Vérification formelle des modèles UML temps-réel

A. Bouamari, M. Mostefai
Laboratoire d'Automatique de Sétif
a.bouamari@arn.dz, mostefai@univ-setif.dz

Résumé- Le model-checking est une technique de vérification formelle très puissante mais ses mécanismes de spécification déployés sont souvent étroits et difficiles à manipuler. Les modèles utilisés sont très complexes et nécessitent généralement des compétences spécialisées dans les énoncés formels. Pour universaliser l'utilisation du langage de modélisation unifié dans le développement des systèmes temps-réel, une technique de vérification des propriétés temporelles sur des statecharts UML est proposée dans cet article. Le système est modélisé par des diagrammes état-transition et les propriétés à vérifier sont exprimées en langage naturel selon des patterns de spécification temps-réel. Les statecharts sont ensuite convertis en automates temporisés et les expressions de propriétés en formules TCTL équivalentes. Ces structures sont analysées par le model-checker Kronos et les résultats d'analyse sont retournés sous forme de notes sur les diagrammes UML sources.

I. INTRODUCTION

Les systèmes temps-réel sont souvent complexes et critiques, et nécessitent un développement rigoureux pour affirmer leur correction fonctionnelle et temporelle [1]. Les techniques dites du model-checking constituent dans ce contexte un très important outil permettant la vérification formelle et automatique d'une large classe de systèmes. En fait, les méthodes de spécification formelles permettent la représentation des systèmes avec précision et donc disposer des procédures permettant une vérification automatique par model-checker des propriétés requises. Cependant, les modèles issus des énoncés formels déployés sont souvent alambiquées et parfois même inexploitables si le nombre d'états est relativement important.

D'autre part, les spécifications qui offrent des mécanismes de modélisation avancés (hiérarchie, historique, synchronisation, etc.) notamment les statecharts (Harel [9] et UML 2.0 [19]) manquent souvent de fondements formels nécessaires pour accomplir une vérification exhaustive et complète de leurs modèles.

Dans la littérature, différentes techniques ont été proposées pour adapter les statecharts UML au contexte de vérification par model-checking des systèmes temps-réel [4]. Ces approches utilisent des restrictions sur l'utilisation de ces diagrammes (seul un sous-ensemble des éléments de modélisation des statecharts est utilisé). Le modèle est ensuite transformé en une configuration de modèles abstraits analysable directement par le model-checker. Au cours de cette translation, tous les détails qui ne sont pas supportés par le model-checker utilisé sont éliminés (hiérarchie…) tout en maintenant la même sémantique opérationnelle du modèle.

D'autres techniques de model-checking des modèles UML temps-réel focalisent plutôt sur l'expression des propriétés à vérifier et utilisent une extension temporelle du langage d'expression des contraintes OCL ou un sous ensemble structuré du langage naturel (l'anglais par exemple) afin de simplifier la spécification des propriétés à vérifier pour des systèmes état-transition.

Dans ce contexte, et pour combiner les avantages du langage de modélisation unifié et des model-checkers, nous présentons une approche permettant la spécification et la vérification automatique des modèles UML temps-réel. Le système et ses propriétés sont complètement spécifiés par des statecharts UML annotés avec des expressions du langage naturel. Les modèles sont ensuite convertis de manière transparente en structures accessibles directement par le model-checker.

II. MODEL-CHECKING

Le model-checking [3] repose sur la modélisation formelle du système par système de transitions étiquetées (automates temporisés, RdP, etc.) et la spécification des propriétés à vérifier par des formules logique (logique temporelle par exemple). L'algorithme de model-checking combine alors le modèle et la formule pour calculer l'ensemble des états accessibles. La propriété est vérifiée s'il y au moins un chemin qui relie l'état initial à l'ensemble des états qui vérifient cette formule. Les model-checkers les plus utilisés dans le contexte temps-réel sont Uppaal [15] et Kronos [20]. Les deux outils utilisent des structures d'automates temporisés pour la modélisation du système et la logique TCTL pour l'expression des propriétés.

Uppaal est très efficace au niveau modélisation et dispose d'un simulateur et d'une interface graphique très performante. Mais l'outil ne permet malheureusement que des formes de synchronisation binaire et ne vérifie qu'une classe restreinte de propriétés d'atteignabilité.

