Contribution à la considération du comportement des plates-formes d'exécution logicielles temps réel

Cédrick Lelionnais

Thèse de Doctorat

Cédrick LELIONNAIS

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'École centrale de Nantes
sous le label de l'Université de Nantes Angers Le Mans

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et Applications
Unité de recherche : Institut de Recherche en Communications et Cybernétique de Nantes (IRCCyN)

Soutenue le 2 juillet 2014

Contribution à la considération du comportement des plates-formes d'exécution logicielles temps réel

JURY

Président :	**M. Patrick MARTINEAU**, Professeur, Polytech'Tours, France
Rapporteurs :	**M. Jean-Philippe BABAU**, Professeur, Université Bretagne Occidentale, France **M. Patrick MARTINEAU**, Professeur, Polytech'Tours, France
Examinateurs :	**M. Sébastien GÉRARD**, Docteur, CEA Saclay Nano-INNOV, France **M. Marc PANTEL**, Maître de conférences, ENSEEIHT-IRIT, France **M. Jérôme DELATOUR**, Enseignant Chercheur, Groupe ESEO, France
Directeur de thèse :	**M. Olivier H. ROUX**, Professeur, École Centrale de Nantes, France

Remerciements

Je tiens en premier lieu à remercier Jérôme Delatour qui m'a co-encadré tout au long de cette thèse. Sa clairvoyance, son soutien, son altruisme et son expertise ont grandement contribué à la concrétisation de ce travail. Je le remercie aussi de m'avoir fait confiance et de m'avoir porté jusque là.

Je remercie également mon directeur de thèse, Olivier H. Roux, pour ses précieux conseils et son orientation précise tout au long de notre recherche. Son savoir m'a permis de prendre les bonnes décisions.

J'adresse aussi mes remerciements à Jean-Philippe Babau et à Patrick Martineau qui ont accepté de juger ce travail en qualité de rapporteurs. Leurs retours pertinents ont été très constructifs pour l'avenir de cette contribution. Merci également à Sébastien Gérard et à Marc Pantel d'avoir consenti à faire partie de mon jury de soutenance de thèse.

Cette thèse est issue d'une collaboration entre l'équipe TRAME du groupe ESEO et l'équipe Temps Réel de l'Institut de Recherche en Communications et Cybernétique de Nantes (IRCCyN). Je tiens par conséquent à remercier l'ensemble des membres de ces deux équipes et plus particulièrement, Charlotte Seidner et Matthias Brun qui ont largement participé à la réussite de cette thèse. Je souhaite également remercier Camille, Frédéric, Mickaël, Jonathan, Guillaume et Sébastien qui m'ont aidé, soutenu et conseillé tout au long de ces années. Je tiens aussi à remercier Ladji qui partageait mon bureau et qui m'a beaucoup épaulé à travers notamment de grands moments de convivialité.

Je veux aussi joindre à mes remerciements, les membres du département SEA de l'ESEO et tout particulièrement Patrick Plainchault, responsable du département qui m'a permis d'enseigner, ainsi que Joseph et Cristina avec qui j'ai eu la chance d'enseigner. Plus largement, je souhaite remercier l'ensemble des membres de l'ESEO avec qui j'ai eu le plaisir de travailler.

Mes remerciements s'adressent une nouvelle fois à Jean-Philippe Babau et Sébastien Faucou qui ont aussi accepté de faire partie de mon comité de thèse.

Enfin, mes derniers remerciements sont réservés à ma famille et mes proches qui m'ont beaucoup soutenu durant cette thèse. Je tiens à remercier plus chaleureusement Elisabeth et Guy, sans qui ce travail n'aurait jamais pu aboutir. Pour terminer, je tiens naturellement à témoigner du soutien exceptionnel que m'a procuré Hélène, les mots qu'elle a su trouver, la force qu'elle m'a donnée, l'attitude positive qu'elle incarnait et bien plus encore, la joie immense qu'elle m'a offerte en mettant au monde nos deux petits garçons.

À Théodore et Léandre.

