Intégration de propriétés temporelles dans des applications à base de composants

Sébastien Saudrais, Olivier Barais, Laurence Duchien, Noël Plouzeau

Intégration de Propriétés Temporelles dans des Applications à Base de Composants

Sébastien Saudrais[†], Olivier Barais[†], Laurence Duchien[††] et Noël Plouzeau[†]

[†]IRISA France, Projet Triskell ⋆ – [††]INRIA France, Projet ADAM
{ssaudrai, barais, plouzeau}@irisa.fr, duchien@lifl.fr

Résumé Prendre en compte le temps dans le développement d'applications à base de composants reste une tâche difficile pour le concepteur car le temps n'est pas quantifiable comme les autres ressources et ne peut donc être structuré de manière isolée. Face à ce défi, cet article présente une technique pour intégrer des propriétés temporelles dans des composants logiciels traditionnels. L'objectif de nos travaux est donc d'aider l'architecte à introduire le temps dans les composants sans interférer avec les fonctionnalités déjà définies. Pour cela, nous définissons des motifs correspondant à des propriétés de temps, comme par exemple le temps de réponse ou la période. Nous intégrons ensuite ces motifs aux composants afin d'obtenir des composants avec des informations de temps. Nous appliquons le principe de la séparation des préoccupations pour permettre la spécification du temps de manière indépendante de la spécification fonctionnelle traditionnelle. Ceci facilite l'intégration des outils de vérification temporelle dans le développement de logiciels en permettant l'emploi des méthodes formelles pour le temps lors de l'assemblage de composants, ceci sans modifier le processus de conception existant.

1 Introduction

Beaucoup d'applications modernes incluent le temps comme une caractéristique importante de leur comportement, comme par exemple les applications de communication (*chat* vidéo, édition de texte concurrente, etc). Fondées généralement sur des approches à base de composants, construites comme un assemblage de modules logiciels réutilisables, ces approches prennent difficilement en compte les propriétés liées au temps lors de la phase de construction de l'application. Or, les composants logiciels impliquent des opérations et des coordinations complexes, et sont écrits principalement dans des langages tels que Java, C++ ou C#. Dès lors, travailler avec des propriétés de temps n'est pas aisé pour le développeur de composants. Des langages de modélisation tel que UML incluent des notions de temps informelles [1] et de fait rarement utilisées lors de l'assemblage des composants. De plus, même si certains langages de programmation et leur cadre logiciel associé incluent aussi des caractéristiques temporelles [10], tous ces modèles de temps ne sont pas suffisamment formels pour les utiliser lors de la conception des composants ou lors de la phase d'assemblage. Par conséquent, bien

⋆ Ce travail est financé par ARTIST2, le réseau d'excellence de la conception de systèmes embarqués.

[1] par exemple en utilisant des profils [26]

que le temps joue un rôle important dans ces applications, il est souvent délaissé au niveau spécification du processus de conception et peu connecté au reste du processus de développement de l'application.

D'un point de vue plus théorique, de nombreux formalismes existent pour décrire les comportements temporels et effectuer des vérifications sur ceux-ci : on compte parmi ces formalismes les réseaux de Petri temporels [23,27] et les automates temporisés[4]. Le formalisme des automates temporisés peut être utilisé pour décrire des applications [5] et peut être vérifié par rapport à des propriétés temporelles. Le formalisme des réseaux de Petri temporels peut servir à modéliser des sytèmes avec des informations temporelles [9]. Dans ce article, nous voulons réconcilier les mécanismes formels de temps avec les architectures construites à base de composants. Pour aboutir à cela, nous fusionnons un méta-modèle classique de composants représentant la définition des concepts manipulés par un architecte avec des extensions pour le comportement et les spécifications formelles de temps. Ces extensions ont été choisies pour promouvoir une bonne séparation des préoccupations selon deux axes : spécification/comportement et synchronisation/temps. Les activités de spécification se fondent sur une logique temporelle avec des extensions qualitatives de temps tandis que le comportement sera représenté par des automates temporisés. Afin d'introduire du temps dans les composants classiques, au niveau spécification et comportemental, nous avons défini un procédé pour aider l'architecte à prendre en compte le temps lors de la conception de l'application.

La suite de cet article est organisée comme suit : la section 2 présente une vue du méta-modèle choisi pour décrire nos composants avec un intérêt spécifique pour les parties sur la spécification et le comportement. La section 3 détaille les formalismes de temps utilisés et montre comment les ajouter dans les composants. Enfin, la section 4 présente les travaux connexes et la section 5 conclut et présente les travaux futurs.

2 Analyse et conception

Notre méta-modèle à composants est fondé sur un sous-ensemble du modèle à composants UML 2.0. Ce sous-ensemble comprend les concepts suivants : composants, ports, interfaces et connecteurs. La notation résultante, définie à l'aide d'un méta-modèle, ressemble à celle utilisée par d'autres approches comme par exemple celles du document [22].

Le modèle de conception est organisé en deux parties principales : la partie structurelle décrivant l'architecture de l'application et la partie comportementale décrivant les interactions du composant avec l'environnement.

