

qui est étendu à un ensemble de machines :

$$config(\{m_1, m_2, \ldots, m_n\}) = \{config(m_1), config(m_2), \ldots, config(m_n)\}.$$

Des quantificateurs universels et existentiels, portant sur les instances d'une classe, peuvent être utilisés dans des machines à états. Une quantification, portant sur une classe C, est une expression de la forme suivante, « for all $x : C : \phi$ » ou bien « for some $x : C : \phi$ ». Intuitivement, une machine à états universellement quantifiée signifie qu'il y a autant d'exécutions concurrentes de copies de cette machine qu'il y a d'instances de la classe C satisfaisant la condition « ϕ ». La quantification existentielle se résume au choix arbitraire d'une instance de C satisfaisant ϕ. Étant donné une population, il est par conséquent possible de transformer tout QHCSM en une grande machine à états sans quantification, que nous utilisons pour la sémantique, mais évitons en pratique vu son coût. Les extensions que nous venons de présenter étaient déjà suggérées dans (Harel, 1986). On pourrait aussi penser à introduire la quantification dans les transitions, comme c'est le cas dans certaines algèbres de processus. L'avantage de notre notation est de rendre la portée de la quantification explicite.

Par exemple, l'état *s1* de la figure 3 décrit une situation où l'alarme envoie le message « disable » à tous ses senseurs, de façon à les désactiver. Il faut remarquer que ceci n'impose aucun ordre de parcours particulier sur l'ensemble `sensors`, évitant ainsi de recourir à des considérations propres à l'implantation, lors des phases d'analyse et de modélisation. Ce mécanisme permet donc de créer des spécifications élégamment abstraites. Lorsque tous les senseurs auront été désactivés, l'alarme pourra entrer dans l'état `off`. La transition vers `off` est une transition de complétion, qui est déclenchée dès que toutes les sous-machines ont terminé leur exécution.

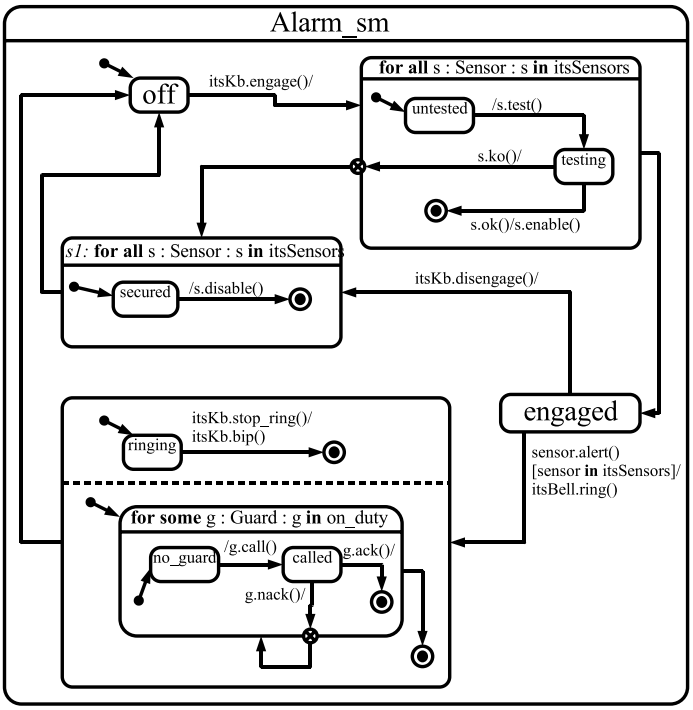


Figure 3. *Machine à états pour la classe* `Alarm`

Nous décrivons les transitions autorisées dans les machines instanciées, autrement dit nous définissons une Sémantique Opérationnelle Structurée à la Plotkin. La première règle est appelée « Top-Level Transition ». Lorsque l'état courant (au niveau le plus élevé) est q, et qu'il existe une transition étiquetée par $e[g]/a$ allant de l'état q à l'état q', la machine peut quitter l'état q et entrer dans l'état q'.

$$\text{TLT}\ \frac{(q, e, g, a, q') \in \delta \quad [\![g]\!]_v = \texttt{true}}{v, (q, c) \xrightarrow{ea} v', (q', c')}$$

où :

– $c' = c\,[q' \leftrightarrow init(q')]$ et $init(q')$ est la configuration initiale de q',

– $v' = [\![a]\!]_v$ est l'évaluation des attributs de A.

Si aucune transition n'est disponible au niveau le plus élevé, des transitions peuvent être tirées dans les sous-machines. Nous modélisons la concurrence par l'entrelacement ; une sous-machine dans laquelle une transition est possible est choisie de manière non déterministe et sa transition est exécutée.

$$\text{LLT}\ \frac{\exists m \in \mathit{dom}(c)\ v, \mathit{config}(m) \xrightarrow{a} v', (q', c')}{v, (q, c) \xrightarrow{a} v', (q, c\,[m \leftrightarrow (q', c')])} \quad \begin{array}{c} \nexists (q, e, g, a, q'') \in \delta \\ [\![g]\!]_v = \texttt{true} \end{array}$$

Les portes d'exception sont représentées par des cercles barrés d'une croix et placées sur les frontières des états. Lorsqu'une porte d'exception est atteinte, la sous-machine est complètement interrompue et la transition est prolongée vers le niveau supérieur. Par exemple, dans l'état supérieur droit de la figure 3, dès qu'un senseur signale qu'il ne fonctionne pas correctement, l'ensemble du protocole de test est avorté et tous les senseurs sont désactivés. Les transitions d'exception ont une priorité supérieure aux transitions situées à l'intérieur des machines.

$$\textsc{Exception}\ \frac{\begin{array}{c}\exists m \in \mathit{dom}(c)\ v, \mathit{config}(m) \xrightarrow{a} v', (q', c') \\ q' \in \mathit{Exc}_i\end{array}}{v, (q, c) \xrightarrow{a} v', (\eta(q'), c')} \nexists (q, e, g, a, q'') \in \delta$$

