RenPar'17 / SympA'2006 / CFSE'5 / JC'2006
Canet en Roussillon, 4 au 6 octobre 2006

PauWare : un modèle de composant basé état

Fabien Romeo, Cyril Ballagny et Franck Barbier

LIUPPA
Université de Pau et des Pays de l'Adour
Av. de l'Université, B.P. 1155
F-64013 PAU - France
fabien.romeo@univ-pau.fr, cyril.ballagny@etud.univ-pau.fr, franck.barbier@franckbarbier.com

Résumé
Issu de l'ingénierie logicielle basée composant (CBSE) et de l'ingénierie dirigée par les modèles (MDE), le modèle de composant PauWare intègre dans ses composants des modèles exécutables basés sur les *Statecharts* de Harel et plus précisément, sur les *StateMachine Diagrams* d'UML 2. La composition dans PauWare est donc prise en compte à la fois aux niveaux structurel et comportemental des composants. Cet article présente les principes de composition propres au modèle de composant PauWare après avoir dressé un historique des relations de composition dont il est issu. PauWare a été implémenté sur les trois plateformes JAVA, *standard*, *enterprise* et *mobile*.

Mots-clés : Composants, Statecharts, Composition logicielle, MDE, UML

Abstract
At the crossroads of Component-Based Software Engineering (CBSE) and Model-Driven Engineering (MDE), PauWare component model embeds into its components executable models of their behavior designed with Harel's Statecharts and more precisely with UML 2 State Machine Diagrams. Composition in PauWare is thus considered at two different levels : structural and behavioral. After a historical outline of the composition relations on which it is based, this paper presents the inner composition principles of the PauWare component model. PauWare has been implemented on the three JAVA platforms, namely standard, enterprise and mobile.

Keywords: Components, Statecharts, Software Composition, MDE, UML

1. Introduction

Les dernières années ont vu s'imposer dans le développement des applications logicielles la reconnaissance des composants logiciels comme paradigme de premier choix. L'idée de structurer l'architecture des systèmes par composition de composants et de sous-composants est maintenant bien adoptée et les bénéfices qui en résultent dans le processus global de développement ne sont plus contestables. Preuve en est le nombre important d'approches qui tirent profit des modèles de composants tels que EJB, COM 2+, CCM, ou Fractal.

L'ingénierie dirigée par les modèles (*Model-Driven Engineering* - MDE [14]) s'intéresse également à la notion de composition notamment à travers l'*Object Management Group* (OMG) qui préconise l'utilisation de son langage de modélisation phare UML dans sa vision du MDE, le *Model-Driven Architecture* (MDA). Dans ses premières versions, UML permettait déjà de modéliser des systèmes par composition avec les relations d'agrégation et de composition. Aujourd'hui, l'introduction des *Component Diagrams* et des *Composite Structure Diagrams* dans UML 2.0 souligne encore cet intérêt [2].

Le MDA, et le MDE en général, préconisent l'utilisation des modèles comme des entités de première classe dans le processus de développement logiciel. Cette approche distingue ainsi deux types de modèles : les *Plaform-Independent Models* (PIM) qui définissent le comportement et les fonctionnalités métier

du logiciel indépendamment de toutes considérations spécifiques à une plateforme technologique particulière et les *Platform-Specific Models* (PSM) qui sont le résultat de transformations de PIM, intégrant en plus des détails d'implémentation du logiciel spécifiques à une plate-forme donnée.

Steve Cook propose que le MDA soit envisagé selon trois angles différents [20]. Le premier implique l'utilisation d'UML pour construire des PIM qui sont ensuite transformés en PSM à partir desquels le code est généré. Le second ne considère pas UML directement mais s'attache plus au méta-modèle d'UML, le *MetaObject Facility* (MOF), pour définir des langages de modélisation et de transformation. Le troisième vise à créer une plateforme d'exécution pour UML qui devient ainsi un langage de programmation de très haut niveau d'abstraction [18, 23].