Par contre, Kronos admet la vérification d'une large classe de propriétés (atteignabilité, sûreté et vivacité) sur des modèles temporisés et combine plusieurs techniques d'analyses (*forward analysis, backward analysis* et *on-the-fly*). L'outil dispose également d'un algorithme de model-checking symbolique ce qui permet de résoudre partiellement le problème d'explosion du nombre d'états [10]. Kronos est cependant destiné aux utilisateurs spécialistes dans les énoncés

formels et n'a pas d'interface graphique ou de module de simulation. Ces sérieuses limites peuvent être contournées si on utilise une autre forme de modélisation combinée avec ce model-checker. C'est la raison pour laquelle notre choix s'est porté sur cet outil de vérification.

Pour accomplir une vérification par Kronos, le système est décrit par un réseau d'automates temporisés et la propriété à vérifier est exprimée en logique temporelle TCTL.

A. Automates temporisés

Les automates temporisés ont été introduits par Alur et Dill [2] afin de permettre la vérification des propriétés de sûreté sur des systèmes de transitions avec des annotations temporelles. Ce mécanisme permet de décrire le comportement de nombreux systèmes temporisés. De plus, cette classe de modèles possède de bonnes propriétés de décidabilité. Le modèle se compose d'un ensemble d'états de base reliés par des transitions. Ces dernières sont annotées avec des expressions de garde et d'affectation. Une transition est produite dans une configuration si la garde est évaluée à *vrai* et si la configuration cible est atteignable (sans violation de son invariant).

Le modèle comprend également un ensemble de variables entières partagées qui peuvent être utilisées dans les expressions de garde ou d'affectation. Les horloges sont des variables à valeurs réelles qui évoluent d'une façon synchrone avec le temps (horloge de référence) et qui peuvent être mises à jour sur les transitions. Pour Kronos, un système est spécifié par un ensemble de fichiers chacun décrivant le comportement d'un composant. Le comportement global du système est définit par l'automate produit (produit synchronisé).

B. TCTL

Le model-checker Kronos est utilisé dans le contexte des systèmes temps-réel. Il utilise TCTL (*Timed Computation Tree Logic*) comme logique temporelle pour l'expression des propriétés à vérifier où des informations quantitatives sur l'écoulement du temps sont associées aux combinateurs temporels habituels de CTL [17]. La logique utilise comme expressions primitives les noms des états, les variables et les horloges du système modélisé. Les syntaxe et sémantique complètes de cette logique sont décrites dans [10].

III. UML

Le langage de modélisation unifié est doté d'un pouvoir expressif et notationnel très riche et convenable pour la plupart des applications informatiques. Le langage propose différents types de diagrammes permettant ainsi la modélisation des aspects statique, dynamique et architectural du système logiciel. Cependant, le langage n'est pas exempt de limitations dont les plus importantes concernent l'expression des caractéristiques permettant la vérification rigoureuse du système produit [14].

L'une des réprimandes majeures du langage de modélisation unifié et particulièrement dans le contexte temps-réel, est sans aucun doute l'absence d'une base formelle robuste. Le problème inhérent issu de cette limite est que certains aspects ne peuvent être capturés d'une façon standard ce qui empêche une analyse formelle des modèles produits. Les mécanismes d'extension d'UML (stéréotypes, valeurs marquées, contraintes et profils) peuvent servir dans ce contexte pour raffiner la sémantique standard d'UML et permettre ainsi l'ajout de nouveaux éléments de modélisation qui seront utilisés dans la création de modèles UML spécifiques.

IV. APPROCHE

Il est clairement distingué que l'utilisation d'un model-checker (particulièrement Kronos) nécessite une pratique démontrée des énoncés en logique formelle et des modèles d'automates temporisés. La technique proposée convoite la vérification formelle des propriétés temps-réel sur modèles UML. Le système est modélisé par statecharts et les propriétés à vérifier par des expressions structurées du langage naturel. Les structures hiérarchiques des statecharts sont aplaties et compilées en automates temporisés et les expressions TCTL sont automatiquement dérivées à partir des expressions du langage naturelle correspondantes selon un système de patterns prédéfinis. La procédure de vérification est alors entièrement automatique et les résultats d'analyse peuvent être utilisés avec les traces de conversion précédentes pour reconstituer les diagrammes UML sources annotés avec des remarques d'analyse.