2.1 Eléments structurels du meta-modèle à composants

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

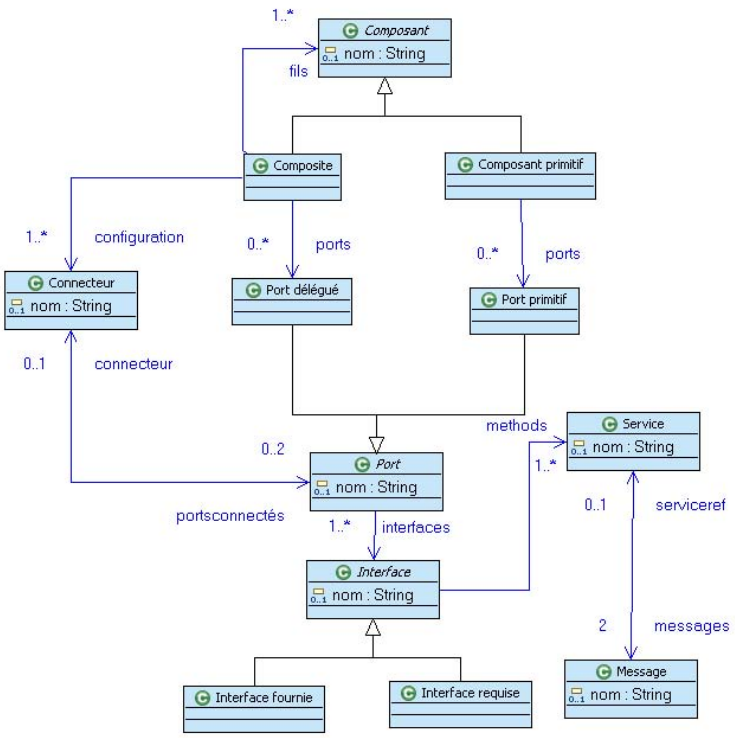

FIG. 1. Partie structurelle du méta-modèle de composant

Dans notre modèle à composants, un composant fournit des *services* et peut requérir des services d'autres composants. Les services sont accessibles par des ports uniquement. Un *port* est un point de connexion sur un composant définissant deux ensembles d'*interfaces* : *fournies* et *requises*.

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

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

requises par l'un sont fournies par l'autre et vice-versa. Les services fournis et requis par un composant fils d'un composant composite sont accessibles grâce à des *ports délégués* qui sont aussi les seuls points d'entrée d'un composant composite. Un port délégué d'un composite est connecté à un et un seul port de composant fils. La partie structurelle est présentée sur la figure 1.

Contrat de composant En UML 2.0, il n'existe pas de sémantique claire pour définir les conditions de compatibilité de deux ports. Il semble raisonnable de décider que dans une architecture, la liaison entre deux ports est correcte si toutes les services fournis par le premier type de port sont requis par l'autre et inversement. Mais la vérification de type, souvent utilisée dans la programmation à objets, ou la correspondance de signature [33] a montré ses limites [24]. La compatibilité entre deux prototypes de services ne peut pas garantir leur bonne utilisation. C'est pourquoi nous enrichissons les propriétés visibles du composant. L'augmentation des capacités d'expression d'un modèle type boîte noire permet de simplifier l'intégration des composants. L'enrichissement des spécifications est souvent utilisé dans les techniques de conception par contrat de développement logiciel assurant des architectures logicielles de haut niveau. Elles garantissent que tous les composants d'un système respectent leurs attentes. Dans notre approche, comme présenté dans [8], nous identifions quatre niveaux de contrats. Le premier niveau, contrats basiques ou syntaxiques, est simplement requis pour que le système fonctionne. Il contient des informations fournies par les prototypes des opérations. Le second niveau, contrats comportementaux, augmente le niveau de confiance dans un contexte séquentiel. C'est un ensemble de contraintes sur les opérations, appelé contrats d'assertion. Ce niveau ajoute des pré et post conditions sur une opération pour garantir un niveau d'utilisation correct. Le troisième niveau traite des contrats de synchronisation, il augmente la confiance dans un contexte distribué ou concurrent. Il spécifie le comportement externe du composant. Le quatrième niveau concerne les contrats de qualité de service (QdS). Il exprime la qualité du service et est la plupart du temps négociable. L'architecte doit être capable de concevoir chaque niveau indépendamment et doit être capable de garantir durant l'étape de conception que l'architecture respecte les contrats des composants.

2.2 Comportement du composant

Avec les définitions d'interface et de service, le composant déclare des éléments structurels sur les services fournis et requis. La spécification de comportement définit l'interaction du composant avec son environnement. Le comportement est décrit par une algèbre de processus pour laquelle nous utilisons le modèle des automates à entrées/sorties [20] pour vérifier le système.

L'algèbre de processus. Pour spécifier le comportement du composant, nous utilisons une algèbre de processus simple, inspirée de FSP [21]. L'algèbre de processus est fondée sur une expression décrivant un ensemble de traces (séquences d'évènements). Appliqué à un composant, un évènement est une abstraction d'un appel de service ou d'une réponse à un appel. Par exemple, un appel de $m1$ sur l'interface $i1$ du port $p1$ est noté par $p1.i1.m1$ et une réponse à ce service $p1.i1.m1\$$. Chaque évènement est

émis par un composant et reçu par un autre. L'appel de $m1$ par l'interface $i1$ du port $p1$ est vu comme l'émission de $!p1.i1.m1$ par le composant $C1$ (noté par un évènement $!C1.p1.i1.m1$). La réception de $p3.i2.m1$ est vue par $C2$ comme $?p3.i2.m1$ (noté par $?C2.p3.i2.m1$ du point de vue de $C2$).

