République Tunisienne
Ministère de l'Enseignement Supérieur,
et de la Recherche Scientifique

Université de Sfax
École Nationale d'Ingénieurs de Sfax

Ecole Doctorale
Sciences et Technologies

Mémoire de *MASTERE*

Nouvelles Technologies des
Systèmes Informatiques Dédiés

N° d'ordre: 49-11

MEMOIRE

Présenté à

L'École Nationale d'Ingénieurs de Sfax

en vue de l'obtention du

MASTERE

Nouvelles Technologies des Systèmes Informatiques Dédiés

Par

Amal GASSARA

Vérification des propriétés non fonctionnelles des RTES distribués dynamiquement reconfigurables

Soutenu le 29 Juillet 2011, devant le jury composé de :

M.	**Mohamed ABID**	*Président*
M.	**Adel MAHFOUDHI**	*Membre*
M.	**Mohamed JMAIEL**	*Encadreur*
M.	**Bechir ZALILA**	*Invité*
Mme.	**Fatma KRICHEN**	*Invitée*

À mes parents,
À mes frères,
À tous ceux que je n'ai pas cités
et qui ne me sont pas moins chers.

Remerciements

C'est avec un grand plaisir que je réserve ces lignes en signe de gratitude et de reconnaissance à tous ceux qui ont contribué à l'élaboration de ce travail.

Je tiens tout d'abord à remercier mes encadreurs M. Mohamed JMAIEL, professeur à l'Ecole Nationale d'Ingénieurs de Sfax (ENIS), M. Bechir ZALILA, assistant à l'ENIS et Mme. Fatma KRICHEN, assistante à l'ENIS, pour la confiance qu'ils m'ont accordé en acceptant de diriger mes travaux de mastère. C'est en fait grâce à leur aide inestimable, aux conseils précieux qu'ils n'ont jamais cessé de me donner que ce manuscrit a pu voir le jour. Je vous serais toujours reconnaissante.

Je tiens à exprimer ma profonde gratitude à M. Mohamed ABID, maître de conférences à l'Ecole Nationale d'Ingénieurs de Sfax pour l'honneur qu'il m'a fait en acceptant de présider le jury de mon mastère.

Mes remerciements s'addressent aussi à M. Adel MAHFOUDHI, maître assistant à la Faculté des Sciences de Sfax, d'avoir accepté de juger mes travaux de mastère.

Plusieurs personnes m'ont aidé énormément pendant ce stage de mastère, je tiens alors à les remercier tout particulièrement M. Slim Kallel, assistant à la la Faculté des Sciences Économiques et de Gestion de Sfax.

J'exprime finalement mes vifs remerciements aux membres de l'unité de Recherche en Développement et Contrôle d'Applications Distribuées pour l'ambiance amicale que nous avons partagée.

Table des matières

Table des figures

Table des listings

Liste des tableaux

Introduction Générale

Les systèmes embarqués sont devenus d'usage plus courant dans notre vie quotidienne. En effet, ils connaissent un essor considérable et ils envahissent les différents domaines d'application. Un système embarqué est constitué de deux parties, matérielle et logicielle, qui sont conçues conjointement pour répondre à des fonctionnalités spécifiques. Le plus souvent ces fonctionnalités sont contraintes par le temps. Par ailleurs, on parle toujours de systèmes embarqués temps réel critiques, car l'échec de leur mission a potentiellement un impact majeur sur la vie humaine, l'état de l'environnement, etc.

Ainsi, développer des systèmes embarqués temps-réel (*RTES* [1]) distribués constitue de nos jours une tâche très difficile. Cette difficulté est due à la complexité et la criticité de ces systèmes et aux contraintes non fonctionnelles qu'ils doivent respecter.

Par ailleurs, un système embarqué ne doit plus se contenter d'offrir des fonctionnalités prédéfinies, mais il doit s'adapter aussi aux changements de son environnement et répondre aux nouvelles exigences des utilisateurs. Cependant, cette adaptation augmente la complexité des *RTES* distribués. En effet, elle peut provoquer le mal fonctionnement du système par la production des anomalies et la perturbation de certaines propriétés non fonctionnelles.

Ainsi, garantir le bon fonctionnement du système et respecter ces contraintes non-fonctionnelles sont devenus aujourd'hui des défis du développement logiciel. Donc, il est nécessaire d'adopter de nouvelles méthodes permettant la vérification le plus tôt possible dans le cycle de vie du système. Ces méthodes doivent également dépasser les limites des techniques de validation traditionnelles.

Dans ce contexte, plusieurs travaux de recherche ont été réalisés dans le but de vérifier certaines propriétés non fonctionnelles comme les propriétés temporelles et de ressources. Les approches proposées [18, 26, 11] ne traitent que des systèmes statiques. Elles ne sont pas destinées à des RTES distribués dynamiquement reconfi-

1. RTES : Real Time Embedded System

gurables puisqu'elles ne permettent pas de vérifier le maintien de ces propriétés après une action de reconfiguration. Par ailleurs, il n'existe pas de travaux qui vérifient à la fois des propriétés temporelles et des propriétés de ressources.

Dans ce travail, nous visons à vérifier certaines propriétés non-fonctionnelles pour les RTES distribués. Nous optons à garantir la validité de ces propriétés même après l'application d'une action de reconfiguration. Les propriétés à vérifier sont des propriétés temporelles (le respect des échéances des tâches, l'absence d'interblocage et l'absence de famine) et des propriétés de ressources (la consommation CPU, la consommation mémoire et la consommation de la bande passante).

Sur le plan pratique, nous réalisons deux plugins Eclipse : le premier présente un framework de modélisation offrant un éditeur graphique pour modéliser des RTES distribués dynamiquement reconfigurables et le deuxième présente un framework de vérification pour assurer le respect des propriétés non fonctionnelles. Ce framework de vérification combine différents formalismes. Nous utilisons l'algorithme d'ordonnancement RMS [25] et le simulateur Cheddar [34] pour vérifier le respect des échéances et la consommation CPU. Nous proposons deux algorithmes pour assurer la vérification de la consommation mémoire et la consommation de la bande passante. Concernant l'absence d'interblocage et l'absence de famine, nous garantissons que ces deux propriétés sont correctes par construction. Pour cela, nous définissons un ensemble de contraintes, inspiré du profil Ravenscar [7], qui doit être respecté dès la construction du système.

Pour valider notre approche, nous considérons une étude de cas de gestion de charge de travail. Nous présentons une modélisation de cette application avec le profil RCA4RTES (Reconfigurable Computing Architectures for Real Time Embedded Systems) en utilisant notre éditeur graphique. Puis, les modèles obtenus seront les entrées à notre framework de vérification pour vérifier certaines propriétés non fonctionnelles.

Ce rapport s'articule autour de cinq chapitres. Dans le premier chapitre, nous présentons l'état de l'art. Le deuxième chapitre est consacré à présenter le travail de thèse dans lequel s'inscrit notre travail et le framework de modélisation. Le troisième chapitre décrit notre approche proposée pour la vérification des propriétés non fonctionnelles en présentant les différents formalismes et outils utilisés. La mise en oeuvre de cette approche est présentée dans le chapitre 4. Dans le chapitre 5, nous présentons l'étude de cas utilisée pour valider notre approche. Enfin, ce rapport est clôturé par une conclusion et des perspectives de notre travail.

Chapitre 1

Etat de l'art

1.1 Introduction

Actuellement, l'évolution de la complexité des systèmes embarqués conduit à augmenter la probabilité de bugs engendrant des défaillances graves du système. Pour palier ce problème, il est nécessaire d'adopter des méthodes de vérification efficaces permettant la livraison des systèmes valides et corrects par rapport à des exigences temporelles et de ressources matérielles.

Dans ce chapitre, nous mettons notre travail dans son contexte. Nous commençons tout d'abord par la présentation des systèmes auxquels nous nous sommes intéressés. Ensuite, nous présentons les formalismes proposés dans la littérature pour la vérification des propriétés non fonctionnelles pour ces systèmes, suivis d'une étude sur les travaux effectués dans le contexte de notre recherche. Finalement, nous clôturons par une synthèse comparative et une conclusion.

1.2 Concepts de base

1.2.1 Les systèmes embarqués

Un système embarqué [22] présente une intégration entre deux parties, matérielle et logicielle, qui sont conçues conjointement afin de répondre à des fonctionnalités dans un domaine spécifique.

Les systèmes embarqués sont présents dans des applications coûteuses comme les applications spatiales, aéronautiques, militaires... où ils prennent une importance considérable. Ces systèmes apparaissent aussi dans des domaines grand public tels que les appareils électroniques (les téléphones portables, les PDA, les appareils

photos...) et dans l'assistance à la conduite automobile (freinage assisté, direction assistée...).

Le système embarqué communique avec son environnement en prenant certaines décisions ou effectuant certains calculs dans des délais de temps bien déterminés. Ceci implique des exigences temporelles plus ou moins fortes tout dépend du domaine d'application visé. Les systèmes embarqués sont souvent des systèmes temps-réel.

1.2.2 Les systèmes temps réel

Un système temps réel est un système informatique qui doit satisfaire les deux contraintes suivantes :

- *Exactitude logique :* calculer les bonnes sorties du système en fonction de ses entrées.
- *Exactitude temporelle :* les résultats de calcul sont présentés au moment opportun (un calcul correct mais hors délai est considéré faux). Autrement, un retard est considéré comme une erreur qui peut engendrer de graves conséquences.

Dans ces systèmes embarqués temps réel, les propriétés non-fonctionnelles ont une très grande importance comme les propriétés fonctionnelles.

1.2.3 Les contraintes des systèmes embarqués temps réel

Les systèmes embarqués temps réel évoluent rapidement et doivent répondre aux exigences requises par les clients. Ces exigences ont un aspect fonctionnel et un aspect non fonctionnel.

- **Les contraintes fonctionnelles :** L'aspect fonctionnel est explicitement décrit dans le cahier de charges. Les contraintes fonctionnelles formalisent les besoins de l'utilisateur. En fait, ils représentent la raison de la création du système. Donc, ce dernier doit offrir des services et des fonctionnalités qui satisfont les exigences de l'utilisateur.
- **Les contraintes non fonctionnelles :** Les contraintes non fonctionnelles sont des qualités, des caractéristiques et des contraintes globales qu'un système ou un sous système doit satisfaire. Ces contraintes sont généralement relatives au temps, aux ressources à caractères limités dont dispose le système, à la sécurité, etc.

  Dans les systèmes embarqués temps réel, il est crucial de prendre en compte ces contraintes très tôt dans le processus du développement. Les erreurs dues à l'omission de ces contraintes sont souvent critiques et difficiles à corriger. Ces

erreurs peuvent mener à des retards et donc à des augmentations indésirables du budget.

1.2.4 Les systèmes distribués

Un système distribué [16] est un ensemble d'entités autonomes de calcul (ordinateurs, PDA, processeurs, processus, etc) inter-connectées par un système de communication. Les différents éléments du système ne fonctionnent pas indépendamment mais collaborent à une ou plusieurs tâches communes.

Les besoins des systèmes distribués sont :

- Intégration d'applications existantes initialement séparées,
- Partage de ressources (programmes, données, services),
- Besoin de communication et de partage d'information,
- Réalisation de systèmes à grande capacité d'évolution.

1.2.5 Les systèmes dynamiquement reconfigurables

Un système est dynamiquement reconfigurable [22] lorsqu'il peut modifier son comportement ou son architecture au cours de l'exécution. Les reconfigurations se produisent selon l'évolution des exigences du contexte d'utilisation et la variation des contraintes de l'environnement d'exécution.

Les architectures logicielles statiques restent stables et constantes dès leur installation. Cependant, les architectures logicielles dynamiques se modifient au cours de l'exécution du système. Dans notre travail, nous nous intéressons aux systèmes à architecture logicielle à base de composants. Donc, dans ce qui suit, nous définissons les architectures logicielles à base de composants ainsi que les architectures logicielles dynamiques.

Architectures logicielles à base de composants

Une architecture à base de composants [5] est une spécification abstraite du système qui contient un ensemble de composants interconnectés entre eux par des connexions.

- **Les composants [5] :** sont définis comme des unités de composition qui décrivent des fonctions spécifiques et possèdent des interfaces requises et des interfaces fournies qui leur permettent de se communiquer avec l'environnement. Ils peuvent être déployés indépendamment et composés avec d'autres

composants.

- **Les connecteurs** [5] **:** représentent les interactions entre les composants comme les appels de procédure, les évènements, etc. Cependant, les connecteurs peuvent aussi représenter des interactions plus complexes, telles qu'un protocole client/serveur ou un lien SQL entre une base de donnée et une application.

L'ensemble de composants connectés entre eux par des connecteurs constituent une configuration du système.

Architectures logicielles dynamiques

Les architectures dynamiques correspondent à des applications possédant des composants qui peuvent être créés, interconnectés et supprimés en cours d'exécution. Nous distinguons différents types de reconfigurations pour concevoir des architectures logicielles dynamiques :

- **Les reconfigurations comportementales** : ce sont les modifications d'un comportement d'une architecture logicielle comme par exemple le changement des propriétés des composants.
- **Les reconfigurations structurelles** [30] : ce type de reconfiguration constitue un ensemble d'opérations agissant sur les composants et les connections entre eux pour modifier la topologie de l'application. Comme opérations structurelles de base, on trouve l'ajout ou la suppression d'un composant, l'ajout ou la suppression d'une connexion, le remplacement d'un composant par un autre.

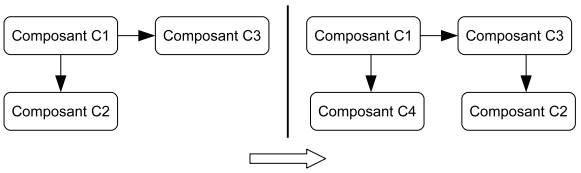

Figure 1.1 – Exemple de reconfigurations structurelles

La figure 1.1 montre un exemple de reconfigurations structurelles. Ces reconfigurations engendrent la suppression de la connexion entre le Composant C1 et le Composant C2 et l'ajout du composant C4 et des deux connexions entre C1 et C4 et entre C3 et C2.

– **Les reconfigurations géographiques** [30] : ce type de reconfiguration agit sur l'emplacement physique d'un composant pour modifier la distribution géographique d'une application. Comme reconfiguration géographiques, on trouve la migration d'un composant d'un noeud à un autre et le déploiement d'un nouveau composant dans un noeud.
La reconfiguration géographique est utilisée pour effectuer la répartition de charge et l'adaptation aux changements des ressources de communication disponibles, etc.

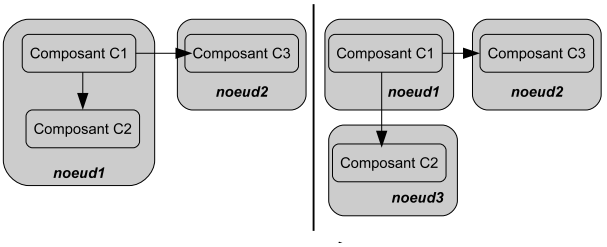

Figure 1.2 – Exemple de reconfiguration géographique

La figure 1.2 montre un exemple d'une reconfiguration géographique qui présente la migration du composant C2 d'un noeud (noeud1) vers un autre (noeud3).

1.3 Étude de l'existant

Pour valider des systèmes informatiques, on peut utiliser des techniques traditionnelles comme le test.

Le test [31] est le processus d'exécuter un programme pour vérifier qu'il satisfait des exigences spécifiques ou identifier la différence entre les résultats obtenus et attendus. Le test trouve les cas où le programme ne respecte pas sa spécification. Alors, il peut être utilisé pour montrer la présence des bugs mais, jamais pour montrer leur absence. De plus, les bugs trouvés tard dans des phases du prototypage du processus de développement des logiciels ont un impact négatif sur le temps de disponibilité sur le marché. Donc cette technique traditionnelle n'est pas suffisante pour garantir la fiabilité des systèmes critiques.

Il existe un autre concept qui consiste à prouver que le système fonctionne correctement. Cette méthode est appelée la vérification formelle. Celle-ci propose de

raisonner rigoureusement, à l'aide d'une logique mathématique, sur des algorithmes afin de démontrer leur validité par rapport à une certaine spécification. Parmi les avantages de la vérification formelle, nous citons :

- L'exploration exhaustive de tous les comportements,
- L'utilisation au plus tôt dans le processus de conception,
- L'automatisation,

Dans ce contexte, nous avons fait une étude concernant les travaux existants qui ont comme objectif la vérification formelle de certaines propriétés non fonctionnelles des systèmes embarqués temps réel. Nous commençons par présenter les différents formalismes utilisés.

1.3.1 Les formalismes

Dans les systèmes embarqués temps réel, il est nécessaire de s'assurer du respect des propriétés non fonctionnelles car ce sont généralement des systèmes critiques. La vérification de ces propriétés est effectuée en se basant sur des formalismes. Ces formalismes doivent être à caractère temporel puisque nous nous sommes intéressés à des systèmes temps réel. Comme exemple de ces formalismes, nous citons les algorithmes d'ordonnancement [34, 15], les automates temporisés [26, 38] ou des extensions des automates temporisés [3, 18], les réseaux de Petri temporisés [9] et le langage TASM [29, 28].

Les automates temporisés

Le modèle des automates temporisés [1] a été bien étudié et plusieurs outils de vérification ont été développés pour ces modèles. Les automates temporisés permettent d'introduire la notion de temps dans la représentation du système.

Définition : Un automate temporisé est un système de transitions disposant d'horloges, qui évoluent continûment, toutes à la même vitesse, permettant de mesurer le temps écoulé depuis leurs dernières initialisations. Un état de l'automate peut comporter une condition sur les horloges, appelée *invariant*, qui doit être satisfaite pendant toute la durée passée dans cet état. Une transition de l'automate est étiquetée par :

- une *garde*, qui exprime une condition sur les valeurs des variables. Cette condition doit généralement être compatible avec l'invariant de l'état origine de la transition et elle doit être satisfaite pour franchir la transition.
- une remise à zéro de certaines horloges (*réinitialisation*).

La figure 1.3 montre l'automate temporisé associé à une minuteur. Après 3 unités de temps sans appui sur le bouton, la lumière s'éteint automatiquement.

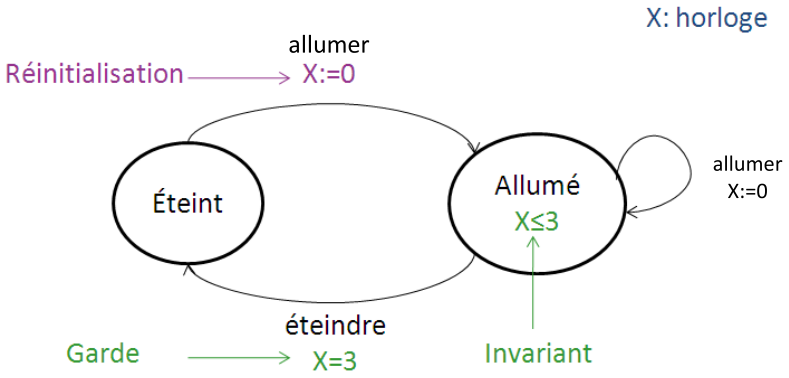

Figure 1.3 – Exemple d'automate temporisé

Il existe des extensions des automates temporisés comme HTA [3]. Les HTAs sont des extensions temps réel des diagrammes d'états UML permettant la modélisation des contraintes temporelles. Ces diagrammes sont étendus avec des horloges, des invariants et des conditions sur les transitions. Ce formalisme est très influencé par les automates temporisés sauf qu'il permet de modéliser des locations (états) hiérarchiques.

Discussion :

Pour s'assurer qu'une spécification du système représentée en automates temporisés satisfait certaines propriétés, il faut spécifier aussi ces propriétés en utilisant des logiques temporelles comme TCTL [36]. Seules les propriétés temporelles peuvent être définies avec ces logiques. Ainsi en utilisant les automates temporisés, nous ne pouvons pas vérifier des propriétés de ressources.

De plus, nous pouvons utiliser les automates temporisés pour réaliser des analyses d'ordonnancement [26, 38]. Cette technique est implantée dans l'outil TIMES [4]. Ainsi, ils permettent de vérifier le respect des échéances de l'ensemble de tâches d'un système.

La vérification des propriétés en utilisant les automates temporisés est effectuée par la technique de vérification de modèles. Grâce à l'existence de plusieurs implantations de l'outil de vérificateur de modèles [37] basées sur ce formalisme, les automates temporisés sont très utilisés pour la vérification des systèmes embarqués.

Le point faible des automates temporisés est le problème d'explosion d'état. Lorsque le nombre de processus du système est très élevé, on aura une explosion du

nombre d'états, ce qui empêche la vérification du modèle par exploration de tous les états du système.

Le point faible des HTAs est l'absence d'outil pour vérifier directement une modélisation en HTA. Il est nécessaire de faire un aplatissement des HTAs en des automates temporisés pour être l'entrées à un vérificateur de modèles.

Les réseaux de Petri temporisés

Pres+ (*Petri net based Represntation for Embedded System*) [9] est un modèle de réseau de Petri étendu pour capturer les caractéristiques des systèmes embarqués temps réel. Il capte explicitement des informations temporelles, permettant la représentation des systèmes aux différents niveaux de granularité et améliore l'expressivité en permettant aux jetons de transmettre des informations. On peut associer aux transitions des conditions sur les valeurs des jetons et des délais minimaux et maximaux.

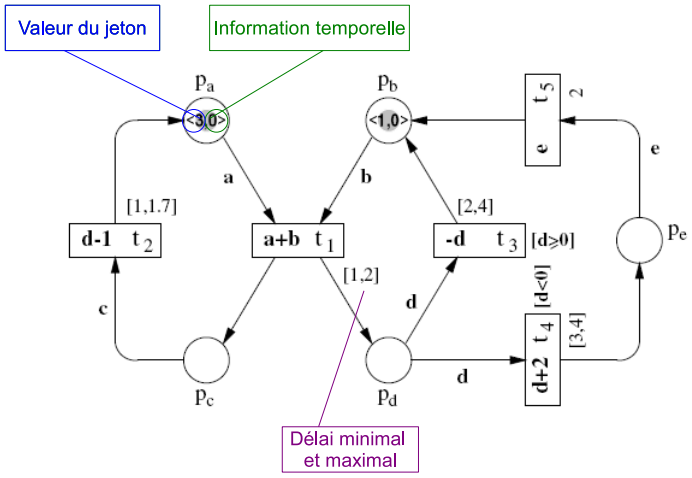

Figure 1.4 – Exemple de modèle en Pres+

Discussion :

Comme les automates temporisés, Pres+ permet de représenter les contraintes temporelles des systèmes embarqués, mais il ne permet pas la représentation des ressources matérielles pour pouvoir respecter leurs contraintes plus tard.

L'utilisation de Pres+ est souvent restreinte à cause de l'inexistence d'outil de vérification basé sur ce formalisme. L'auteur de [9] a recours à une traduction des

modèles Pres+ en automates temporisés pour utiliser le vérificateur de modèles UPPAAL [2] désigné aux automates temporisés.

L'ordonnancement

Un algorithme d'ordonnancement permet de fixer à chaque instant la tâche à exécuter par le processeur quand on a plusieurs tâches qui veulent en accéder. Le choix de cette tâche est assuré par une politique d'ordonnancement. Pour les politiques d'ordonnancement classiques [25] comme RMS, DMS, EDF et LLF, il existe des méthodes d'analyse permettant de s'assurer que la politique choisie permet de respecter les échéances des tâches.

On distingue deux types d'ordonnancement temps réel : l'ordonnancement par priorité statique et l'ordinnancement par priorité dynamique. Nous présentons ainsi un aperçu sur les algorithmes les plus utilisés.