Les doubles cercles représentent des états terminaux. Lorsque l'exécution d'une sous-machine atteint un tel état, cette exécution est considérée comme terminée. Certaines transitions spéciales, appelées *transitions de complétion*, partant d'un état q, peuvent être exécutées lorsque toutes les sous-machines de q sont terminées. Syntaxiquement, ces transitions sont représentées comme des flèches sans étiquettes, partant de la frontière d'un état, mais pas de portes d'exception.

$$\textsc{Completion}\ \frac{\forall m \in \mathit{dom}(c)\,(\mathit{config}(m) = (q'', c'') \wedge q'' \in m.F)}{v, (q, c) \xrightarrow{a} v', (q', c')}\ (q, q') \in \kappa$$

Pour chaque objet, on construit une stratégie, en appliquant la règle suivante : après tout préfixe d'exécution w, l'objet propose d'exécuter les actions $\{a_1, \ldots, a_n\}$ ssi $\forall i : 1 \leq i \leq n : w$ mène à un état q_i dans lequel une transition étiquetée par a_i est activée. Formellement,

$$a \in f(w) \text{ si et seulement si } \exists c\,.\,\mathit{init}(m) \xrightarrow{w}{}^{*}\ c \wedge \exists c'\,.\,c \xrightarrow{a} c'$$

Il s'agit d'une sémantique de trace, qui néglige la distinction entre choix interne et choix externe.

2.3. *Diagrammes de séquence*

Nos diagrammes de séquence sont proches des MSC (ITU, 2004) et des diagrammes d'interaction d'UML 2.0 (OMG, 2003). Les scénarios de base sont décrits par des diagrammes de séquence, qui peuvent être ensuite combinés à l'aide d'opérateurs de contrôle de flux, de façon à former des comportements plus complexes.

La sémantique d'un diagramme de séquence de base est standard. Un ordre partiel entre les différents événements peut être dérivé de ce diagramme, en appliquant la règle suivante : si un événement e est dessiné plus haut qu'un événement e', sur la même ligne d'instance, e doit nécessairement précéder e'. Ceci revient à dire qu'un objet n'est autorisé à exécuter une action que si toutes les actions au-dessus de celle-ci sur sa ligne d'instance ont été exécutées. Puisque nos messages sont synchrones, l'envoi et la réception constituent un seul événement.

Les scénarios peuvent être combinés en utilisant les opérateurs de contrôle de flux suivants : l'alternative, la mise en séquence et la composition parallèle. La combinaison de l'alternative et des conditions logiques permet d'exprimer des choix de type « if-then-else ».

Nous autorisons également la quantification existentielle et universelle des instances d'objet. Graphiquement, la quantification universelle se représente par un rectangle à ligne continue, tandis que la quantification existentielle est décrite par un rectangle à ligne discontinue. Par exemple, dans la figure 6a, la ligne d'instance dénommée `c : Client` représente un objet de la classe « Client ». Cette figure illustre également l'utilisation de « conditions de liaison » : celles-ci sont dessinées comme des flèches liant une instance (x) à une instance quantifiée (y). Cette flèche est décorée d'une expression booléenne, locale à x. Elle signifie que seuls des objets satisfaisant cette condition sont autorisés à être liés à y.

Définition 4 (QHMSC). *La syntaxe abstraite de QHMSC est définie par la grammaire suivante, dans laquelle B représente un diagramme de séquence de base :*

$$\begin{array}{rcl} M & ::= & B \\ & | & M_1 \parallel M_2 \mid M_1 + M_2 \mid M_1 \,;\, M_2 \mid M_1^* \\ & | & \phi \Rightarrow M \mid \phi \wedge M \\ & | & \forall x : C : \phi\,(M_1) \mid \exists x : C : \phi\,(M_1)\,. \end{array}$$

Toutes les instances apparaissant dans un QHMSC doivent se trouver sous la portée d'un quantificateur. Graphiquement, les variables libres, telles que `server`, sont par conséquent implicitement universellement quantifiées. On suppose qu'elles appartiennent à leur propre classe `Server`.

La signification d'un QHMSC ne peut se définir que par rapport à une population d'objets donnée. La signature de la fonction sémantique est donc $[\![\cdot]\!]$: QHMSC $\rightarrow$ $\mathbb{P} \rightarrow 2^{\Sigma^*(p)}$. Lorsqu'une population est donnée, tous les quantificateurs sont remplacés par de véritables valeurs d'objet, ce qui revient à lier l'ensemble des variables et, ainsi, à ce que toutes les lignes d'instances se réfèrent à de véritables objets. Nous donnons une sémantique classique des opérateurs de contrôle de flux, similaire à celle des expressions rationnelles : les diagrammes de base décrivent des langages finis, le choix correspond à l'union de langages, la composition parallèle à l'entrelacement (*shuffling*), la boucle à l'étoile de Kleene, la séquence à la concaténation de langages ; nous nous écartons ici de la sémantique faible de l'ITU (2000) pour une sémantique forte plus classique. La construction $\phi \Rightarrow M$ signifie « exécuter M si ϕ est vraie, ne

rien faire sinon ». Enfin $\phi \wedge M$ signifie « ϕ doit être vraie et M est exécuté ». Si ϕ n'est pas vraie, le langage décrit est vide. La sémantique d'un HMSC clos (où toutes les variables sont liées) est donnée par $[\![\cdot]\!]$: HMSC $\rightarrow (\mathbb{O} \rightarrow \mathbb{V}\text{al}) \rightarrow 2^{\Sigma^*}$. Cette fonction dépend évidemment des valeurs des attributs, puisqu'il est nécessaire d'évaluer les conditions de garde (ϕ).