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

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

Definition 1. *(Automate à entrée/sortie) Un automate à entrée/sortie est un n-uplet* (S, L, T, s_0) *avec :*

- S *est un ensemble fini non vide d'états ;*
- L *est un ensemble fini non vide de labels.* $L = I \bigcup O$ *où* I *est un ensemble d'entrées et* O *les sorties et* $I \bigcap O = \emptyset$ *;*
- $T \subseteq S(L \bigcup \{\tau\})S$ *est un ensemble fini de transition où* τ *est une action interne non observable, ;*
- $s_0 \in S$ *est l'état initial.*

Composition d'automates à entrée/sortie La composition de composants dans notre modèle est fondée sur la synchronisation d'une sortie d'un composant avec l'entrée du composant qui lui est connectée.

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

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

2.3 Exemple

La figure 2 illustre notre modèle avec un exemple de composant `AudioPlayer`. Ce composant fournit une interface `IAPoutsound` qui contient deux services `launch` et `sound`. Ce composant est un composite de trois composants `Decoder`, `Extraction` et `Source`. La partie haute de la figure montre la représentation structurelle de ce composant dans notre modèle. La partie basse montre l'automate $A1$ décrivant le comportement du composant `Decoder`.

3 Ajout des propriétés temporelles dans les composants

Quand toutes les propriétés fonctionnelles sont définies pour le composant, l'architecte peut lui ajouter des propriétés extra-fonctionnelles. Ces propriétés sont souvent

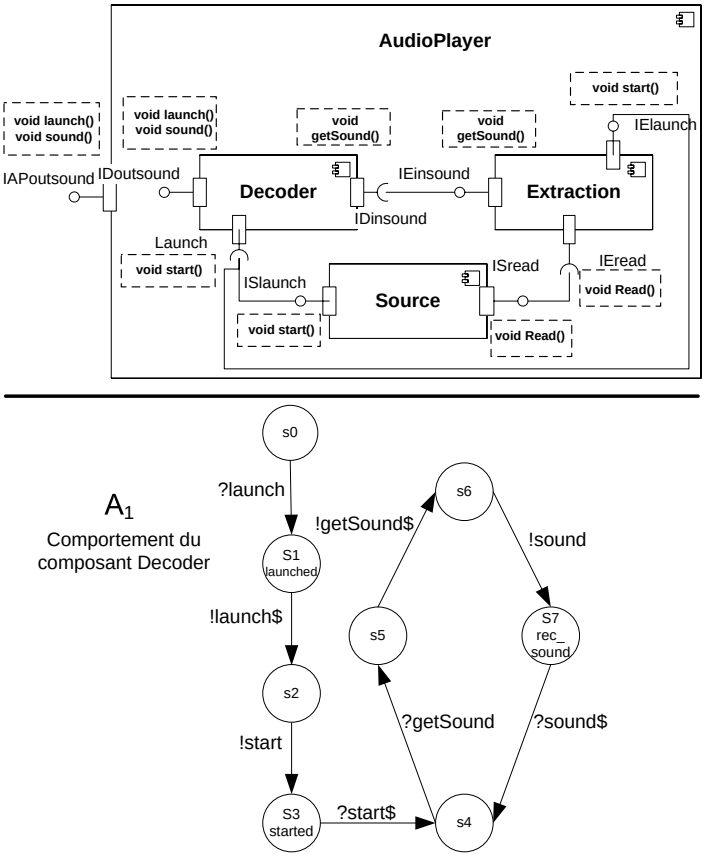

FIG. 2. Exemple : Un composant AudioPlayer

orthogonales aux propriétés fonctionnelles (intersection non vide entre les propriétés) et difficile à ajouter. Les propriétés de temps constituent une catégorie importante des propriétés extra-fonctionnelles. Nous définissons une méthode pour ajouter les propriétés temporelles dans les composants après avoir défini les propriétés fonctionnelles. Nous modifions certaines parties du composant sans pour autant en changer ses fonctionnalités. Les informations temporelles peuvent être ajoutées dans les composants à deux endroits : sur le comportement du composant lui-même et dans les contrats attachés aux interfaces requises. Ces deux emplacements sont les plus adaptés à l'ajout des informations de temps, car ils représentent ce que le composant fournit et requiert et sont utilisés lors de la composition des composants. Pour ajouter du temps dans le comportement du composant, nous utilisons la théorie des automates temporisés [4] et nous définissons un ensemble de motifs pour aider l'architecte à ajouter les informations. Pour les propriétés temporelles dans les contrats, nous utilisons une logique temporelle temporisée (*Timed Computation Tree Logic* [3]). Nous définissons aussi des motifs représentant les structures les plus fréquentes pour aider à l'écriture des contrats temporels.

3.1 Ajout du temps dans le comportement des composants

Pour décrire le comportement temporel, nous avons choisi le formalisme des automates temporisés (AT). Ces AT vont remplacer le comportement originel du composant dans sa description.

Automates temporisés Un automate temporisé est un automate étendu avec des horloges qui sont un ensemble de variables augmentant uniformément avec le temps. Formellement, un automate temporisé est défini comme suit.