- **Ordonnancement par priorité statique** :
  Comme exemple de ce type d'ordonnancement, nous trouvons les algorithmes RMS et DMS.
  Avec RMS (Rate Monotic Scheduling) [25], les priorités des tâches sont affectées selon leurs périodes. La tâche ayant la période la plus petite est la tâche la plus prioritaire. Avec RMS, l'ordonnancement d'un ensemble de tâches est garanti si le taux d'utilisation du processeur est inférieur ou égale à $n(2^{1/n}-1)$, où n est le nombre de tâches périodiques. C'est une condition suffisante. Les tâches peuvent parfois être ordonnancées même si la condition suffisante n'est pas satisfaite.
  Avec DMS (Deadline Monotic Scheduling) [24], les priorités des tâches sont affectées selon leurs échéances. La tâche ayant l'échéance la plus petite est la tâche la plus prioritaire. Sa condition d'ordonnançabilité est la même que RMS.
- **Ordonnancement par priorité dynamique** :
  Pour ce type d'ordonnancement, il existe les algorithmes EDF et LLF :
  Avec EDF (Earliest Deadline First) [25], les priorités des tâches sont affectées selon leurs échéances. La tâche périodique ayant l'échéance la plus proche est la tâche la plus prioritaire. Cette priorité change au cours du temps. Un ensemble de tâches est ordonnançable avec EDF que si le taux d'utilisation du processeur est inférieur ou égale à 1.
  Avec LLF (Least Laxity First) [20], les priorités des tâches sont affectées selon

leurs laxités. La tâche ayant la laxité la plus petite est celle la plus prioritaire. La laxité est donnée par la formule ci-dessous :

$$Laxité = Échéance - TempsDeCalculRestant - TempsCourant$$

La condition d'ordonnançabilité de LLF est la même que EDF.

Discussion : Le tableau 1.1 illustre une comparaison entre les algorithmes présentés. EDF et LLF sont des algorithmes d'ordonnancement dynamique et permettent une utilisation à 100 % du CPU. Avec ces deux algorithmes, l'échéance la plus proche et la laxité sont calculés à chaque instant et ainsi les priorités des tâches changent. Par conséquent, cela entraîne à générer un comportement supplémentaire du système. C'est-à-dire l'ordonnanceur exige une allocation du CPU. De plus, ces algorithmes, en particulier LLF, se caractérise par une complexité de mise en oeuvre. Pour ces raisons, l'utilisation de EDF et LLF est souvent restreinte.

RMS et DMS se comportent mieux que EDF et LLF en cas de surcharge et s'implantent facilement avec les systèmes d'exploitation classiques. De plus, ils ne conduisent pas à un comportement supplémentaire comme le cas des algorithmes dynamiques. Étant donné que RMS est l'algorithme optimal pour un ordonnancement statique et son implantation est simple, il est considéré le meilleur.

Table 1.1 – Tableau comparatif des algorithmes d'ordonnancement

caractéristiques	**RMS**	**DMS**	**EDF**	**LLF**
Priorité	statique	statique	dynamique	dynamique
Utilisation du CPU	$\leq 69\%$ qd $n \to \infty$	$\leq 69\%$ qd $n \to \infty$	$\leq 100\%$	$\leq 100\%$
Simplicité de mise en oeuvre	++	-	-	- -
Optimalité	+	-	+	-
Meilleur en cas de surcharge	+	+	-	-
Répandu avec les exécutifs classiques	+	+	-	

Les machines d'états abstraites temporisés

Une machine en ASM (*Abstract State Machine*) est un ensemble fini de règles ayant la structure suivante :

$$
\begin{aligned}
R_1 &\equiv \text{if } cond_1 \text{ then } effect_1 \\
R_2 &\equiv \text{if } cond_2 \text{ then } effect_2 \\
&\vdots \\
R_n &\equiv \text{if } cond_n \text{ then } effect_n
\end{aligned}
$$

TASM (*Timed Abstract State Machine*) [29, 28] est une extension des ASM fournissant une sémantique pour exprimer le temps et la consommation des ressources. Le listing 1.1 présente un exemple d'une rèle en TASM. L'éxécution de cette règle prend entre 1 et 3 unités de temps et consomme 2000 unités d'énergie.

```
1 R1: Pickup from Press
2 {
3      t := [1, 3];
4      power := 2000;
5     if armbpos = atpress and armb = empty and press_block = available then
6        press_block := notavailable;
7        press := empty;
8        armb := loaded;
9 }
```

Listing 1.1 – Exemple d'une règle d'une machine en TASM

TASM permet aussi la composition hiérarchique (`sub ASM`, `function ASM`), la composition parallèle (plusieurs `main machine`) et la synchronisation entre ces machines. Elle permet de spécifier trois aspects du comportement des systèmes temps réel : le comportement fonctionnel, le comportement temporelle et la consommation des ressources.

Discussion :

Bien que TASM combine la spécification fonctionnelle et non fonctionnelle du système, mais il y a un manque d'outils de vérification basés sur ce formalisme. De plus, TASM est un langage textuel, il n'est pas donc aussi simple de modéliser une application par ce langage.

1.3.2 Vérificateurs de modèles

Dans le but de vérifier qu'une spécification formelle d'un système est correcte, il est nécessaire de recourir à des techniques facilitant ce processus comme la

vérification de modèles (eng. model checking).

La vérification de modèles [37] est une approche désignée à la vérification formelle. Elle est utilisée pour déterminer si un modèle d'un système satisfait sa spécification.

Comme il est illustré dans la figure 1.5, les deux entrées au vérificateur de modèles sont le modèle du système (comportements possibles du système) et les propriétés à vérifier (comportements attendus), exprimées en logique temporelle. Le vérificateur modèle examine tous les comportements possibles du systèmes, si un comportement non attendu est détecté, ce dernier fournit un contre exemple qui consiste en un scénario indiquant les circonstances dans lesquelles le modèle peut avoir un comportement non souhaité.

L'étude approfondie de cette approche a été suivie du développement d'outils de vérification comme HyTech [17], Kronos [10] ou UPPAAL [2], etc. Ces outils ont montré leur efficacité pour la validation de nombreuses études de cas.

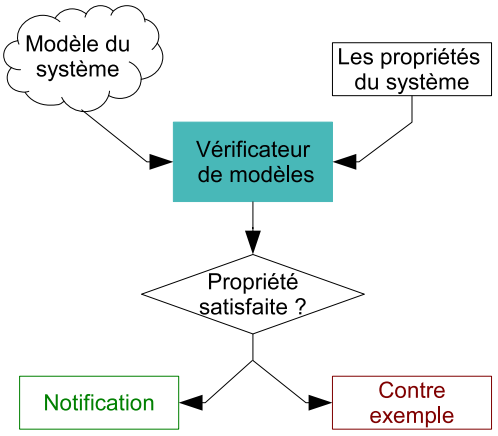

Figure 1.5 – Le *working flow* du vérificateur de modèles

Discussion :

La technique de vérification de modèles est basée sur une exploration exhaustive de tous les états du système définis par un modèle. Ce dernier est exprimé dans un langage formel comme les automates, les réseaux de Petri, etc. La plupart des étapes de cette approche sont complètement automatisées, ce qui facilite leur utilisation par un utilisateur qui n'est pas expert dans le domaine. Cependant, cette technique est dédié aux systèmes finis. Par ailleurs, un système possédant un nombre d'états très élevé, peut mener au problème d'explosion d'états.

1.3.3 Frameworks de vérification

Il existe plusieurs travaux qui ont proposé des frameworks pour la vérification. Ces travaux utilisent souvent la technique de la vérification de modèles.

Cheddar

Pour analyser les politiques d'ordonnancement appliquées pour un ensemble de tâches, il existe des simulateurs. Dans ce cadre, un framework [34] a été proposé et développé en Ada. Ce framework, appelé Cheddar, permet d'évaluer des critères de performance comme les contraintes temporelles.

Cheddar offre deux techniques de vérification : des tests de faisabilité et un environnement de simulation. Les tests de faisabilité permettent à l'utilisateur de prédire le respect des contraintes temporelles d'une application sans faire une simulation de l'ordonnancement. Quant à l'environnement de simulation, effectue une analyse d'un ordonnancement précédemment calculé pour vérifier les propriétés.

Les principales fonctionnalités du framework sont :

- Calcul d'ordonnancement pour la plupart des algorithmes "standards" comme RMS, DMS, EDF, LLF
- Support des tâches périodiques, apériodiques, activées aléatoirement (loi de Poisson, Exponentiel).
- Possibilité d'appliquer des tests de faisabilité dans le cas préemptif ou non préemptif :
  - Test du temps de réponse des tâches par rapport à leurs échéances (pour EDF, LLF, DMS et RMS).
  - Test sur les temps de blocage sur des ressources partagées par rapport à une borne (avec PCP, PIP ou IPCP),
  - Test sur le taux d'utilisation processeur par rapport à une borne (avec EDF, LLF, RMS et DMS).
- A partir d'un ordonnancement, l'environnement de simulation permet d'obtenir :
  - Les temps de blocage sur des ressources partagées.
  - Les temps de réponse des tâches.
  - Nombre de préemptions.
  - Recherche d'interblocage, d'inversion de priorité.
  - Echéances non respectées.

Discussion :

Le framework est conçu d'être facilement connecté à des outils CASE par l'échange des données sous forme XML. Il est aussi flexible, il peut être étendu pour exécuter des algorithmes d'ordonnancement spécifiques, ou faire des analyses d'événements spécifiques ou ordonnancer des tâches avec des modèles d'activation particuliers.

Les ordonnanceurs, les modèles d'activation des tâches et les analyseurs d'événements exprimés avec le langage Ada, ne sont pas compilés, mais interprétés par le framework lors de la simulation. Cette solution permet d'écrire et de tester facilement des extensions du framework, sans une connaissance approfondie du framework et du langage Ada. Cependant, Cheddar ne permet pas de vérifier certaines propriétés de performance comme la consommation mémoire et la consommation de la bande passante.

TASM toolset

Le TASM Toolset [29, 28] implémente les caractéristiques du langage TASM (section 1.3.1) via un éditeur, un simulateur et un analyseur.

L'éditeur permet de décrire textuellement la spécification TASM. Le simulateur permet la visualisation du comportement dynamique du système spécifié par le langage TASM. Cette visualisation comprend les dépendances de temps entre les mains machines parallèles, l'exécution pas à pas des variables d'environnement, et la consommation des ressources.

L'analyseur permet la vérification des propriétés définies dans la spécification TASM. TASM toolset intègre :

- **UPPAAL pour la vérification des propriétés temporelles** : TASM toolset effectue une traduction de la spécification TASM vers les automates temporisés.
- **SAT4J SAT Solver pour la vérification de la complétude et la consistance de la spécification** : cette vérification est assurée en traduisant les règles des machines d'état dans une formule booléenne.

Discussion :

L'apport de TASM toolset est de fournir une approche basée sur la spécification pour développer des systèmes embarqués temps réel. Les trois aspects du comportement de ces systèmes (fonctionnel, temporel et de ressource) peuvent être spécifiés, analysés et testés à partir des étapes initiales de la conception.

Cependant, il ne permet pas de traiter des systèmes dynamiquement reconfigurables.

ModSyn

ModSyn [11] (Model driven Co-synthesis for embedded system) est un framework basé sur une approche MDE (Model Design Engineering) pour la vérification formelle des systèmes embarqués. Cette approche permet la génération automatique d'un réseau d'automates temporisés à partir d'une spécification fonctionnelle d'une application embarquée présentée par un diagramme de classes UML et des diagrammes de séquences. Les automates automatisés seront obtenus automatiquement à partir d'une représentation basée sur MOF(Meta Object Facility), pour être une entrée à un outil de vérification formelle comme le vérificateur de modèles UPPAAL afin de vérifier certaines propriétés temporelles et fonctionnelles du système.

Les modèles UML sont traduits à des modèles conformes à un meta modèle IAMM (Internal Application Meta Model) puis ces modèles sont traduits en un réseau d'automates temporisés conforme à un meta modèle LTAMM (Labeled Timed Automata Meta Model). La transformation entre les modèles est implantée avec le langage Xtend du framework open ArchitectureWare.

Discussion :

Cependant que cette approche permet la vérification formelle de certaines propriétés des systèmes embarqués temps réel, elle ne garantit pas que ces propriétés sont vérifiées pour des systèmes dynamiquement reconfigurables. Elle ne permet pas aussi de vérifier des propriétés de ressource comme la consommation CPU, mémoire et bande passante des Bus. De plus, elle n'est pas destinée à des systèmes distribués.

DREAM

Dans le cadre de la vérification des principales propriétés QoS (Quality of service) des systèmes embarqués temps réel ayant une architecture à base de composants et qui utilisent le modèle de communication publish/subscribe, Dream (Distributed Realtime Embedded Analysis Method) [26] a été proposé. C'est un framework qui permet la simulation et la vérification des systèmes DRE (Distributed Real time Embedded) et qui utilise le formalisme des automates temporisés et le vérificateur de modèles pour l'analyse d'ordonnancement et la vérification des propriétés QoS.

Les auteurs proposent une sématique spécifique (DRE Semantic Domain) pour vérifier l'ordonnancement non préemptif des applications avioniques construites

sur la platforme Boieng Bold Stroke. En utilisant les automates temporisés, cette sémantique permet de modéliser les composants de base des systèmes DRE comme : les minuteries, les ordonnanceurs et les tâches. Dream utilise la transformation des modèles (eng. model trasformation) pour automatiser la tranformation des modèles de Boieng Bold Stroke en automates temporisés.

Discussion :

Dream permet ainsi de vérifier l'absence d'interblocage, la consommation du mémoire et l'ordonnancement non préemptif du système. Cependant, seulement l'ordonnancement non préemptif est assuré avec ce framework. Dream ne permet pas de vérifier la consommation de la bande passante des Bus. Toutefois, il est intéressant de tenir compte de cette propriété puisque Dream traite des systèmes distribués. Par ailleurs, ce framework ne prend pas en considération le cas des systèmes reconfigurables.

VERTAF

VERTAF (*Verifable Embedded Real Time Application Framework*) [18] est un framework qui intègre la conception et la vérification formelle des systèmes embarqués temps réel. L'automatisation de ce framework est illustré dans la transformation des modèles, l'ordonnancement, la vérification formelle et la génération du code à partir des modèles UML spécifiés par l'utilisateur. Trois diagrammes UML sont utilisés : le diagramme de classes avec déploiement, le digramme d'états-transitions pour chaque classe et le diagramme de séquences.

Afin d'assurer l'ordonnancement, VERTAF effectue la traduction des diagrammes d'UML en des réseaux de Petri, RTPN (Real-Time Petri Nets) ou CCPN (Complexe Choice Petri Nets). Il fait recourt à deux algorithmes d'ordonnancement :

- Si le système est sans RTOS, l'ordonnancement quasi-dynamique (QDS) est appliqué, qui nécessite un RTPN. QDS prépare le système pour être généré comme un seul noyau exécutif temps réel avec un ordonnanceur.
- Si le système est avec RTOS, Extended quasi-static scheduling (EQSS) est appliqué, qui nécessite un CCPN. EQSS prépare le système pour être généré comme un ensemble de plusieurs tâches qui peuvent être programmées et envoyées par un RTOS pris en charge.

Comme les modèles RTPN et CCPN sont générés automatiquement à partir des diagrammes de séquences, des ETAs (Extended Timed Automata) sont aussi générés à partir des diagrammes d'états-transitions pour effectuer la vérification formelle.

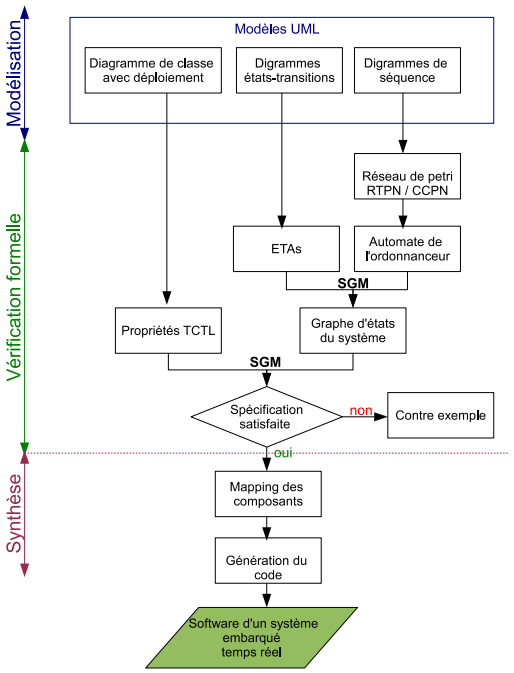

Figure 1.6 – Le *working flow* du VERTAF

Tous les ETAs générés sont fusionnés, avec un ETA ordonnancé généré dans la phase d'ordonnancement, en un seul graphe d'états. VERTAF utilise le paradigme de vérification de modèles pour la vérification formelle. Ainsi, les ETAs seront les entrées au vérificateur de modèles SGM (State Graph Manipulators).

Il y a deux classes de propriétés qui peuvent être de vérifiées avec VERTAF : (1) Propriétés définis par le système comme l'absence d'interblocage et la vivacité, (2) Propriétés définies par l'utilisateur à partir de la spécification UML. Toutes ces propriétés sont automatiquement traduites en spécifications TCTL pour être vérifier par SGM.

Discussion :

VERTAF est un framework proposé pour le développement des systèmes embarqués temps réel. Il est le résultat de l'intégration de trois technologies : la réutilisation des composants logicielles, la synthèse formelle et la vérification formelle. Cependant, VERTAF n'accepte en entrée que des modèles UML, alors que UML n'est pas spécifique à la modélisation des systèmes embarqués temps réel distribués dynamiquement reconfigurables.

De plus, bien que cet outil vérifie différentes propriétés du système, il ne permet

de vérifier aucune propriété liée aux ressources comme la mémoire, le CPU ou le Bus, notant que ces ressources sont de capacités limitées pour les systèmes embarqués.

1.4 Synthèse et objectifs

Dans notre travail de recherche, nous nous intéressons à la vérification formelle des propriétés non-fonctionnelles des RTES distribués et en particulier aux systèmes dynamiquement reconfigurables pour garantir le bon fonctionnement de tels systèmes.

Dans la littérature, plusieurs travaux de recherche ont proposé des approches permettant la vérification des propriétés non-fonctionnelles comme les propriétés temporelles et les propriétés de ressources. Ces derniers ont proposé des solutions basées sur différents langages formels et outils de vérification. Cependant, ces solutions sont destinées aux systèmes embarqués temps réel statiques. Aussi, la plupart de ces solutions ne sont pas destinées à des systèmes distribués. Ceci est illustré dans le tableau 1.2.

Table 1.2 – Tableau récapitulatif des frameworks existants

Frameworks	Systèmes embarqués temps réel	Systèmes distribués	Systèmes reconfigurables
VERTAF 2004	✓	✘	✘
CHEDDAR 2004	✓	✘	✘
DREAM 2006	✓	✓	✘
TasmToolset 2007	✓	✘	✘
ModSyn 2009	✓	✘	✘
Notre framework	✓	✓	✓

Cependant, l'évolution des exigences du contexte d'utilisation et la variation des contraintes de l'environnement d'exécution pour ces systèmes, rendent la reconfiguration dynamique de plus en plus importante. Mais, la reconfiguration au cours de l'exécution du système peut provoquer le mal fonctionnement du système par la production des anomalies et la perturbation de certaines propriétés non fonctionnelles. Il est donc nécessaire de vérifier la validité des propriétés non fonctionnelles après

l'exécution d'une action de reconfiguration.

Par ailleurs, les travaux existants ne permettent pas de vérifier toutes les propriétés que nous optons les préserver. Le tableau 1.3 présente, en guise de conclusion et de résumé, les propriétés vérifiées par certains framework, que nous comparons aux propriétés assurées par notre framework proposé dans le cadre de ce travail.

Notre objectif est de proposer un framework permettant la modélisation des systèmes embarqués temps réel distribués dynamiquement reconfigurables. Ce framework doit aussi permettre la vérification de certaines propriétés non fonctionnelles (consommation CPU, consommation mémoire, consommation de la bande passante, respect des échéances, absence d'interblocage et absence de famine). La vérification doit être basée sur des langages formels.

A l'issue de l'étude effectuée sur les formalismes, nous pouvons conclure que :

- Pour les propriétés de ressources (consommation mémoire, consommation CPU et consommation de la bande passante), elles peuvent être traitées par le langage TASM. Mais nous remarquons un manque d'outils de vérification basés sur ce formalisme.
- Pour le respect des échéances, il peut être vérifié soit par les automates temporisés, soit par les algorithmes d'ordonnancement. Mais, l'utilisation des automates temporisés est limitée par le problème d'explosion d'états.
- Pour les deux propriétés (absence d'interblocage et absence de famine), elles peuvent être vérifiées généralement en se basant sur des systèmes de transitions comme les automates temporisés et les réseaux de petri temporisés puisqu'il existe la technique de vérification de modèles. Cette technique est complètement automatique et elle est utilisée avec succès dans la pratique. Mais elle est limitée par le problème d'explosion d'espace et nécessite un modèle avec un nombre fini d'états.

Table 1.3 – Tableau récapitulatif des approches existantes

Approches et frameworks	Respect des échéances	Consommation CPU	Consommation mémoire	Largeur de la bande passante	Absence d'interblocage	Absence de famine
VERTAF 2004	✓	✘	✘	✘	✓	✓
CHEDDAR 2004	✓	✓	✘	✘	✘	✘
DREAM 2006	✓	✘	✓	✘	✓	✘
TasmToolset 2007	✘	✓	✓	✓	✓	✘
Amnel 2001	✓	✘	✘	✘	✓	✘
Gumzej 2004	✓	✘	✘	✘	✘	✘
Zaharia 2008	✓	✘	✘	✘	✓	✓
Boutekkouk 2009	✘	✘	✓	✓	✓	✘
Nascimento 2009	✘	✘	✘	✘	✓	✘
Notre framework	✓	✓	✓	✓	✓	✓

1.5 Conclusion

Dans ce chapitre, nous avons présenté les techniques de vérification et plus précisément la vérification formelle des propriétés non fonctionnelles des RTES distribués. Ensuite, nous avons présenté les formalismes utilisés pour la vérification de ces propriétés. Puis nous nous sommes intéressés aux frameworks de vérification existants.

Suite à notre étude bibliographique, nous décrivons dans le chapitre 3, notre approche proposée. Cette approche permet la vérification de certaines propriétés non fonctionnelles des RTES distribués dynamiquement reconfigurables tout en surmontant les limites des approches existantes. L'entrée à cette vérification est le résultat obtenue par un framework de modélisation. Ce framework est présenté dans le chapitre suivant.

Chapitre 2

Framework de modélisation

2.1 Introduction

Ce mastère s'inscrit dans le cadre d'un travail de thèse [16, 22]. Cette thèse offre tout un processus de développement pour les systèmes embarqués temps réel distribués dynamiquement reconfigurables. Ce processus comporte trois étapes : une étape de modélisation, une étape de vérification formelle et une étape de génération de code basée sur un support d'exécution. Pour la première étape, une approche dirigée par les modèles, nommée RCA4RTES[1], a été proposée.

Dans ce chapitre, nous présentons la première étape de modélisation pour procéder à notre travail qui s'intéresse à la deuxième étape (vérification formelle). Nous commençons par présenter le processus de développement. Puis nous mettons l'accent sur l'approche dirigée par les modèles RCA4RTES et la méthodologie de modélisation proposées. Enfin, nous clôturons par une conclusion.

2.2 Processus de développement

Le processus de développement destiné aux RTES distribués dynamiquement reconfigurables est automatisé. Il est constitué par trois étapes (figure 2.1) :

- **Étape de modélisation** : le concepteur modélise son application en utilisant UML, le profil MARTE et le profil RCA4RTES (section 2.5).
- **Étape de vérification formelle** : la modélisation présentée lors de l'étape précédente sera vérifiée pour garantir le respect de certaines propriétés non fonctionnelles. Si les propriétés ne sont pas vérifiées, le concepteur doit retour-

1. RCA4RTES : Reconfigurable Computing Architectures for Real Time Embedded Systems

ner à la première étape pour corriger sa modélisation. Sinon, nous passons à l'étape suivante. Cette étape sera présenté dans le chapitre 3.