Seule l'opération de suppression des quantificateurs est détaillée ci-dessous, car il s'agit de la partie la plus originale de notre langage ; la formalisation des autres opérations est sans surprise, sinon fastidieuse. Considérons une population p fixée et posons $\{o_1, \ldots, o_n\} = \{o | cl(o,p) \sqsubseteq C\}$. Nous définissons alors :

$$i(\forall x : C : \phi\,(M)\,, p) = (\phi \Rightarrow i(M,p))[x/o_1] \| \ldots \| (\phi \Rightarrow i(M,p))[x/o_n]$$

$$i(\exists x : C : \phi\,(M)\,, p) = (\phi \wedge i(M,p))[x/o_1] + \ldots + (\phi \wedge i(M,p))[x/o_n]$$

Comme expliqué ci-dessus, *i* supprime tous les quantificateurs et renvoie un HMSC clos, dont toutes les lignes d'instances sont étiquetées par des noms de véritables objets. Nous pouvons alors simplement écrire la fonction sémantique d'un QHMSC comme $[\![\cdot]\!]$: QHMSC $\rightarrow \mathbb{P} \rightarrow \Sigma^*$, avec :

$$[\![M]\!]p = [\![i(M,p)]\!]val(p).$$

3. Synthèse

Nous considérons l'élaboration des modèles de comportement d'un objet comme une activité incrémentale. D'abord, l'analyste décrit les interactions souhaitées entre les objets avec des scénarios. Tous ces scénarios sont combinés dans un unique QHMSC pour former une description inter-objet complète du système. Ensuite, l'analyste conçoit une machine à états qui doit exhiber le comportement décrit par ce QHMSC pour chaque classe. Dans cette section, nous proposons un algorithme pour réaliser cette activité. Nous définissons d'abord le problème de la synthèse d'une machine à états à partir d'un QHMSC puis nous présentons l'algorithme résolvant ce problème.

3.1. *Problème*

Notre algorithme prend un QHMSC et retourne pour chaque classe sa machine à états, *synth*() : QHMSC $\rightarrow \mathbb{C} \rightarrow$ QHCSM. Notre but idéal serait d'obtenir, pour toute population, les mêmes comportements pour le QHMSC et ses machines à états.

$$\forall p \in \mathbb{P} : [\![M]\!]p = Out\,([\![\mathit{synth}(M)]\!]p)$$

Par définition, le système d'objets instanciés à partir d'une machine à états est distribué, c'est-à-dire que le comportement de chaque objet est uniquement déterminé par

sa vue locale de l'exécution. Les stratégies sont locales. Cependant, il y a une différence inévitable entre la vue globale (inter-objet) et la vue locale (intra-objet). On peut concevoir une spécification inter-objet qui nécessite une information d'un autre objet, et donc ne peut être réalisée par une spécification intra-objet (Uchitel, 2003 ; Uchitel *et al.*, 2001b ; Alur *et al.*, 2000). Nous suivons l'état de l'art des algorithmes de synthèse, et résolvons un problème plus réaliste : construire la meilleure sur-approximation distribuée d'une spécification inter-objet.

Définition 5 (sur-approximation optimale). *Étant donné une population p, un système d'objets distribué $\sigma \in DOS(p)$ est la* sur-approximation optimale *d'un langage $L \subseteq \Sigma^{\infty}(p)$ ssi*

1) $L \subseteq Out(\sigma)$ (σ produit au moins toutes les traces de L),

2) $\forall \sigma' \in DOS(p) : L \subseteq Out(\sigma') \Rightarrow Out(\sigma) \subseteq Out(\sigma')$ (σ est minimal).

Notre problème de synthèse est alors simplement :

Définition 6 (problème de la synthèse). *Le* problème de la synthèse *est, un QHMSC M et un ensemble fini de classes $\mathbb{C}$ étant donnés, de retourner un modèle sous forme de machine à états synth$(M) : \mathbb{C} \rightarrow$ QHCSM tel que $\forall p \in \mathbb{P} :$ [[synth(M)]]p est la sur-approximation optimale de [[M]]p.*

Remarquons d'abord que la seule façon d'obtenir une sur-approximation optimale du QHMSC est de laisser chaque objet effectuer une action e, après une exécution partielle w, quand w ressemble *localement* à une autre exécution w' de laquelle e pourrait être une continuation possible. La meilleure sur-approximation distribuée d'un langage L est donc obtenue quand chaque objet ne fait rien de plus que ce lemme. Un objet décide d'effectuer l'action e après le préfixe w si *et seulement* s'il y a un mot w' qui est équivalent à w du point de vue de cet objet ($w|_{\Sigma(o)} = w'|_{\Sigma(o)}$), et $w'e$ est le préfixe d'un mot de L.

Lemme 1 (critère d'optimalité pour la sur-approximation). *Soient $\sigma \in DOS(p)$ et $L \subseteq \Sigma^{\infty}(p)$, σ est la meilleure sur-approximation L ssi $\forall o \in$ dom(p), $\forall e \in \Sigma(o)$, $\forall w \in \Sigma^*$, $e \in \sigma(o)(w) \iff \big(\exists w', w'' \in \Sigma^* : w'|_{\Sigma_r(o)} = w|_{\Sigma_r(o)} \wedge w'ew'' \in L\big)$.*

3.2. *Algorithme*

Nous suivons les techniques habituelles pour synthétiser des machines à états à partir de HMSC (Krüger *et al.*, 1999), mais nous ajoutons une nouvelle étape pour traiter la quantification. Notre but est de synthétiser une machine à états optimale (au sens du lemme 1) pour une classe donnée C à partir d'un QHMSC. Pour ce faire, notre algorithme effectue successivement les étapes suivantes :

1) **Instanciation** du QHMSC. On insère une instance, appelée « self », partout où une instance de classe effective C apparaît dans le QHMSC. La propriété essentielle préservée par cet algorithme, présenté dans le tableau 1, est donnée dans le lemme 2.