Definition 2. *(Automate temporisé) Un automate temporisé est un n-uplet* $A =< S, X, L, T, \iota, P >$ *où :*

- S *est un ensemble fini de localités ;*
- X *est un ensemble fini d'horloges. A chaque horloge, nous assignons une estimation* $v \in V$, $v(x) \in R^+$ *pour tout* $x \in X$ *;*
- L *est un ensemble fini de labels ;*
- T *est un ensemble fini de transitions. Chaque transition* t *est un tuple* $< s, l, \psi, , s' >$ *où* $s, s' \in S$, $l \in L$, $\psi \in \Psi_X$ *est la condition de progression,* Ψ_X *est l'ensemble des prédicats sur* X *définis par* $x \sim c$ *où* $x \in X$ *et* $\sim \in \{<, \leq, =, \geq, >\}$ *et* $c \in N$. *;*
- ι *est l'invariant de* A. $\iota \in \Phi_X$ *où* Φ_X *est l'ensemble des fonctions* $\phi : S \rightarrow \Psi_X$ *liant chaque localité* s *à un prédicat* ψ *;*
- P *associe un ensemble de propositions atomiques à un état.*

Un état d'un automate temporisé est un couple *localité et estimation d'horloges* satisfaisant l'invariant de la localité. Deux types de transition sont possibles entre les états : discrètes sans incrément de temps et avec incrément de temps.

Motifs temporels Pour faciliter l'ajout de temps dans le comportement, nous définissons un ensemble de motifs temporels basés sur ceux définis partiellement dans [14] : temps de réponse, délai, temps d'exécution, période d'appel de service, durée, etc. Nous expliquons ici deux motifs principaux : temps de réponse et temps d'exécution.

Temps de réponse Le motif *temps de réponse* (RT) décrit comment exprimer un temps de réponse avec un automate temporisé. Le temps de réponse est le temps entre l'appel de service et son acquittement. Par exemple, pour exprimer un temps de réponse sur le service $getSound$, une horloge doit être initialisée lors de l'appel à $getSound$ et doit être comparée à une valeur lors de la réception de son acquittement. Ce motif requiert trois paramètres : l'appel de service $service$, l'opérateur de comparaison $\sim \in \{<, \leq, =, \geq, >\}$ et la valeur c. L'automate RT de la Figure 3 représente le motif générique du temps de réponse. Le motif est composé de trois localités, deux transitions et une horloge. Cette dernière est initialisée sur la première transition avec l'appel de service et vérifiée sur la seconde avec l'acquittement de l'appel. La deuxième transition a la propriété $call\_service\_begin$ et la troisième $call\_service\_end$. Ces propriétés seront utilisées lors la vérification des contrats. Quand le motif est ajouté au comportement du composant, les deux transitions ne sont pas obligatoirement consécutives, d'autres

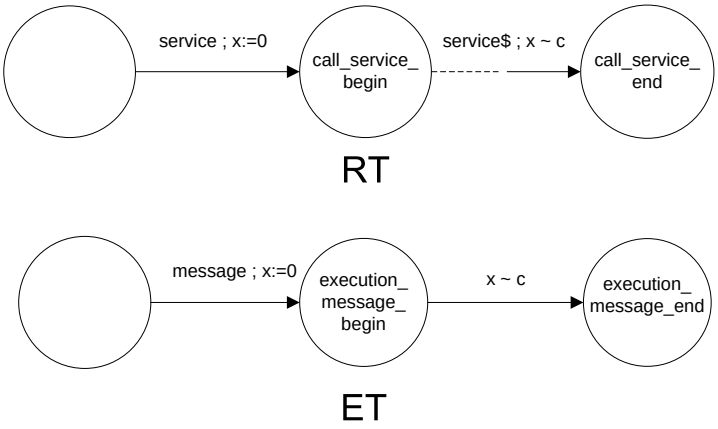

FIG. 3. Motifs temps de réponse et temps d'exécution

transitions peuvent être présentes entre elles. Ceci est représenté sur la partie de la figure RT par la ligne pointillée entre la seconde location et la seconde transition.

Temps d'exécution Le motif *temps d'exécution* (ET) est utilisé pour représenter une exécution avec un automate temporisé. Le temps d'exécution est le temps pris pour effectuer un traitement. Par exemple, après la réception de la réponse à un appel de service, le composant peut avoir besoin d'un temps pour traiter cette réponse. Le motif a trois paramètres : le message $message$, l'opérateur de comparaison $\sim$ et la valeur c. L'automate ET de la Figure 3 représente le motif générique de temps d'exécution. Le motif comprend trois localités, deux transitions et une horloge. L'horloge est initialisée sur la première transition avec le message et comparée sur la seconde sans aucun label. La seconde localité a la propriété $execution\_message\_begin$ et la troisième $execution\_message\_end$. Contrairement au motif temps de réponse, les deux transitions de ce motif doivent être consécutives. En effet, le composant ayant reçu des données ne peut rien faire d'autre. C'est pourquoi la seconde transition et la troisième localité n'existent pas dans le comportement d'origine et seront créées lorsque le motif sera appliqué. Cette façon d'ajouter ce motif n'est pas la seule. Par exemple, le composant peut être capable de recevoir des informations ou d'en envoyer pendant le traitement. Dans ce cas, la garde de la seconde transition est ajoutée à toutes les transitions sortantes de la seconde localité de l'automate d'origine.