A. Spécification

Cette étape est cruciale pour la pertinence des résultats d'une vérification par model-checking (*spécifier pour vérifier*) et il est fondamental de maintenir une équivalence optimale entre le modèle utilisé et la réalité. Le système est modélisé par une configuration de statecharts et les propriétés à vérifier sont spécifiées par des expressions du langage naturel. Cette forme de spécification est très appropriée pour les concepteurs du fait de la commodité d'utilisation de diagrammes UML et de leur expressivité contrairement aux langages formels qui sont plutôt ardus et nécessitent des compétences spécialisées souvent rares.

A.1 statecharts

Les statecharts présentent un outil très adéquat pour la modélisation de la dynamique des systèmes réactifs avec leurs concepts d'abstraction hiérarchique, synchronisation, concurrence, etc. [19].

Les statecharts ou machines d'états modélisent des séquences d'états par lesquels passe un objet ou une interaction en réaction à des événements qu'ils peuvent recevoir, ainsi que leurs effets réceptifs. Une machine d'états spécifie le comportement des instances de l'élément source détenteur

(*owner*). Cet élément est un classeur qui peut être une classe, un cas d'utilisation, une collaboration ou une méthode. Les machines d'états peuvent être également structurées pour modéliser des situations plus complexes (accès concurrents, etc.).

Le principe de base d'exécution d'une machine d'états est qu'elle traite un seul événement à la fois et finit de traiter toutes ses conséquences avant d'en traiter un autre (*run-to-completion*). Les actions sont instantanées et les événements ne sont jamais simultanés. Les activités liées aux transitions sont exécutées grâce à la sémantique *run-to-completion*. Si un objet reconnaît un événement alors qu'il est en train d'exécuter une étape *run-to-completion*, l'occurrence de l'événement est placée dans un pool d'événement jusqu'à l'achèvement de cette étape.

L'occurrence d'un événement dans une machine d'états, peut engendrer le déclenchement d'une transition. Quatre types d'événements déclencheurs des transitions peuvent être distingués : événement d'appel (*call event*), de changement (*change event*), de signal (*signal event*) et temporel (*Time Event*).

Les principales déficiences affirmées quant à la modélisation des systèmes temps-réel par statecharts concernent les points de variation sémantique. Cette forme de modélisation concède différentes interprétations sémantiques des modèles de statecharts. Ceci est n'a aucun effet si ces diagrammes sont utilisés pour simple documentation. Cependant, et dans le cas d'une vérification formelle, il est nécessaire de se rendre compte de l'interprétation sémantique exacte du contexte d'implémentation [6].

Dans UML 2.0 [19], de nouveaux éléments de modélisation du temps simple ont été ajoutés. Des contraintes temps-réel peuvent être désormais exprimées sur des diagrammes UML (Statecharts, Séquences …) explicitement et d'une manière standard et significative. Donc, et contrairement aux travaux précédents, les statecharts UML permettent l'expression des contraintes temporelles de base et aucune extension temporisée ne sera proposée dans ce contexte. Cependant quelques restrictions doivent être imposées sur cette utilisation compte tenu des limitations d'expression présentes sur les model-checkers :

- les gardes doivent avoir une forme simple proche de celle utilisée par Kronos ;
- les opérations de transformations de données complexes sur transitions ne sont pas permises.

Pour simuler le comportement des statecharts par automates temporisés, il serait fondamental de tenir compte des points suivant :

- ✓ la communication par événements dans un statechart est asynchrone. Celle d'un automate temporisé est synchrone mais un automate temporisé qui implémente un pool d'événements peut simuler une communication par ces derniers. Dans notre contexte, le système est considéré comme plus rapide que son environnement et finira de traiter les conséquences d'un événement avant l'émission de l'événement suivant.
- ✓ L'interaction d'un système de statecharts n'est pas assez précise dans le standard de l'OMG. Elle dépend du domaine d'application et du niveau d'abstraction du modèle. Une vue proche de l'implémentation doit prendre en considération le matériel et le système d'exploitation de support.