PauWare s'inscrit dans cette troisième façon d'appréhender le MDA. Il utilise ainsi des modèles éprouvés en modélisation, à savoir les *Statecharts* [11] d'UML, pour définir le comportement métier de ses composants logiciels qui sont alors directement exécutables.

Dans la section 2, cet article présente le modèle de composant PauWare en considérant tout d'abord les relations de composition dont il s'inspire, puis à travers les principes qui le régissent. Dans la section 3, nous aborderons plus en profondeur les techniques de composition de PauWare. La section 4 conclura notre travail et en présentera les perspectives envisagées.

2. Le modèle de composant PauWare

Dans des travaux précédents [6], nous nous sommes appuyés sur les relations d'agrégation et de composition dans UML pour définir une relation de composition logicielle axée sur la structure et les propriétés non-fonctionnelles des composants. Nous rappelons ces travaux dans la première partie de cette section, puis dans la seconde, nous présentons les principes de composition de notre modèle de composant qui prend également en compte le comportement fonctionnel des composants.

2.1. Historique : l'agrégation/composition dans UML

Le développement basé composant est apparu dans le prolongement de la technologie objet en s'appuyant et en étendant ses principes fondateurs [19]. La notion de composition au sens du CBSE se retrouve ainsi à l'état embryonnaire dans la technologie objet. UML, qui unifie les principales méthodes orientées-objet des années 90, intègre donc dès ses premières versions des mécanismes permettant de modéliser le logiciel comme un assemblage d'entités plus petites. Cette décomposition/composition hiérarchique est modélisable grâce aux relations d'agrégation et de composition.

Cependant, la sémantique [13] de ces deux relations donnée par UML souffre d'imprécisions et d'incohérences, ce qui empêche de les utiliser précisemment [3]. Elles sont généralement «comprises »comme dans [21] : *L'agrégation permet de représenter des relations de type maître et esclaves, tout et parties ou composé et composants. [...] La contenance physique est un cas particulier de l'agrégation, appelé composition*. Cela se traduit dans le méta-modèle UML par un attribut *aggregation* de type *AggregationKind* qui définit le type d'agrégation de la relation (ou de l'association exactement), c'est-à-dire agrégation ou composition. Le type agrégation est représenté au niveau modèle par un losange blanc (*white diamond*) et le type composition par un losange noir (*black diamond*).

Ainsi l'agrégation et la composition sont définies comme des types d'agrégation, la composition définissant un couplage plus fort entre composite et composants en liant leurs cycles de vie et en contraignant un composant à n'appartenir qu'à une seule composition et par conséquent à un seul composite. De façon générale, ces deux relations sont conceptuellement de même niveau et doivent être plutôt envisagées comme des sous-types d'un même type de relation. D'où l'idée dans [4], de proposer un méta-modèle UML plus formel qui définit l'agrégation et la composition comme des sous-types d'une relation plus abstraite, la relation Tout-Partie.

La relation Tout-Partie est inspirée de la théorie mathématique de Lesniewski [16], la méréologie. Cette

base théorique a donc permis à [4] de spécifier cette relation au niveau du méta-modèle d'UML en dégageant des propriétés primaires intrisèques à tout type de relation Tout-Partie (nature binaire, propriété émergente, asymétrie, ...) et des propriétés secondaires (encapsulation, transitivité, séparabilité, ...) permettant de spécialiser cette relation, et notamment de définir les sous-types agrégation et composition. Ce travail a ensuite été envisagé sur l'assemblage des composants logiciels dans [6].

Bien qu'elle enrichisse la sémantique d'UML, cette proposition ne répond pas entièrement aux problèmes du CBSE. En effet, elle envisage la composition plutôt d'une manière statique ou structurelle, alors que les composants logiciels ont de plus en plus vocation à être des entités dynamiques au comportement autonome, notamment avec le rapprochement des concepts d'agent et de composant [9] ou encore avec l'émergence de l'informatique dite «autonomique »[15]. La version 2 d'UML a étoffé les diagrammes de composants en ajoutant les *Composite Structure Diagrams* qui permettent d'attacher des comportements aux composants [7] mais les connexions entre diagrammes de structure et de comportement restent floues. C'est pourquoi nous proposons un modèle de composant prenant en compte la composition au niveau comportemental.