Table des matières

Introduction

Contexte

Les Systèmes Embarqués Temps Réel (SETRs) sont rencontrés dans de nombreux secteurs d'activités tels que l'aéronautique, l'automobile, la téléphonie mobile ou bien encore la robotique, etc. Les ingénieurs des SETRs sont alors confrontés à produire de plus en plus de systèmes complexes et innovants [11], avec des cycles de développement réduits à moindres coûts de production [44]. Optimiser le temps de mise sur le marché (*time to market*) de ces systèmes est devenu un enjeu majeur pour les entreprises du domaine dans le but d'être plus compétitives. Afin de répondre à ces contraintes de développement, trois axes se démarquent aujourd'hui pour améliorer la productivité des SETRs :

- **La portabilité des applications temps réel :** doit être assurée pour s'adapter facilement aux exigences hétérogènes des supports d'exécution matériels ;
- **La réutilisabilité des processus de développement** doit aussi permettre une migration rapide de ces applications temps réel dans le cas de déploiements vers d'autres supports d'exécution ;
- **La maintenabilité des SETRs** doit enfin faciliter l'intervention des parties prenantes selon leurs préoccupations métiers en vue de modification ou de réparation.

Afin de décomplexifier et d'alléger la mise en œuvre des SETRs, les systèmes d'exploitation temps réel (RTOSs, *Real-Time Operating Systems*) [89] sont devenus des éléments incontournables du développement de tels systèmes. En effet, du fait de la grande hétérogénéité des supports d'exécution matériels, les RTOSs apportent un niveau d'abstraction nécessaire au portage des applications logicielles. Ces RTOSs représentent par conséquent des supports d'exécution (ou *plates-formes*) logiciels pour ces applications.

Toutefois, ces RTOSs se sont beaucoup diversifiés ; ce qui rend leur considération moins triviale à travers les processus de développement des SETRs. Les RTOSs présentent des mécanismes et offrent des services qui conditionnent fortement les instructions classiques de la programmation [76]. Cette diversité de mise en œuvre et ce savoir-faire ont contraint les développeurs à s'abstraire aussi des RTOSs. L'Ingénierie Dirigée par les Modèles (IDM) [60] prône cette indépendance en offrant un cadre méthodologique aux ingénieurs du génie logiciel pour s'abstraire des plates-formes d'exécution durant la phase de conception.

Une telle initiative met en exergue l'utilisation de langages de modélisation et d'outils de transformation pour représenter et manipuler des modèles conceptuels de haut niveau d'abstraction. Cette avancée a permis aux développeurs d'établir eux-mêmes des suites d'outils (*tool suites* en anglais) en vue de partiellement automatiser leurs activités de développement. Ces suites d'outils s'appuient sur une architecture dirigée par les modèles (MDA, *Model Driven Architecture*) dans laquelle une description conceptuelle de haut niveau d'abstraction permet la représentation de l'application indépendamment de la plate-forme (PIM, *Platform Independant Model*). Des transformations de modèles se succèdent alors pour raffiner ce PIM et ainsi générer une représentation de l'application dépendante de la plate-forme visée pour l'exécution (PSM, *Platform Specific Model*). La notion de modèle de description de plate-forme (PDM, *Platform Description Model*) est aussi évoquée au sein du

MDA dans le but de mieux la considérer à travers les transformations IDM.

Problématique générale

Dans ce contexte, un grand nombre de suites d'outils ont été développées à ce jour. Parmi celles rencontrées, deux grandes familles se distinguent pour soit générer du code exécutif, soit générer des modèles formels. Les motivations avancées par ces contributions résultent de l'intégration d'activités de développement au plus tôt dans la conception. De ce fait, elles permettent respectivement d'appliquer des activités de :

- **Déploiement des applications logicielles :** le niveau d'abstraction évoqué précédemment oblige les développeurs à raffiner les modèles conceptuels jusqu'à des modèles de code implémentable et spécifique à une plate-forme particulière. Le déploiement se résume alors à un *mapping* des concepts applicatifs vers les services offerts par la plate-forme logicielle dans le but de les exécuter ;
- **Vérification de propriétés non-fonctionnelles :** une telle abstraction implique aussi de vérifier sur ces modèles conceptuels que les SETRs puissent satisfaire les contraintes exigées par les applications logicielles comme par exemple les contraintes temporelles.