A.2. Pattern de spécification temps-réel

Du coté spécification, l'utilisation du model-checker Kronos nécessite pareillement une maîtrise démontrée des énoncés en logique temporelle. La démarche propose de ce fait des patterns de spécification temps-réel permettant ainsi d'assister l'expression des propriétés à vérifier. Les patterns proposés utilisent le langage naturel pour permettre une expression simplifiée et délibérée des difficultés de TCTL. La structure template des patterns de spécification proposés est comme suit :

- **Name.** Identification du pattern.
- **Classification.** catégorie du pattern
- **Intent.** description du pattern.
- **Natural Language Specification.** Spécification langage naturel.
- **Mappings.** Spécification concrète du pattern avec :
  - TCTL,
  - Syntaxe Kronos.
- **Special cases.** Cas particuliers du pattern
- **Examples and Known Uses.** Exemples connus du pattern.
- **Relationships.** relations avec autres patterns.

Fig. 1. Template du pattern de spécification temps-réel.

Natural Language Specification. Pour mieux simplifier la spécification, une grammaire structurée (du langage naturel) peut aisément servir pour l'expression des propriétés à vérifier sur des modèles UML. La grammaire est principalement basée sur la classification des patterns temps-réel et leurs étendues.

Patterns de spécification. L'approche utilise des patterns pour présenter, codifier et réutiliser des spécifications de propriétés pour model-checking. Nous utiliserons la notion de patterns de spécifications proposée par [7] comme éléments de base pour la spécification des propriétés. Des translations directes de ces patterns en CTL existent et nous proposons donc une spécialisation de ces derniers pour prendre en charge l'aspect quantitatif du temps. Rappelons qu'à ce stade, on s'intéresse uniquement à l'expression des propriétés à vérifier et non pas aux annotations temporelles sur le modèle. Ces dernières sont complètement traitées par des modèles de statecharts.

Chaque pattern possède deux portées, une temporelle (*temporal scope*) et une autre temps-réel (*real time scope*). Ces

portées présentent les étendues qualitative et quantitative d'exécution du modèle sur lesquelles le pattern est valable.

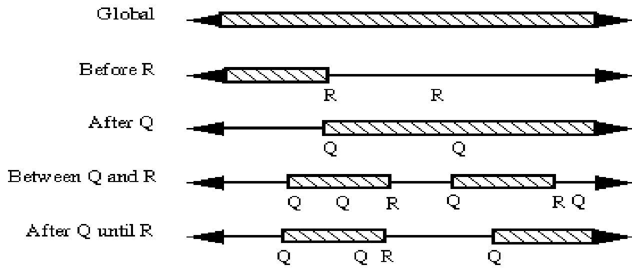

Fig. 2. Temporal Scopes [7].

La figure 3 montre les principaux patterns retenus dans l'extension temps-réel et qui ont été initialement introduits par [7] dans le cadre d'une vérification qualitative.

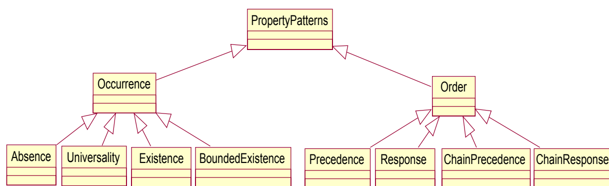

Fig. 3. Patterns de spécification de propriétés [7].

Universality. Un événement/état donné doit avoir lieu dans tout le scope (le scope entièrement).
Absence. Un événement/état donné ne peut pas avoir lieu.
Existence. Un événement/état donné doit avoir lieu.
Bounded existence. Un événement/état donné doit avoir lieu k fois dans le scope. Une variante de ce pattern spécifie la plus petite (et la plus grande) occurrence k.
Precedence. Un événement/état P doit être toujours précédé par un événement/état Q dans le scope.
Response. Un événement/état P doit être toujours suivi par un événement/état Q dans le scope.
Chain of Precedence. Une séquence d'événements/états P1,…, Pn doit être toujours précédée par une séquence d'événements\états Q1,…, Qm. Ce pattern est une généralisation du *Precedence pattern*.
Chain of Response. Une séquence d'événements/états P1,…, Pn doit être toujours suivie par une séquence d'événements/états Q1,…, Qm. Ce pattern est une généralisation du *Response pattern*.