– **Étape de génération de code** : à partir des modèles définis dans la première étape, on aura une génération de code qui est basée sur un support d'exécution.

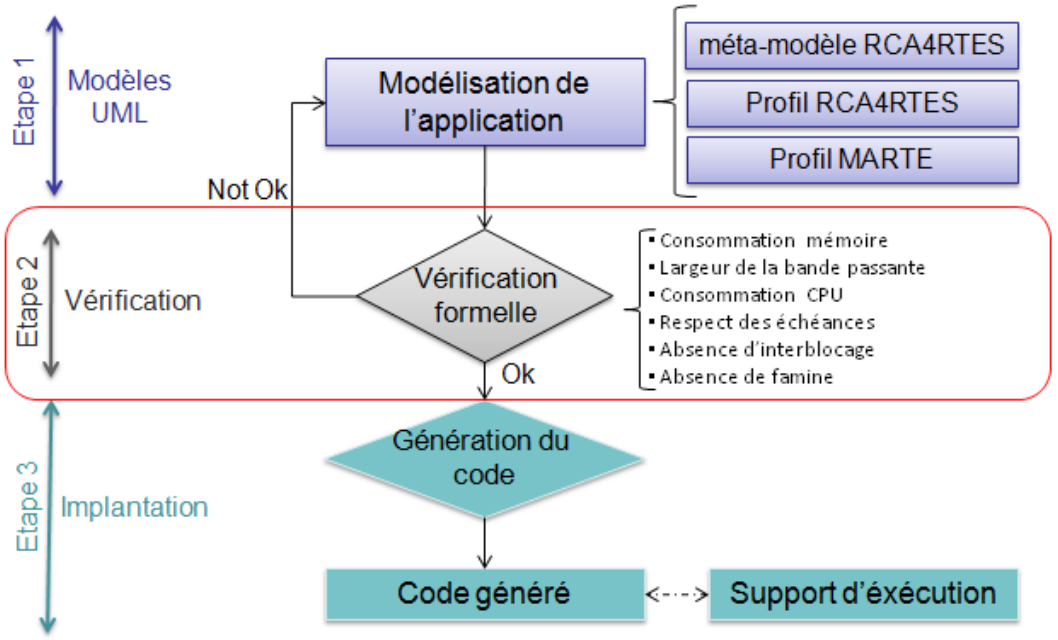

Figure 2.1 – Processus de développement

2.3 Approche dirigée par les modèles RCA4RTES

Une reconfiguration dynamique représente une modification architecturale ou comportementale d'une architecture logicielle en cours d'exécution. Plus précisément, l'évolution des besoins des utilisateurs et le changement de contraintes de l'environnement d'exécution exigent des systèmes embarqués temps réel adaptatives. Pour assurer la modélisation de telles applications, plusieurs travaux sont proposés dans ce contexte. L'article [21] présente un état d'art bien détaillé. Les travaux les plus puissants proposés sont les deux standards AADL [32] et MARTE [27].

Avec ces deux standards, la reconfiguration est présentée par un ensemble de configurations (ou modes) et des transitions entre eux. Ainsi, l'inconvénient majeur est la définition d'un ensemble statique de configurations du système. Toutes les configurations possibles sont prédéfinies a priori.

Pour palier cet inconvénient, l'approche dirigée par les modèles RCA4RTES introduit la notion de metaMode, qui représente et caractérise un ensemble de configurations au lieu de définir chacune d'entre elles. Ainsi, nous pouvons définir une reconfiguration dynamique avec un nombre non prédéfini de modes.

2.3.1 Notion de MetaMode

Un metaMode est une représentation abstraite d'un ensemble de configurations. Il est défini par des types de composants et de connexions entre eux, aussi bien des contraintes structurelles et des contraintes non-fonctionnelles.

Un mode appartenant à un metaMode est défini par un ensemble d'instances de ses composants et de ses connexions logicielles respectant ces contraintes structurelles et non fonctionnelles.

2.3.2 Reconfiguration dynamique niveau MetaMode

Les reconfigurations dynamiques sont spécifiées via des digrammes d'états-transitions composés par des metaModes et des transitions entre eux. Une transition niveau metaMode présente un ensemble de reconfigurations entre les modes. Lorsqu'un événement (présenté par une transition niveau metaMode) est déclenché, l'action de reconfiguration est appliquée sur le mode courant à l'un des modes appartenant au metaMode cible (comme il est illustré dans la figure 2.2).

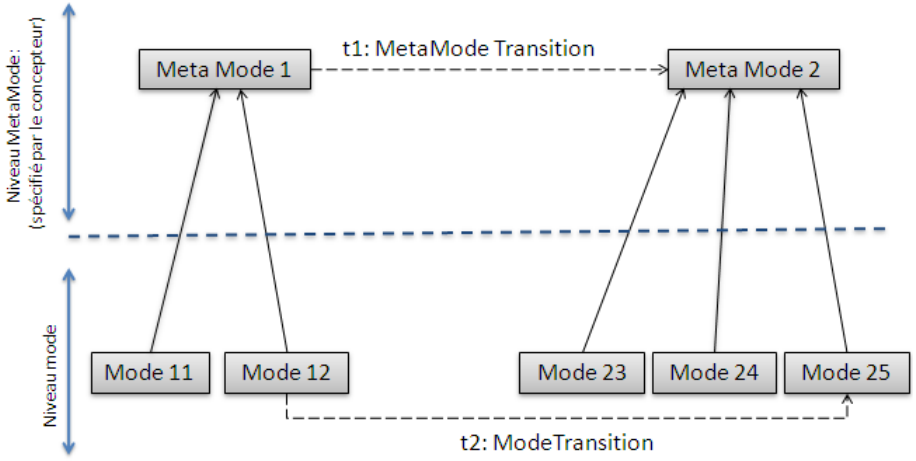

Figure 2.2 – Modélisation niveau MetaMode

Le choix du mode cible est assuré par des politiques de reconfiguration comme :

- Le choix du mode possédant un nombre limité de composants
- Le choix du mode nécessitant un nombre minimal d'actions de reconfiguration
- Le choix du mode exigeant une consommation minimale de ressources matérielles

La figure 2.2 explique le concept précédent par un exemple. Nous avons deux metaModes : *MetaMode1* et *MetaMode2*. La transition *t1* constitue une transition entre les deux metaModes. La transition *t2* est une transition parmi les transitions possibles déduites automatiquement à partir de *t1* selon les politiques de reconfigurations définies. Ainsi, la configuration (le mode) courante *Mode12* est remplacée automatiquement par la configuration *Mode25*.

Chaque metaMode doit être alloué sur l'architecture matérielle du système. Cette allocation est appliquée en se référant à des contraintes d'allocation. Ces contraintes indiquent le nombre d'instances d'un composant allouant chaque CPU puisqu'un composant peut allouer un ou plusieurs CPUs dans le cas d'une architecture matérielle multiprocesseur.

2.3.3 Méthodologie de modélisation

L'approche RCA4RTES propose une méthodologie pour la spécification des RTES distribués dynamiquement reconfigurables. Le concepteur doit suivre les étapes suivantes pour modéliser son application :

- **Etape 1 : Spécification de la partie logicielle** : Le concepteur commence par spécifier l'aspect dynamique de son application via le diagramme états-transitions. Ce diagramme est composé par l'ensemble des metaModes du système et les transitions entre eux. Il spécifie aussi les exigences de reconfiguration comme le mode initial et les événements déclencheurs des actions de reconfiguration.

  Puis, en utilisant le profil RCA4RTES, le concepteur spécifie, pour chaque metaMode, l'ensemble des composants et les connexions entre eux ainsi que les contraintes structurelles et non fonctionnelles correspondantes.
- **Etape 2 : Spécification de la partie matérielle** : Comme il s'agit de systèmes distribués, le concepteur doit définir l'architecture de déploiement de l'application par l'intermédiaire d'un diagramme de déploiement. Ce diagramme comporte les différents noeuds du système et les liens entre eux.

  Ensuite, pour chaque noeud défini, il spécifie son architecture matérielle en

utilisant le profil MARTE. Il identifie les composants matériels tels que les processeurs, les mémoires, les Bus, etc.

– **Etape 3 : Allocation de la partie logicielle sur la partie matérielle** : Dans cette étape, le concepteur fait l'allocation des metaModes définis à la première étape sur l'architecture matérielle définie à la deuxième étape. Ainsi, l'allocation est définie à partir des modèles logiciels sur des instances matérielles. Certaines contraintes doivent être définies pour spécifier les politiques d'allocation. Ces contraintes d'allocation sont décrites en utilisant le langage VSL (Value Specification Language) du profil MARTE [27].

Les nouveaux concepts, définis dans cette section, permettent de spécifier les reconfigurations dynamiques et d'introduire des contraintes temporelles et de ressources dans la modélisation haute niveau des systèmes embarqués temps réel distribués à base de composants. Tous ces concepts sont traduits par un méta-modèle (section 2.4) et un profil (section 2.5) comme implantation de ce méta-modèle.

2.4 Le méta-modèle RCA4RTES

Le méta-modèle RCA4RTES est composé par quatre paquetages (figure 2.3) qui sont décris sous forme de notations UML.

1. **SWConfRTES** : permet la définition des configurations en spécifiant les metaModes du système (Figure 2.4).
2. **SWConstraintRTES** : spécifie les contraintes d'allocation ainsi que les contraintes structurelles et non-fonctionnelles qui doivent être respectées par le metaMode (Figure 2.7).
3. **SWEventRTES** : spécifie les différents événements possibles qui peuvent activer une reconfiguration (Figure 2.8).
4. **SWReconfRTES** : spécifie les reconfigurations dynamiques entre les metaModes. Ce paquetage importe SWConfRTES et SWEventRTES pour definir respectivement les metaModes et les événements (Figure 2.9).

Dans la suite, nous allons détailler les paquetages du méta-modèle. Le paquetage SWConfRTES (Figure 2.4) introduit la méta-classe MetaMode, qui permet de présenter des applications à plusieurs configurations. Un MetaMode est décrit par un ensemble de composants structurés et de connexions.

– Un composant structuré, qui est représenté par la méta-classe StructuredComponent, présente une tâche périodique, sporadique ou apériodique.

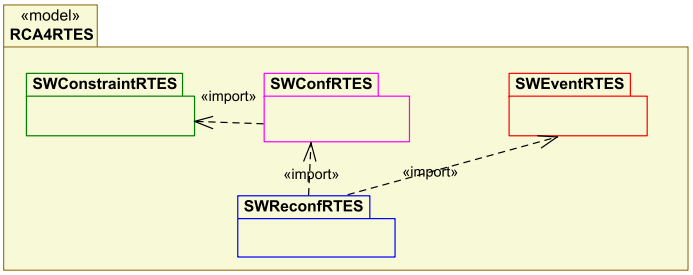

Figure 2.3 – Le Méta-Modèle RCA4RTES

– Une connexion, qui est présentée par la méta-classe Connector, relit deux ou plusieurs composants structurés.

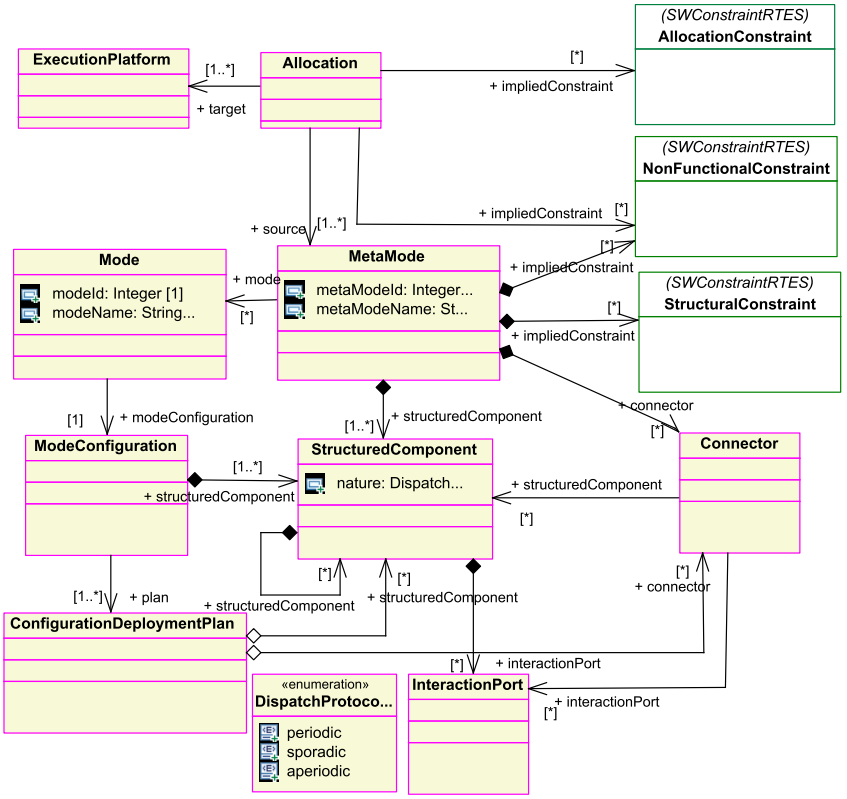

Figure 2.4 – Le paquetage SWConfRTES

Un MetaMode est caractérisé aussi par des contraintes qui sont définies dans les paquetage *SWConstraintRTES*. La méta-classe *Allocation* permet de spécifier l'allocation des MetaModes sur les supports d'exécution. Cette allocation a des contraintes non fonctionnelles et des contraintes d'allocation qui doivent être respectées. Le méta-modèle présente trois types de contraintes qui sont décrites dans le paquetage SWConstraintRTES (figure 2.7) :

– **Des contraintes structurelles** reliées à la structure de l'architecture. Par exemple, on peut imposer le nombre minimale et maximale d'instances d'un type de composant et on peut imposer le nombre maximal qu'on peut avoir pour un type de connexion. Citons par exemple les deux contraintes structurelles suivantes correspondantes à la spécification de la figure 2.5 :
  - $composant1 \rightarrow size > 0$ *and* $composant1 \rightarrow size < 4$: c'est-à-dire on peut avoir au moins une instance du composant1 et au maximum 3 instances.
  - $compsant1.composant2 \rightarrow size = 4$: c'est-à-dire une instance du composant1 (c'est le composant possédant l'interface fournie) peut être connectée à 4 instances du composant2 (c'est le composant possédant l'interface requise).

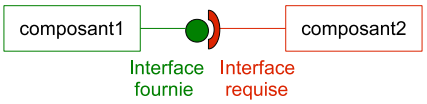

Figure 2.5 – Architecture à base de composant

– **Des contraintes non fonctionnelles** spécifient des conditions sur les propriétés non fonctionnelles des éléments associés au metaMode. Par exemple, on peut préciser la fréquence du processeur selon son taux d'utilisation :
procUtiliz > (90, percent) ? clockFreq==(60,MHz) : clockFreq==(20,MHz)
– **Des contraintes d'allocation** spécifient la politique d'allocation utilisée du modèle logiciel (metaMode) sur une architecture matérielle multiprocesseur figée. Par exemple pour l'allocation présentée dans la figure 2.6, nous avons la contrainte suivante :
(Composant.size % 2==0) ? a1=1 : a2=1 : c'est-à-dire si le numéro de l'instance du composant est paire, alors elle alloue le CPU1. Sinon, elle alloue le CPU2.

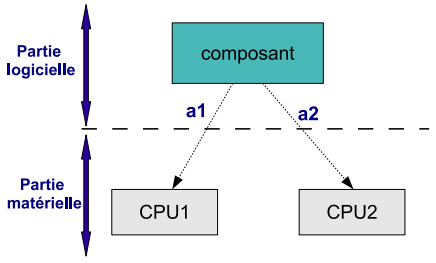

Figure 2.6 – L'allocation d'un composant

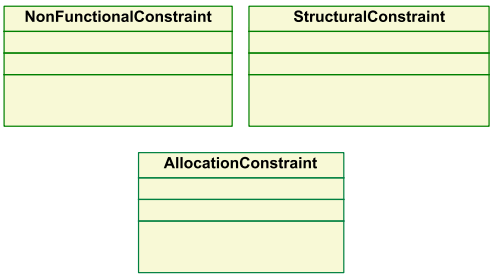

Figure 2.7 – Le paquetage SWConstraintRTES

Le paquetage SWEventRTES comporte les types d'événement menant à la reconfiguration d'un RTES. L'énumeration MetaModeChangeEventKind definie dans le paquetage SWEventRTES (Figure 2.8) présente deux types d'évenements qui peuvent lancer une transition :

- Un evenement d'application présente un changement de configuration répondant aux exigences des utilisateurs,
- Un evenement d'infrastructure présente une variation de la situation de l'infrastructure.

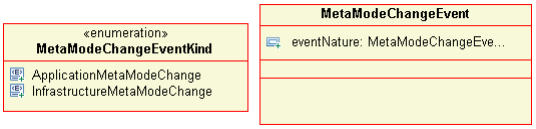

Figure 2.8 – Le paquetage SWEventRTES

Le paquetage SWReconfRTES, présenté dans la figure 2.9, permet de décrire les reconfigurations dynamiques des RTES. La méta-classe SoftwareSystem, qui présente la machine d'état du système, est composée d'un ensemble de transitions et de MetaModes. Lorsqu'un événement est déclenché, une transition permet de passer d'un MetaMode à un autre. Pour chaque transition, une activité de reconfiguration est associée. Elle représente un algorithme de passage de la configuration actuelle vers la configuration cible.

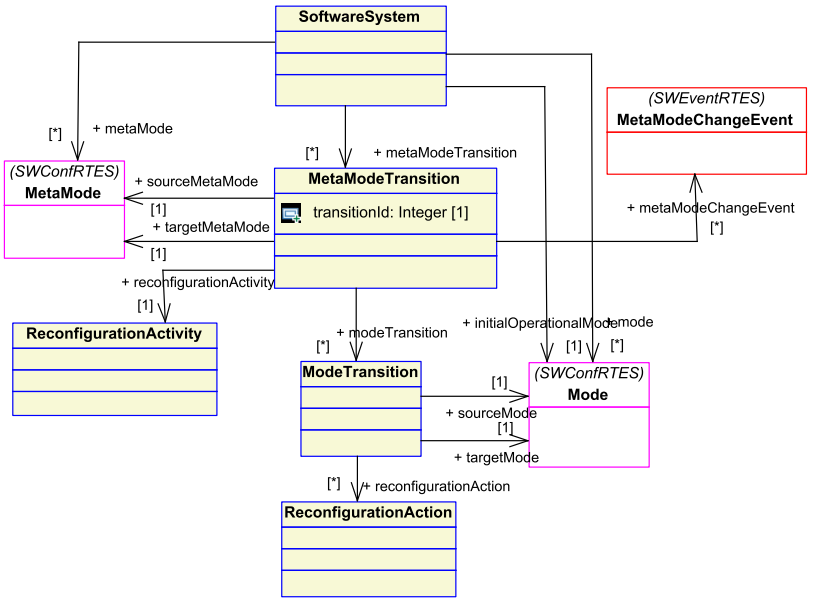

Figure 2.9 – Le paquetage SWReconfRTES

2.5 Le profil RCA4RTES

Pour spécifier les RTES distribués dynamiquement reconfigurables, un profil a été établi (Figure 2.10) présentant une implantation du meta-modèle RCA4RTES. Ce profil importe les profils NFPs et VSL du profil MARTE [27] et les NFP types basiques de la bibliothèque MARTE [27] (figure 2.11).

Dans RCA4RTES, une tâche est définie par un ensemble de caractéristiques :

- **Nature** : définit la nature de la tâche qui peut être périodique, sporadique ou apériodique.
- **Period** : définit la période d'une tâche périodique. Elle est utilisée aussi pour définir le temps minimal entre deux activations successives d'une tâche sporadique. Elle est de type *NFP_Duration* de la bibliothèque MARTE.
- **Deadline** : définit l'échéance d'une tâche périodique ou sporadique. Elle est de type *NFP_Duration* de la bibliothèque MARTE.
- **StartTime** : définit le temps de début d'une tâche apériodique. Elle est de type *NFP_DateTime* de la bibliothèque MARTE.
- **EndTime** : définit le temps de fin d'une tâche apériodique. Elle est de type *NFP_DateTime* de la bibliothèque MARTE.
- **WCET1(Processeur 1 MHZ)** : définit le pire temps d'éxécution (worst case

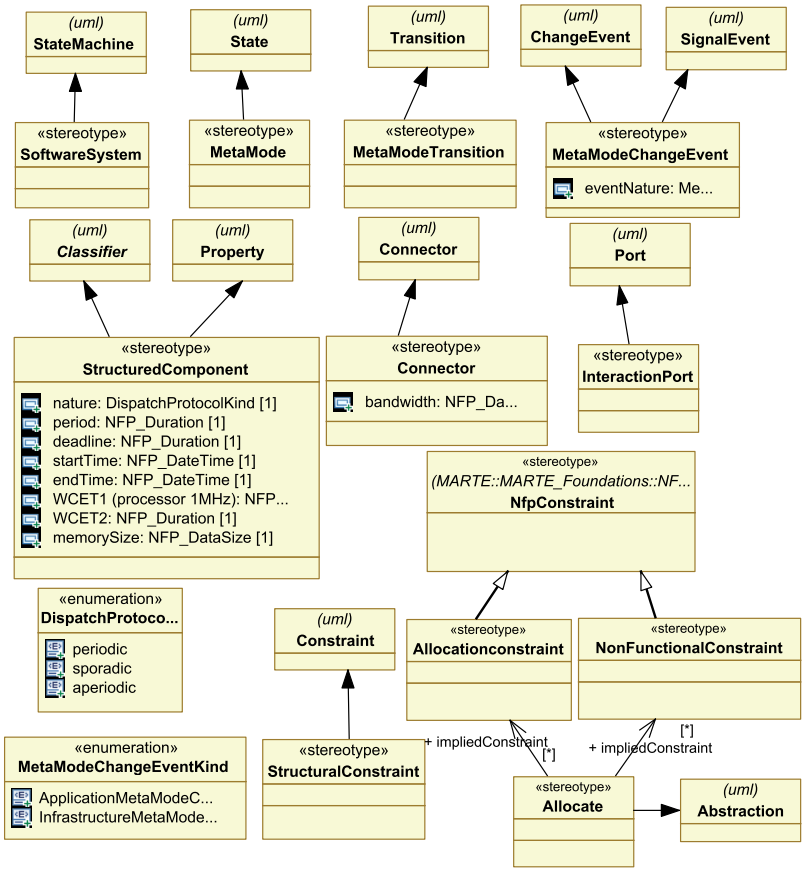

Figure 2.10 – Description du profil RCA4RTES

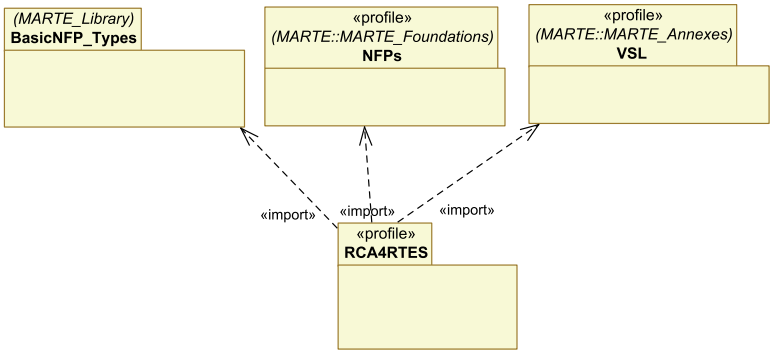

Figure 2.11 – Les dépendances du profil RCA4RTES

execution time) des tâches périodiques et sporadiques pour un processeur de fréquence 1 MHZ. Il est de type *NFP_Duration* de la bibliothèque MARTE.