Pourtant, ces deux familles révèlent un problème dual dans l'interprétation comportementale des plates-formes d'exécution logicielles temps réel. D'un côté, les suites qui se focalisent sur la génération de code ne proposent pas de formalisation des applications une fois déployées. De l'autre côté, les suites qui se focalisent sur la génération de modèles formels manquent d'exhaustivité dans la prise en compte des mécanismes et des services des plates-formes. Ces suites offrent certes la possibilité d'appliquer des activités de vérification, mais celles-ci n'interviennent qu'en amont du déploiement.

Le fait de ne pas rendre possible l'application d'activités de vérification après déploiement peut engendrer des retours chronophages sur la conception d'un système. Cette réalité contredit les gages d'amélioration de la productivité mentionnés plus haut. Existe-t-il alors une solution pour mener à la fois des activités de déploiement et des activités de vérification, à un niveau de conception détaillée des SETRs au sein d'une même suite d'outils ? Comment procéder pour considérer au mieux le comportement des plates-formes à travers une telle suite d'outils ?

Objectifs de recherche

Pour répondre à ces questions, cette thèse complète en partie une stratégie de déploiement d'applications temps réel multiplates-formes dans un contexte IDM de conception détaillée des SETRs. Cette stratégie est supportée par une suite d'outils appelée SEXPISTOOLS (pour *Software Execution Platform Inside Tools*) en cours d'élaboration au sein de notre équipe. L'orientation prise est d'offrir la possibilité de générer, indépendamment de la plate-forme envisagée pour le déploiement, à la fois du code et des modèles formels tout en considérant le comportement des plates-formes d'exécution logicielles temps réel. Afin de répondre aux trois critères d'amélioration de la productivité que nous avons introduits, SEXPISTOOLS s'appuie sur les points suivants pour :

- **Considérer un plus grand nombre de plates-formes d'exécution logicielles temps réel** en paramètre des processus de génération dans le but de produire un déploiement multiplates-formes ;
- **Utiliser des règles de transformation génériques** les plus indépendantes possible au regard de la plate-forme considérée en paramètre des processus de génération ;

– **Séparer les préoccupations métiers** (i.e., le choix de déploiement d'application, la considération des plates-formes, les règles de transformation et les activités de vérification et de validation) afin de clarifier les interventions de chaque spécialiste du domaine métier.

Afin de garantir ces préconisations, la notion de PDM a été introduite dans SEXPISTOOLS. Cependant, la version proposée ne précise pas comment représenter le comportement des plates-formes. De plus, le processus de déploiement actuellement développé ne permet pas de générer des modèles formels de conception détaillée des SETRs. Notre objectif est donc double :

1. Représenter le comportement des plates-formes et par conséquent l'intégrer dans un PDM.
2. Considérer ce comportement à travers un processus de génération de modèles formels.

Notre approche s'appuie sur une transformation visant à construire un modèle formel entier du SETR à concevoir, en incluant la plate-forme d'exécution et ce quelque soit la plate-forme considérée. Cette transformation compose plusieurs fragments de modèles formels indépendamment de la plate-forme ciblée. Chacun de ces fragments représente une partie du modèle formel qui capture le comportement du SETR déployé.

Précisons toutefois que l'objectif n'est pas de définir un outil de vérification et de validation. Nous cherchons seulement à générer des modèles comportementaux formels d'applications déployées qui serviront de supports de vérification.

Contributions

Cette étude contribue à la représentation et à la considération du comportement des plates-formes d'exécution logicielles dans une IDM dédiée au déploiement d'applications temps réel, en se focalisant sur les cinq points suivants :

Une extension d'un langage de modélisation des plates-formes d'exécution logicielles temps réel est apportée pour représenter le comportement de ces plates-formes. Ce langage est un outil de SEXPISTOOLS, appelé RTEPML (*Real-Time Embedded Platform Modeling Language*). RTEPML [12] permet de décrire structurellement ces plates-formes dans un contexte de développement utilisant des DSMLS (Domaine Specific Modeling Languages) comme alternative aux langages de modélisation généralistes : GPMLS (*Generic Purpose Modeling Languages*, dont UML [65] est un représentant, y compris dans ses formes profilées). Une adaptation est aussi établie au sein de RTEPML, quant à la traduction formelle du comportement en Réseaux de Petri Temporels (TPN, *Time Petri Net* en anglais) [59] [10].