Les scopes temps-réel peuvent être :

1. Time Interval : peut être after, before ou exactly.
2. Time Duration : min, max ou immediately.
3. Periodic : cas particulier du *response pattern* (P est toujours suivi par P).

Le premier scope est applicable sur tous les patterns. Le deuxième est utilisable uniquement avec les patterns présence et absence.

Une structure générale de la formule est associée à chaque triplet *TemporalScope-Pattern-RealTimeScope* et peut être instanciée par simple substitution des paramètres spécifiques par les expressions relatives au modèle. La figure 4 présente le pattern de réponse temps-réel.

RT response pattern specification

Classification
It is a category of order patterns.
Intent
This pattern describes the relation of priority between couples of states\events S and P, if the first state\event S occurs, then the other state\event P eventually holds.....

Natural Language Specification
TemporalScope, if P holds then S eventually holds, RealTimeScope.

Mappings

- ***TCTL « S responds P »***
  - Globally $\forall\Box(S \Rightarrow \forall\Diamond_{\sim k}(P))$
    - Max interval
    - min interval
    - exactly
    - ...
  - Before R ....
  - After Q ....
  - Between Q and R ....
  - After Q until R ....
- ***Syntax Kronos***
  - Globally ab (Sa →ad~k (P))
    - Max interval
    - min interval
    - exactly
    - ...
  - Before R ....
  - After Q ....
  - ....

Special Cases
Periodic actions are special cases of response pattern.

Examples and Known Uses
This pattern is used in real time specifications
....

Relationships
This pattern is a specialization of chain response pattern
....

Fig.4. Real time response pattern.

B. Transformation des modèles

L'algorithme d'aplatissement parcourt la structure du statechart à la recherche des états composites et qui peuvent être de type *BASIC* (simple), *AND* (concurrent) ou *XOR* (séquentiel). L'état est converti selon son type [5] :

1. *états de base* :
   a. chaque état du statechart est converti en une *location* de l'automate cible ;
   b. les invariants des états sont établis en fonction des conditions sur les transitions sortantes : l'invariant doit être inférieur ou égal à la borne supérieure (de l'horloge) spécifiée dans les transitions sortantes. Les états sans transitions

temporisées doivent avoir la condition d'invariance *TRUE* ;

c. les gardes des transitions sans contraintes temporelles doivent être *TRUE* ;

d. les gardes des transitions temporisées avec intervalle [min, max] doivent avoir la forme : *clock ≥ min and clock ≤ max* (ou *clock=max* si *max =min*) ;

e. les événements générés par une transition et envoyés aux autres objets doivent être déclarés comme événements de synchronisation dans l'automate associé à l'objet destinataire. Les objets destinataires sont identifiés par le diagramme d'objets correspondant ;

2. *états composites séquentiels* : un statechart contenant un ou plusieurs états composites séquentiels est translaté en un seul automate temporisé ayant une *location* pour chaque état ou sous état direct. Il est nécessaire de définir une variable horloge spécifique pour chaque état composite pour modéliser les contraintes de temps sur les transitions entre ses sous états.
3. *états composites concurrents* : la translation nécessite un automate temporisé pour chacune des régions orthogonales de l'état original. Cet ensemble d'automates doit être activé en même temps : ceci est obtenu par un événement de synchronisation (*start*). Si un état concurrent est abandonné, les automates correspondants doivent être suspendus (placés dans un état d'attente). Ceci est réalisé par un événement de synchronisation *stop*. La translation de chaque région orthogonale est réalisée selon les règles précédentes avec deux exceptions vu le besoin de synchroniser les automates des sous états quand un état composite est activé :
   a. un état initial d'attente est ajouté à chaque automate dérivé d'un sous état parallèle. Cet état est abandonné si l'événement *start* est reçu.
   b. pour chaque état des sous automates est ajoutée une transition sortante avec l'événement *stop* comme événement déclencheur. La destination de cette transition est l'état d'attente initial.
4. Dans Kronos, chaque composant est représenté par un automate. Tandis qu'un statechart UML décrit le comportement de toutes les instances d'une classe. Pour déterminer *le nombre d'automates* à générer, on doit tenir compte des diagrammes d'objets UML : un automate est associé à chaque instance de la classe.
5. *conditions de garde* : en général les conditions de garde ne peuvent pas être modélisées par des automates temporisés. Cependant des transitions contrôlées par gardes avec des variables booléennes peuvent être converties dans la notation de TA en divisant l'état source de la transition en plusieurs sous états. Un pour chaque valeur supportée par la variable booléenne. Ce genre de translation est viable seulement pour les variables booléennes ou variables à nombre limité de valeurs supportées sinon la tille de l'automate résultant peut être très importante.
6. translation des instances multiples : UML permet d'avoir des événements paramétrés *Send(i)* et *i* indique l'instance originaire de l'événement. Pour simuler ça sous Kronos, il faut dupliquer l'automate (un pour chaque instance) aussi bien que les événements. La procédure de translation est la suivante :
   a. pour chaque événement paramétrique un ensemble d'événements est défini correspondant aux instances des classes indiquées par le diagramme d'objets ;
   b. les événements de synchronisation sont associés aux objets correspondants.
   c. Attention aux objets qui interagissent avec plusieurs instances de la même classe.