$$
\begin{aligned}
rcl\mathit{inst}(C, B) &= B \quad \text{où } B \in \mathsf{MSC} \\
\mathit{inst}(C, M_1 \parallel M_2) &= \mathit{inst}(C, M_1) \parallel \mathit{inst}(C, M_2) \\
\mathit{inst}(C, M_1 \,;\, M_2) &= \mathit{inst}(C, M_1) \,;\, \mathit{inst}(C, M_2) \\
\mathit{inst}(C, M_1 + M_2) &= \mathit{inst}(C, M_1) + \mathit{inst}(C, M_2) \\
\mathit{inst}(C, M^*) &= \mathit{inst}(C, M)^*
\end{aligned}
$$

$$
\mathit{inst}(C, \forall x : C' : \phi\,(M)) = \begin{cases} \text{if } C \sqsubseteq C' \qquad (\phi \Rightarrow \mathit{inst}(C, M))[x/\mathit{self}] \\ \qquad \parallel \forall x : C' : \phi \wedge x \neq \mathit{self}\,(\mathit{inst}(C, M)) \\ \text{else } \forall x : C' : \phi\,(\mathit{inst}(C, M)) \end{cases}
$$

$$
\mathit{inst}(C, \exists x : C' : \phi\,(M)) = \begin{cases} \text{if } C \sqsubseteq C' \qquad (\phi \wedge \mathit{inst}(C, M))[x/\mathit{self}] \\ \qquad + \exists x : C' : \phi \wedge x \neq \mathit{self}\,(\mathit{inst}(C, M)) \\ \text{else } \exists x : C' : \phi\,(\mathit{inst}(C, M)) \end{cases}
$$

Tableau 1. *Algorithme d'instanciation*

2) **Projection** du QHMSC sur l'instance « self ». Cette étape élimine toutes les lignes de vie qui n'appartiennent pas à « self » dans le QHMSC. Cette opération abstrait donc le QHMSC, conservant uniquement la vue locale de l'instance « self ».

3) **Traduction** du projeté du QHMSC vers une machine à états, associant chaque structure de contrôle à la construction correspondante dans les machines à états. Cette étape est simple s'il y a suffisamment de constructions dans le langage des machines à états (Ziadi *et al.*, 2004). Comme nous avons ajouté la notation de quantification dans les *deux* langages, nous évitons les difficultés rencontrées dans les précédents algorithmes de synthèse (Whittle *et al.*, 2002).

Notre algorithme d'instanciation n'altère pas la sémantique du QHMSC, pour des classes non vides. Il insère une nouvelle instance « *self* », qui n'est *jamais quantifiée*.

Lemme 2. *Pour toute population p, telle que* $p(\mathit{self}) = C$,

$$
[\![\mathit{inst}(C, M)|_{\mathit{self}}]\!]p = \big([\![M]\!]p\big)|_{\Sigma(\mathit{self})}
$$

Ainsi *self* peut être n'importe quelle instance de C dans une population quelconque sans changer la sémantique du diagramme.

La construction d'une machine à états à partir d'un QHMSC est inspirée de la construction d'automates à partir d'expressions rationnelles (Hopcroft *et al.*, 2001). La fonction *sm* qui synthétise cette machine à états est définie inductivement sur la structure du QHMSC et présentée figure 4. Elle procède par cas :

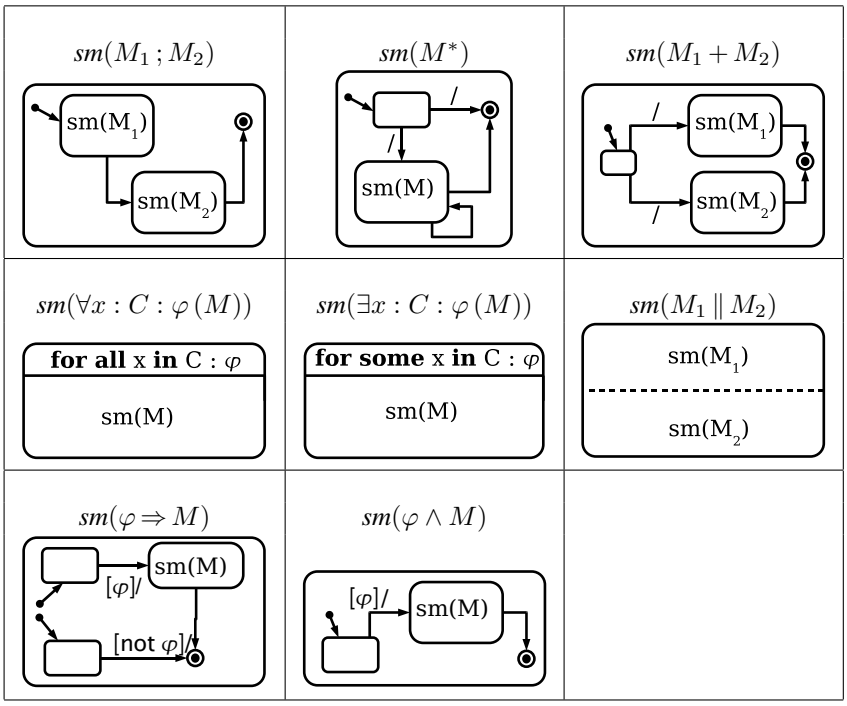


Figure 4. *Définition de la fonction sm*

– M est un diagramme de base : puisque M est fini, il a un nombre fini d'exécutions. Celles-ci peuvent être aisément reconnues par une machine à états, qui les énumère toutes. Les états de cet automate sont les coupures (ou idéaux) du diagramme ;

– $M = M_1 ; M_2$: $sm(M_2)$ est exécutée dès que $sm(M_1)$ termine, grâce à une transition de complétion ;