2.2. Principes

Dans PauWare, le principe est d'incorporer à chaque composant un *Statechart* [11] définissant le comportement du composant par un ensemble d'états et de transitions. Les figures 1 et 2 spécifient en UML un exemple de composant PauWare, nommé *PauWare Component*, respectivement du point de vue de sa structure et du point de vue de son comportement. Le comportement représenté par le *Statechart* de la figure 2 est exécuté par un moteur intégré au composant, le *Statechart_monitor* de la figure 1. Détaillons le comportement du composant *PauWare Component*.

Dans son exécution, le composant ne peut être que dans un de ses deux états mutuellement exclusif *Idle* ou *Busy*. En suivant le formalisme des *Statecharts*, le composant se trouve à l'initialisation dans l'état *Idle*. Dans cet état, une requête sur le service *go* spécifié dans l'interface de service fonctionnel du composant provoquerait alors un événement qui déclencherait une transition de l'état *Idle* à l'état *Busy*, toute autre requête de service serait inopérante. L'état *Busy* étant un état composite divisé en régions parallèles, le composant se trouverait ensuite simultanément dans les sous-états *S1*, *S2* et *S3*. L'entrée dans l'état *S1*, et par conséquent dans son sous-état initial *S11*, exécuterait l'action *w* implémentée par le composant. De même le passage dans l'état *S2* et donc dans son sous-état initial *S22* provoquerait l'envoi d'un signal *request_h* sur le composant *self*, c'est-à-dire sur lui-même.

Cet exemple montre les relations qu'il existe entre la structure des composants PauWare et leur comportement. Le modèle de composant PauWare associe ainsi plusieurs éléments :
- les services fournis du composant correspondent à des événements du *Statechart* du composant ;
- les actions métier du composant correspondent à des actions du *Statechart* du composant. Le formalisme *entry / action* est employé pour une action exécutée à l'entrée d'un état, *exit / action* à la sortie d'un état, *do / action* durant l'état ou *event / action* lors d'un changement d'état ;
- les services requis du composant correspondent à des signaux du *Statechart* du composant, c'est-à-dire des messages envoyés à un autre *State Machine* et détectés comme des événements par le receveur. Le formalisme d'OCL 2 suivant est employé : ^*receiver.event*. Les signaux seront examinés en détail dans la section 3.

Lorsque ces éléments sont internes au composant ceux-ci ne sont pas exposés dans les interfaces du composant. C'est le cas du service *request_h* de l'exemple qui peut être considéré à la fois comme un événement du *Statechart* et un envoi de signal du *Statechart* à lui-même.

La figure 1 montre également un autre type de service représenté par le service *reset*. Ce service qui apparaît dans l'interface de configuration du composant n'est plus un service fonctionnel, mais un service destiné à la configuration ou au management du composant comme utilisé dans [24]. Par cette interface, le système de management accède directement au *Statechart* qui contrôle le composant et peut ainsi reconfigurer le composant, par exemple en le forçant dans un état particulier.