Du côté propriétés, les expressions TCTL sont directement dérivées à partir de patterns utilisés et instanciées par substitution des paramètres par les valeurs correspondantes du modèle de l'application. Ces formules sont ensuite aplaties de façon à correspondre toujours aux éléments du modèle d'automates résultat. Chaque conversion d'un état composite en sous-états implique la conversion des expressions TCTL qui utilisent le nom de l'état composite comme argument. Les formules TCTL résultantes doivent être conformes au modèle d'automates final.

C. Processus de traitement de modèles

La figure 5 présente le processus général envisagé pour la vérification des modèles UML et identifie les fonctionnalités et informations de base nécessaires à ce processus.

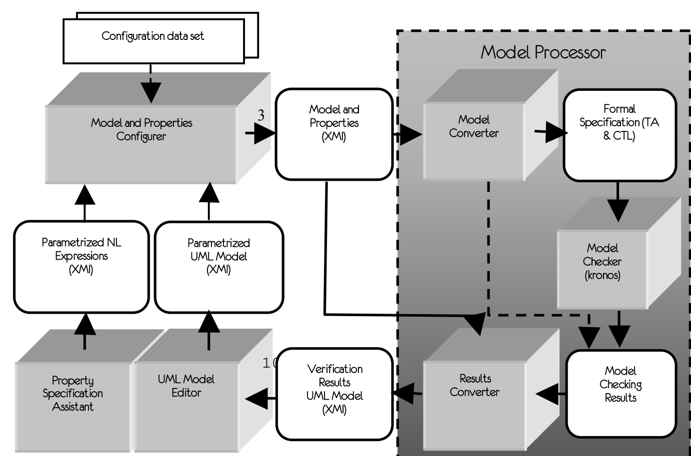

Fig. 5. *Processus de vérification formelle des modèles UML.*

Le processus est une instanciation du framework de traitement de modèles proposé dans le profil UML pour la spécification de l'ordonnançabilité, de la performance et du temps [18] adopté par L'OMG. Le processus se base cependant sur des modèles de statecharts et utilise des model-checkers comme outils d'analyse.

La fonction *Model Editor* est chargée de la création et modification des modèles UML. Les modèles produits peuvent être paramétriques ce qui permet l'analyse du même modèle pour différentes valeurs des mêmes paramètres sans avoir à reproduire un nouveau modèle pour chaque nouvelle valeur. Cependant, il faudra faire appel à la fonction *Model Configurer* chargée de reproduction de modèle différent par la substitution d'un ensemble approprié de valeurs de paramètres. Les variables paramétriques peuvent être spécifiques au modèle de statecharts comme (valeurs d'horloges par exemple), des propriétés à vérifier et les instructions de vérification (type opération Kronos…).

L'objectif de la fonction *Property Specification Assistant* est d'assister l'analyste à la dérivation et l'instanciation des propriétés à vérifier.