– $sm(M^*)$: soit $sm(M)$ n'est pas exécutée (sans itération), soit elle est exécutée et quand elle termine la machine choisit de manière non déterministe de relancer $sm(M)$ ou de terminer ;

– $M = M_1 \parallel M_2$: dans ce cas, $sm(M_1)$ et $sm(M_2)$ produisent deux machines à états S_1 et S_2. Une nouvelle machine possédant uniquement une région orthogonale est construite. Cette région contient S_1 et S_2. Grâce aux préconditions énoncées plus haut, on a $[\![sm(M_1 \parallel M_2)]\!]\, p = [\![M_1 \parallel M_2]\!]\, p$;

– $M = M_1 + M_2$: la nouvelle machine à états choisit de manière non déterministe entre $sm(M_1)$ et $sm(M_2)$. Cela correspond à des ϵ-transitions dans la théorie classique des automates. Notre sémantique étant linéaire, on obtient alors l'égalité $[\![sm(M_1 + M_2)]\!]\, p = [\![M_1 + M_2]\!]\, p$;

– $M = \forall x : C : \varphi(M)$: simplement quantifier universellement *sm*(M), avec l'expression for all $x : C : \varphi$. Encore une fois, la sémantique est préservée grâce aux préconditions ;

– $M = \exists x : C\,(M)$: quantifier de manière existentielle *sm*(M) avec l'expression for some $x : C : \psi$. La sémantique est également préservée ;

– $\varphi \Rightarrow M$: signifie « si la condition φ est vraie, se comporter comme M, sinon ne rien faire ». Sa traduction repose sur des invariants, voir la section 2.2 ;

– $\varphi \wedge M$: signifie que φ est vraie avant de lancer M. Des invariants sont aussi utilisés.

Par construction, la machine à états résultant est la meilleure implantation possible du QHMSC.

4. Exemple

Nous avons appliqué notre algorithme à un extrait du problème CTAS (NASA AMES, 2006), le « Simple Update System » (SUS). C'est un système très simple dont la structure est présentée à la figure 5. Il est constitué d'un serveur central auquel les clients se connectent. Certains opérateurs sont autorisés à accéder à ce serveur ; ils peuvent actionner manuellement une mise à jour. Quand cela se produit, le serveur doit envoyer une requête à *tous* les clients connectés, leur demandant de récupérer des données à partir de leur source de données (scénario « update », voir figure 7a). Les clients choisissent un de leurs miroirs de téléchargement et commencent à recevoir ces nouvelles données. Si le transfert échoue, ils notifient le serveur qui déconnectera tous les clients après avoir reçu la réponse de tous. Si tous les clients réussissent à obtenir les données mises à jour, tout va bien. Bien que ce soit un petit exemple, il comporte les principales particularités du système TRACON du CTAS (Whittle *et al.*, 2002 ; Bontemps *et al.*, 2003) qui nous ont conduits à étendre la notation des machines à états et des scénarios.

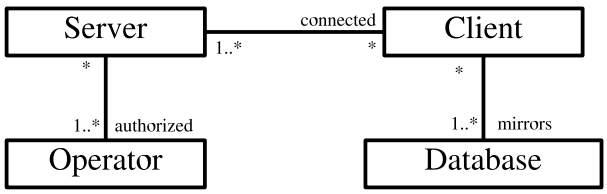
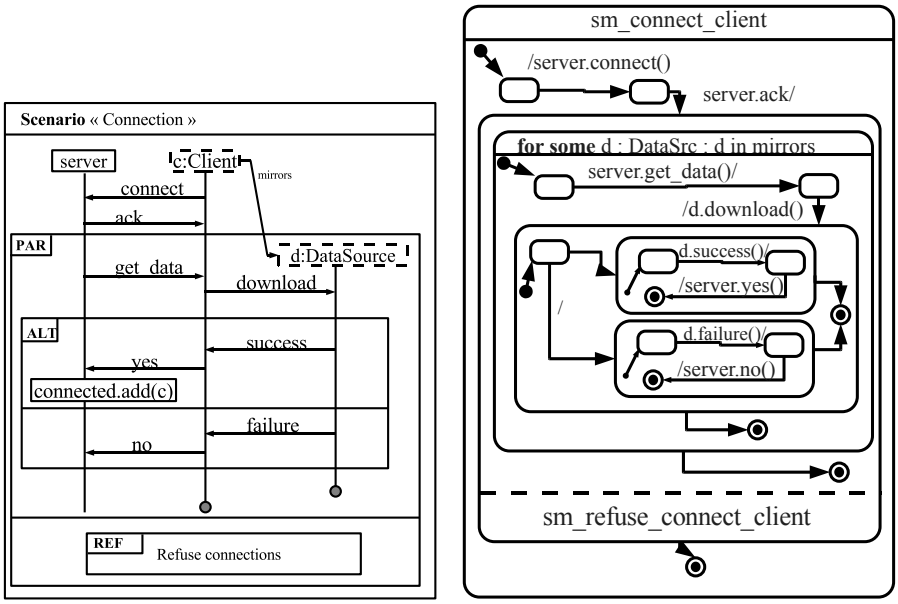


Figure 5. *Diagramme de classes pour SUS*

Les deux principaux scénarios, c'est-à-dire « connection » et « update », sont présentés dans les figures 6a et 7a. Ces scénarios référencent un autre scénario nommé « Disconnect All », non représenté ici. Notre algorithme de synthèse a été appliqué au scénario « connection » pour la classe `Client` (voir figure 6b) et au scénario « update » pour `Server` (voir figure 7b).