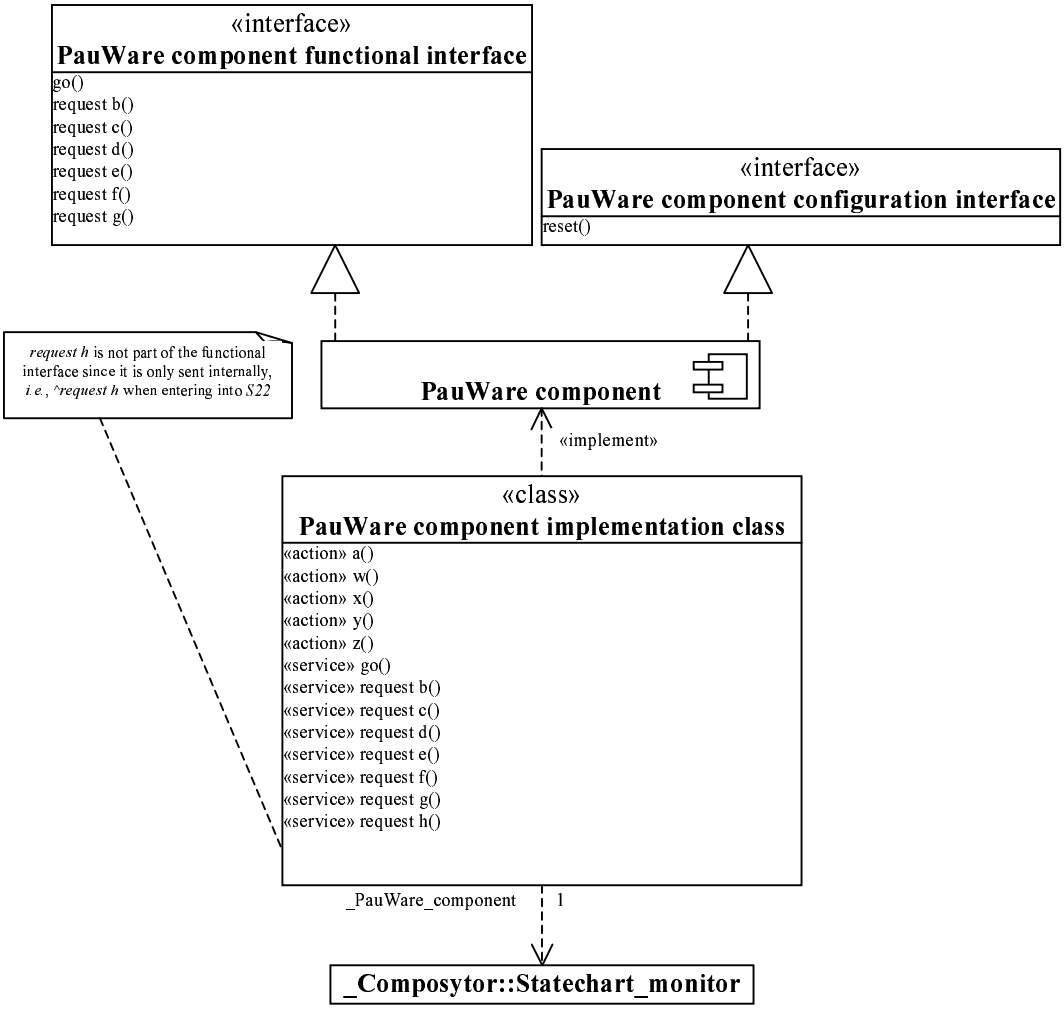

FIG. 1 – Structure du composant *PauWare Component*

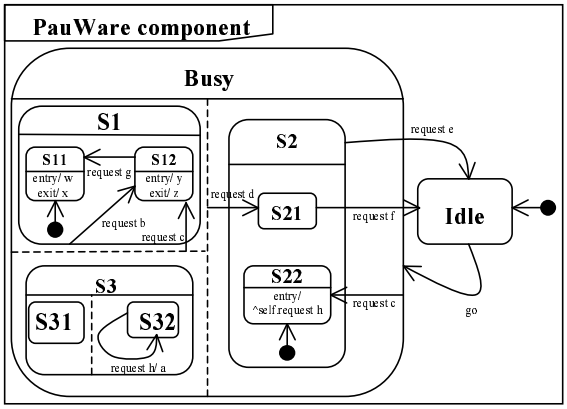

FIG. 2 – Comportement du composant *PauWare Component*

Les fondements du modèle de composants PauWare reposent sur la réification du comportement des composants en *Statecharts* exécutables et accessibles par le modèle, permettant d'envisager des mécanismes de composition basés sur la composition des *Statecharts*. Ces mécanismes seront détaillés dans la section 3.

3. Les techniques de composition

A travers les diagrammes de composants, la notation UML 2.0 propose un modèle d'assemblage des composants suivants deux directions : une horizontale et une verticale. Néanmoins, ces formes d'assemblage n'envisagent la composition qu'au niveau de la description statique du système. Nous avons retenu ces deux formes. Le modèle de composant PauWare propose de coupler la composition au niveau statique avec une composition dynamique, c'est-à-dire tenant compte du comportement des composants mis en jeu.