– **WCET2** : définit le pire temps d'exécution constant pour une tâche. Ce temps est indépendant du processeur. Il est de type *NFP_Duration* de la bibliothèque MARTE.

– **MemorySize** : définit l'espace mémoire nécessaire pour l'exécution d'une tâche. Il est de type *NFP_DataSize* de la bibliothèque MARTE.

Pour le pire temps d'exécution, deux propriétés sont définies : le WCET1 qui dépend de la fréquence du processeur et WCET2 qui lui est indépendant. Le WCET1 est défini par le rapport du nombre d'instructions sur la fréquence du processeur. La valeur de WCET1 varie selon la fréquence du processeur. Par contre, la valeur de WCET2 est fixe. Ainsi, la valeur de WCET d'une tâche est la somme de WCET1 et WCET2.

Ces propriétés sont proposées dans le profil afin de pouvoir vérifier certaines propriétés non fonctionnelles telles que la consommation CPU, la consommation mémoire, etc.

Les composants sont liés par des connexions. Pour caractériser ces connexions, le stereotype *Connector* qui étend la meta-classe Connector d'UML a été défini. Il est caractérisé par la propriété `bandwidth` de type *NFP_DataTxRate* de la bibliothèque MARTE.

Pour assurer l'allocation des metaModes sur le support d'exécution, un stereotype *allocate* qui étend la méta-classe Abstraction a été défini. Ce stéréotype est associé aux contraintes d'allocation et les contraintes non fonctionnelles.

2.6 Conclusion

Dans ce chapitre, nous avons présenté le processus de développement proposé par l'approche RCA4RTES pour les RTES distribués dynamiquement reconfigurables. Puis, nous avons présenté la première étape de ce processus. Cette étape sera réalisée via un framework de modélisation pour ces systèmes. La modélisation obtenue par ce framework va être l'entrée à un framework de vérification de certaines propriétés non fonctionnelles qui sera présenté dans le chapitre suivant.

Chapitre 3

Étude théorique : Framework de vérification

3.1 Introduction

Ce travail de mastère illustre une intégration des solutions de modélisation et de vérification des RTES distribués dynamiquement reconfigurables. Dans le chapitre précédent, nous avons présenté le framework de modélisation. Dans ce chapitre, nous présentons notre approche proposée pour assurer la vérification des différentes propriétés non fonctionnelles pour ces systèmes en décrivant les outils et les formalismes utilisés.

3.2 Propriétés non fonctionnelles à vérifier

Dans notre travail, nous avons considéré deux types de propriétés non fonctionnelles : des propriétés de ressources et des propriétés temporelles.

3.2.1 Propriétés de ressources

Les applications logicielles embarquées sont généralement allouées sur des architectures matérielles distribuées avec des ressources à caractère limités comme la taille des mémoires, la bande passante des Bus et la capacité des processeurs, etc... Par conséquent, les concepteurs des systèmes embarqués doivent être concernés par l'allocation des ressources, assurant qu'ils ne dépassent pas leurs limites. Nous considérons dans notre travail trois propriétés de ressources :

- **Consommation CPU** : Le taux d'utilisation d'un CPU ne doit pas dépasser une borne bien déterminée. Cette borne varie selon la politique d'ordonnancement utilisée par le système.
- **Consommation mémoire** : Pour un système, il ne faut pas avoir de fuite mémoire. C'est-à-dire, la somme des allocations mémoire par les tâches du système ne dépasse la taille de la mémoire matérielle.
- **Consommation de la bande passante** : Cette propriété nécessite la vérification de l'existence d'un Bus matériel pour toute connexion logicielle ainsi que l'absence d'un débordement de la bande passante de ce Bus. Donc, la somme des bandes passantes des connexions logicielles projetées sur un Bus ne doit pas dépasser la bande passante de ce Bus.

3.2.2 Propriétés temporelles

Parmi les propriétés temporelles nous citons :

- **Respect des échéances** : Le temps de réponse d'une tâche ne doit pas dépasser son échéance.
- **Absence d'interblocage** [8] : Le système ne se bloque pas dans une situation dont deux ou plusieurs processus sont en attente les uns des autres.
- **Absence de famine** [8] : Absence d'une situation dans laquelle les processus travaillent mais n'effectuent aucun progrès.

3.3 Framework de vérification

Les propriétés non fonctionnelles à vérifier doivent être conservées après chaque action de reconfiguration. Selon la spécification RCA4RTES, la reconfiguration est définie niveau metaMode. L'action de reconfiguration est appliquée d'un metaMode à un autre. Donc, il suffit de vérifier la validité de ces propriétés pour chaque metaMode du système. Nous avons vérifié des propriétés de ressources (la consommation CPU, la consommation mémoire et la largeur de la bande passante) ainsi que des propriétés temporelles (le respect des échéances, l'absence d'interblocage et de famine) pour assurer un bon fonctionnement du système.

En se référant à l'existant, la vérification est appliquée à une configuration bien déterminée du système. Dans notre cas, nous vérifions tout un style architectural (ou modèle) pouvant avoir un nombre indéfini d'instances logicielles.

Comme le metaMode peut avoir plusieurs instances (ou modes), il faut vérifier

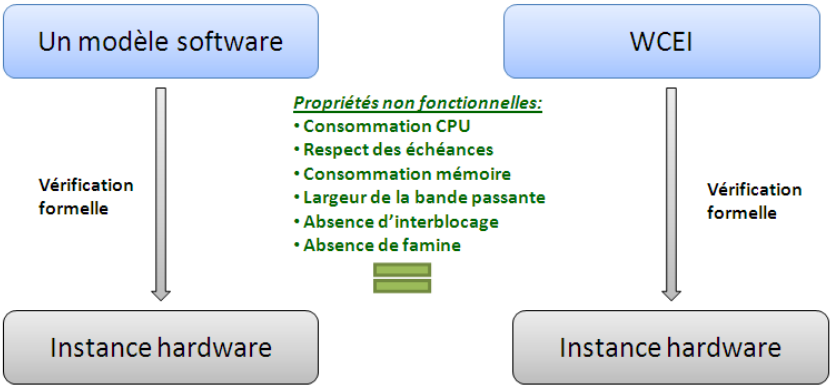

Figure 3.1 – Vérification formelle d'un metaMode

alors tous les modes. Mais il est difficile de prévoir le nombre de modes possibles. Pour cela, nous avons proposé comme une solution d'exercer la vérification des propriétés pour une instance logicielle de manière que la validité de cette instance implique la validité de toutes les instances du metaMode. Cette instance logicielle représente l'instance dans le pire cas d'exécution d'un metaMode (voir figure 3.1), notée WCEI (Worst Case Execution Instance). Donc, pour chaque propriété, nous devons définir l'instance adéquate.

Dans notre travail, nous considérons trois types de tâches [19] :

- **Tâche périodique :** elle se caractérise par un intervalle de temps constant entre deux activations successives. Elle est définie par trois paramètres : une échéance (D_p), une période (P_p) et une capacité (C_p).
- **Tâche sporadique :** elle peut être activée à un instant aléatoire mais elle se caractérise par un délai minimal entre deux activations successives (P_{sp}). Elle est définie aussi par une échéance (D_{sp}) et une capacité (C_{sp}).
- **Tâche apériodique :** elle est activée une seule fois et elle est caractérisée par un temps d'arrivé et une capacité (C_{ap}).

3.3.1 Les propriétés de ressources

Comme ces propriétés dépendent des ressources matérielles, nous avons défini cette instance (WCEI) par le nombre maximal d'instances des composants et de connexions. Ainsi, nous aurons une consommation maximale de ressources matérielles (CPUs, mémoires et Bus).

L'instance dans le pire cas d'exécution est interprétée à partir des modèles de l'application. Nous avons proposé un algorithme permettant de valider la spécification et

de déterminer cette instance pour chaque metaMode afin d'être par la suite vérifiée.

Tout d'abord, pour déterminer le WCEI correspondant aux propriétés de ressources, nous avons besoin du nombre maximal de tâches et de connexions possibles. Ce nombre est défini par le concepteur via des contraintes structurelles. Si ces contraintes sont indéfinies alors notre algorithme demande au concepteur de compléter les contraintes manquantes.

Ensuite, notre algorithme vérifie que la spécification est correcte. Un composant requis doit être toujours connecté à un composant fourni correspondant. Afin de garantir cela, nous devons parcourir touts les types de connexions et vérifier que le nombre de composant fourni multiplié par son nombre maximale de connexions ne dépasse pas le nombre de composants requis. Ainsi nous garantissons qu'il n'aura pas de composants requis libres. Dans le cas échéant, notre algorithme demande au concepteur de rectifier le nombre maximal du composant requis.

Ainsi, nous pouvons créer les différentes tâches et connexions du metaMode pour vérifier ensuite les propriétés de ressources.

La consommation CPU

Nous avons recouru aux algorithmes d'ordonnancement comme formalisme pour vérifier la consommation CPU et le respect des échéances. D'après l'étude comparative entre les différents algorithmes d'ordonnancement dans le premier chapitre, nous avons choisi l'algorithme d'ordonnancement RMS [25] et le framework Cheddar [34] comme outil de vérification.

L'algorithme d'ordonnancement RMS impose quelques hypothèses :

– Les tâches sont périodiques, préemptives et indépendantes

– L'échéance d'une tâche est en fin de période ($D = P$)

Ainsi, nous pouvons traiter que les tâches périodiques. Mais dans notre cas, nous traitons aussi les tâches sporadiques et apériodiques. Donc nous proposant une solution pour tenir en compte de ces deux types de tâches.

Dans un premier temps, puisque nous analysons l'instance logicielle dans le pire cas d'exécution, nous considérons le pire cas d'exécution des tâches sporadiques. C'est le cas où elles se déclenchent le plus souvent possible (deux invocations consécutives sont séparées par le délai minimal). De ce fait, nous considérons la tâche sporadique comme étant une tâche périodique avec C_p=C_{sp} et D_p=P_p=D_{sp}=P_{sp}.

Pour une configuration de n tâches périodiques et Sporadiques T= $\{t1, t2,..,tn\}$, la séquence produite par l'algorithme d'ordonnancement RMS est cyclique et elle se

répète de manière similaire. L'étude de la séquence d'exécution produite peut se limiter à un temps appelé période d'étude ou méta-période et notée P_{etud}. Après chaque période d'étude, le système se trouve dans l'état initial. Cette période représente le $PPCM$ (Plus Petit Multiple Commun) des périodes des tâches périodiques et sporadiques. Elle est donnée par :

P_{etud} = [0, PPCM $\{P_i\}$] $i \in [1, n]$

Dans un second temps, nous faisons la simulation d'une tâche apériodique sur un intervalle de temps I, de manière qu'elle sera exécutée une seule fois. Dans le cas où il existe des tâches périodiques et/ou Sporadique, cet intervalle I représente le plus petit multiple du méta-période incluant la date de début et de fin de toutes les tâches apériodiques. S'il existe que des tâches apériodiques, cet Intervalle I sera la date de fin de la dernière tâche apériodique à exécuter.

La vérification de la consommation CPU consiste à comparer le facteur d'utilisation du processeur à une borne bien déterminée. Ce facteur est calculé suivant la formule $\sum_{i=0}^{n}(Ci/Pi)$ [25], où n est le nombre des tâches périodiques. En se basant sur l'algorithme d'ordonnancement RMS, le facteur d'utilisation du processeur doit être inférieur ou égale à $n(2^{1/n}\text{-}1)$ [25], où n est le nombre des tâches périodiques. Ce test est appliqué à tous les processeurs du système.

Pour mettre en oeuvre ce test avec Cheddar, nous traduisons les modèles définis, à partir d'un fichier XMI, en un fichier XML respectant le format de l'entrée du simulateur Cheddar. Ce fichier contient l'ensemble des tâches d'un WCEI, les CPUs et l'allocation des CPUs par ces tâches.

La figure 3.2 illustre la manière dont nous avons adapté le fichier XMI pour être une entrée à l'outil Cheddar. Ainsi, nous vérifions la consommation CPU.

La consommation mémoire

Chaque tâche réserve un espace mémoire pour assurer son exécution. Cet espace mémoire représente la pile de tâche qui manipule les données. Nous avons négligé l'allocation dynamique des tâches. L'allocation mémoire des tâches périodiques et sporadiques est permanente tout au long l'exécution du système. Par contre, l'allocation des tâches apériodiques est préservée uniquement durant leurs exécutions.

Dans notre travail, nous optons à vérifier, à chaque instant t, que la consommation mémoire par toutes les tâches du système exécutées sur le même noeud soit inférieur à la taille de sa mémoire. Pour cela, nous avons élaboré un algorithme (voir Annexe A). Cet algorithme permet de vérifier la consommation mémoire pour les

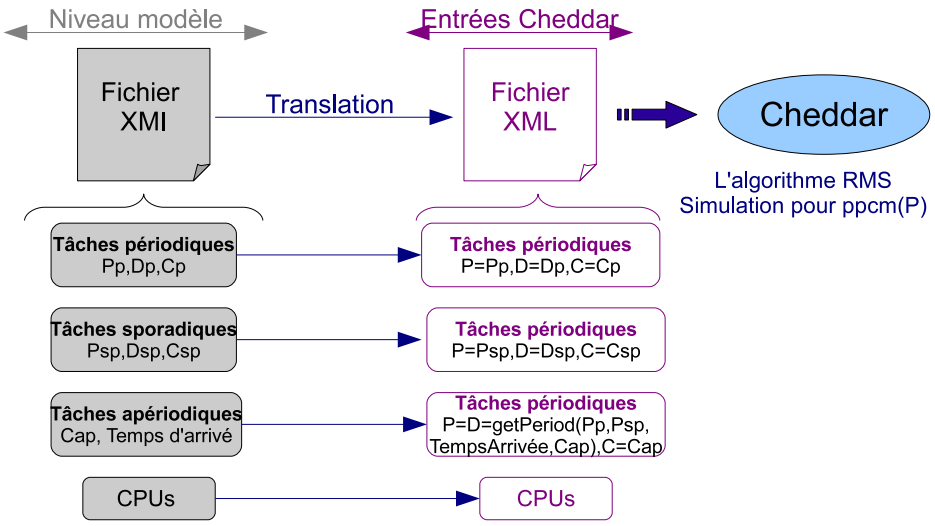

Figure 3.2 – Transformation du WCEI en entrée de Cheddar

WCEIs des metaModes du système. Pour un WCEI donnée, l'algorithme traite tous les noeuds du système. Nous supposons qu'un noeud possède une seule mémoire avec une taille définie. Nous distinguons trois cas :

1. Existence des tâches périodiques et/ou sporadiques seulement : Puisque ces types de tâches sont allouées dans la mémoire même si elles sont inactivées, nous calculons leur consommation mémoire par la somme des espaces mémoires alloués par ces tâches. Si cette somme dépasse la taille mémoire du noeud, la vérification sera interrompue en avertissant d'un débordement de la mémoire.
2. Existence des tâches apériodiques seulement : Comme premier test, nous calculons la consommation mémoire des tâches de la même manière que le cas précédent en considérant les tâches apériodiques comme des tâches périodiques. Si cette somme calculée ne dépasse pas la taille mémoire du noeud, alors la consommation mémoire est vérifiée. Dans le cas échéant, nous calculons autrement. Nous détectons le chevauchement des intervalles de temps des tâches apériodiques qui exige l'espace mémoire maximale. En fait, une tâche apériodique alloue de l'espace mémoire que lorsqu'elle est en cours d'exécution puisqu'elle est activée une seule fois. Pour cela, nous avons crée des intervalles élémentaires selon les date de début et de fin des tâches comme il est illustré dan la figure 3.3. Ensuite, nous calculons, durant chaque intervalle, la somme des mémoires allouées par ces tâches apériodiques. Si la valeur maximale calculée ne dépasse pas la taille mémoire, alors la consommation mémoire est bien vérifiée.

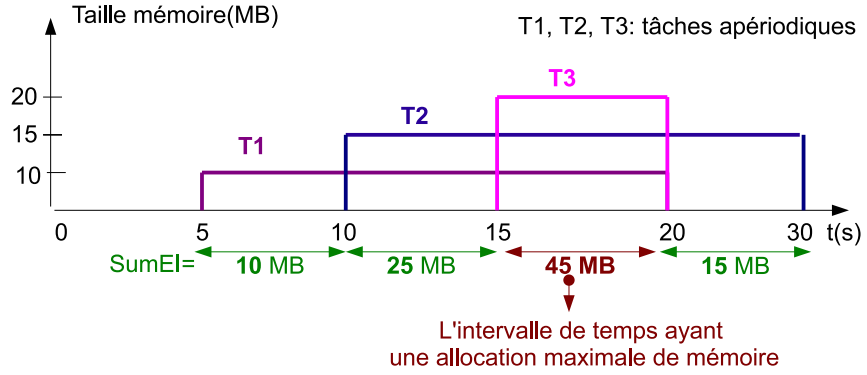

Figure 3.3 – Intervalles de temps des tâches apériodiques

3. Existence des tâches périodiques, sporadiques et apériodiques : Dans ce cas, nous faisons la somme des deux valeurs calculées précédemment. Si ce résultat ne dépasse pas la taille mémoire, alors la consommation mémoire est vérifiée. La procédure 3.1 permet de détecter le chevauchement des intervalles de temps des tâches apériodiques qui requiert l'espace mémoire le plus élevé.

```
1  Procedure verificationAperiodicThread()
2  MaxMemory=0;
3  createElementaryIntervals(aperiodicThreads);
4  for each elementary interval i do
5      sumEI=getAllocatedMemorySize(i);
6      if sumEI> MaxMemory then
7          MaxMemory=sumEI;
8          if (isAllAperiodicThread()) then
9              if (MaxMemory>MemorySize) then
10                 Exit("Memory consumption not verified");
11             endIf
12         else
13             if (MaxMemory+ allocatedMemorySize(periodicThreads,
                 sporadicThreads)>MemorySize) then
14                 Exit("Memory consumption not verified");
15             endIf
16         endIf
17     endIf
18 endFor
19 endProcedure
```

Algorithme 3.1 - Vérification mémoire dans le cas d'existence des tâches apériodiques

Enfin, pour garantir une vérification des systèmes reconfigurables, cet algorithme doit être appliqué pour le WCEI de chaque metaMode du système.

La consommation de la bande passante

Une architecture logicielle basée composant est caractérisée par des connexions logicielles entre les composants. Il est nécessaire de vérifier l'existence d'un Bus matériel assurant chacune de ces connexions logicielles. Ces dernières peuvent être locales ou distantes. Une connexion locale relie deux composants allouant deux CPUs d'un même noeud. Ces deux CPUs doivent être connectés par un Bus. Une connexion distante relie deux composants allouant deux CPUs de deux noeuds différents. Nous vérifions que chaque CPU est connecté au port du noeud par un Bus et que les noeuds sont connectés aussi par un Bus.

Par ailleurs, il est nécessaire de vérifier que tous les Bus du système assurent, durant l'exécution du système, l'échange des données sans avoir un débordement de la bande passante. Dans ce contexte, nous avons proposé un algorithme (voir Annexe B) qui permet la vérification de la consommation de la bande passante. Dans cet algorithme, nous distinguons trois cas :

1. Le cas du Bus local : ce Bus permet de connecter les CPUs d'un noeud du système. Il supporte les types de connexions logicielles locales. Donc, nous devons vérifier que la somme des bandes passantes de toutes les connexions locales projetées sur ce Bus est inférieure ou égale à sa bande passante. La procédure 3.2 permet la vérification de la consommation de la bande passante d'un seul Bus local. Un Bus peut lier plus que deux CPUs (Bus2.1 de l'exemple 3.4). Donc, nous calculons la somme des bandes passantes de tous les types de connexions possibles entre chaque couple de CPUs. Pour un couple de CPUs, nous calculons la bande passante de chaque type de connexion en multipliant sa bande passante (`BWcnxType`) par le nombre de connexions maximales projetées sur le Bus. Ce nombre est égal au minimum des deux valeurs suivantes :
   - Le nombre de composant possédant l'interface requise de la connexion (`nbRequiredTh`).
   - Le nombre de composant possédant l'interface fournie (`nbProvidedTh`) multipliée par le nombre de connexions maximale de l'interface fournie (`nbCnxMax`). Une interface fournie peut être connectée à `nbCnxMax` interfaces requises.

La valeur calculée (`SumBW`) sera comparée à la bande passante du Bus pour vérifier l'absence d'un débordement. Cette procédure est appelée pour chaque Bus d'un noeud.

Mais, il existe des cas où le Bus local est de type `Access`. C'est-à-dire, il peut lier un CPU à un autre, comme il peut lier un CPU avec le port du noeud. Dans ce cas, nous devons ajouter à la valeur calculée précédemment, la somme des bandes passantes des connexions distantes mappées sur ce Bus.

```
1  Procedure verificationBW (Bus)
2  SumBW=0 // la somme des bandes passantes des connexions mappées sue le Bus
3  For each couple of cpu connected by the Bus
4      For each software connection type
5          sumBW = sumBW + (BWcnxType*min(nbRequiredTh,(nbProvidedTh*
             nbCnxMax)))
6          // la somme est appliquée seulement pour les cnxType entre le couple du
             CPUs courant
7          if (sumBW > BW(Bus)) then
8              Exist ('bandwidth_overflow')
9          EndIf
10     EndFor
11 EndFor
12 If(getType(Bus)=AccessType) then
13     //le Bus peut relier les CPUs avec les ports du noeud
14     For each CPU(i) isLinkedToThePortOfNode(Bus)
15         SumBW=+ SumBw(CPU(i))
16         //sumBW(CPU(i)) est une fonction qui retourne la bande passante des
             connexions distantes ayant des composants allouant le CPU(i)
17         if (sumBW > BW(Bus)) then
18             Exist ('bandwidth_overflow')
19         EndIf
20     endFor
21 endIf
22 endProcedure
```

Algorithme 3.2 - Vérification de la bande passante d'un Bus interne

2. Le cas du Bus qui connecte le CPU avec le port du noeud : pour vérifier la consommation de la bande passante pour ce type de Bus (le cas des Bus1.1 et Bus2.2 de l'exemple 3.4), nous parcourons toutes les connexions logicielles distantes projetées sur ce Bus et nous calculons la somme de leurs bandes passantes. Cette somme est calculée de la même manière que le cas précédent.
3. Le cas du Bus externe : il relie les noeuds du système. Nous calculons la

somme des bandes passantes des connexions distantes projetées sur le Bus de la même manière que précédemment sauf que le nombre de connexions maximale pour un type de connexion sera égal au nombre de composants possédant les interfaces requises.

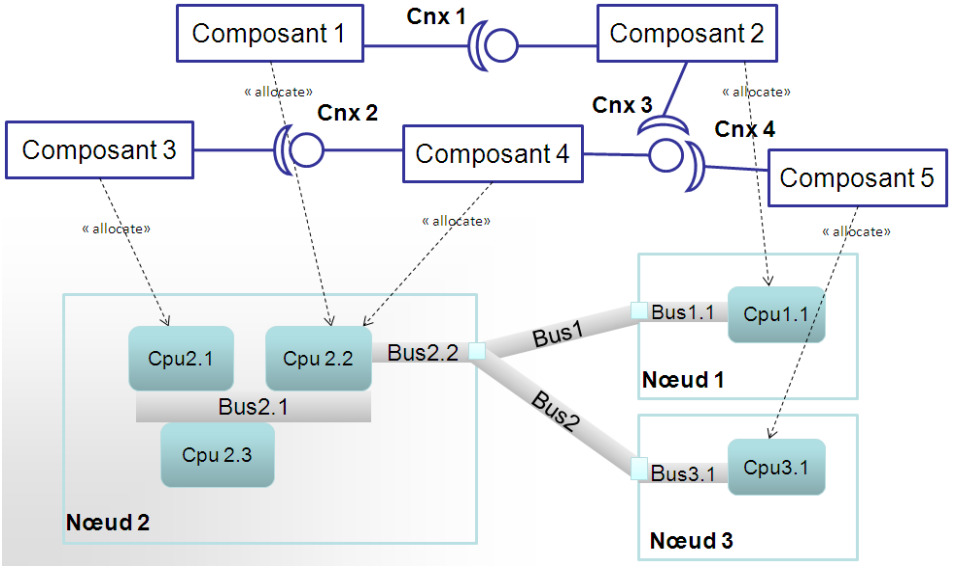

Figure 3.4 – Une architecture logicielle allouant une architecture matérielle

Pour garantir une vérification des systèmes reconfigurables, cet algorithme doit être appliqué au WCEI de tous les metaModes du système.