(a) Scénario « Connection » (b) Machine à états pour « Client »

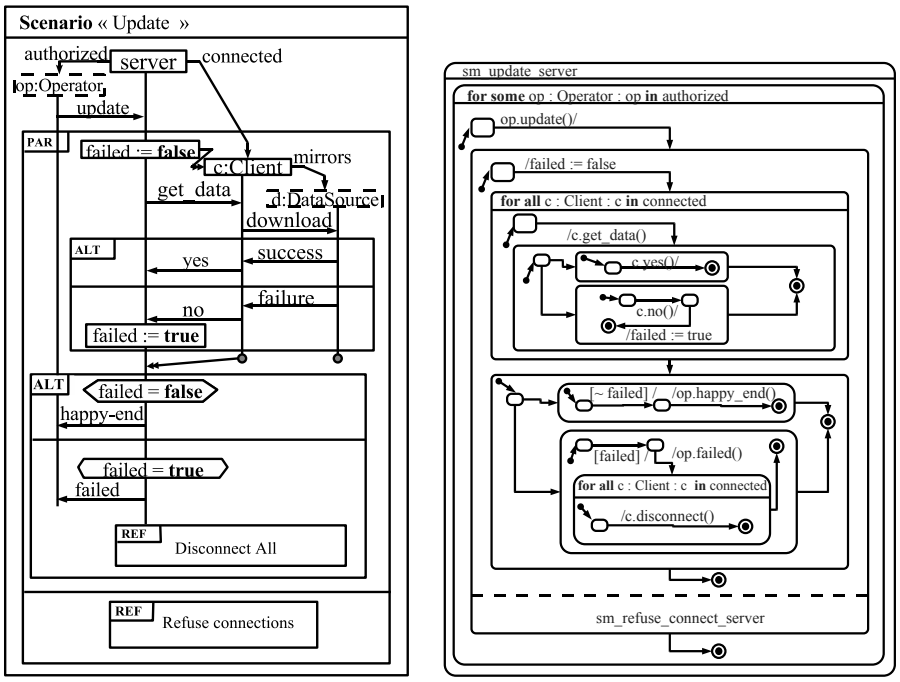
Figure 6. *Synthèse du client à partir du scénario « Connection »*

Dans cet exemple, les comportements induits ne présentent pas de surprise. Ceci indique que les machines reçoivent l'information nécessaire à leur tâche. L'élaboration des MSCs nécessaires est ici simple, mais ne nous permet pas d'extrapoler pour des exemples plus complexes.

5. Conclusion

Nous avons présenté une extension simple de deux notations de comportement pour les systèmes orientés objet, pour permettre des descriptions de tels systèmes au niveau de leurs classes. Cette extension ajoute la quantification universelle et existentielle sur les instances aux notations usuelles des machines à états hiérarchiques/concurrentes (Statecharts) et des scénarios de haut niveau (HMSC). Les algorithmes de synthèse classiques, construisant un ensemble de machines à états à partir d'un modèle structuré composé de scénarios, ont été subséquemment modifiés pour fonctionner au niveau classe. Nous avons illustré l'application de cet algorithme avec un petit exemple.

Des questions importantes sont toutefois encore ouvertes : Comment peut-on détecter et énoncer les scénarios induits dans les QHMSC ? Quelles contraintes syntaxiques sont nécessaires pour s'assurer qu'aucun scénario induit n'apparaît, comme



(a) Scénario « Update » (b) Machine à états pour « Server »

Figure 7. *Synthèse du serveur à partir du scénario « Update »*

pour les MSC d'instance chez Finkbeiner *et al.* (2001) ? Quelle méthode doit être suivie pour prendre plusieurs scénarios de classe indépendants et dériver un scénario intégré nécessaire pour notre algorithme ? Les utilisateurs produisent souvent des scénarios instanciés, quelle méthode utiliser pour les généraliser ? Nous devons générer du code exécutable à partir de nos modèles, pour que notre algorithme soit vraiment utile au sein de l'approche MDA ou pour faire du prototypage rapide. Dans ce cas, comment peut-on surmonter la différence entre les programmes générés et une implantation existante, un problème déjà soulevé par Whittle *et al.* (2002) ? Est-il possible d'intégrer d'autres constructions utiles comme la variabilité (pour les lignes de produits logiciels), les contraintes temps-réel, les communications asynchrones, la composition séquentielle faible ou la préemption ? Peut-on modifier d'autres notations, comme les diagrammes d'activité, pour prendre en compte le comportement au niveau classe ?

Finalement, notre notation est limitée car elle ne permet pas de créer ou détruire des instances. Il est possible de contourner ce problème en ajoutant des objets initialement inactifs et en introduisant une classe spéciale de construction d'objets (Factory) qui implante la création et la destruction, mais une solution plus élégante serait bienvenue.

6. Bibliographie

Abdalla M., Khendek F., Butler G., « New results on deriving SDL specifications from MSCs », *SDL Forum*, 1999, p. 51-66.

Alur R., Etessami K., Yannakakis M., « Inference of Message Sequence Charts », *Proceedings of 22nd International Conference on Software Engineering*, 2000, p. 304-313.

Amyot D., Eberlein A., « An Evaluation of Scenario Notations for Telecommunication Systems Development », *Telecommunications Systems Journal*, vol. 24, n° 1, 2003, p. 61-94.

Biermann A. W., Krishnaswamy R., « Constructing Programs from Example Computations », *IEEE Transactions on Software Engineering (TSE)*, vol. SE-2, n° 3, 1976, p. 141-153.

Bontemps Y., Heymans P., Kugler H., « Applying LSCs to the specification of an Air Traffic Control system », *in* S. Uchitel, F. Bordeleau (eds), *Proc. of the 2nd Int. Workshop on "Scenarios and State Machines : Models, Algorithms and Tools" (SCESM'03), at the 25th Int. Conf. on Soft. Eng. (ICSE'03)*, IEEE, Portland, OR, USA, May, 2003.

Bontemps Y., Schobbens P.-Y., Löding C., « Synthesizing Open Reactive Systems from Scenario-Based Specifications », *Fundamenta Informaticae*, vol. XX, 2004, p. 1-31.