3.1. La composition horizontale

La composition horizontale structurelle est la forme de composition privilégiée dans les modèles actuels de composants (EJB, CCM, .NET, ...). Elle doit être vue comme une forme de collaboration entre composants : un composant A requiert les services fournis par un composant B. L'ensemble des services fournis par B est décrit dans une interface, de même pour les services requis par A. L'assemblage structurel entre ces deux interfaces est ensuite réalisé, au sein de la notation UML 2.0, par des connecteurs d'assemblage. Il résulte de cet assemblage la création d'un canal d'événements. Ce canal est soit unidirectionnel (de A vers B), soit bidirectionnel, si on considère respectivement un événement de signal (asynchrone) ou un événement d'appel (synchrone) [25, 5]. Cette différence est due au fait que l'appelant d'un événement d'appel ne reprend son flot d'exécution que lorsque l'appelé a terminé le traitement de l'événement, il est donc nécessaire que l'appelant soit notifié de la fin de ce traitement.

La génération d'événements et l'invocation d'opérations sont les deux modes de communication retenus dans les *Statecharts* [12] et dans le modèle de composant PauWare. Néanmoins, la spécification UML impose la communication asynchrone, donc par événements de signal, dans le cadre de la composition horizontale. Ce mode de communication permet d'éviter les phénomènes de réentrance lors d'un cycle *run-to-completion* [25].

En effet, une machine à états ne doit gérer qu'une occurence d'événement par cycle, sans quoi l'intégrité de la machine n'est pas garantie. Rappelons qu'à chaque service fourni correspond un événement dans la machine à états du composant (cf. section 2.2). Considérons le diagramme de séquence de la figure 3 montrant la communication, par événements d'appel, entre deux composants.

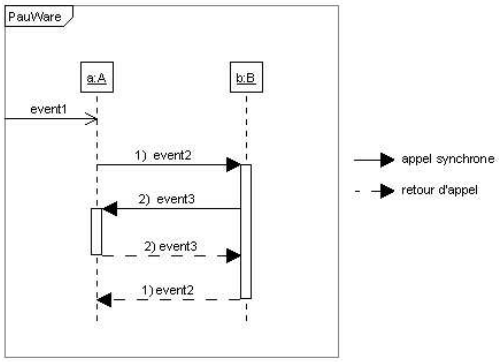

FIG. 3 – Séquençage d'événements avec réentrance dans le composant A

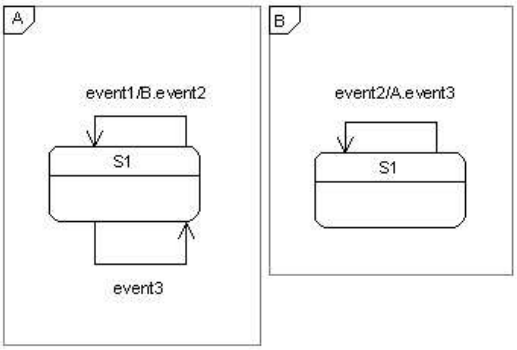

FIG. 4 – Machine à états des composants A et B avec événement d'appel

Un composant A reçoit un événement `event1` provoquant l'occurence d'un événement `event2` dans un composant B lui-même provoquant l'occurence d'un nouvel événement (`event3`) dans A. Le cycle *run-to-completion* du traitement de l'événement `event1` est donc en cours alors que le composant A doit traiter une nouvelle occurence d'événement. Ceci viole la sémantique du *run-to-completion*. Il est donc nécessaire de procéder par événement de signal (cf. figure 5) : `event3` est alors mis dans le pool des événements de A et ne sera traité que lors du prochain cycle *run-to-completion*. Cette différence s'exprime au niveau du diagramme d'états du composant B (cf. figure 6) par le symbole `^` préfixant l'action `A.event3`. Notons que ce symbole est connu dans l'*Object Constraint Language* (OCL 2.0)[10] comme étant l'opérateur `hasSent`.