Comportement temporel Après avoir défini un ensemble de motifs de temps, nous allons maintenant expliquer comment les ajouter au comportement du composant. L'architecte choisit les différents motifs qu'il souhaite ajouter au composant. Une fois les motifs sélectionnés, ils seront automatiquement intégrés au comportement du composant en transformant l'automate originel en automate temporisé. Les motifs seront combinés successivement avec cet automate pour obtenir le comportement temporisé final. Nous illustrons le processus d'ajout en déroulant pas à pas l'ajout de deux motifs au comportement du composant $decoder$.

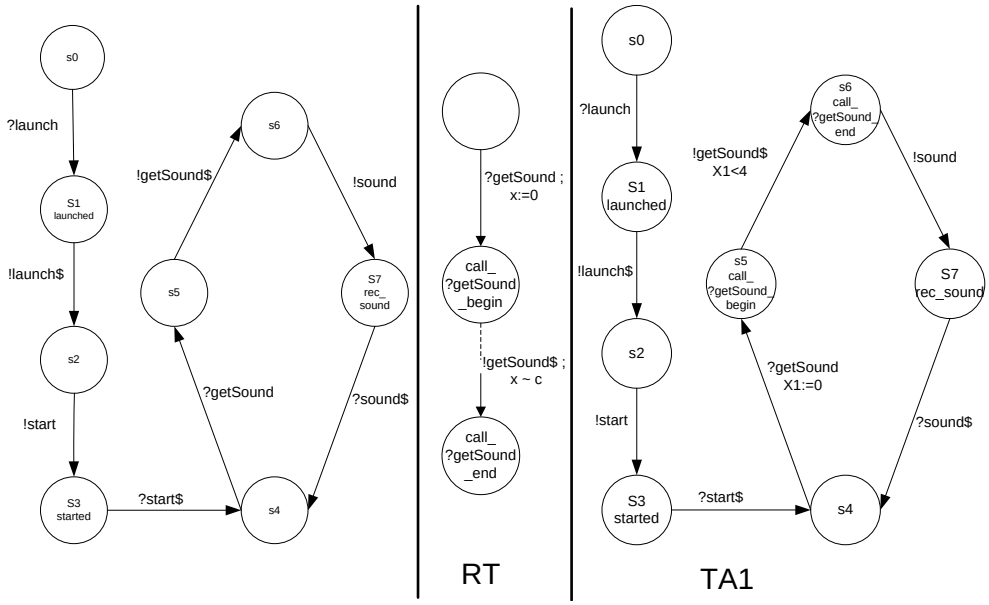

FIG. 4. Ajout du motif temps de réponse

Premièrement, comme illustré Figure 4, le motif temps de réponse est sélectionné avec les paramètres suivants : $getSound$ pour l'appel de service, l'opérateur $<$ et la valeur 4. Une horloge $x1$ est d'abord ajoutée à l'automate. Celle-ci est initialisée sur la transition $?getSound$ partant de la localité $s5$. La propriété $call\_getSound\_begin$ est ajoutée à la localité $s6$ cible de cette transition. La transition portant l'acquittement de l'appel de service est ensuite sélectionnée. La garde $x1 < 4$ est ajoutée à cette transition et la propriété $call\_getSound\_begin$ est ajoutée à la localité cible.

Nous appliquons ensuite le motif temps d'exécution avec les paramètres $!getSound\$$, l'opérateur $<$ et la valeur 2. Une deuxième horloge $x2$ est ajoutée et initialisée sur la transition portant $!getSound\$$. Une nouvelle localité $s6\_exec$ est ajoutée. Les transitions sortantes de $s6$ deviennent les transitions sortantes de $s6\_exec$. Une nouvelle transition entre $s6$ et $s6\_exec$ est ajoutée avec la garde $x2 < 2$. Les propriétés $execution\_!getSound\$\_begin$ et $execution\_!getSound\$\_end$ sont ajoutées respectivement aux localités $s6$ et $s6\_exec$. Le nouvel automate du comportement est montré par l'automate $TA2$ de la figure 5. Ce nouveau comportement du composant ne change pas le comportement d'origine : on peut obtenir $A1$ à partir de $TA2$ en enlevant les horloges puis les transitions sans étiquette.

3.2 Ajout de temps dans les contrats

Le second emplacement où le temps est ajouté est dans les contrats des composants. Les trois premiers niveaux de contrats sont utilisés pour assembler les composants en respectant les propriétés syntaxiques, comportementales et de synchronisation [6]. Nous