Bordeleau F., Cameron D., « On the Relationship between use-case maps and Message Sequence Charts », *in* E. Sherratt (ed.), *SAM 2000, 2nd Workshop on SDL and MSC, Col de Porte, Grenoble, France, June 26-28, 2000*, VERIMAG, IRISA, SDL Forum, 2000, p. 123-138.

Buhr R., « Use Case Maps as Architectural Entities for Complex Systems », *IEEE Transactions on Software Engineering*, vol. 24, n° 2, 1998, p. 1131-1155.

Engels A. G., Languages for analysis and testing of events sequences, PhD thesis, Technische Universiteit Eindhoven, 2001.

Finkbeiner B., Krüger I. H., « Using Message Sequence Charts for Component-based Formal Verification », *Proc. of OOPSLA 2001 Workshop on Specification and Verification of Component-Based Systems*, Tampa Bay, FL, USA, October, 2001.

Harel D., Statecharts : A Visual Formalism for Complex Systems, Technical report, Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel, July, 1986. February 1984, extensively revised February 1986.

Harel D., « From Play-In Scenarios to Code : An Achievable Dream », *IEEE Computer*, vol. 34, n° 1, 2001, p. 53-60.

Harel D., Kugler H., « Synthesizing State-Based Object Systems from LSC Specifications », *International Journal of Foundations of Computer Science*, vol. 13, n° 1, 2002, p. 5-51. (Preliminary version appeared in, *Proc. Fifth Int. Conf. on Implementation and Application of Automata (CIAA 2000)*, LNCS 2088, July 2000.).

Harel D., Kugler H., Pnueli A., « Synthesis Revisited : Generating Statechart Models from Scenario-Based Requirements », *in* H.-J. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, G. Taentzer (eds), *Formal Methods in Software and Systems Modeling, Essays Dedicated to Hartmut Ehrig, on the Occasion of His 60th Birthday*, vol. 3393 of *Lecture Notes in Computer Science*, Springer, 2005, p. 309-324.

Hélouët L., Jard C., « La manipulation formelle de scénarios. L'exemple des Message Sequence Charts. », *Proc. of MSR'01 (Colloque Francophone sur la Modélisation des Systèmes Réactifs)*, 2001.

Hopcroft J. E., Motwani R., Ullman J. D., *Introduction to Automata Theory, Languages, and Computation*, 2nd edn, Addison-Wesley, 2001.

ITU, *MSC-2000 : ITU-T Recommendation Z.120 : Message Sequence Chart (MSC)*, International Telecommunication Union (prev. CCITT), Geneva. 2000, `http://www.itu.int/`.

ITU, *UCM : ITU-T Recommendation Z.152 : Use Case Maps (UCM)*, International Telecommunication Union (prev. CCITT), Geneva. September, 2003, `http://www.itu.int/`.

ITU, *MSC-2004 : ITU-T Recommendation Z.120 : Message Sequence Chart (MSC)*, International Telecommunication Union (prev. CCITT), Geneva. 2004.

Jarke M., Bui X. T., Carroll J. M., « Scenario Management : an Interdisciplinary Approach », *Requirements Engineering Journal*, vol. 3, n° 3-4, 1998, p. 155-173.

Khendek F., Robert G., Butler G., « Implementability of Message Sequence Charts », *1st Conference on SDL and MSCs (SAM98), Berlin*, July 29, 1998.

Koskimies K., Männistö T., Systä T., Tuomi J., SCED : A Tool for Dynamic Modelling of Object Systems, Technical Report n° Report A-1996-4, Department of Computer Science, University of Tampere, July, 1996.

Krüger I., Grosu R., Scholz P., Broy M., « From MSCs to Statecharts », *in* F. J. Rammig (ed.), *DIPES*, vol. 155 of *IFIP Conference Proceedings*, Kluwer Academic Publishers, 1999, p. 61-72.

Leue S., Mehrmann L., Rezai M., « Synthesizing ROOM Models from Message Sequence Charts Specifications », *Proc. of 13th IEEE Conference on Automated Software Engineering*, Honolulu, Hawaii, October, 1998.

Liang H., Dingel J., Diskin Z., « A comparative survey of scenario-based to state-based model synthesis approaches », *in* J. Whittle, L. Geiger, M. Meisinger (eds), *SCESM '06 : Proceedings of the 2006 International Workshop on Scenarios and State Machines : Models, Algorithms, and Tools, Shanghai, China, May 27, 2006*, ACM, 2006, p. 5-12.

Mansurov N., Zhukov D., « Automatic synthesis of SDL models in use case methodology », *SDL Forum*, 1999, p. 225-240.

Marelly R., Harel D., Kugler H., « Multiple Instances and Symbolic Variables in Executable Sequence Charts », *Proc. 17th Ann. ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02)*, Seattle, WA, 2002, p. 83-100.

Martínez H. N. C., « Synthesizing State-Machine Behaviour from UML Collaborations and Use Case Maps », *in* A. Prinz, R. Reed, J. Reed (eds), *SDL 2005 : Model Driven, 12th International SDL Forum, Grimstad, Norway, June 20-23, 2005, Proceedings*, vol. 3530 of *Lecture Notes in Computer Science*, Springer, 2005, p. 339-359.

NASA AMES, *Overview of Center TRACON Automation System (CTAS)*. November, 2006, `http://ctas.arc.nasa.gov`.

OMG, *UML Specification (2.0)*. September, 2003, `http://www.omg.org/uml`.

Uchitel S., Elaboration of Behaviour Models and Scenario-based Specifications using Implied Scenarios, PhD thesis, Imperial College London, January, 2003.

Uchitel S., Chatley R., Kramer J., Magee J., « System architecture : the context for scenario-based model synthesis », *SIGSOFT '04/FSE-12 : Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, ACM Press, New York, NY, USA, 2004a, p. 33-42.