3.3.2 Les propriétés temporelles

Respect des échéances des tâches

Pour vérifier le respect des échéances des tâches, nous avons déterminer le WCEI de la même manière que les propriétés de ressources (section 3.3.1). Puis, nous avons utilisé l'algorithme d'ordonnancement RMS [25] et le framework Cheddar [34] comme outil de vérification. L'adaptation du WCEI en entrée de Cheddar est de la même manière que celle utilisé pour la vérification de la consommation CPU (section 3.3.1).

Le respect des échéances consiste à vérifier, après l'ordonnancement des tâches, que le temps de réponse de chaque tâche ne dépasse pas la valeur de son échéance.

Absence d'interblocage et absence de famine

Nous consacrons cette section pour la vérification des deux dernières propriétés : absence d'interblocage et de famine afin d'assurer un bon fonctionnement des RTES

distribués dynamiquement reconfigurables.

Il existe des formalismes et des outils comme les vérificateurs de modèles permettant la vérification de ces propriétés. L'entrée de ces outils est une spécification d'une configuration du système ayant un nombre fini d'états. Dans notre cas, on doit vérifier toutes les configurations de chaque metaMode du système. Mais, comme le nombre de ces configurations est indéfini, c'est difficile de les parcourir et de les vérifier une par une.

Dans le cas des propriétés de ressources, nous avons arrivé de déterminer une instance, qui est le WCEI. La validité du WCEI implique la validité de toutes les configurations du metaMode. Mais, pour ces deux propriétés (absence d'interblocage et de famine), nous ne pouvons pas définir une configuration capturant le pire cas d'exécution d'un metaMode. Pour cela, nous avons adopté une autre approche. C'est une approche qui permet de garantir que ces propriétés soient correctes par construction. Elle impose des règles qui doivent être respectées par le concepteur lors de la modélisation de son application.

Dans ce cadre, nous nous sommes inspirés du profil Ravenscar [7, 23] pour définir un ensemble de contraintes qui doit être respecté lors de la construction des RTES distribués dynamiquement reconfigurables.

Le profil Ravenscar

Le profil Ravenscar [7, 8, 23] est dédié aux systèmes temps réel nécessitant une grande sûreté de fonctionnement. Il présente un ensemble de restrictions qui limitent l'usage du langage Ada ou Java pour garantir une analyse statique d'ordonnancement et l'absence d'interblocage et de famine.

Pour garantir une analysabilité statique des tâches selon RMA (Rate Monotonic Analysis), Ravenscar recommande l'utilisation des tâches cycliques. Le déclenchement d'un cycle est effectué soit par un événement temporel (c'est le cas des tâches périodiques), soit par un événement extérieur tout en spécifiant un temps minimal entre deux événements successives (c'est le cas des tâches sporadiques). Ravenscar interdit l'utilisation des tâches qui peuvent être déclenchées à un instant inconnu.

Dans tout système, les tâches interagissent suite à la concurrence des ressources partagées, l'échange de données et la nécessité de synchroniser leurs activités. L'utilisation incontrôlée de ces interactions peut conduire à un certain nombre de problèmes comme l'interblocage et la famine.

Pour garantir l'absence de ces problèmes, Ravenscar recommande une communi-

cation asynchrone entre les tâches. Elles n'interagissent pas directement, mais cette interaction doit être via des objets. Ces objets doivent être protégés à la concurrence de manière que deux tâches ne peuvent pas accéder à l'objet simultanément. Ainsi un problème de blocage peut survenir lors de l'attente des tâches pour accéder aux objets protégés. Pour remédier ce problème, Ravenscar impose l'utilisation du protocole PCP (Priority Ceilling Protocol) [33]. Ceci garantit l'absence d'interblocage, l'absence de famine.

Les règles inspirées du Ravenscar

Dans notre travail, puisque nous traitons trois types de tâches : périodique, sporadique et apériodique, nous émulons les tâches apériodiques à des tâches périodiques comme il est décrit dans la section 3.3.1. Aussi, la communication entre les tâches doit être assurée par des objets protégés en utilisant le protocole PCP.

Les contraintes de Ravenscar sont établies et prouvées que pour des applications locales. Comme nous nous sommes intéressés aux systèmes distribués, la non fiabilité des couches de transport rend le temps de construction et d'envoi des messages indéterministe. Ceci peut engendrer la perte des messages. Pour cela, nous proposons l'utilisation d'une couche de transport fiable comme SpaceWire [12]. Ainsi, nous pouvons considérer les applications distribuées comme des applications locales. Donc, nous garantissons le maintien d'absence d'interblocage et de famine aux systèmes distribués.

3.4 Conclusion

Dans ce chapitre, nous avons présenté notre approche proposée pour la vérification des propriétés non fonctionnelles des systèmes embarqués temps réel distribués dynamiquement reconfigurables. Cette approche sera validée par une étude de cas présentée dans le chapitre suivant.

Chapitre 4

Réalisation

4.1 Introduction

Dans les deux chapitres précédents, nous avons présenté le framework de modélisation ainsi que notre approche proposée pour la vérification des propriétés non fonctionnelles des RTES distibués dynamiquement reconfigurables. Dans ce chapitre, nous allons présenter les différentes technologies de développement utilisées pour mettre en œuvre notre approche. Ensuite, nous allons présenter la description du framework de modélisation et enfin la réalisation et la description du framework de vérification.

4.2 Technologies de développement

Pour bien concevoir un framework de modélisation et de vérification, nous avons choisi de travailler avec l'environnement Eclipse. C'est un éditeur qui s'adapte mieux à notre contexte de développement puisqu'il fournit la possibilité de faire des extensions tout en bénéficiant des plugins existants.

4.2.1 Environnement de développement : Eclipse

Eclipse[1] est un environnement de développement et de modélisation générique, ouvert et extensible. Il est portable et il présente un environnement de développement intégré IDE (Integrated Development Environment) conçu pour fournir une plateforme modulaire pour la réalisation des développements informatiques. La spécificité d'Eclipse vient du fait que son architecture est totalement développée autour de la

1. ECLIPSE : http ://www.eclipse.org/downloads/

notion de plugin. Son architecture est composée de trois niveaux : Plateforme Eclipse, JDT, PDE.

1. **Plateforme Eclipse** :

   La plateforme Eclipse constitue une infrastructure de programmation indépendante de tout langage. Eclipse utilise énormément le concept de modules plugin dans son architecture. D'ailleurs, hormis le noyau de la plateforme nommé "Runtime", tout le reste de la plate-forme est développé sous la forme de plugins. Ce concept permet de fournir un mécanisme d'extension de la plateforme.

2. **JDT**

   JDT (Java Developpement Tools) est un environnement de développement intégré pour Java, qui utilise les services de la Plateforme : éditeur de codes, gestion des versions, compilation incrémentale. Ces outils dotent la plateforme d'un environnement de développement intégré java complet. Il présente la partie la plus visible du projet Eclipse, car c'est l'environnement qui s'ouvre quand nous lançons Eclipse. Il propose un IDE déjà utilisable pour le développement java professionnel, avec une documentation complète.

3. **PDE**

   Le PDE (Plugin Developpement Environment) enrichit la JDT pour offrir un support fonctionnel de développement des plugins. Il propose un ensemble d'outils pour rendre le développement de plugins Eclipse plus facile. Grace à PDE on peut indiquer les ressources nécessaires à l'exécution, définir les points d'extension, associer des fichiers XML aux plugins pour permettre une validation des ressources.

4.2.2 Langage de programmation

JAVA

Le langage JAVA est la base de l'implantation de notre travail. Ce langage de programmation a été introduit par l'entreprise Sun Microsystems qui a été aquise en 2009 par la société Oracle. L'avantage majeur de ce langage est sa portabilité sur plusieurs systèmes d'exploitation. En outre, Java représente un environnement de programmation très développé comprenant de nombreuses bibliothèques standards. Il se base sur le déploiement d'une machine virtuelle qui est indépendante du support

physique. De ce fait, nous avons adopté ce langage comme outil de développement de nos plugins.

XMI

Le XML Metadata Interchange (XMI) [14] est un standard de l'OMG permettant l'échange des metadonnées via des balises XML (eng. Extensible Markup Language). Il définit comment des balises XML sont employées pour représenter un modèle MOF (eng. Meta Object Facility) [13]. Il peut être alors utilisé pour toutes meta-données dont le meta-modèle peut être exprimé en MOF.

En outre, XMI est un format standard de représentation de modèles UML qui permet l'échange d'informations. Il permet de représenter les spécifications du langage UML et de les valider en se basant sur le langage XML. Il essaie d'employer un langage basé sur des balises extensibles pour représenter des objets et leurs associations.

4.2.3 API

JDOM [2] est une API du langage Java développée indépendamment de Sun Microsystems. Elle permet de manipuler des donnés XML plus simplement qu'avec les API classiques. Son utilisation est pratique pour tout développeur Java et repose sur les API XML de Sun.

JDOM utilise DOM pour manipuler les éléments d'un Document Object Model spécifique (créé grâce à un constructeur basé sur SAX). DOM est une spécification du W3C qui propose une API permettant de construire, de parcourir et de mettre à jour un document XML. Le principal rôle de DOM est de fournir une représentation d'un document XML sous la forme d'un arbre d'objets et d'en permettre la manipulation.

4.3 Description du framework de modélisation

Dans le cadre d'un projet de fin d'étude, nous avons développé un framework de modélisation. Ce framework offre un éditeur graphique qui permet la modélisation et la spécification des RTES distribués dynamiquement reconfigurables selon le profil RCA4RTES. Dans cette section, nous décrivons cet éditeur graphique et l'application du profil RCA4RTES.

2. JDOM : http ://www.jdom.org/

4.3.1 Description de l'éditeur graphique

Notre éditeur graphique de modélisation est présenté sous forme de deux plugins Eclipse. Ces plugins sont exportés en fichiers archives JAR. Le concepteur peut les utiliser en les intégrant dans Eclipse avec ses dépendances. Deux nouveaux éditeurs de diagrammes sont crées : le diagramme d'états-transitions (Figure 4.1) et le diagramme de composant (Figure 4.2). Ainsi, l'utilisateur peut spécifier son système en utilisant une palette qui offre les éléments nécessaires.

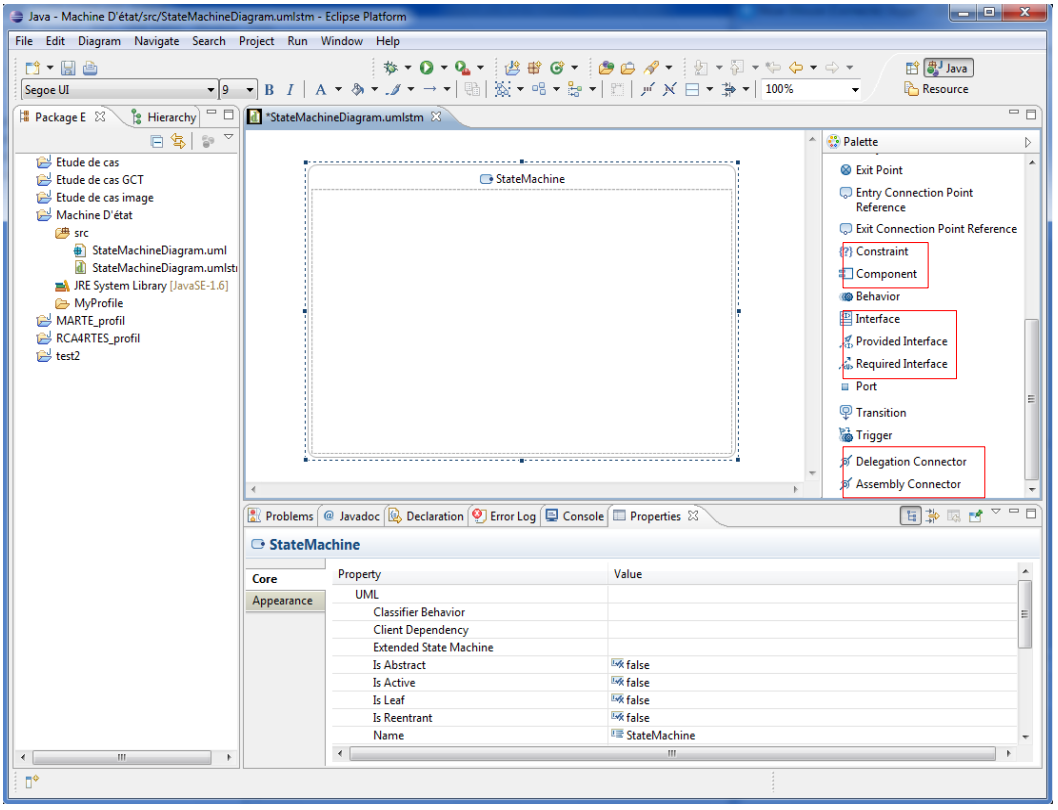

Figure 4.1 – Editeur graphique du diagramme d'états-transitions

4.3.2 Intégration du profil RCA4RTES

Pour utiliser le profil RCA4RTES, le concepteur doit appliquer les différents stéréotypes du profil aux éléments de ses diagrammes. Le stéréotype s'ajoute automatiquement dans le dessin graphique (Figure 4.3). Nous citons comme exemples de stéréotypes : `MetaMode`, `StructuredComponent`, `StructuralConstraint`, `AllocationConstraint`, `MetaModeTransition`, etc.

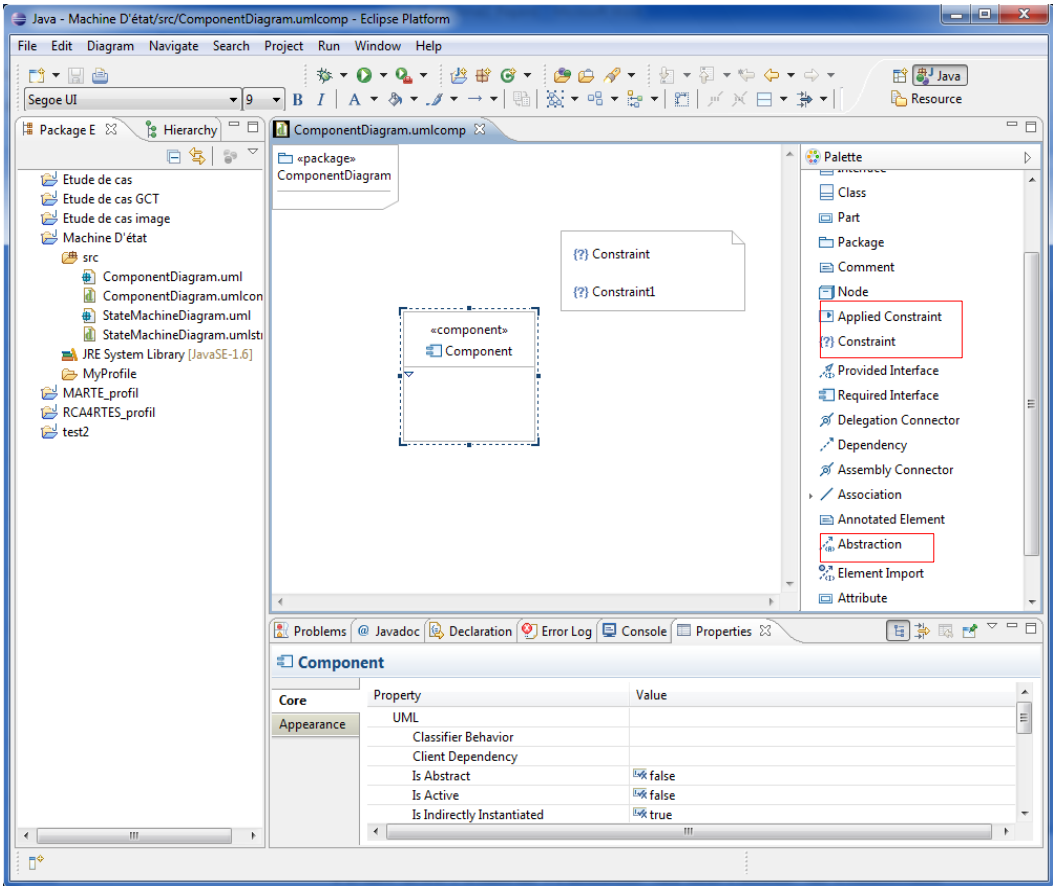

Figure 4.2 – Editeur graphique du diagramme de composant

4.4 Framework de vérification

4.4.1 L'entrée au framework de vérification

En utilisant notre framework de modélisation, le concepteur peut spécifier des systèmes embarqués temps réel distribués dynamiquement reconfigurables. Lors de cette phase, un fichier XMI sera généré automatiquement pour chaque diagramme. A chaque modification des diagrammes, le fichier XMI sera aussi modifié. Ce fichier comporte tous les informations concernant la spécification d'une application (les metaModes, les composants logiciels et matériels, les contraintes structurelles et les contraintes d'allocation, etc).

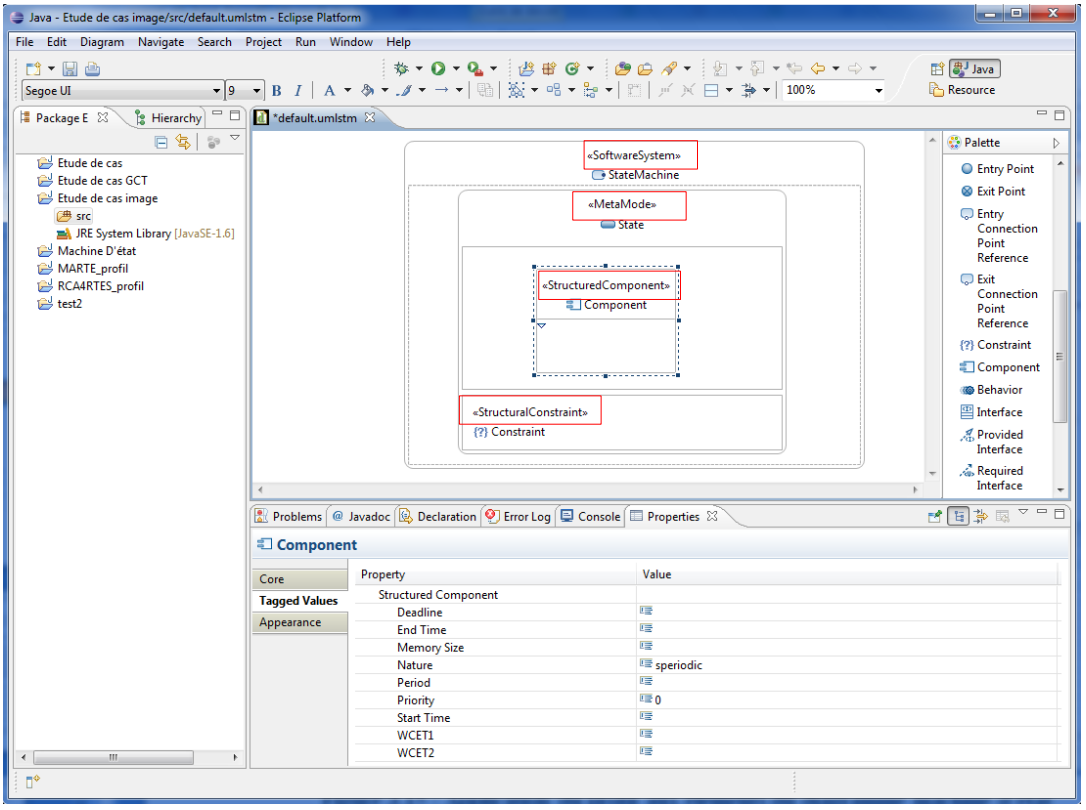

Figure 4.3 – Application du profil aux éléments du diagramme états-transitions

4.4.2 Réalisation du framework

Consommation CPU et respect des échéances

Nous avons utilisé le framework Cheddar comme outil de simulation d'ordonnancement temps réel pour vérifier la consommation CPU et le respect des échéances. Pour cela, nous avons intégré cet outil dans notre framework de vérification. Comme nous visons à élaborer un framework avec une interface ergonomique et conviviale, l'exécution de Cheddar doit être en arrière plan. De ce fait, seulement le résultat sera affiché sans avoir besoin d'accéder à l'interface graphique du Cheddar.

Pour mettre en œuvre ceci, nous avons développé un petit programme en Ada (voir annexe C). L'interface du framework Cheddar est décrite dans un package "call_framework.ads". Il contient tous les services fournis par le framework. Donc, notre programme fait appel à ce package pour se servir des fonctionnalités de Cheddar.