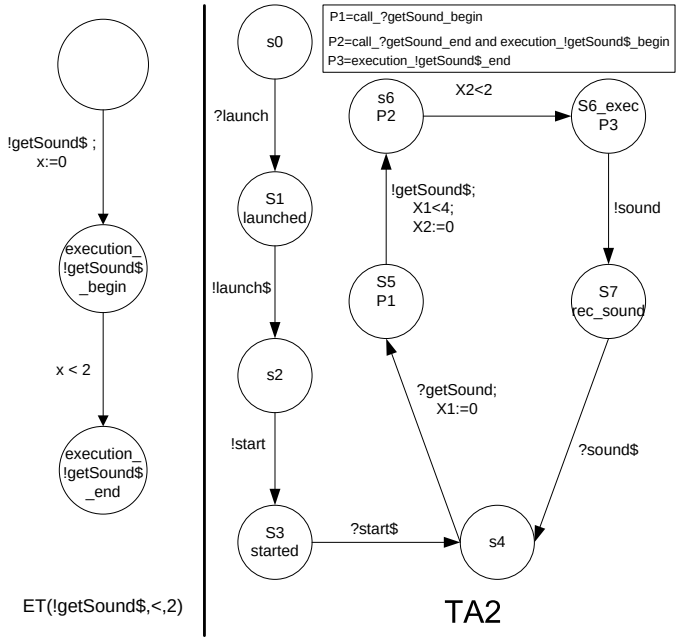

FIG. 5. Ajout du motif temps d'exécution

ajoutons un quatrième niveau de contrats pour la qualité de service de temps. Pour exprimer ces contrats temporels, nous utilisons une logique temporelle temporisée, TCTL. Ces nouveaux contrats seront utilisés lors de la phase de composition des automates temporisés pour valider la compatibilité entre les composants.

TCTL TCTL est une extension de la logique temporelle CTL [13] avec des opérateurs quantitatifs temporels. En CTL, la formule $\exists\Diamond p$ exprime que le prédicat p peut devenir vrai le long de certains chemins d'exécution, mais sans information sur le moment où il devient vrai. L'extension TCTL peut répondre à cette deuxième question : on peut enrichir la formule précédente afin de préciser le moment où p devient vrai. Par exemple la formule $\exists\Diamond_{<5}p$ est vraie le long des chemins d'exécution où p est vraie avant 5 unités de temps.

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

Definition 3. *(Syntaxe) Les formules ψ en TCTL sont définies par :*
$\psi := p|false|\psi_1 \rightarrow \psi_2|\exists\psi_1 U_{\sim c}\psi_2|\forall\psi_1 U_{\sim c}\psi_2$
où $p \in P$, $c \in N$, et $\sim \in \{<, \leq, =, \geq, >\}$.

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

– $\forall\Box_{\sim c}\psi$ pour $\neg\exists\Diamond_{\sim c}\neg\psi$ (toutes les localités le long de toutes les exécutions).

Nous interdisons l'utilisation de plus d'une horloge dans la même expression pour éviter le problème de l'analyse en avant [11].

Contrats temporels A l'instar des contrats des trois premiers niveaux, un contrat de qualité de service est attaché aux interfaces requises. Afin de faciliter la définition des contrats temporels, nous définissons un ensemble de motifs s'inspirant de [18]. Ces motifs de contrats sont des squelettes qui devront être complétés par l'architecte et produiront des formules TCTL. Celui-ci pourra aussi écrire les contrats directement en TCTL. Pour créer un nouveau contrat temporel, l'architecte sélectionne le motif désiré et fournit les valeurs des paramètres. Des exemples de motifs sont :

– temps de réponse de c de l'appel de service foo : $call\_foo\_begin \rightarrow \forall\Box(\forall\Diamond_{\sim c} call\_foo\_end)$
– période de c de la propriété p : $\forall\Box(\forall\Diamond_{\sim c} p)$
– temps de c entre deux propriétés $p1$ et $p2$: $p1 \Rightarrow \forall\Box(\forall\Diamond_{\sim c} p2)$

Certains contrats sont automatiquement créés lorsque des motifs de temps sont ajoutés par l'architecte. Par exemple, lors du choix du motif de temps de réponse sur un appel de service externe, ($?getSound$ par exemple), un contrat est implicitement créé avec ce motif. En effet ce type de propriétés temporelles mettant en jeu des acteurs extérieurs au composant peut se répercuter sur le comportement et sur ce que le composant requiert. La formule TCTL est donc automatiquement créée avec les paramètres du motif.

3.3 Vérifier les propriétés de temps lors de la composition de composants

Pour vérifier les propriétés de temps lors du processus de composition, nous utilisons le *model-checker* Kronos [12] qui permet de vérifier si un automate temporisé satisfait une formule TCTL ou non. Le comportement de chaque composant primitif est décrit par un automate temporisé et les contrats temporels sont exprimés à l'aide de la logique TCTL. Lorsque l'automate temporisé ne satisfait pas la formule, Kronos identifie les localités où la formule n'est pas vérifiée. Par exemple, si les contrats requis par l'environnement sur le composant $decoder$ sont :

– recevoir $sound$ périodiquement toutes les 7 unités de temps : $\forall\Box(\forall\Diamond_{<7} rec\_sound)$
– recevoir $sound$ au plus 5 unités de temps après $launch$: $launched \Rightarrow \forall\Box(\forall\Diamond_{<5} rec\_sound)$

Kronos répond $vrai$ quand la formule est vérifiée sur l'automate temporisé. Sinon, Kronos répond par la négative et fournit les localités précédant celle où rec_sound est vraie.

L'utilisation des motifs et des contrats TCTL présentés dans cette section permet d'intégrer les informations liées au temps dans une modélisation à base de composants. Du fait de l'orthogonalité entre les contrats TCTL par rapport aux autres contrats liés au comportement et grâce aux motifs, l'étape d'intégration est bien isolée des autres tâches de développement de l'application.