La mise en œuvre d'un processus de génération de modèles formels de SETRs au sein d'une IDM est proposée pour considérer à la fois l'applicatif et le comportement des plates-formes d'exécution. Elle complète le processus déjà mis en œuvre au sein de SEXPISTOOLS pour le déploiement multiplates-formes d'applications temps réel. Ce processus s'appuie sur des règles de transformation indépendantes de toute plate-forme pour composer des modèles comportementaux en TPN d'applications déployées.

Une formalisation de la composition de modèles comportementaux en TPN d'applications déployées est décrite pour augmenter la confiance dans l'implémentation de cette composition au sein du processus de génération.

Un exemple d'application pour illustrer le cas d'un déploiement multiplates-formes est présenté. Le comportement de deux plates-formes d'exécution logicielles temps réel (OSEK/VDX [66] et VXWORKS [93]) est considéré à travers ce déploiement.

Plan

Ce mémoire est constitué de deux parties distinctes. La première partie vise à cibler davantage la contribution de cette thèse en matière de considération du comportement des plates-formes d'exécution logicielles temps réel dans une IDM. La seconde partie expose cette contribution qui propose une telle considération dans un processus de génération de modèles formels en TPN d'applications déployées.

La première partie est divisée en deux chapitres.

Le chapitre 1 détaille la problématique de cette étude de manière à identifier les besoins nécessaires pour représenter et considérer le comportement des plates-formes.

Le chapitre 2 présente notre suite d'outils SEXPISTOOLS et un état de l'art des travaux qui contribuent à la considération du comportement des plates-formes dans un processus de génération. Une synthèse vient positionner nos axes de recherche vis-à-vis des contributions arborées dans ce chapitre pour compléter SEXPISTOOLS. Ce positionnement tient compte des critères d'amélioration de la productivité évoqués dans cette introduction.

La seconde partie englobe les quatre derniers chapitres.

Le chapitre 3 vise à enrichir le langage de modélisation RTEPML pour décrire le comportement des plates-formes. Il définit les nouveaux concepts utiles à une telle description en intégrant ceux des TPNs pour traduire formellement le comportement.

Le chapitre 4 développe la stratégie de composition adoptée pour compléter le processus de déploiement avec un processus de considération du comportement. Les règles de transformation nécessaires à la composition sont présentées.

Le chapitre 5 formalise la composition avancée en complément du processus exposé dans le chapitre 4. Une nouvelle définition de la composition de TPNs est aussi donnée pour éviter toute ambiguïté de synchronisation des descriptions.

Le chapitre 6 synthétise cette contribution en s'appuyant sur un cas d'étude simplifié. Un exemple d'implantation du processus est illustré dans l'optique de générer des modèles formels en TPN exploitables avec l'outil de vérification Roméo [53].

Enfin, la présentation de ce mémoire se termine par une conclusion générale dans laquelle un bilan de la thèse est dressé. Des perspectives d'évolution des contributions présentées sont aussi apportées.

Première partie

Problématique détaillée et état de l’art

1 Problématique détaillée

Sommaire

Ce chapitre émet des hypothèses sur le contexte de cette étude, tout en posant le vocabulaire employé, pour guider de manière didactique le lecteur à la problématique détaillée de cette thèse.

Ce chapitre s'organise en quatre sections. La première contextualise notre étude vers la conception détaillée des systèmes embarquées temps réel. L'ingénierie adoptée est ensuite présentée dans le but d'encadrer notre contexte de travail : l'IDM. La section suivante caractérise la problématique de cette étude sur la considération des plates-formes logicielles d'exécution en modélisant leur comportement. Enfin, nous synthétiserons les grandes lignes de la problématique dans la dernière section.

1.1 Conception détaillée des systèmes embarqués temps réel

Le contexte général de la conception détaillée des SETR est posé ici pour focaliser le domaine métier sur celui du logiciel embarqué temps réel.

1.1.1 Les systèmes embarqués temps réel

Définition d'un système embarqué

Un système embarqué est un système, à la fois électronique et informatique, qui permet de contrôler et de commander un procédé physique [89]. Chaque système mis en œuvre doit répondre à des exigences fonctionnelles de l'équipement dans lequel il est intégré [1]. La conception de tels systèmes impose des activités qui concernent, d'une part, le traitement de l'équipement ; d'autre part, les intéractions avec l'équipement (voire d'autres équipements) et celles avec l'environnement physique ou humain [80] [2]. Le système est parfois qualifié de système "enfoui" de part l'absence d'interface homme-machine (IHM) et, par conséquent, l'absence d'intéraction directe avec un utilisateur.