L'exécution de ce programme génère un exécutable (cheddar.exe) qui permet d'effectuer la simulation sans accéder à une interface graphique. L'entrée de cet

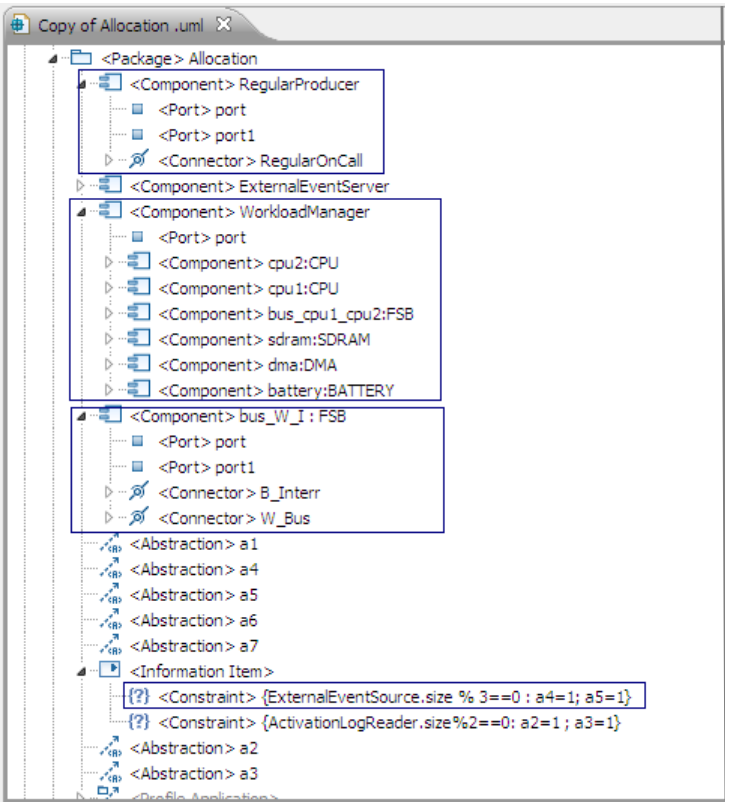

Figure 4.4 – Structure du fichier XMI correspondant à l'allocation

exécutable est un fichier XML (listing 4.1) comportant lensemble des tâches à ordonnancer. Pour celà, nous devons adapter notre fichier XMI, généré lors de la phase de modélisation, en un fichier XML pour qu'il respecte le format du listing 4.1. Ainsi, nous avons utilisé JDOM pour éffectuer cette adaptation.

```
1  <?xml version="1.0" standalone="yes"?>
2  <?xml-stylesheet type="text/xsl" href="cheddar_project.xsl"?>
3  <cheddar>
4  <processors>
5    <processor>
6      <name>InteruptionSimulator_Cpu</name>
7      <scheduler> RATE_MONOTONIC_PROTOCOL </scheduler>
8    </processor>
9  </processors>
10 <tasks>
11   <task task_type="PERIODIC_TYPE" >
12     <cpu_name>InteruptionSimulator_Cpu</cpu_name>
13     <address_space_name>IS</address_space_name>
14     <name>ExternalEventSource1</name>
15     <capacity> 2</capacity>
```

```
16      <start_time> 0</start_time>
17      <deadline> 5000</deadline>
18      <blocking_time> 0</blocking_time>
19      <priority> 1</priority>
20      <period> 5000</period>
21    </task>
22  </tasks>
23  </cheddar>
```

Listing 4.1 – Exemple d'un fichier XML (entrée à Cheddar)

Suite au lancement de "cheddar.exe", un deuxième fichier XML sera généré comportant le résultat obtenu. Ce résultat sera affiché dans l'interface de notre framework. Le listing 4.2 décrit un exemple d'un fichier XML généré par Cheddar.

```
1  <results>
2      <scheduling_feasibility>
3          <title>Scheduling feasibility, Processor <processor>
             InteruptionSimulator_Cpu</processor></title>
4          1) Feasibility test based on the processor utilization factor :
5          <line>- The base period is<base_period> 5000</base_period></line>
6          <line>-<unit> 4992</unit> units of time are unused in the base
             period.</line>
7          <line>- Processor utilization factor with period is <period_factor
             >0.00160</period_factor></line>
8          <line>- In the preemptive case, with RM, the task set is
             schedulable because the processor utilization factor <
             factor_bound>0.00160</factor_bound> is equal or less than <
             factor_bound>1.00000</factor_bound></line>
9      </scheduling_feasibility>
10     <scheduling_simulation>
11         <title>Scheduling simulation, Processor <processor>
             InteruptionSimulator_Cpu</processor> :</title>
12         <line>- Number of preemptions : <number_of_preemption> 0</
             number_of_preemption></line>
13         - Task response time computed from simulation :
14         <task>ExternalEventSource => 8/worst </task>
15         <task>ActivationLogReader => 6/worst </task>
16         - No deadline missed in the computed scheduling : the task set
             seems to be schedulable.
17     </scheduling_simulation>
18 <results>
```

Listing 4.2 – Exemple d'un fichier XML (Sortie de Cheddar)

Algorithme de vérification de la consommation mémoire

Nous avons implanté l'algorithme de vérification de la consommation mémoire en utilisant le langage de programmation JAVA. Les entrées à cet algorithme sont extraites du fichier XMI. Nous avons utilisé aussi JDOM pour parcourir ce fichier et avoir les données nécessaires à l'éxécution de notre algorithme. Parser un fichier XML revient à le transformer en une arborescence JDOM. Le listing 4.3 présente un exemple pour récupérer des éléments d'un fichier XMI. En suivant cette méthode, nous avons extrait les données nécessaires pour la vérification de la consommation mémoire : les metaMode du système, les tâches de chaque metaMode, les contraintes structurelles et les noeuds du systèmes avec leurs tailles mémoires.

```
1  public class JDOM{
2     static org.jdom.Document document;
3     static Element racine;
4     public static void main(String[] args){
5        //On crée une instance de SAXBuilder
6        SAXBuilder sxb = new SAXBuilder();
7        try{
8        //On crée un nouveau document JDOM avec en argument le fichier XML
9        document = sxb.build(new File("fichier.xmi"));}
10       catch(Exception e){}
11       //On initialise un nouvel élément racine avec l'élément racine du document.
12       racine = document.getRootElement();
13       getAll();}
14    static void getAll(){
15          //On crée une List contenant tous les noeuds "processeur" de l'Element racine
16          List listProcesseur = racine.getChildren("processor");
17           Iterator i = listProcesseur.iterator();
18          while(i.hasNext()){
19               Element courant = (Element)i.next();}
20          }
21 }
```

Listing 4.3 – Explorer un fichier XMI

L'implantation de notre algorithme de vérification de la consommation mémoire est présenté par le listing 4.4.

```
1  public class MemoryVerification {
2    public List<Node> nodes=new ArrayList<Node>();//ensemble des nodes du
       système
3    private List<Task> tasksP, tasksSp, tasksAp;//ensemble des tâches d'un node à
       vérifier
```

```
4   private boolean esistenceTaskP, esistenceTaskSp, esistenceTaskAp;
5   private float EndPoints[];
6   public void memoryUsage(){
7     for(int i=0;i<nodes.size();i++){
8       float memorySize=nodes.get(i).memorySize;
9       //si toutes les tâches sont périodiques et sporadiques:
10      if(!esistenceTaskAp){
11        if(sumTaskP(tasksP)+sumTaskSp(tasksSp)<= memorySize){
12          System.out.print("Memory usage is verified");
13        }else{
14          System.out.print("Memory usage not verified");}
15      }else{// il existe des tâches apériodiques
16        if (sumTaskP(tasksP)+sumTaskSp(tasksSp)+sumTaskAp(tasksAp)<=
            memorySize){
17          System.out.print("Memory usage is verified");}
18        else{//détecter le chevauchement des tâches apériodiques ayant une allocation
            mémoire maximale
19          EndPoints=ElementaryIntervals(tasksAp);//Les intervalles de temps des
              tâches apériodiques
20          EndPoints=sort(EndPoints);//Les intervalles de temps triés
21          tasksAp=sortTasksAp(tasksAp);//Les tâches apériodiques triées selon les
              dates de début
22          int j=0;
23          float memApMax=0;
24          boolean stop=false;
25          float sumIE=0;//Mémoire allouée durant un intervalle de temps
26          while(j<EndPoints.length-1 && !stop){
27            sumIE=multiOverlap(EndPoints[j],EndPoints[j+1],tasksAp);//
                calculer sumIE pour les tâches chevauchées durant un intervalle de temps
28            if(sumIE>memApMax){
29              memApMax=sumIE;
30              //S'il existe que des tâches apériodiques
31              if(esistenceTaskAp&&!esistenceTaskP&&!esistenceTaskSp){
32                if(memApMax>memorySize){
33                  stop=true;
34                  System.out.println("Memory usage is not verified");}
35              }
36              //S'il coexiste des tâches périodiques et/ou sporadiques
37              if(esistenceTaskAp&&(esistenceTaskP||esistenceTaskSp)){
38                if(memApMax+sumTaskP(tasksP)+sumTaskSp(tasksSp)>memorySize){
39                  stop=true;
40                  System.out.println("Memory usage is not verified");}
41              }
```

```
42              }
43              j++;}
44            if(stop==false){
45              System.out.println("Memory usage is verified");}
46          }
47        }
48      }
49    }
50  }
```

Listing 4.4 – Algorithme de la vérification mémoire en JAVA

Algorithme de vérification de la consommation de la bande passante

Nous avons implanté notre algorithme de vérification de la consommation de la bande passante des Bus en utilisant le langage de programmation JAVA. L'implantation de cet algorithme est illustré dans le listing 4.5. Les entrées à cet algorithme sont définies aussi dans le fichier XMI. Nous avons exploiter ce fichier avec JDOM de la même manière que dans la section précédente. Les données nécessaires à l'éxécution de cet algorithme sont : l'ensemble des metaModes du système, les tâches de chaque metaMode, les noeuds du système avec les CPUs et les Bus associés, les contraintes structurelles et les contraintes d'allocation.

```
1  public class BandwidthVerification {
2    public List<Node> nodes=new ArrayList<Node>();
3    public List<ExternBus> buses=new ArrayList<ExternBus>();
4    public List<Connection> conections=new ArrayList<Connection>();
5    private List<Connection> distantsConections=new ArrayList<Connection>();
6    public void BandwidthUsage(){
7      classifyConnection(conections);//classifier les connexions distantes et locales
8      //vérifier la bande passante des Bus de chaque noeud
9      for (int i=0;i<nodes.size();i++){
10       //vérifier la bande passante de chaque Bus reliant les CPUs
11       for(int j=0;j<nodes.get(i).busesInterCpus.size();j++){
12         verificationForLocalConnection(nodes.get(i).busesInterCpus.get(j));
13                 //si le Bus est de type Access
14                 if(nodes.get(i).busesInterCpus.get(j).Access==true){
15                     verificationForDistantConnection(nodes.get(i).
                         busesInterCpus.get(j));}}//vérifier la bande passante des
                         connexions distantes mappées sur ce Bus
16       //verifier la bande passnate des Bus reliant les CPUs avec les ports des noeuds
17       for(int j=0;j<nodes.get(i).busesCpusNode.size();j++){
```

```
18          verificationOfBusCpuToNode(nodes.get(i).busesCpusNode.get(j));}
19      }
20      //vérifier la bande passante des Bus reliant les noeuds du système
21      for(int j=0;j<buses.size();j++){
22        verificationOfExternBus(buses.get(j));}
23    }
24  }
```

Listing 4.5 – Algorithme de la vérification de la bande passante en JAVA

Le listing 4.6 présente l'implantation de la procédure `verificationOfExternBus(ExternBus bus)`. Cette procédure permet de vérifier la bande passante d'un Bus qui relie des noeuds du système.

```
1  void verificationOfExternBus(ExternBus bus){
2      float SumConnection=0; //Somme des connexions mappées sur le Bus
3      for(int i=0;i<distantsConections.size();i++){
4        Task providedTh=distantsdConections.get(i).providedThread;
5        Task requiredTh=distantsdConections.get(i).requiredThread;
6        //si la tâche fournie et la tâche requise sont allouées sur des noeuds reliés par le Bus
7        if(bus.nodes.contains(providedTh.node) && bus.nodes.contains(
           requiredTh.node)){
8          SumConnection +=distantsConections.get(i).bandwidth * requiredTh.
             nbInstance;}
9        if(SumConnection>bus.bandwidth){
10         System.out.println("Bandwidth overflow");
11         System.exit(0);}
12     }
13     System.out.println("Bandwidth usage is verified");}
14 }
```

Listing 4.6 – Une procédure permettant de vérifier la bande passante des Bus reliant les noeuds en JAVA

4.4.3 Réalisation d'un plugin

Notre framework de vérification est un plugin Eclipse sous forme de menu. Un menu est défini par un ensemble de commandes (eng. commands). La déclaration d'une commande est le lien entre au moins un menu et un comportement (eng. handler). La mise en place d'une commande est illustré dans la figure 4.5. Une commande crée est associée à une catégorie. Elle appartient à un menu et elle est définie par un raccorci clavier (eng. key binding) et un comportement décri dans une

classe qui sera éxécuté après le clic sur la commande.

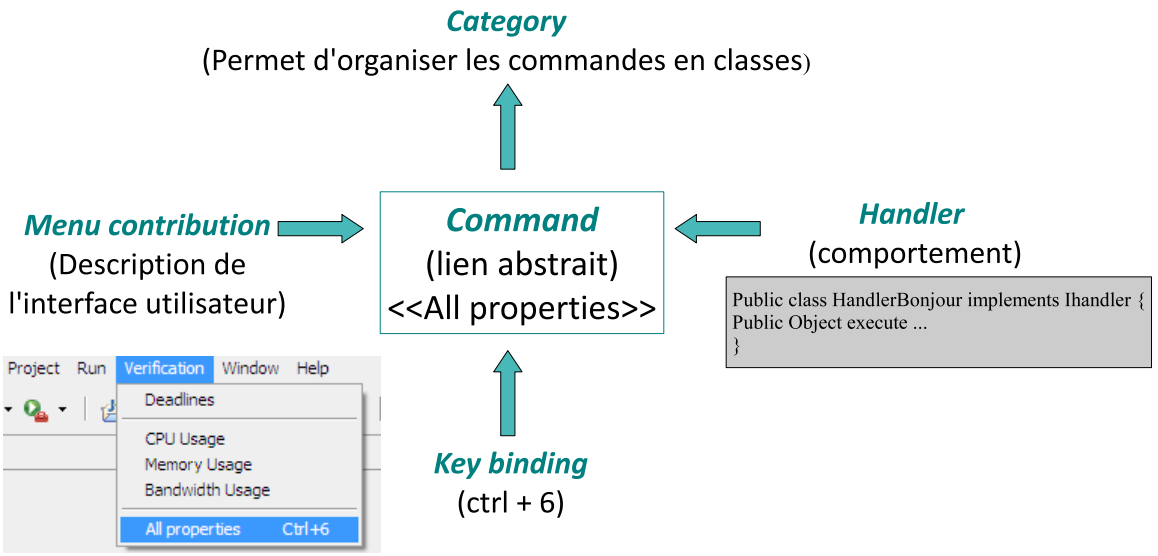

Figure 4.5 – Mise en place d'une commande

4.4.4 Description du framework de vérification

Notre framework de vérification est présenté sous forme d'un plugin Eclipse. Ce plugin est exporté en fichier archive JAR. L'utilisateur peut vérifier une propriété en sélectionnant une commande du menu "Verification" (*Deadlines*, *CPU usage*, *Memory usage*, *bandwidth usage*). Il peut aussi vérifier toutes ces propriétés en choisissant la commande *All properties* du menu "Verification". Il peut aussi vérifier ces propriétés en utilisant les icônes de la barre d'outils (figure 4.6).

4.4.5 Conclusion

Dans ce chapitre, nous avons présenté un framework de modélisation des RTES dynamiquement reconfigurables offrant un éditeur graphique et un profil UML sous Eclipse. Ensuite, nous avons détaillé les différentes étapes de développement et de création d'un framework de vérification des propriétés non fonctionnelles de ces systèmes.

Afin de valider notre approche, nous allons présenter dans le chapitre suivant une étude de cas de gestion de charges de travail.

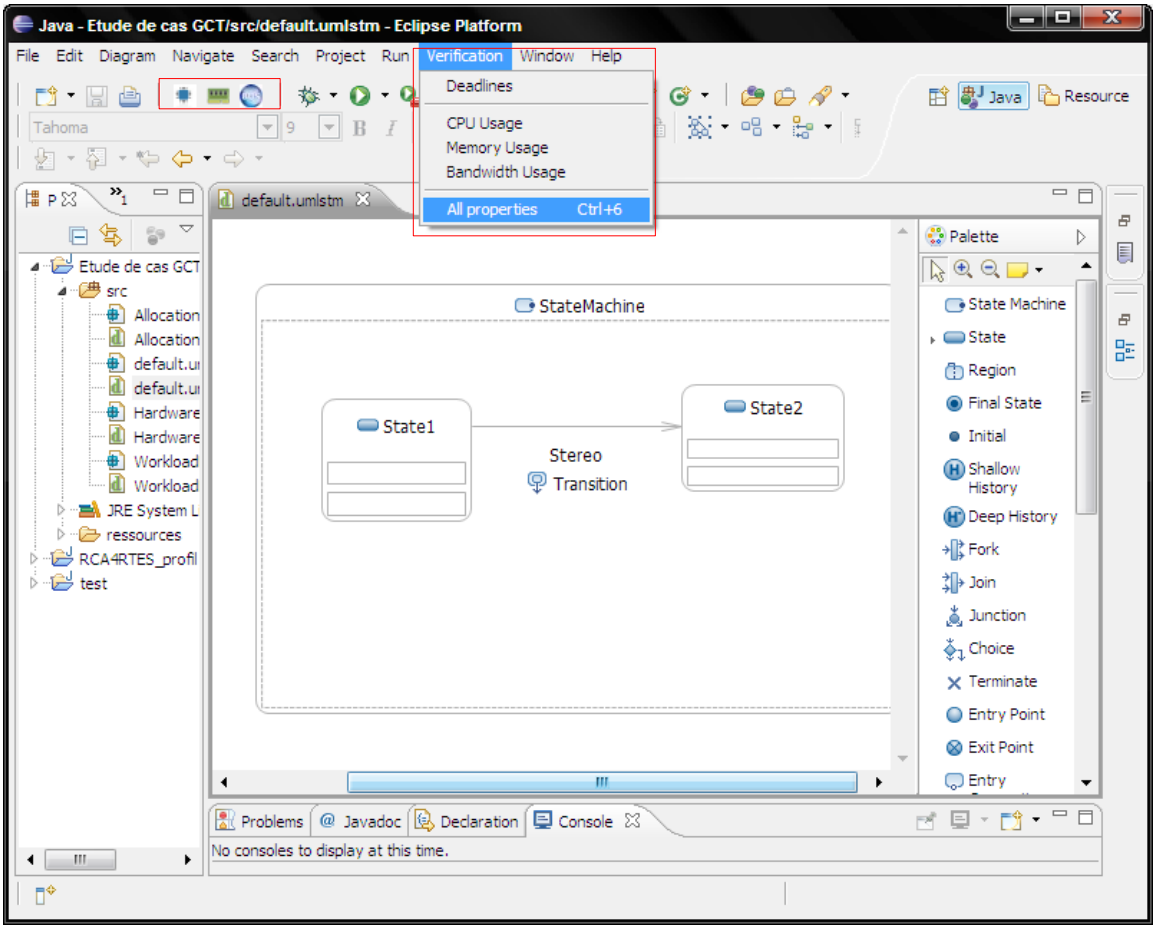

Figure 4.6 – Description du framework de vérification

Chapitre 5

Étude de cas

5.1 Introduction

Dans les chapitres précédents, nous avons présenté une approche théorique permettant la vérification des propriétés non fonctionnelles des RTES distribués dynamiquement reconfigurables. Dans ce chapitre, nous présentons dans un premier temps une étude de cas de gestion de charge de travail (eng. Workload Manager) pour valider notre approche : nous la modélisons tout en suivant le processus proposé dans le chapitre 2. Dans un second temps, nous appliquons les différents outils proposés pour vérifier les propriétés non fonctionnelles et nous présentons les résultats obtenus.

5.2 Etude de cas : Gestion de charge de travail

Cette application [8] représente un système de gestion de charge de travail. Il s'agit d'un processus périodique pouvant gérer des charges de travail variables. Les travaux réguliers sont effectués par un processus périodique de haute priorité. Les travaux supplémentaires (irréguliers) sont délégués à un processus sporadique de moindre priorité. Par ailleurs, le système est capable de recevoir des interruptions venant de l'extérieur. Ces interruptions sont reçues par un processus de très haute priorité et sont enregistrées dans un tampon spécifique. Le traitement de ces interruptions est délégué à un processus sporadique de très faible priorité qui est réveillé de temps en temps par le processus périodique principal.

Cette application comporte deux nœuds (`WorkLoadManager` et `interruptionSimulator`). Le processus Workload_Manager reçoit des inter-

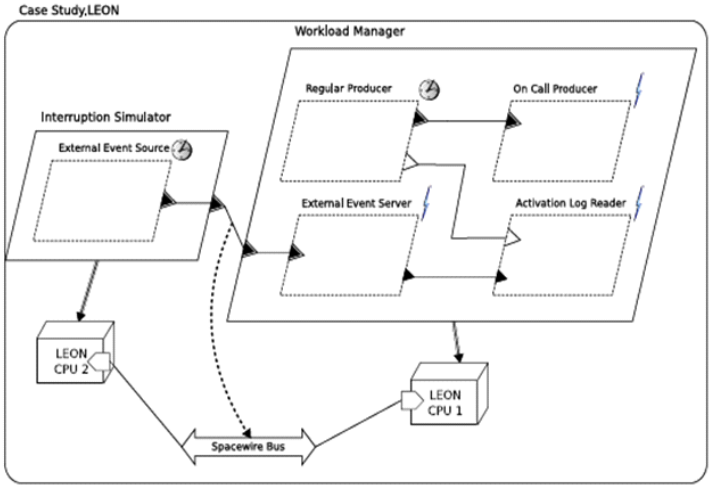

Figure 5.1 – Gestion de charge de travail

ruptions extérieures. Ces interruptions sont envoyées de manière aléatoire sous forme de messages par un second processus (Interruption_Simulator). Chacun des deux processus s'exécute sur un nœud. Ils sont reliés par l'intermédiaire d'un Bus SpaceWire [12].

5.3 Framework de modélisation

Dans cette section, nous présentons la modélisation du système de gestion de charges de travail pour pouvoir par la suite vérifier ses propriétés non-fonctionnelles en se basant sur cette modélisation.

5.3.1 Etape1 : Spécification de la partie logicielle

Cette étude de cas comporte trois metaModes :

- Le metaMode `Insecure Workload Manager` : le système permet d'accomplir les commandes régulières et de répondre aux interruptions externes d'une manière non sécurisée.
- Le metaMode `Secure Workload Manager` : le système permet d'accomplir les commandes régulières et de répondre aux interruptions externes d'une manière sécurisée.
- Le metaMode `Intern Workload Manager` : le système permet seulement d'ac-

Table 5.1 – Les propriétés non fonctionnelles des composants

Structured Component	Nature	Period Deadline	WCET1 Processor 1Ghz	WCET2	Priority	Memory Size
Regular Producer	periodic	1000ms	498ms	2ms	7	50MB
On Call Producer	sporadic	1000ms	200ms	2ms	5	25MB
Activation Log Reader	sporadic	1000ms	125ms	4ms	3	20MB
External Event Server	sporadic	5000ms	2ms	0	11	50MB
External Event Source	sporadic	5000ms	2ms	0	11	50MB

complir les commandes régulières.

Nous définissons ces trois metaModes et les transitions entre eux via le diagramme états-transitions comme illustré dans la figure 5.2. Par exemple, la transition du metaMode insecure vers le metaMode secure est déclenchée quand l'utilisateur demande que l'exécution du système se fasse en mode sécurisé.

Ensuite, nous définissons les politiques de reconfiguration qui aident à choisir une des configurations du metaMode cible. Par la suite, nous décrivons chaque metaMode défini via un diagramme de composant. Par manque d'espace, nous détaillons, dans la suite, le metaMode `Insecure Workload Manager` seulement.

Le MetaMode `Insecure Workload Manager` :

La figure présente le metaMode `Insecure workload Manager`. Il est défini par cinq composants.

- Regular Producer : effectue la charge de travail régulière. Il délègue, sous des conditions spécifiques, la charge de travail supplémentaire et le traitement des interruptions extérieures à d'autres processus légers.
- On Call Producer : effectue la charge de travail supplémentaire.
- Activation Log Reader : effectue une quantité de travail correspondant au traitement de la dernière interruption reçue.
- External Event Server : effectue la réception des interruptions extérieures et leur enregistrement dans un tampon spécifique.
- External Event Source : présente le source des interruptions extérieures.

Le diagramme de composant est accompagné par des contraintes structurelles (définies dans la figure 5.3) et des caractéristiques pour chaque composant (définies dans le tableau 5.1).