4 Travaux connexes

Le premier objectif de notre article est d'intégrer du temps en tant que qualité de service dans un modèle à composants. De nombreux résultats de recherche ont montré l'utilité de langages dédiés pour décrire les architectures logicielles appelés aussi langage de description d'architecture (ADL). Grâce à la précision de la sémantique de ces langages, des suites d'outils ont été développées pour analyser la cohérence des architectures logicielles et pour les prototyper. SOFA [16], par exemple, étend le langage de description d'interface de l'OMG (IDL *Interface Description Language*) afin de spécifier l'architecture des applications à composants. SOFA fournit aussi une algèbre de processus pour spécifier le comportement externe du composant. Mais en utilisant SOFA, l'architecte ne peut pas décrire la QdS fournie et requise des composants.

Le standard AADL [28] est l'un des premiers ADLs fournissant un mécanisme pour spécifier les quatre niveaux de contrat. Mais AADL est une abstraction de bas niveau et est fortement connecté avec l'implantation. De plus, AADL n'est pas encore connecté avec des outils utilisant les informations de QdS pour analyser la cohérence de l'architecture.

Dans le monde UML, de nombreux profils existent pour la conception de systèmes temps-réel : *Scheduling, Performance and Time* (SPT-UML) [26] de l'OMG, MARTE [1]. Ces profils définissent des concepts et une syntaxe pour décrir des systèmes temps-réel mais la sémantique de ces profils reste imprécise. CQML [32] est un langage lexical pour décrire les spécifications de QdS. Il peut être intégré avec UML et utilisé à différents niveaux d'abstraction. Cependant CQML est intégré à peu d'outils et ne peut donc pas être intégré efficacement dans le processus de développement. Le projet OMEGA [2] fournit des méthodes formelles pour vérifier la cohérence de modèles UML 2. L'approche OMEGA propose un profil UML dédié au développement des systèmes temps réel embarqués critiques. La modélisation est basée sur le langage UML étendu à l'aide du mécanisme de stéreotype. Ce profil définit un raffinement formel du profil SPT-UML de l'OMG. Ce profil associé à la boîte à outils IFx permet de simuler et de valider formellement des propriétés dynamiques de modèles UML. Pour le moment, ce profil permet uniquement de spécifier l'architecture et ne fournit pas d'outils pour générer du code pour une partie de l'implantation.

UPPAAL [19] est un environnement d'outils intégrés pour modéliser, valider et vérifier des systèmes temps-réel modélisés par des réseaux d'automates temporisés. UPPAAL permet d'évaluer des formules CTL sur les automates mais pas TCTL et ne peut donc pas évaluer le quatrième niveau de contrats. Bip [7] est un cadre logiciel pour modéliser des composants hétérogènes. Les composants Bip peuvent être étendus avec des horloges mais le modèle de temps est discret et simulé.

Les travaux de [31] proposent l'ajout d'interfaces temporisées pour les composants. Les interfaces temporisées définissent des propriétés temporelles que l'on souhaite intégrer aux composants. Contrairement à notre approche, dans leurs travaux les automates des interfaces ne sont pas directement ajoutés au comportement du composant. La cohérence entre les automates temporisés définis au niveau des interfaces des composants et les automates traduisant le comportement des composants doit être vérifiée pour garantir que la propriété de temps désirée est compatible avec le composant.

Notre second objectif est d'avoir une bonne séparation des préoccupations afin de pouvoir ajouter la qualité de service de temps sans interférer sur les fonctionnalités des composants. Dans [17], Klein et al. proposent un opérateur de composition (tissage) fondé sur la sémantique des scénarios. Ce travail utilise les *Message Sequence Charts* (MSC) comme langage de scénarios, mais les MSC et les automates à E/S utilisés pour spécifier le comportement sont des langages identiques du point de vue de l'opérateur de tissage. Cependant, l'opérateur de tissage peut aider l'architecte à intégrer la spécification de l'aspect comportemental mais il n'est pas compatible avec les automates temporisés et l'intégration de la QdS dans la spécification du composant. Notre approche peut être vue comme un premier travail pour la composition de temps. Nous ne fournissons pas actuellement de langage de point de coupe définissant la façon dont nous intégrons nos motifs.

5 Conclusion et perspectives

La séparation des préoccupations permet une conception plus facile, augmente la testabilité et la maintenabilité de l'application. La séparation des préoccupations est souvent utilisée pour placer dans différentes unités des préoccupations techniques comme la sécurité, la persistance ou la traçabilité. Cet article considère le temps comme une préoccupation et propose des méthodes pour aider l'architecte à intégrer des informations de qualité de service temporelle pendant la spécification et la conception d'application à composants. L'approche met en avant des motifs de conception pour le comportement et la définition de contrats. L'introduction de ces motifs dans les composants se fait automatiquement, l'architecte n'ayant qu'à paramétrer les motifs choisis.