La fonction *Model Processor* est chargée d'analyser le modèle UML et de générer les résultats d'analyse correspondants qui seront retournés dans un format accessible par l'outil d'édition. Cette fonction est décomposée en trois unités de fonctionnalité plus fine : *Model Converter*, *Model Checker* et *Results Converter*.

La fonction primaire du *Model Converter* est de convertir le modèle UML et les propriétés à vérifier en modèles spécifiques au domaine d'analyse. Les fichiers comportant les modèles résultats prendront les extensions *.tg* pour les graphes d'automates et *.tctl* pour les propriétés à vérifier. Un troisième fichier texte doit être également généré pour les séquences d'instructions Kronos. Le *Model Converter* est également responsable de détecter toutes les inconsistances des annotations utilisées. Telles inconsistances doivent être renvoyées au *Model Editor* via *Results Converter*. Le résultat de l'extraction (modèle du domaine, propriétés et instructions Kronos) est ensuite utilisé par la fonction *Model Checker*.

Le *Model Checker* constitue le noyau de l'analyse. Deux genres de résultats sont produits par le modèle : des validations (propriété vérifiée) sous formes de notes (XML ou HTML) ou des contre-exemples dans le cas d'échec de validation. Les résultats du model-checking sont spécifiques à l'outil; la raison pour laquelle une autre fonction de conversion (*Results Converter)* doit être utilisée.

L'objectif de la fonction *Results Converter* est de reconstruire les modèles UML *sources* annotés avec les résultats de l'analyse avant d'être retournés au *Model Editor* pour éventuelles revues et inspections.

V. ASSISTANT DE SPECIFICATION DE PROPRIETES

L'objectif de la fonction *Property Specification Assistant* est d'assister à la dérivation et l'instanciation des propriétés à vérifier et consiste à :

Etape1. Spécification de la propriété à vérifier :

Chaque fois qu'une pièce est introduite, alors il y a réponse de la machine dans un temps max de 10 s

Etape2. Instanciation de la propriété :
Pattern : Order/Response ;
Temporal scope : Globally, et
Real Time scope : TimeInterval/Maximum.

Globally, if P holds then S eventually holds within the interval of k units of time.

Etape2. Dérivation de la propriété :

$$\forall\Box\left(S \Rightarrow \forall \Diamond_{\sim k} (P)\right)$$

Etape3. Instanciation de l'expression TCTL correspondante :

$$\forall\Box\left(\text{Pièce} \Rightarrow \forall \Diamond_{\leq 10} (\text{Réponse})\right)$$

Etape4. Les propriétés du domaine sont réécrites selon la grammaire résultante de notre classification de patterns :

$$\forall\Box\left(\text{Pièce} \Rightarrow \forall \Diamond_{\leq 10} (\text{café} \vee \text{thé} \vee \text{rendre})\right)$$

VI. TRAVAUX CONNEXES

L'utilisation des statecharts dans le contexte du model-checking temps-réel a été sujet de plusieurs travaux de recherche ces dernières années. De nouvelles variantes temporisées ont été proposées comme dans le cas de *Timed statecharts* [11]. Des tentatives d'extension, formalisation et translation des statecharts UML ont été également proposées depuis l'adoption du standard de modélisation objet [5], [6], [16]…etc.

Du coté spécification de propriétés, [8] a proposé une formalisation du langage OCL pour la prise en charge des aspects temporelle. Les auteurs de [12] proposent un sous ensemble structuré du langage naturel pour la dérivation et l'instanciation des propriétés à vérifier. Une extension temps-réel des patterns de spécification de Dwyer [7] a été également proposée dans [13] où une nouvelle classification (quantitative) de patterns est proposée.

Notre travail propose une combinaison des techniques de modélisation par statecharts et une extension temps-réel des patterns de spécification et comprend la modélisation et l'aplatissement des structures hiérarchiques des statecharts pour et assiste à la dérivation et l'instanciation des formules TCTL.

VII. CONCLUSION ET PERSPECTIVES

L'utilisation des méthodes formelles dans le contexte du développement d'un système temps-réel permet d'exprimer ce système avec précision et donc disposer d'un support considérable d'outils d'analyse et de vérification. Dans ce contexte, nous avons proposé une technique permettant la spécification et la vérification formelle des propriétés temporelles sur des modèles UML. Au début, des statecharts UML annotés avec des expressions du langage naturel sont utilisés pour la spécification de la dynamique du système et de ses propriétés. Ces modèles sont automatiquement convertis en structures équivalentes d'automates temporisés et d'expressions TCTL. Ces structures sont analysées par le model-checker Kronos et les résultats de vérification sont retournés dans un format accessible par l'outil de modélisation.