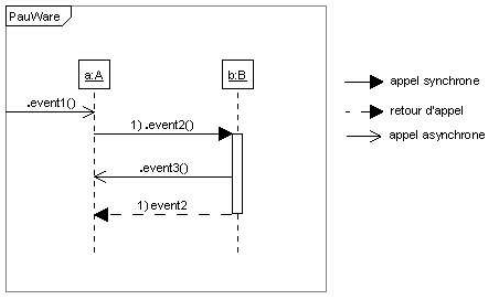

FIG. 5 – Séquençage d'événements sans réentrance

3.2. Besoin de composition verticale

En plus de la composition horizontale, il est intéressant de considérer la composition verticale également appelée composition hiérarchique [17] ou récursive [8]. Dans ce type de composition, un composant de granularité supérieure, le composite, utilise les services de composants de granularité moindre, les sous-composants, eux-même pouvant être des composants composites.

Parmi les modèles exploitant ce type de composition, nous pouvons citer le modèle de composant Fractal. Il possède une sémantique de composition verticale de composants avec partageabilité [8], c'est-à-dire qu'un composant peut être un sous-composant de plusieurs composants distincts. Il ne s'agit toujours que de composition structurelle.

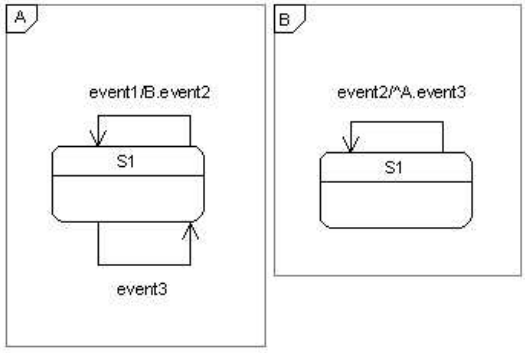

FIG. 6 – Machine à états des composants A et B avec événement de signal

Le modèle de composant PauWare reprend la sémantique de la composition discutée dans [4] et envisage la composition verticale, au niveau structurel et comportemental, de composants non partagés et aux cycles de vie coïncidents. La partageabilité n'est pas permise car il faut pouvoir garantir l'intégrité du comportement des composés en cas d'accès extérieur à un sous-composant [22]. Le cycle de vie du composé doit coïncider avec le cycle de vie de ses sous-composants :
– si le composé préexiste à son sous-composant, le composé ne peut pas s'appuyer de façon sûre sur les états de son composant ;
– si un sous-composant préexiste à son composé, la composition des comportements peut donner un comportement global incohérent.

D'après le formalisme des *Statecharts* [11, 25], deux états sont soit mutuellement exclusifs (decomposition en XOR), soit orthogonaux (décomposition en AND), c'est-à-dire concurrents et indépendants, soit imbriqués (un état est alors un sous-état de l'autre). A l'instar du modèle Cobalt [22], le comportement global d'un composant composite est le résultat de la composition de son comportement propre avec le comportement de ses sous-composants. Afin de préserver la notion d'autonomie des composants, nous proposons que le comportement global du composite résulte de la composition orthogonale (AND) du comportement des composants et du comportement propre du composite. De plus, le comportement global d'un sous-composant est modélisé comme un état de sous-machine [25] pour le composé (cf. figure 7). Cette notion permet de différencier les comportements de plusieurs instances d'un même type de sous-composant dans un composé.

La spécification des *Statecharts* [11] précise qu'une occurrence d'événement est diffusée entre toutes les régions orthogonales d'un état ainsi que vers ses sous-états. Une occurrence d'événement peut donc déclencher plusieurs transitions au sein d'une même machine. Il existe alors un mécanisme de priorité permettant de résoudre les conflits de transitions déclenchables à plusieurs niveaux de la hiérarchie des états.