Le travail présenté dans cet article fait partie d'une approche globale qui a pour objectif la réduction de l'écart entre le modèle de spécification et l'implémentation [30]. L'approche propose un processus unifié pour la conception et l'implémentation des composants. Elle peut aider les architectes dans la gestion du temps dans leur application, en fournissant un ensemble d'outils vérifiant la cohérence des composants tout au long du développement de l'application. La partie spécification abordée dans cet article permet d'obtenir une description formelle de l'application. L'ajout d'un motif est vu comme une opération du méta-modèle des automates temporisés implanté en Kermeta [25]. La partie fonctionnelle de cette description est ensuite fournie à un programmeur tandis que la partie qualité de service temporelle est transmise à un logiciel moniteur afin de vérifier l'implémentation à l'exécution [29]. Ce moniteur est construit automatiquement par une transformation de modèles écrites en Kermeta. A partir d'automates temporisés nous construisons automatiquement une application Giotto [15] (cadre logiciel offrant une abstraction de temps).

Nous travaillons actuellement à la mise en œuvre de ces motifs comme un aspect au niveau modèle (au sens de la conception par aspects). Le but est de décrire un nouvel élément au niveau modèle pour réutiliser des modèles de qualité de service. De plus, nous travaillons à la définition d'un langage de coupe pour simplifier l'intégration d'un même modèle de qualité de service dans plusieurs composants de l'architecture. Cela permettra de construire une couche de qualité de service pouvant être composée avec d'autres aspects de l'architecture.

Références

1. MARTE UML profile RFP. voted at OMG. http ://www.omg.org/cgi-bin/doc ?realtime/2005-02-06.
2. Webpage of the OMEGA IST project. http ://www-omega.imag.fr/.
3. R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1) :2–34, 1993.
4. R. Alur and D.L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2) :183–235, 1994.
5. T. Amnell, E. Fersman, P. Pettersson, W. Yi, and H. Sun. Code synthesis for timed automata. *Nordic J. of Computing*, 9(4) :269–300, 2002.
6. O. Barais. *Construire et Maîtriser l'Evolution d'une Architecture à base de Composants*. PhD thesis, LIFL, INRIA Futurs, Lille,France, 2005.
7. A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in bip. In *SEFM '06 : Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.
8. A. Beugnard, J-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7) :38–45, 1999.
9. A. Bobbio. System modelling with petri nets. In *Colombo and A. Saiz de Bustamante, editors, Systems Reliability Assessment*, 1990.
10. G. Bollella and J. Gosling. The real-time specification for java. *Computer*, 33(6) :47–54, 2000.
11. P. Bouyer. Untameable timed automata ! In *STACS '03 : Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science*, pages 620–631, London, UK, 2003. Springer-Verlag.
12. M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos : A model-checking tool for real-time systems. In *Proc. 1998 Computer-Aided Verification, CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.
13. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, 1986.
14. S. Graf and I. Ober. A real-time profile for UML and how to adapt it to SDL. In *SDL Forum 2003, July 1-4, Stuttgart*, volume 2708 of *LNCS*, July 2003.
15. T.A. Henzinger, C.M. Kirsch, and B. Horowitz. Giotto : A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1) :84–99, January 2003.
16. T. Kalibera and P. Tuma. Distributed component system based on architecture description : The sofa experience. In *On the Move to Meaningful Internet Systems - DOA, CoopIS and ODBASE*, pages 981–994, London, UK, October 2002. Springer-Verlag. *ISBN : 3-540-00106-9*.
17. J. Klein, L. Hélouet, and J.M. Jézéquel. Semantic-based weaving of scenarios. In *proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, Bonn, Germany, March 2006. ACM.
18. S. Konrad and B. H.C.Cheng. Real-time specification patterns. In *ICSE27*, pages 372–381, May 2005.

19. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2) :134–152, Oct 1997.
20. N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3) :219–246, 1989.
21. J. Magee. Behavioral analysis of software architectures using ltsa. In *Proceedings of the 21st international conference on Software engineering*, pages 634–637. IEEE Computer Society Press, 1999. *ISBN : 1-58113-074-0*.
22. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. In *IEEE Transactions on Software Engineering*, volume 26, page 23, January 2000.
23. P. M. Merlin. *A study of the recoverability of computing systems.* PhD thesis, Department of Information and Computer Science, University of California, Irvine, CA, 1974.
24. B. Meyer. Applying "design by contract". *Computer*, 25(10) :40–51, 1992.
25. P-A. Muller, F. Fleurey, and J-M. Jézéquel. Weaving executability into object-oriented meta-languages. In L. C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2005.
26. OMG. *UML Profile for Schedulability, Performance, and Time Specification*, January 2005. Version 1.1.
27. C. Ramchandani. *Analysis of asynchronous concurrent systems by timed petri nets*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.
28. As-2 Embedded Computing Systems Committee SAE. Architecture Analysis & Design Language (AADL). SAE Standards n^o AS5506, November 2004.
29. S. Saudrais, O. Barais, and L. Duchien. Using model-driven engineering to generate qos monitors from a formal specification. *edoc-workshop*, page 45, 2006.
30. S Saudrais, O. Barais, and N. Plouzeau. Composants avec propriétés temporelles. In M. C. Oussalah, F. Oquendo, D. Tamzalit, and T. Khammaci, editors, *CAL*, pages 143–149. Hermes Science, 2006.
31. B. Schätz. Interface descriptions for embedded components. In *3rd Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER'05)*, Paderborn, 2005.
32. J. Øyvind Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, Department for Informatics, University of Oslo, June 2001.
33. A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4) :333–369, 1997.