Catégorisation

Une spécificité qualifie un système embarqué de "temps réel" (ou SETR) lorsque ce dernier est contraint par l'évolution dynamique ou comportementale du procédé auquel il est lié [31] [89]. Un système temps réel ne signifie pas pour autant qu'il est rapide, il doit plutôt réagir aux stimulii extérieurs à l'équipement dans un temps pertinent pour l'équipement. Cela implique que la satisfaction comportementale d'un SETR résulte de l'approbation d'une multitude de contraintes induites par le procédé physique. Ces contraintes sont considérées de différentes manières : elles peuvent être de sûreté de fonctionnement [17], d'enfouissement [5] ou encore temporelles [2]. Des nuances sont alors à préciser selon le respect absolu ou partiel de ces contraintes par le système. Dans ce sens, une catégorisation des SETRs a permis de les classer en fonction du niveau de contraintes respectées. Ainsi, ces SETRs sont qualifiés de :

- critiques s'ils nécessitent le respect strict des contraintes ;
- fermes s'ils admettent le non respect de certaines contraintes suivant le contexte de l'application ;
- souples s'ils tolèrent une probabilité de contraintes non satisfaites.

Hypothèse 1 *Les systèmes concernés par cette étude sont embarqués et temps réel. Le champ des applications couvert par ces systèmes peut s'étendre jusqu'aux systèmes critiques. Toutefois, l'objectif de cette thèse se limitera aux contraintes de sûreté de fonctionnement et temporelles.*

Mise en œuvre

La mise en œuvre d'un SETR repose sur un ensemble de composantes illustrées sur la figure 1.1. Il peut être vu comme une superposition verticale de composants tels que (1) l'application logicielle qui contrôle le procédé physique, (2) le support d'exécution matériel et/ou logiciel qui exécute l'application et (3) des périphériques tels que les capteurs (fournissant des informations comme des mesures ou des évènements) et les actionneurs (commandés par l'application en fonction des informations et des évènements traités).

Le composant central d'un SETR est par conséquent le support d'exécution agencé en une ou deux plates-formes. L'une, matérielle, s'articule autour de quatre architectures : (a) monoprocesseur, (b) multiprocesseur ou (c) multicœur pour paralléliser les traitements

1. Par exemple, un système de régulation de vitesse dans un véhicule automobile, ou bien encore un programmateur dans une machine à laver

2. Par exemple, le vent qui ralentirait le véhicule, ou bien encore, l'individu qui arrêterait sa machine à laver en cours de fonctionnement

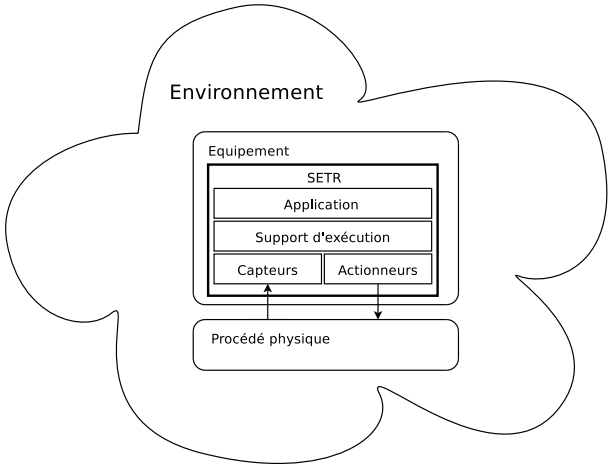


FIGURE 1.1 – Constitution d'un SETR au sein d'un équipement

lorsqu'une puissance de calcul est nécessaire et enfin (d) distribuée pour servir une installation éclatée géographiquement ; tandis que l'autre, logicielle, se concrétise par un programme dont l'exécution est (a) autonome sur le support d'exécution matériel, ou bien (b) dépendant d'un système d'exploitation qui est une couche intermédiaire entre les deux plates-formes. Pour résumer, le traitement d'une application embarquée temps réel s'effectue par l'implantation des fonctionnalités du système soit dans des circuits matériels spécifiques tels que les ASIC (*Application Specific Integrated Circuit*) et les FPGA (*Field Programmable Gate Array*), soit dans un programme logiciel s'exécutant sur un processeur. Des techniques de conception conjointe (*co-design*) existent pour mettre en œuvre la configuration matérielle et la programmation logicielle [94] [82].