Dans le cas de notre composition, les deux sous-composants répondent aux mêmes événements, or il est important de pouvoir adresser un événement à un sous-composant particulier. Nous avons donc introduit la notion d'espace de nom pour les événements afin de limiter leur portée. Nous avons adopté le formalisme suivant :

[*<état de sous-machine>*' : :']* *événement*

Pour adresser un événement à un sous-composant particulier, il faut préfixer le nom de l'événement par le nom de l'état de sous-machine dans le composé suivi du séparateur d'espace de nom. Si nous reprenons l'exemple de la figure 7, pour adresser un événement uniquement à `sousComposant1`, il faut nommer l'événement :

sousComposant1 : :un_evenement

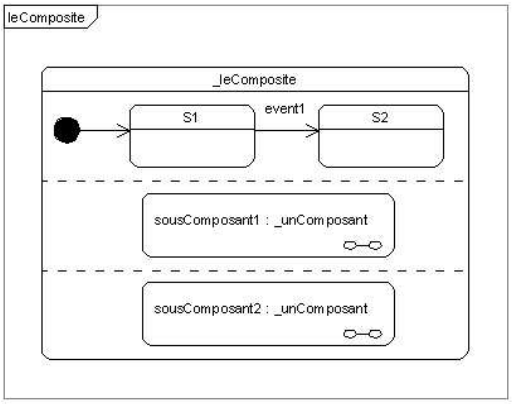

FIG. 7 – Modélisation du comportement d'un composé

L'expression *un_evenement* adresse l'événement à toutes les régions orthogonales du comportement global du composé. Avec un composé ayant une vue boîte blanche sur ses sous-composants, il est possible de limiter la portée d'un événement à un éventuel sous-composant de son sous-composant, donc en profondeur.

3.3. Implémentation

Le modèle de composant PauWare est mis en œuvre grâce à la librairie *PauWare* [1] qui est un moteur d'exécution pour les UML *2 State Machine Diagram*. Elle est écrite en Java et est compatible avec les plate-formes J2EE, J2ME, J2SE. La libraire implémente les techniques de compositions décrites.

Chaque état du composant est une instance d'une sous-classe d'`AbstractStatechart`. Le comportement global du composant est géré par un *monitor* qui est une instance d'une sous-classe d'`AbstractStatechart_monitor`. La sous-classe à instancier dépend du type de plate-forme envisagée pour l'exécution.

Chaque transition de la machine à état est ensuite enregistrée dans le *monitor* en précisant l'événement déclencheur, une éventuelle garde, ainsi que l'action associée suivant la sémantique définie dans UML 2.0 [25]. De plus, suivant les principes du MDE et afin de se concentrer sur la modélisation, l'outil *XMI2PauWare* [1] permet de générer le code associé à un diagramme de machine à états, décrit en XMI. Enfin, la librairie implémente les techniques de compositions décrites. Lorsqu'un composant est composable verticalement, il doit implémenter l'interface `Composable`. Ceci permet de rendre accessible son comportement afin de le composer avec le comportement d'un composant de granularité supérieure.

4. Conclusion

Nous avons présenté PauWare, un modèle de composant qui prend en compte le comportement et les états des composants. Un composant PauWare intégre un *Statechart* modélisant son comportement qui est rendu directement exécutable par la mise en œuvre d'une méthode issue du MDA. Nous avons également présenté les mécanismes de composition qui tirent profit de la réification du comportement des composants PauWare. Alors que la plupart des modèles de composants ne considèrent que la composition horizontale, PauWare permet de plus de composer verticalement les composants à partir d'états qui traduisent leur comportement.

En rendant ainsi le comportement des composants explicite et accessible tout au long du cycle de vie des composants, on améliore leur compréhension par des tiers et par conséquent leur composabilité et leur réutilisabilité. Cela est d'autant plus crucial que les composants sont de plus en plus sujets à des

traitements réalisés par des entités logicielles tant au niveau modèle avec le MDE qu'au niveau de leur exécution avec l'*autonomic computing*.