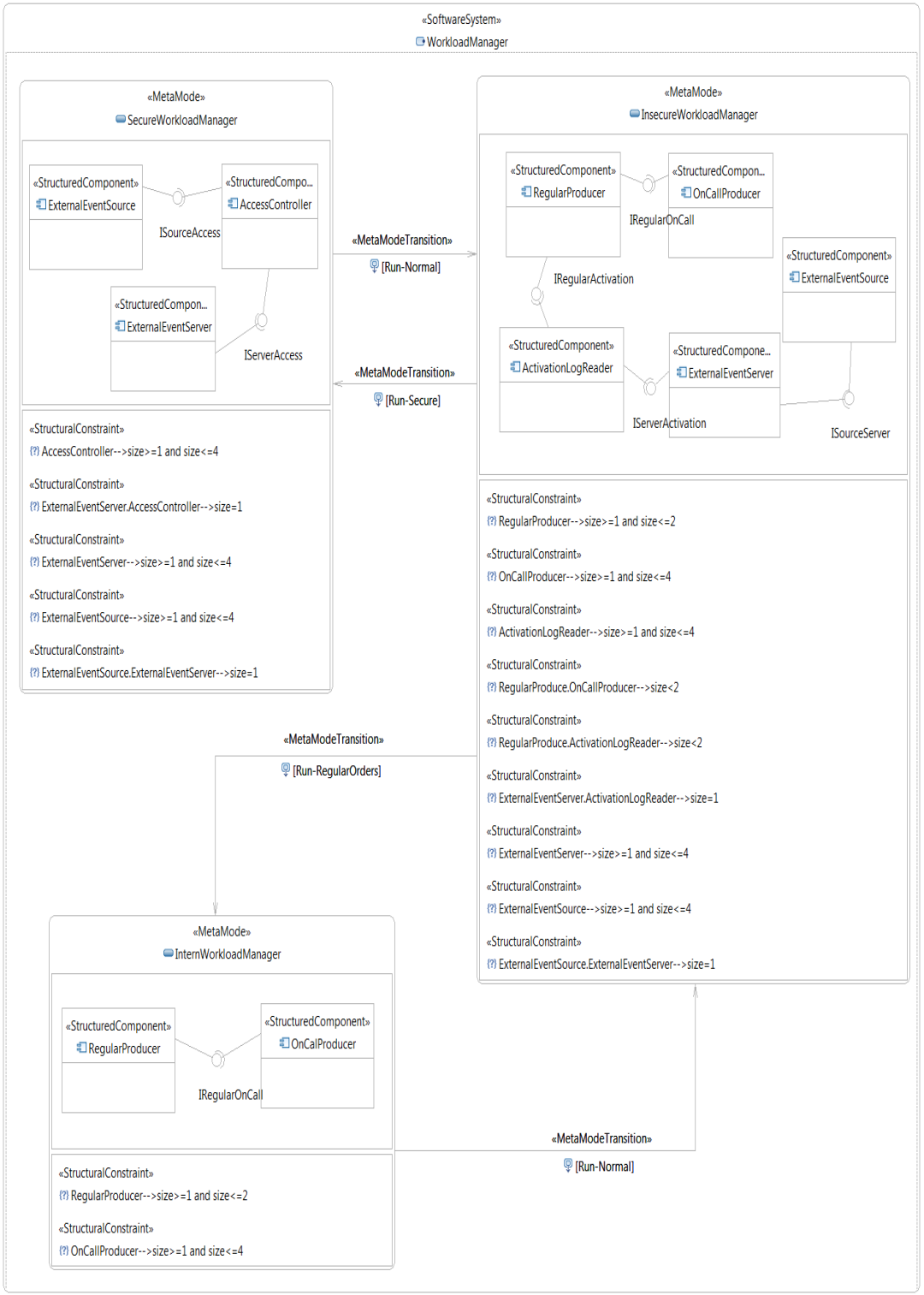

Figure 5.2 – Le diagramme états-transitions

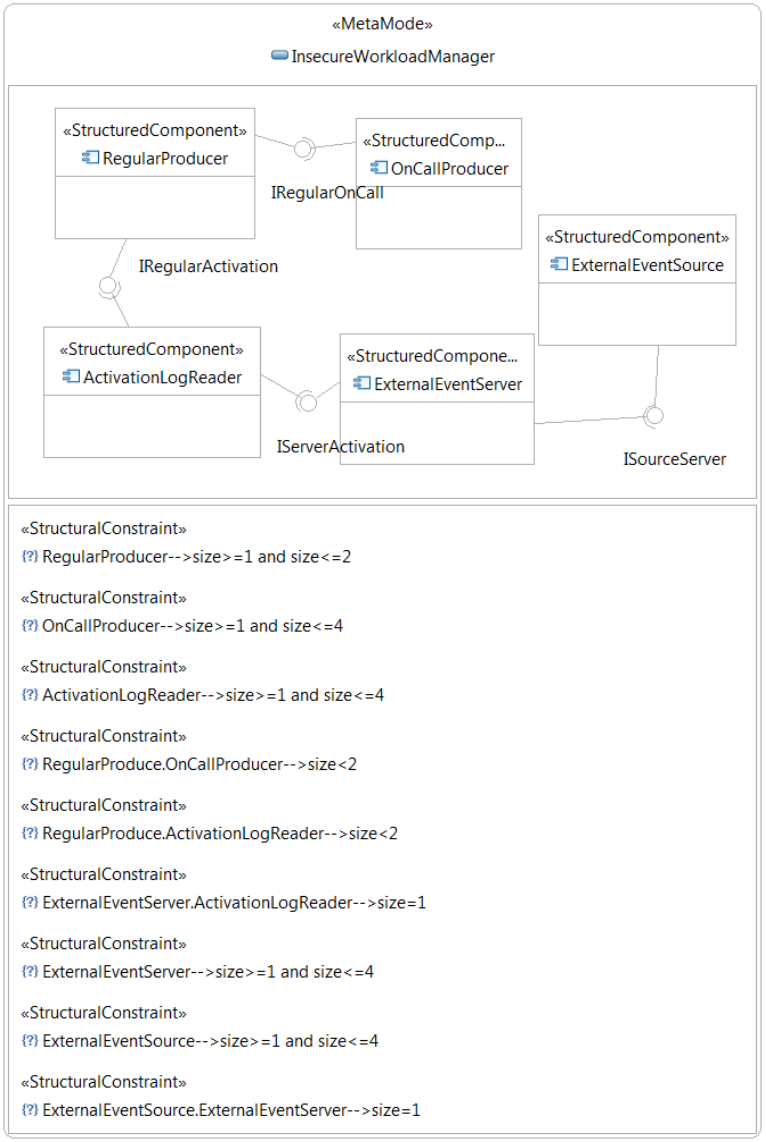

Figure 5.3 – Le MetaMode `Insecure Workload Manager`

5.3.2 Eatpe2 : Spécification de la partie matérielle

Après avoir spécifié la partie logicielle de notre application, nous passons à la spécification de la partie matérielle. Puisqu'il s'agit d'une application distribuée, nous décrivons l'architecture de déploiement (des nœuds et des liens entre eux) via le diagramme de déploiement. Dans notre cas, nous avons deux nœuds liés par un Bus SpaceWire.

Le nœud `workload Manger` possède deux processurs (CPU1 avec une fréquence de 800MHZ et CPU2 avec une fréquence de 600MHZ), une RAM de taille 500MB, un DMA, une batterie et un Bus avec une bande passante de 150 b/s pour assurer la communication entre ces composants (figure 5.4). Le nœud `Interruption Simulator` possède un seul CPU avec une fréquence de 800MHZ, une RAM de taille 250MB, un DMA et un Bus avec une bande passante de 110 b/s (figure 5.5).

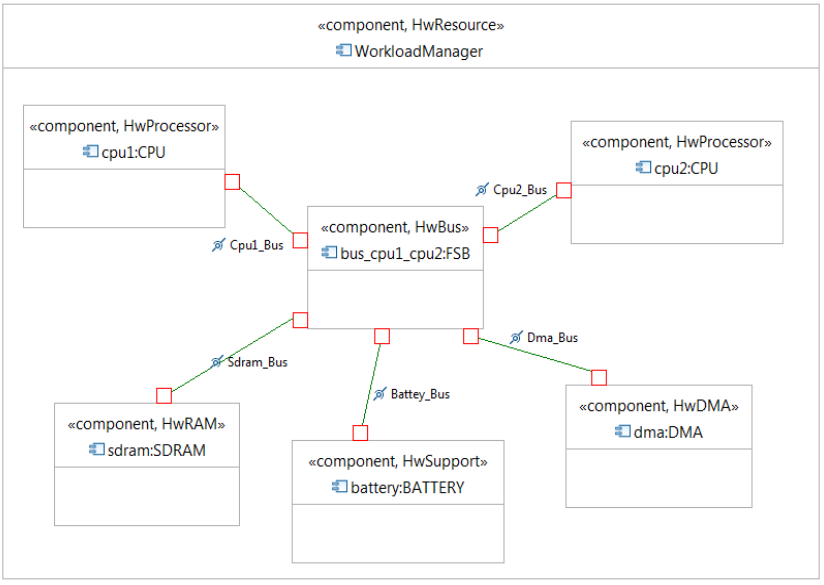

Figure 5.4 – L'architecture matérielle du noeud `workLoad Manager`

5.3.3 Etape3 : Allocation de la partie logicielle sur la partie matérielle

Notre MetaMode `Insecure Workload Manager` devrait être alloué sur les ressources matérielles (Figure 5.6). Pour spécifier cette allocation, nous avons appliqué le stéréotype Allocate et le stéréotype AllocationConstraint du profil RCA4RTES. Les contraintes d'allocation décrivent les politiques d'allocation des modèles (Me-

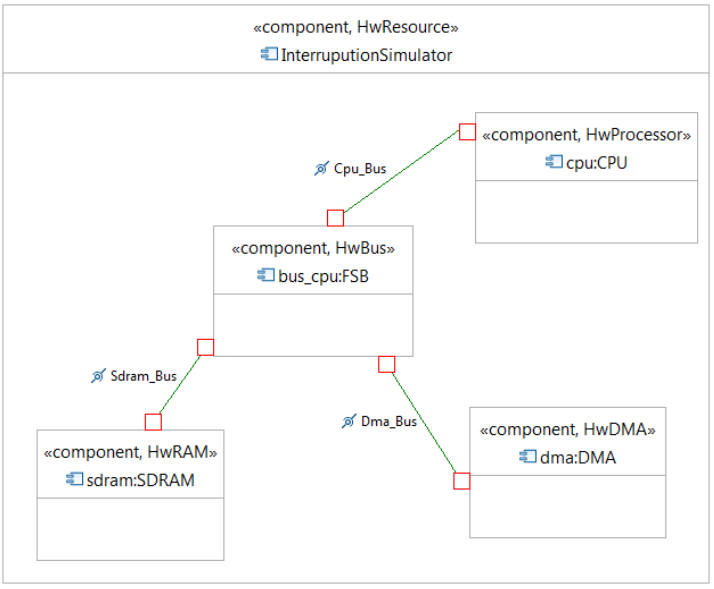

Figure 5.5 – L'architecture matérielle du noeud `Interruption Simulator`

taModes) sur l'architecture matérielle. Par exemple, les instances du composant `ActivationLogReader` sont allouées sur les deux processeurs CPU1 et CPU2 de la ressource matérielle `WorkloadManager`. La moitié de ces instances sera allouée sur le CPU1 et l'autre moitié sur le CPU2. Ceci est traduit par la contrainte d'allocation « `ActivationLogReader.size%2==0 : a1=1 ;a3=1` »

Après la spécification de notre étude de cas, nous utilisons notre framework de vérification pour vérifier l'ensemble des propriétés non fonctionnelles.

5.4 Framework de vérification

La spécification du système est enregistrée dans un fichier XMI qui sera l'entrée à notre framework de vérification. Les propriétés non fonctionnelles sont testées pour chaque metaMode du système.

5.4.1 La consommation CPU et le respect des échéances

À partir de la modélisation de notre étude de cas, notre framework de vérification fait appel au simulateur Cheddar pour vérifier la consommation CPU et le respect des échéances. Pour chaque metaMode du système, notre framework parcourt tous les CPUs du système (cpu1 :CPU et cpu2 :CPU du noeud `Workload Manager` et cpu :CPU du noeud `Interruption Simulator`). En premier temps, notre framework

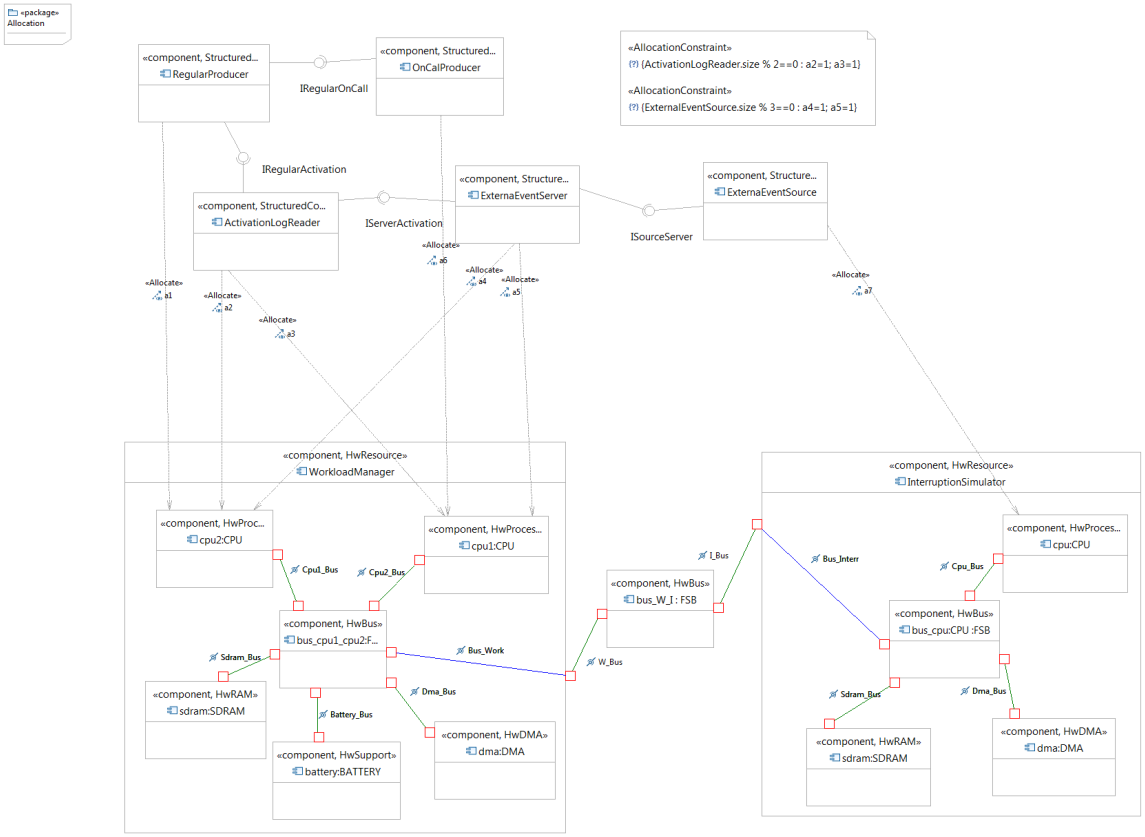

Figure 5.6 – Allocation du metaMode Insecure work load Manager sur le hardware du nœud workload Manager et Interruption Simulator

vérifie que le taux d'utilisation de chaque CPU est inférieur à la borne calculée. En deuxième temps, il vérifie que toutes les tâches allouant le CPU respectent leurs échéances. Le résultat obtenu est illustré dans la figure 5.7 et la figure 5.8.

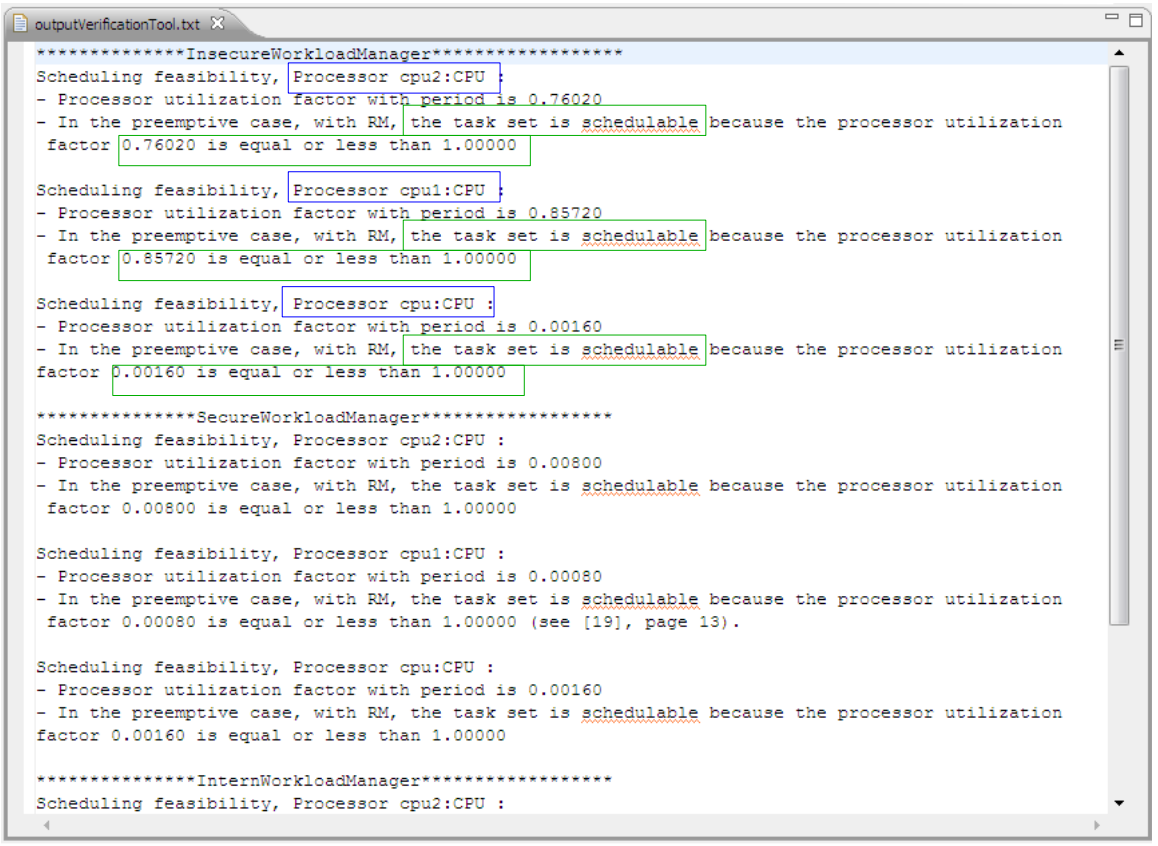

Figure 5.7 – Vérification de la consommation CPU

5.4.2 La consommation mémoire

En utilisant notre algorithme de vérification mémoire, notre framework permet de vérifier cette propriété. Pour chaque metaMode du système, notre framework parcourt les noeuds du système et vérifie que la mémoire allouée par les tâches ne dépasse pas leur taille mémoire. Le résultat obtenu dans la figure 5.9 illustre que cette propriété est vérifiée pour les deux metaModes `InsecureWorkloadManager` et `InterWorkloadManager`. Cependant, notre framework détecte que cette propriété n'est pas vérifiée au niveau du noeud `workloadManager` durant le metaMode `SecureWorkloadManager`.

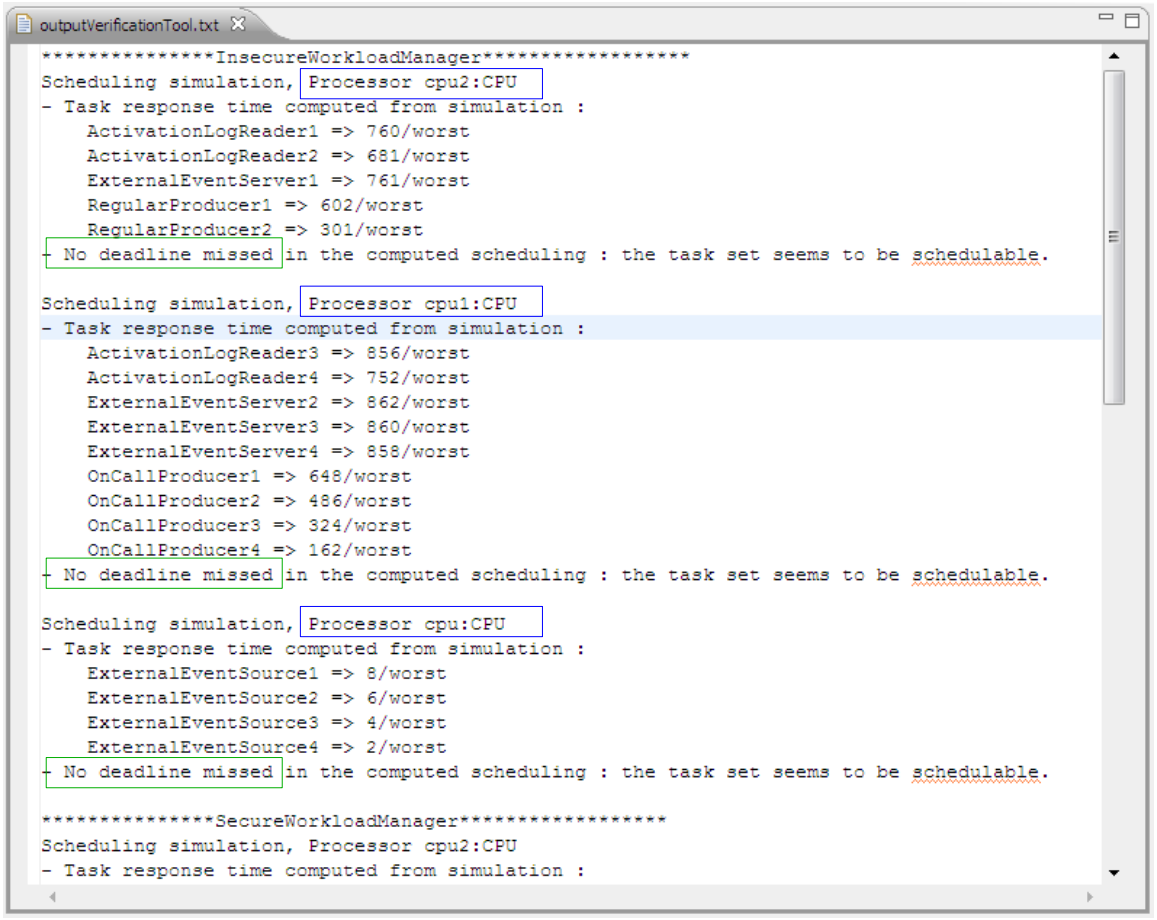

Figure 5.8 – Vérification du respect des échéances

5.4.3 La consommation de la bande passante

En utilisant notre algorithme de vérification de la bande passante, notre framework permet de vérifier cette propriété. Pour chaque metaMode du système, notre framework parcourt les Bus de tous les noeuds du système (bus_cpu1_cpu2 et bus_cpu), ainsi que les Bus reliant les noeuds (bus_W_I). Puis, il vérifie que la bande passante des tâches mappées sur chaque Bus ne dépasse pas sa bande passante. Le résultat obtenu dans la figure 5.10 illustre que cette propriété est vérifiée pour notre étude de cas.