Hypothèse 2 *Nous admettons ici que la plate-forme logicielle est vue comme un exécutif indépendant de la plate-forme matérielle. Par la suite, nous nous focaliserons sur la mise en œuvre d'applications embarquées s'exécutant sur des plates-formes logicielles ; chaque plate-forme intègrera un système d'exploitation temps réel dans une architecture monoprocesseur. Le système d'exploitation intégré ne sera pas imposé, mais le concepteur pourra plutôt orienter son choix parmi un ensemble de systèmes d'exploitation disponibles.*

Les plates-formes logicielles d'exécution

La mise en œuvre logicielle d'un SETR s'appuie aussi sur la capacité à établir un lien entre l'occurrence d'un évènement et le traitement d'une action fonctionnelle associée à cet évènement. Deux approches découlent de cette particularité, à savoir l'approche « synchrone » et l'approche « asynchrone » [78].

L'approche synchrone est basée sur le déclenchement synchronisé des actions par leurs évènements avec la fin de l'action qui était en cours de traitement. Chaque occurrence est ainsi mémorisée et considérée simultanément en temps nul par le système vis-à-vis de la dynamique du procédé contrôlé. Il en résulte une non concurrence des actions. La programmation des systèmes synchrones est effectuée sous la forme de systèmes échantillonnés dans lesquels la durée d'exécution des actions est inférieure à la période d'échantillonnage [16].

L'approche asynchrone, contrairement à l'approche synchrone, est basée sur le déclenchement des actions de façon asynchrone durant l'action en cours de traitement. Les occurrences d'évènements sont observées continuellement et considérées aussitôt en temps non nul par le système. Il en résulte un parallélisme et une mise en concurrence des actions.

L'approche asynchrone correspond mieux à notre champ d'étude, étant donné que nous voulons considérer la concurrence au sein des plates-formes logicielles d'exécution. Les systèmes d'exploitation propices à un tel paradigme sont appelés multitâches. Dans ce cadre, les actions prévues par une application sont amenées à être interrompues durant leurs traitements ; ces actions sont communément appelées des tâches qui sont des unités, ou fils, d'exécution séquentielle. La concurrence entre les tâches est orchestrée par des mécanismes offerts par les systèmes d'exploitation, portant sur : la synchronisation, la communication et l'accès à des ressources partagées [3].

Les accès concurrents aux ressources logicielles qui composent l'exécutif sont dirigés par une politique d'ordonnancement. Chaque politique est appliquée par un ordonnanceur qui est lui-même un composant de l'exécutif et peut être définie « en ligne » ou « hors ligne ». Une politique dite « en ligne » privilégie l'exécution de la tâche de plus forte priorité. Une distinction est faite quant à la définition des priorités selon une base de critères liés à l'ensemble des tâches concurrentes, tels que leurs échéances ou leurs périodes d'exécution [54]. Notons, toutefois, que l'interruption ou la préemption des tâches concurrentes avec ce genre de politique peut être inhibée au profit d'une autre. Une politique dite « hors ligne » impose la séquentialité des tâches à exécuter. L'ordonnanceur est dans ce cas considéré comme un séquenceur.

Outre la politique d'ordonnancement qui consiste à allouer les ressources d'un système d'exploitation dans une approche asynchrone, l'observation continue des occurrences des évènements est différenciée selon deux démarches.

Une démarche de cadencement par les évènements (*event triggered*) oblige le système à réagir à chaque évènement par un appel à l'exécutif. Les exécutifs temps réel OSEK/VDX-OS [66], ARINC-653 [1] et VXWORKS [93] sont des exemples mettant en œuvre cette démarche.

Une démarche de cadencement par le temps (*time triggered*) oblige le système à réagir à un seul évènement externe provenant d'une horloge périodique par un appel à l'exécutif. Le procédé est ainsi scruté à chaque période. Ce principe coïncide avec une planification des tâches « hors ligne ». Les exécutifs temps réel MARS [45] et OASIS [22] sont des exemples mettant en œuvre cette démarche.