Des travaux supplémentaires sont encore nécessaires pour déduire complètement et sûrement le comportement global du logiciel du comportement individuel de ses composants. L'encapsulation que nous avons imposée pour la composition verticale permet d'éviter des problèmes épineux dus à la partageabilité, essentiellement des problèmes de synchronisation et de cohérence des *Statecharts*. Nous envisageons de plus amples recherches pour traiter ces problèmes qui ne peuvent être évités pour la composition horizontale puisque dans ce cas l'encapsulation n'est pas réalisable. Nous avons notamment initié des travaux pour traiter ce problème à l'aide d'un système de management supervisant le comportement de l'assemblage des composants [24].

Bibliographie

1. Pauware open source software. http ://www.pauware.com.
2. Franck Barbier. *UML 2 et MDE - Ingénierie des modèles avec études de cas*. Dunod, 2005.
3. Franck Barbier et Brian Henderson-Sellers. Controversies about the black and white diamonds. *IEEE Transactions on Software Engineering*, 29(11) :1056, 2003.
4. Franck Barbier, Brian Henderson-Sellers, Annig Le Parc, et Jean-Michel Bruel. Formalization of the whole-part relationship in the unified modeling language. *IEEE Trans. Software Eng.*, 29(5) :459–470, 2003.
5. Mikaël Beauvois. *Composition comportementale de composants*. PhD thesis, INPG, Grenoble, 2005.
6. Nicolas Belloir, Jean-Michel Bruel, et Franck Barbier. Formalisation de la relation Tout-Partie : application à l'assemblage des composants logiciels. In *Journées composants : flexibilité du système au langage*, pages 99–108, Besançon, France, LIFC/LIP6, 25-26 Octobre 2001.
7. Morgan Bjorkander et Cris Kobryn. Architecting systems with uml 2.0. *IEEE Software*, 2003.
8. Eric Bruneton, Thierry Coupaye, et Jean-Bernard Stefani. Recursive and dynamic software composition with sharing. In *the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, Malaga, Spain, 2002.
9. Martin L. Griss et Gilda Pour. Accelerating development with agent components. *Computer*, 34(5) :37–43, 2001.
10. Object Management Group. UML OCL 2.0 specification, june 2005.
11. David Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, June 1987.
12. David Harel et Eran Gery. Executable object modeling with statecharts. *IEEE computer*, 1997.
13. David Harel et Bernhard Rumpe. Meaningful modeling : What's the semantics of "semantics" ? *Computer, IEEE*, octobre 2004.
14. Steve Kent. Model driven engineering. In *3rd International Conference on Integrated Formal Method*, volume 2335 of *LNCS*, pages 286–298, Turku, Finland, 2002. Springer.
15. Jeffrey Kephart et David Chess. The vision of autonomic computing. In *Computer Magazine*, volume 36, pages 41–50. IEEE Computer Society, 2003.
16. Stanislaw Lesniewski. *O podstawach matematyki*. Przeglad Filosoficzny, 1927-1931.
17. Jeff Magee, Naranker Dulay, Susan Eisenbach, et Jeff Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.
18. Stephen J. Mellor et Marc J. Balcer. *Executable UML : A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.
19. Bertrand Meyer. On to components. *Computer (IEEE)*, 32 :139–140, 1999.
20. Granville Miller, Scott Ambler, Steve Cook, Stephen Mellor, Karl Frank, et Jon Kern. Model driven architecture : the realities, a year later. In *OOPSLA '04 : Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 138–140, New York, NY, USA, 2004. ACM Press.
21. Pierre-Alain Muller. *Modélisation objet avec UML*. Eyrolles, 1997.
22. Chabane Oussalah, Martine Magnan, et Sylvain Vauttier. Modélisation du comportement des objets

composites. In *the 3rd International Symposium on Programming and Systems*, Alger, Algérie, 1997.
23. Chris Raistrick, Paul Francis, et John Wright. *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004.
24. Fabien Romeo et Franck Barbier. Management of wireless software components. In *the 10th International Workshop on Component-Oriented Programming in the 19th European Conference on Object-Oriented Programming*, Glasgow, Scotland, July 2005.
25. James Rumbaugh, Ivar Jacobson, et Grady Booch. *UML 2.0 Guide de référence*. CampusPress, 2004.