Uchitel S., Kramer J., « A Workbench for Synthesizing Behaviour Models from Scenarios », *Proc. of the 23rd IEEE International Conference on Software Engineering (ICSE'01)*, ACM, 2001a.

Uchitel S., Kramer J., Magee J., « Detecting Implied Scenarios in Message Sequence Chart Specifications », *in* V. Gruhn (ed.), *Proceedings of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC/FSE-01)*, vol. 26, 5 of *Software Engineering Notes*, ACM Press, New York, 10–14, 2001b, p. 74-82.

Uchitel S., Kramer J., Magee J., « Synthesis of Behavioral Models from Scenarios », *IEEE Trans. Software Eng*, vol. 29, n° 2, 2003, p. 99-115.

Uchitel S., Kramer J., Magee J., « Incremental elaboration of scenario-based specifications and behavior models using implied scenarios », *ACM Trans. Softw. Eng. Methodol*, vol. 13, n° 1, 2004b, p. 37-85.

von der Beeck M., « A structured operational semantics for UML-statecharts », *Jour. on Software and Systems Modeling*, vol. 1, n° 2, 2002, p. 130-141.

Weidenhaupt K., Pohl K., Jarke M., Haumer P., « Scenario Usage in System Development : A Report on Current Practice », *IEEE Software*, vol. 15, n° 2, 1998, p. 34-45.

Whittle J., Schumann J., « Generating Statechart Designs from Scenarios », *22nd International Conference on Software Engineering (ICSE 2000)*, ACM, Limerick, Ireland, June, 2000, p. 314-323.

Whittle J., Schumann J., « Statechart Synthesis from Scenarios : an Air Traffic Control Case Study », *Proc. of "Scenarios and State-Machines : models, algorithms and tools" workshop at the 24th Int. Conf. on Software Engineering (ICSE 2002)*, ACM, Orlando, FL, May, 20th, 2002. `http://www.cs.tut.fi/~tsysta/ICSE/papers/`.

Yamanaka M., Komura S., Kato J., Ichikawa H., « Deriving Protocols from Message Sequence Charts in a Communicating Processes Model », *IEICE Transactions on Information and Systems*, vol. E79-D, n° 11, 1996, p. 1533-1544.

Ziadi T., Hélouët L., Jézéquel J.-M., « Revisiting Statechart Synthesis with an Algebraic Approach », *Proc. of 26th International Conference on Software Engineering (ICSE)*, IEEE Computer Society, Edinburgh, May, 2004, p. 242-251.

Article reçu le 20 juin 2006
Article accepté le 18 janvier 2007

Yves Bontemps *est maître et docteur en Informatique des Facultés Universitaires Notre-Dame de la Paix (FUNDP), Université de Namur en Belgique. En 2005, il a achevé sa thèse de doctorat sous la direction du Professeur Pierre-Yves Schobbens, comme chercheur aspirant du Fonds National de la Recherche Scientifique. Il est actuellement collaborateur scientifique de l'Institut d'Informatique de l'Université de Namur.*

Germain Saval *est titulaire d'un diplôme d'études spécialisées en Informatique de l'Université Pierre et Marie Curie Paris VI et du Conservatoire National des Arts et Métiers. Il poursuit une recherche doctorale sous la direction du Professeur Patrick Heymans sur le thème du génie*

logiciel. Il est actuellement assistant au sein de l'Institut d'Informatique de l'Université de Namur.

Pierre-Yves Schobbens *est ingénieur et docteur de l'Université Catholique de Louvain. Il a été chargé de recherche au CNRS à Nancy. Il est actuellement professeur aux Facultés Universitaires Notre-Dame de la Paix, Université de Namur. Ses intérêts de recherche sont le génie logiciel formel, les logiques pour le génie logiciel, le développement orienté agents, les lignes de produits logiciels.*

Patrick Heymans *est maître et docteur en Informatique de l'Université de Namur en Belgique, où il est maintenant professeur. Il est membre du centre de recherche PRECISE (Precise REsearch Centre in Information Systems Engineering). Il compte parmi ses principaux intérêts de recherche en génie logiciel, les langages de modélisation, l'ingénierie des exigences et les lignes de produits logiciels.*

ANNEXE POUR LE SERVICE FABRICATION

A FOURNIR PAR LES AUTEURS AVEC UN EXEMPLAIRE PAPIER
DE LEUR ARTICLE ET LE COPYRIGHT SIGNE PAR COURRIER
LE FICHIER PDF CORRESPONDANT SERA ENVOYE PAR E-MAIL

1. ARTICLE POUR LA REVUE :
   RSTI - TSI – 26/2007. AFADL 2006

2. AUTEURS :
   Yves Bontemps — Germain Saval
   Pierre-Yves Schobbens — Patrick Heymans

3. TITRE DE L'ARTICLE :
   Synthèse de diagrammes d'états par classe
   à partir de diagrammes de séquence

4. TITRE ABRÉGÉ POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :
   Synthèse de diagrammes d'états par classe

5. DATE DE CETTE VERSION :
   31 août 2007

6. COORDONNÉES DES AUTEURS :

   – adresse postale :

   Institut d'Informatique, FUNDP, Namur
   rue Grandgagnage, 21
   B5000 - Namur (Belgique)

   {ybo,gsa,pys,phe}@info.fundp.ac.be

   – téléphone : +32 (0) 81 72 49 85

   – télécopie : +32 (0) 81 72 49 67

   – e-mail : pys@info.fundp.ac.be

7. LOGICIEL UTILISÉ POUR LA PRÉPARATION DE CET ARTICLE :
   LaTeX, avec le fichier de style `article-hermes2.cls`,
   version 1.23 du 02/08/2006.

8. FORMULAIRE DE COPYRIGHT :
   Retourner le formulaire de copyright signé par les auteurs, téléchargé sur :
   `http://www.revuesonline.com`

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél. : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com