Hypothèse 3 *La mise en œuvre logicielle des plates-formes d'exécution vise des systèmes d'exploitation multitâches conformément à l'approche asynchrone. Nous nous pencherons sur des systèmes, dont les exécutifs seront régis par une politique d'ordonnancement « en ligne », cadencés par les évènements (event triggered).*

1.1.2 Analyse et conception orientée objet

La plate-forme au cœur de la conception

Le développement des SETRS suit majoritairement le parangon de développement nommé cycle en V (figure 1.2) qui, initialement, a été adopté pour le génie logiciel. Ce cycle de développement anticipe les tests attendus de la partie montante dès les phases de spécification et de conception de la partie descendante. Cette stratégie offre l'avantage de vérifier et de valider, de manière ascendante, le développement du SETR selon des exigences fonctionnelles (i.e., ce que doit faire le SETR) et non fonctionnelles (i.e., la manière dont le SETR exécute ce qu'il doit faire) établies respectivement en spécification et en conception.

Comme illustré en gris sur la figure 1.2, la conception est divisée en deux niveaux d'abstraction :

3. Notons que ces mécanismes sont gérés par l'exécutif du système d'exploitation

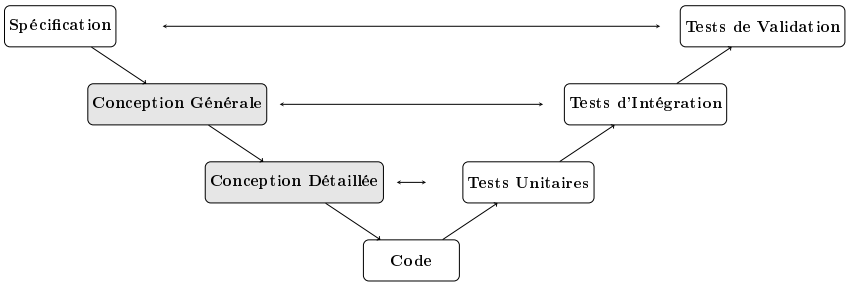


FIGURE 1.2 – Place de la conception dans le cycle en V

– **La conception générale** représente l'étape de plus haut niveau. Elle définit l'architecture logicielle du système à concevoir vis-à-vis de la spécification. Les entités fonctionnelles sont modélisées et agencées structurellement afin de représenter les échanges internes, mais aussi externes au système.
– **La conception détaillée** représente l'étape de plus bas niveau. Elle identifie les composants logiciels du système vis-à-vis de l'architecture précédemment définie. Chaque composant est décrit de manière exhaustive en précisant chaque traitement unitaire à exécuter.

Hypothèse 4 *Dans notre cas d'étude, nous différencierons la conception détaillée de la conception générale, de part la prise en compte des plates-formes logicielles d'exécution temps réel.*

Vers une conception orientée objet et détaillée

Devant la complexité et le besoin de diminuer les temps de développement des SETRs, des méthodologies d'analyse et de conception se sont succédées [7] en se calquant sur le cycle en V. La première grande évolution connue réside dans une approche dite fonctionnelle. Cette approche trouve son origine dans les langages procéduraux. Elle met en évidence les fonctions décomposables du système à implémenter en s'appuyant sur une hiérarchie descendante. La représentation des données avec cette approche est dissociée du traitement de ces données à travers les fonctions décomposées. Basée sur cette approche, la méthode graphique SA-RT (*Structured Analysis for Real-Time*) [37][92] est venue enrichir les premières méthodes fonctionnelles, telles que SADT (*Structured Analysis Design Technique*) [57], afin de s'adapter au domaine du temps réel. Cet enrichissement est fondé sur une vision dynamique du système réagissant aux stimuli de l'environnement dans lequel il se trouve et contraint par le temps.

Les approches fonctionnelles ont permis d'améliorer la vérification (i.e., les tests d'intégration et unitaires) et la validation (i.e., les tests de validation) structurées des fonctions des systèmes respectivement durant les phases de conception et de spécification. Seulement, dans une approche structurée, l'évolution des besoins tout au long du développement peut entraîner une dégénérescence de la topologie du parcours hiérarchique des fonctions du système. Ces approches devenues caduques, ont laissé peu à peu la place à d'autres approches de développement dites orientée objet [9] [77].