Le travail constitue un exorde et nécessite plusieurs améliorations au niveau des structures déployées. Il peut être également complété et étendu afin de supporter les autres concepts de modélisation hiérarchiques avancées tels que concurrence interne, connecteurs historiques, actions, synchronisation …etc.

REFERENCES

[1] R. Alur & T.A. Henzinger. "Logics and models of real time: a survey". *In Real Time: Theory in Practice, number 600 in Lecture Notes in Computer Science, Springer-Verlag*, pp. 74–106,1992.

[2] R. Alur and D.L. Dill, "A theory of timed automata". *Theoretical Computer Science*, vol. 126, pp. 183-235, 1994.

[3] E.M. Clarke, O. Grumberg & D.A. Peled, *Model Checking*. The MIT Press, 1999.

[4] A. David, and M.O.Möller, "From HUPPAAL to UPPAAL: A Translation from Hierarchical Timed Automata to Flat Timed Automata". *Research Series RS-01-11 BRICS*, Department of Computer Science, University of Aarhus, March 2001.

[5] V. Del Bianco, L. Lavazza, M. Mauri. "Model Checking UML Specifications of Real Time Software". *Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'02)*, p. 203. 2002.

[6] K. Diethers, U. Goltz, and M. Huhn. "Model Checking UML statecharts with Time". *Proc. Workshop on Critical Systems Development with UML UML '02,* J.-M. Jézéquel, H. Hußmann, and S. Cook, eds., 2002.

[7] Matthew B. Dwyer, George S. Avrunin and James C. Corbett. "Patterns in Property Specifications for Finite-state Verification". *Proceedings of the 21st international conference on Software Engineering*. Los Angeles, California, United States. pp: 411 – 420. 1999.

[8] S. Flake, and W. Mueller. "A UML Profile for Real-Time Constraints with the OCL". *Proc. Int'l Conf. Unified Modelling Language,* J.-M. Jézéquel, H. Hußmann, S. Cook, eds., 2002.

[9] D. Harel, "statecharts: A visual formalism for complex systems". *Science of computer programming*, 8: pp. 231-274, 1987.

[10] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. "Symbolic model checking for real-time systems, Information and Computation". 111(2): pp. 193–244, 1994.

[11] Y. Kesten, and A. Pnueli. "Timed and Hybrid statecharts and their Textual Representation". In J. Vytopil, editor, Formal Techniques in Real-Time and Fault-Tolerant Systems, volume 571 of Lecture Notes in Computer Science, pages 591-619. Springer-Verlag, 1992.

[12] S. Konrad and B. H.C. Cheng. "Automated Analysis of Natural Language Properties for UML Models". In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM*, number 3844 in Lecture Notes in Computer Science, pages 48-57. Springer Verlag, 2006.

[13] S. Konrad and B.H.C. Cheng. "Real-time Specification Patterns". In Proceedings of the 27th International Conference on Software Engineering (ICSE05), St Louis, MO, USA, May 2005.

[14] K.C. Lano & A. Evans. "Rigorous development in UML". *ETAPS '99*, LNCS. 1999.

[15] 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, October 1997.

[16] T. Schäfer, A. Knapp & S. Merz. "Model checking UML state machines and collaborations". *Electronic Notes in Theoretical Computer Science* 47, pp. 1-13, 2001.

[17] P. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussine, A. Petit. *Vérification de logiciels : techniques et outils du model-checking*. Vuibert, Paris. 1999.

[18] Object Management Group. UML Profile for Schedulability, Performance, and Time Specification, January 2005, V.1.1, formal/05-01-02.

[19] Object Management Group. Unified Modelling Language: UML 2.0 Superstructure Specification, august 2005, V. 2.0, formal/05-07-04.

[20] S. Yovine, "Kronos: A verification tool for real time systems", *Int. Journal of Software Tools for Technology Transfer*, 1: pp. 123-133, October 1997.