5.5 Conclusion

À travers l'étude de cas de gestion de charge de travail, nous avons pu tester notre approche et mettre en évidence le processus de développement des RTES

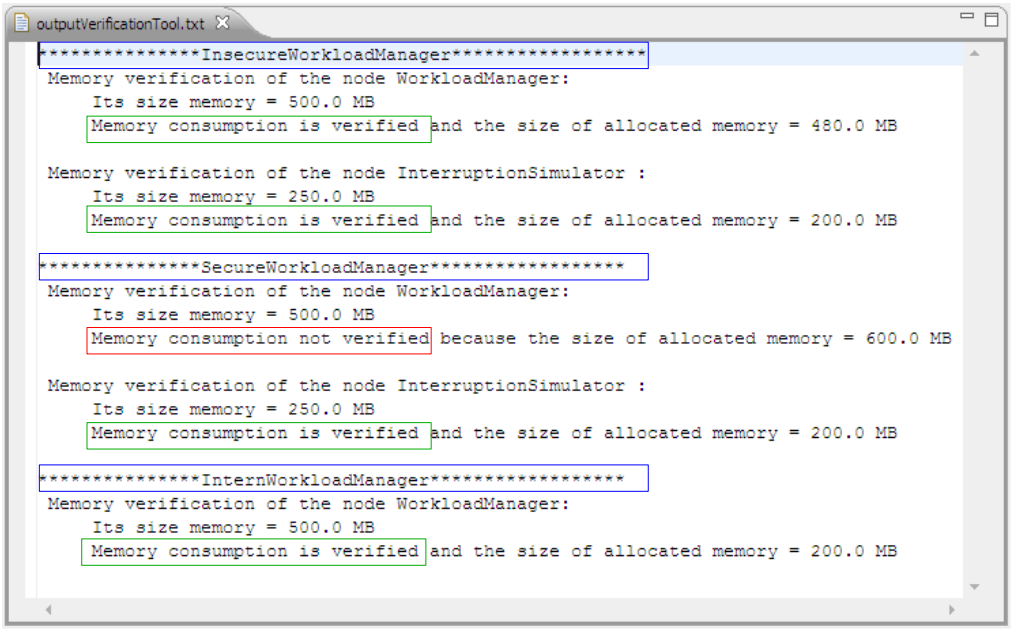

Figure 5.9 – Vérification de la consommation mémoire

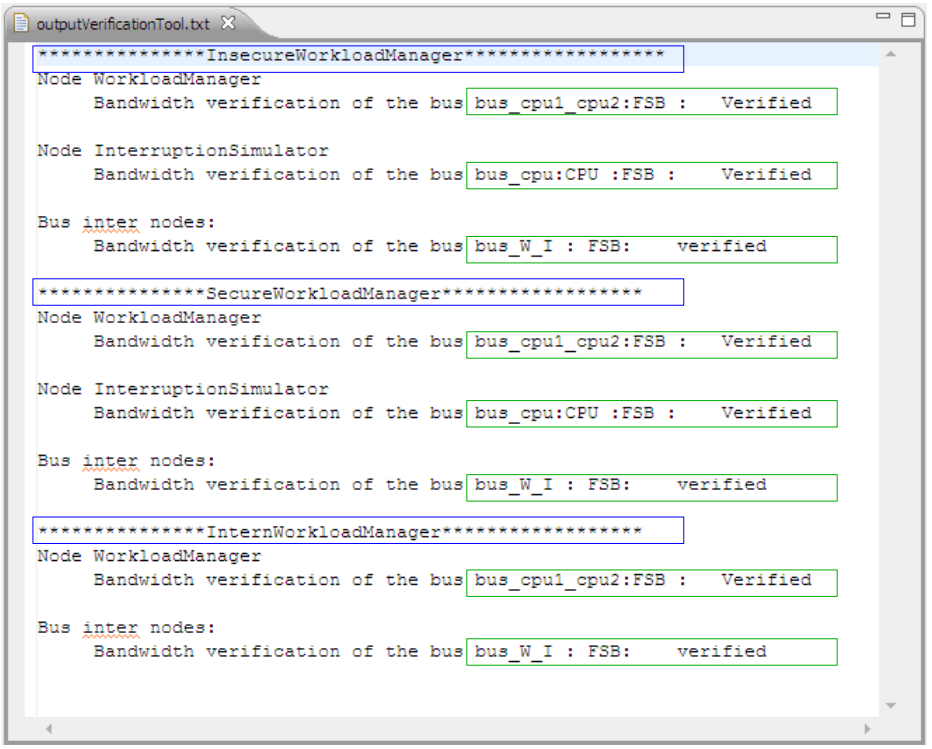

Figure 5.10 – Vérification de la consommation de la bande passante

distribués dynamiquement reconfigurables suivant le profil RCA4RTES. Nous avons

modélisé tout d'abord notre système en utilisant notre framework de modélisation. Puis, nous avons vérifié certaines propriétés non fonctionnelles (consommation CPU, respect des échéances, consommation mémoire, consommation de la bande passante) en utilisant notre framework de vérification élaboré.

Conclusion générale et perspectives

En guise de conclusion, nous avons proposé dans le cadre de nos travaux de mastère un framework de modélisation et de vérification destiné aux RTES distribués dynamiquement reconfigurables. Ce framework offre d'une part un éditeur graphique permettant la modélisation. D'autre part, il offre un outil permettant la vérification des propriétés non-fonctionnelles de ces systèmes.

Le framework de vérification proposé assure une vérification des propriétés temporelles (le respect des échéances, l'absence d'interblocage et l'absence de famine) ainsi que des propriétés de ressources (la consommation CPU, la consommation mémoire et la consommation de la bande passante). Ce framework combine différents formalismes pour assurer la vérification de ces propriétés. Il utilise les algorithmes d'ordonnancement et le simulateur Cheddar pour la vérification de la consommation CPU et le respect des échéances. Il fait recourt à deux algorithmes que nous avons développés pour vérifier la consommation mémoire et la consommation de la bande passante. Pour l'absence d'interblocage et de famine, nous avons garantit le respect de ces deux propriétés dès la construction du système puisque nous avons imposé un ensemble de contraintes inspiré du profil Ravenscar.

Notre framework permet la vérification des propriétés non fonctionnelles pour des systèmes dynamiquement reconfigurables. Ce processus de vérification est appliqué niveau modélisation. À ce niveau, la vérification est plus facile et efficace. Par contre celle niveau implantation est très difficile en terme de temps et de coût.

Comme perspective, notre framework peut être étendu pour supporter la vérification d'autres propriétés. Nous visons à améliorer notre framework de manière qu'il permet de donner des propositions dans le cas où les propriétés non fonctionnelles ne sont pas vérifiées. Ces propositions aident le concepteur à corriger la modélisation de son système pour qu'il respecte ses propriétés non fonctionnelles.

Nous envisageons également, à moyen terme, de mettre en œuvre tout le pro-

cessus de développement automatisé basé sur une approche dirigée par les modèles pour les RTES dynamiquement reconfigurables. Nous visons donc à générer le code à partir des modèles. La génération du code est appliquée que si les propriétés non-fonctionnelles sont vérifiées.

Annexe A

Algorithme : Vérification mémoire

```
1  Begin
2  If (isAllPeriodicAnd/OrSporadicThread()) then
3      if (allocatedMemorySize(periodicThreads,sporadicThreads)>MemorySize)
         then
4          Exit(”Memory_consumption_is_not_verified”);
5      else Exit (”Memory_consumption_is_verified”);
6      endIf
7  else//existence of aperiodic thread
8      if (allocatedMemorySize(periodicThreads,sporadicThreads,
         aperiodicThreads)<= MemorySize) then
9          Exit (”Memory_consumption_is_verified”);
10     else
11         createElementaryIntervals(aperiodicThreads);
12         MaxMemory=0;
13         for each elementary interval i do
14         sumEI=getAllocatedMemorySize(i);
15         if sumEI> MaxMemory then
16             MaxMemory=sumEI;
17             if (isAllAperiodicThread()) then
18                 if (MaxMemory>MemorySize) then
19                     Exit(”Memory_consumption_not_verified”);
20                 endIf
21             else
22                 if (MaxMemory+ allocatedMemorySize(periodicThreads,
                     sporadicThreads)>MemorySize) then
23                     Exit(”Memory_consumption_not_verified”);
24                 endIf
25             endIf
26         endIf
```

```
27          endFor
28          Exit("Memory consumption is verified");
29      endIf
30  endIf
31  End
```

Algorithme A.1 - Vérification mémoire

Annexe B

Algorithme : Vérification de la consommation de la bande passante

```
1  //vérifier la bande passante des Bus de chaque noeud
2  Begin
3  For each node of the system
4      //vérifier la bande passante de chaque Bus reliant les CPUs
5      for each BusCpuToCpu
6          verificationBW(Bus)
7      endFor
8      //vérifier la bande passante de chaque Bus reliant le CPU avec le port du noeud
9      for each BusCpuToNode
10         if(SumBw(CPU)>BandWidth(BusCpuToNode)) then
11             Exit('bandwidth overflow')
12         endIf
13     endFor
14     //vérifier la bande passante des Bus reliant les noeuds du système
15     for each BusNodeToNode
16         verificationOfExternBus(Bus)
17     endFor
18 endFor
19 End
20
21 float Function SumBw(CPU)
22 SumBW=0 // la somme des bandes passante des connexions mappées sur le lien physique
     reliant le CPU avec le port du noeud
23 For each distributed software connection type
```

```
24      if(requiredTh isAllocatedOn(CPU)) then
25          sumBW = sumBW + (BWcnxType*nbRequiredTh)
26          // nbRequiredTh est le nombre des tâches requises allouant le CPU
27      else if(providedTh isAllocatedOn(CPU)) then
28          sumBW = sumBW + (BWcnxType*min(nbRequiredTh,(nbProvidedTh*nbCnxMax)
              ))
29          // nbProvidedTh est le nombre des tâches fournies allouant le CPU
30      endIf
31  endFor
32  return SumBW
33
34  Procedure verificationBW(Bus)
35  SumBW=0 // la somme des bandes passantes des connexions mappées sue le Bus
36  For each couple of cpu connected by the Bus
37      For each software connection type
38          sumBW = sumBW + (BWcnxType*min(nbRequiredTh,(nbProvidedTh*nbCnxMax)
              ))
39          // la somme est appliquée seulement pour les cnxType entre le couple du CPUs
               courant
40          if (sumBW > BandWidth(Bus)) then
41              Exit('bandwidth_overflow')
42          EndIf
43      EndFor
44  EndFor
45  If(getType(Bus)=AccessType) then
46      //le Bus peut relier les CPUs avec les ports du noeud
47      For each CPU(i) isLinkedToThePortOfNode(Bus)
48          SumBW=+ SumBw(CPU(i))
49          //sumBW(CPU(i)) est une fonction qui retourne la bande passante des connexions
               distantes ayant des composants allouant le CPU(i)
50          if (sumBW > BW(Bus)) then
51              Exist ('bandwidth_overflow')
52          EndIf
53      endFor
54  endIf
55  endProcedure
56
57  Procedure verificationOfExternBus(Bus)
58  SumBW=0 // la somme des bandes passantes des connexions mappées sue le Bus
59  For each distant software connection type
60      sumBW = sumBW + (BWcnxType*nbRequiredTh)
61      // la somme est appliquée seulement pour les cnxType entre les deux noeuds connectés
           par le Bus
```

```
62      if (sumBW > BandWidth(Bus)) then
63          Exist ('bandwidth_overflow')
64      EndIf
65  EndFor
66  endProcedure
```

Algorithme B.1 - Vérification de la consommation de la bande passante

Annexe C

Programme Ada : Intègration de Cheddar

```
1  procedure IntegratedCheddar is
2     Sys : System;
3     Response_List : Framework_Response_Table;
4     Request_List : Framework_Request_Table;
5     A_Request : Framework_Request;
6     A_Param : Parameter_ptr;
7     Dir : Unbounded_String_List;
8     //The XML file to read
9     Input_File : Boolean := False;
10    Input_File_Name : Unbounded_String;
11    //The XML file to write
12    Output_File : Boolean := False;
13    Output_File_Name : Unbounded_String;
14    //The duration on which we must compute the scheduling
15    Period_String : Unbounded_String;
16    Period : Natural;
17    Ok : Boolean;
18    procedure Usage is
19    begin
20       Put_Line ("Usage : IntegratedCheddar [switch] period");
21       Put_Line ("   switch can be :");
22             Put_Line("-o file-name, write the scheduling into the XML file  file-
                 name ");
23       Put_Line("-i file-name, read the system to analyze from the XML file  file-
           name ");
24    end Usage;
25 BEGIN
```

```
26    //Get arguments
27    loop
28       case GNAT.Command_Line.Getopt ("i: o:") is
29          when ASCII.NUL =>
30             exit;
31
32          when 'i' =>
33             Input_File := True;
34             Input_File_Name := To_Unbounded_String (GNAT.Command_Line.
                 Parameter);
35
36          when 'o' =>
37             Output_File := True;
38             Output_File_Name := To_Unbounded_String (GNAT.Command_Line.
                 Parameter);
39
40          when others =>
41             Usage;
42             OS_Exit (0);
43       end case;
44    END LOOP;
45    loop
46       declare
47          S : constant String := GNAT.Command_Line.Get_Argument (
              Do_Expansion => True);
48       begin
49         exit when S'Length = 0;
50 ␣␣␣␣␣␣␣Period_String := Period_String & S;
51 ␣␣␣␣␣␣end;
52 ␣␣␣␣end loop;
53 ␣␣␣␣//Check the period on which we will compute the scheduling
54 ␣␣␣␣To_Natural (Period_String, Period, Ok);
55 ␣␣␣␣if not Ok then
56 ␣␣␣␣␣␣␣␣Raise_Exception
57 ␣␣␣␣␣␣␣␣(Constraint_Error'Identity, "The period to compute the scheduling must be
     a numeric value");
58     END IF;
59     if (not Input_File) or (not Output_File) then
60        Usage;
61        OS_Exit (0);
62     end if;
63    Call_Framework.Initialize (False);
64    Put("<results>");
```

```
65    //Test on periodic tasks
66    Read_From_Xml_File (Sys, dir, Input_File_Name);
67    Initialize (Response_List);
68    Initialize (Request_List);
69    Initialize (A_Request);
70    A_Request.Statement := Scheduling_Feasibility_Basics;
71    Add (Request_List, A_Request);
72    Sequential_Framework_Request(Sys,Request_List,Response_List,Total_Order
        ,Xml_Output);
73    Put(Response_List);
74    Initialize (Response_List);
75    Initialize (Request_List);
76    Initialize (A_Request);
77    A_Request.Statement := Scheduling_Simulation_Time_Line;
78    A_Param := new Parameter (Integer_Parameter);
79    A_Param.Name := To_Unbounded_String ("period");
80    A_Param.Integer_value := Period;
81    Add (A_Request.Param, A_Param);
82    A_Param := new Parameter (Boolean_Parameter);
83    A_Param.Name := To_Unbounded_String ("schedule_with_offsets");
84    A_Param.Boolean_value := True;
85    Add (A_Request.Param, A_Param);
86    A_Param := new Parameter (Boolean_Parameter);
87    A_Param.Name := To_Unbounded_String ("schedule_with_precedencies");
88    A_Param.Boolean_value := True;
89    Add (A_Request.Param, A_Param);
90    A_Param := new Parameter (Boolean_Parameter);
91    A_Param.Name := To_Unbounded_String ("schedule_with_resources");
92    A_Param.Boolean_value := True;
93    Add (A_Request.Param, A_Param);
94    A_Param := new Parameter (Integer_Parameter);
95    A_Param.Name := To_Unbounded_String ("seed_value");
96    A_Param.Integer_value := 0;
97    Add (A_Request.Param, A_Param);
98    Add (Request_List, A_Request);
99    Sequential_Framework_Request(Sys,Request_List,Response_List,Total_Order
        ,Xml_Output);
100   Put (Response_List);
101   Initialize (Request_List);
102   Initialize (Response_List);
103   Initialize (A_Request);
104   A_Request.Statement := Scheduling_Simulation_Basics;
105   Add (Request_List, A_Request);
```

```
106     Sequential_Framework_Request(Sys,Request_List,Response_List,Total_Order
          ,Xml_Output);
107     Put(Response_List);
108     Put("</results>");
109  end IntegratedCheddar ;
```

Listing C.1 – Intègration de Cheddar

Bibliographie

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 1994.

[2] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim Guldstrand Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. Uppaal - now, next, and future. In *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*, MOVEP '00, pages 99–124, London, UK, 2001. Springer-Verlag.

[3] Tobias Amnell, Alexandre David, Elena Fersman, M. Oliver Möller, Paul Pettersson, and Wang Yi. Tools for real-time UML : Formal verification and code synthesis. ECOOP 2001 SIVOES Workshop, 2001.

[4] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times : a tool for schedulability analysis and code generation of real-time systems. In *In Proc. of FORMATS 2003, number 2791 in LNCS*, pages 60–72. Springer-Verlag, 2003.

[5] Franck Barbier, Corine Cauvet, Mourad Oussalah, Dominique Rieu, Sondes Bennasri, and Carine Souveyet. Composants dans l'ingénierie des systèmes d'information : concepts clés et techniques de réutilisation. In *Proceedings of the Information Interaction Intelligence - Actes des 2emes Assises nationales du GdR I3*, pages 95–117, Nancy, France, 2002.

[6] Fateh Boutekkouk and Mohamed Benmohammed. Uml modeling and formal verification of control/data driven embedded systems. In *Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 311–316, Washington, DC, USA, 2009. IEEE Computer Society.

[7] Alan Burns. The ravenscar profile. *Ada Lett.*, XIX :49–52, December 1999.

[8] Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the use of the ada ravenscar profile in high integrity systems. *Ada Lett.*, XXIV :1–74, June 2004.

[9] Luis Alejandro Cortes, Luis Alej, Ro Corts, Petru Eles, and Zebo Peng. Verification of real-time embedded systems using petri net models and timed automata, 2002.

[10] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool kronos. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control : verification and control*, pages 208–219, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[11] Francisco Assis Moreira do Nascimento, Marcio Ferreira da Silva Oliveira, and Flávio Rech Wagner. Formal verification for embedded systems design based on mde. In *Proceedings of the 10th International Embedded Systems Symposium*, pages 159–170, Langenargen, Germany, September 2009. Springer.

[12] ECSS-E-50-12A. Spacewire - links, nodes, routers and networks. European Space Agency, January 2003. Technical report.

[13] Object Management Group. MetaObject Facility. http ://www.omg.org/mof/.

[14] Object Management Group. Xml metadata interchange (xmi), version 2.0. http ://cgi.omg.org/docs/formal/03-05-02.pdf, May 2003.

[15] Roman Gumzej, Matjaz Colnaric, and Wolfgang A. Halang. Temporal feasibility verification of specification pearl designs. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 249–252, Vienna, Austria, May 2004. IEEE Computer Society.

[16] Brahim Hamid and Fatma Krichen. Model-based engineering for dynamic reconfiguration in DRTES. In *ECSA '10 : Proceedings of the Fourth European Conference on Software Architecture. WORKSHOP SESSION : VIII Nordic Workshop on Model-Driven Software Engineering*, pages 269–276, New York, NY, USA, 2010. ACM.

[17] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. A user guide to hytech. In *Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 41–71, London, UK, 1995. Springer-Verlag.

[18] Pao-Ann Hsiung and Shang-Wei Lin. Formal design and verification of real-time embedded software. In *Proceedings of the Second Asian Symposium on Programming Languages and Systems*, pages 382–397, Taipei, Taiwan, November 2004. Springer.

[19] Damir Isovic and Gerhard Fohler. Efficient Scheduling of Sporadic, Aperiodic, and Periodic Tasks with Complex Constraints. In *Proceedings of the 21st IEEE conference on Real-time systems symposium*, pages 207–216, Washington, DC, USA, November 2000. IEEE Computer Society.

[20] Li Jie, Guo Ruifeng, and Shao Zhixiang. The research of scheduling algorithms in real-time system. In *International Conference on Computer and Communication Technologies in Agriculture Engineering*, 2010.

[21] Fatma Krichen. Position paper : Advances in Reconfigurable Distributed Real Time Embedded Systems. In *International workshop on Distributed Architecture modeling for Novel component based Embedded systems (DANCE 2010)*, Tozeur, Tunisie, May 2010. IEEE Computer Society.

[22] Fatma Krichen, Brahim Hamid, Bechir Zalila, and Bernard Coulette. Designing Dynamic Reconfiguration of Distributed Real Time Embedded Systems. In *Proceedings of the 10th annual international conference on New Technologies of Distributed Systems (NOTERE 2010)*, Tozeur, Tunisie, June 2010. IEEE Computer Society.

[23] Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-java : a high integrity profile for real-time java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 131–140, New York, NY, USA, 2002. ACM.

[24] Joseph Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Inf. Process. Lett.*, pages 115–118, 1980.

[25] Chang L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1) :46–61, 1973.

[26] Gabor Madl, Sherif Abdelwahed, and Douglas C. Schmidt. Verifying distributed real-time properties of embedded systems via graph transformations and model checking. *Real-Time Systems*, 33(1–3) :77–100, 2006.

[27] OMG. A UML Profile for MARTE : Modeling and Analysis of Real-Time Embedded systems,Beta 2. http ://www.omgmarte.org/Documents/Specifications/08-06-09.pdf, June 2008.

[28] Martin Ouimet, Guillaume Berteau, and Kristina Lundqvist. Modeling an electronic throttle controller using the timed abstract state machine language and toolset. In *Workshops and Symposia at MoDELS in Software Engineering*, pages 32–41, Genoa, Italy, October 2006. Springer.

[29] Martin Ouimet and Kristina Lundqvist. The tasm toolset : Specification, simulation, and formal verification of real-time systems (tool paper). In *Proceedings of the 19th International Conference on Computer Aided Verification*, pages 126–130, Berlin, Germany, July 2007. Springer.

[30] J.M. Purtilo and C.R. Hofmeister. Dynamic reconfiguration of distributed programs. In *Distributed Computing Systems, 1991., 11th International Conference on*, pages 560 –571, may 1991.

[31] Thomas Reinbacher. Introduction to embedded software veri
cation, 2008.

[32] SAE. Architecture Analysis & Design Language (AADL). http ://www.sae.org/technical/standards/AS5506A, January 2009.

[33] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *IEEE Transactions on Computers*, 39 :1175–1185, 1990.

[34] Frank Singhoff, Jérôme Legrand, L. Nana, and Lionel Marcé. Cheddar : a flexible real time scheduling framework. In *Proceedings of the 2004 annual ACM SIGAda international conference on Ada*, pages 1–8, New York, NY, USA, 2004. ACM.

[35] Farn Wang. Formal verification of timed systems : A survey and perspective. In *Proceedings of the IEEE*, pages 1283–1305. IEEE Computer System, 2004.

[36] Farn Wang, Ieee Computer Society, Geng dian Huang, and Fang Yu. Tctl inevitability analysis of dense-time systems. *IEEE Trans. Softw. Eng*, 2006 :176–187, 2003.

[37] Sergio Yovine. Model checking timed automata. In *In European Educational Forum : School on Embedded Systems*, pages 114–152. Springer-Verlag, 1998.

[38] Tudor Zaharia and Piroska Haller. Formal verification and implementation of real time operating system based applications. In *Proceedings of the 4th International Conference on Intelligent Computer Communication and Processing*, pages 299–302. IEEE Computer System, August 2008.

Vérification des propriétés non fonctionnelles des RTES distribués dynamiquement reconfigurables

Amal GASSARA

الخلاصة: إن إعادة التشكيل الديناميكية للأنظمة المضمنة الآنية الموزعة يمكن أن تؤدي إلى تعطيل النظام واختلال بعض الخصائص الغير الوظيفية. ولذلك فمن الضروري التحقق من صحة هذه الخصائص لهذه الأنظمة. في عملنا، اقترحنا إطار عمل يمكّن من التحقق من بعض الخصائص الزمنية و خصائص الموارد انطلاقا من نماذج محددة. يجمع هذا الإطار مختلف الأدوات و الشكليات.

المفاتيح: الأنظمة المضمنة الآنية، إعادة التشكيل الديناميكية، الموزعة، التحقق، الخصائص الغير الوظيفية

Résumé : La reconfiguration dynamique des systèmes embarqués temps réel distribués (RTES) peut provoquer le mal fonctionnement du système et la perturbation de certaines propriétés non fonctionnelles. Donc, il s'avère nécessaire de vérifier la validité des propriétés non fonctionnelles pour ces systèmes. Dans notre travail, nous avons proposé un framework qui permet la vérification de certaines propriétés temporelles et de ressources à partir des modèles définis. Ce framework combine différents outils et formalismes.

Mots clés: RTES, reconfiguration dynamique, distribués, vérification, propriétés non fonctionnelles

Abstract: Dynamic reconfiguration of distributed real-time embedded systems (RTES) can cause the non-preservation of some non-functional properties. So, it is necessary to check the maintain of these properties. In our work, we proposed a framework that allows the verification of a set of temporal and resource properties at design level. This framework combines different tools and formalisms.

Key-words: RTES, dynamic reconfiguration, distributed, verification, non-functional properties