L'approche orientée objet est aujourd'hui largement adoptée par les concepteurs du génie logiciel. Avec cette approche, un système peut être représenté de manière abstraite par un modèle qui rassemble un ensemble d'objets dissociés, identifiés et définis par des propriétés. Ces propriétés représentent soit des attributs qui sont des données caractérisant l'état de l'objet, soit des méthodes (ou opérations) qui sont des actions qui caractérisent le comportement de l'objet en agissant sur les valeurs des attributs. La particularité de cette approche tient alors du fait qu'elle rassemble les données et leurs traitements associés au

sein d'un unique objet. La fonctionnalité du système représenté prend son sens dans les relations (e.g., héritage, agrégation, composition, etc.) qui permettent aux différents objets d'interagir.

Au fil du temps, des méthodes basées sur cette approche sont apparues pour améliorer le développement des SETRs. Au début elles contribuaient davantage à l'analyse des besoins (i.e., la spécification) en terme de description des spécificités temps réel au sein d'objets composés. Afin de valider ces descriptions au cours de la spécification, les méthodes HOOD/PNO (*Hierarchical Object Oriented Design with Petri Net Objects*) [67] et UML/PNO (*Unified Modeling Language with Petri Net Objects*) [24] ont vu le jour en apportant une représentation comportementale à ces objets. Une traduction est par conséquent semi-automatisée pour générer un représentation formelle de ces objets en réseaux de Petri [69] et ainsi y appliquer des activités de simulation et de vérification formelle. Les outils de vérification employés pour de telles activités s'appuient sur des méthodes de preuve formelle ou bien de *model-checking* [74] [62].

Depuis d'autres travaux ont étendu leurs recherches afin de mettre en œuvre des méthodes orientées objets plus portées vers la conception des SETRs. Cette volonté émane du besoin de répondre aux exigences non-fonctionnelles des SETRs et de vérifier ces exigences dès la phase de conception. Dans un contexte UML, la méthodologie ACCOR [70] vient étendre la spécification d'objets temps réel du SETR à concevoir avec des concepts actifs. Ces concepts actifs [36] regroupent à la fois :

- **Des boîtes aux lettres** pour échanger des messages de données entre les objets temps réel ;
- **Un contrôleur de concurrence et d'état** pour arbitrer le traitement de ces messages suivant les états et la mise en concurrence des objets.

Les objets actifs ainsi obtenus constituent un modèle de conception générale qui peut ensuite être formalisé pour vérifier des propriétés non-fonctionnelles telles que des propriétés de sûreté de fonctionnement ou de non blocage. Toujours dans un contexte UML, une autre contribution a posé un cadre méthodologique intégrant des exigences non-fonctionnelles temporelles dans la conception des SETRs. Cette méthodologie [34] associe UML avec SYSML qui a été étendu pour décrire ces exigences temporelles sous la forme de diagrammes. La vérification de propriétés temporelles peut s'opérer suite à la génération d'observateurs formels à partir des diagrammes d'exigences.

À ce jour, aucune méthodologie n'est supposée offrir la possibilité d'appliquer des activités de vérification en phase de conception détaillée. En effet, les méthodes proposées se veulent générales pour ne pas dépendre d'une plate-forme logicielle d'exécution temps réel spécifique. La diagramme d'activités de la figure 1.3 résume globalement cette situation.

Sur ce diagramme, le flux qui mène la conception générale d'un système (i.e., modélisation des objets actifs) à son déploiement (i.e., implémentation du code spécifique à une plate-forme d'exécution) [4] est représenté. La vérification impliquant au préalable une formalisation est ici distinguée des autres acteurs du domaine métier (i.e., ceux du génie logiciel embarqué temps réel) qui interviennent entre la conception et le déploiement. De manière itérative, le diagnostic donné par la vérification peut orienter le flux des activités vers la redéfinition du modèle d'objets actifs. Cette approche valide l'intégration du système dès la phase de conception. Une fois le modèle d'objets actifs validé, ce dernier doit être détaillé unitairement pour spécifier chaque objet en fonction de la plate-forme d'exécution. Cependant, la vérification sur ce modèle détaillé n'étant pas appliquée, des tests unitaires

4. Nous considérons le déploiement comme la mise en exécution de l'application sur la plate-forme logicielle

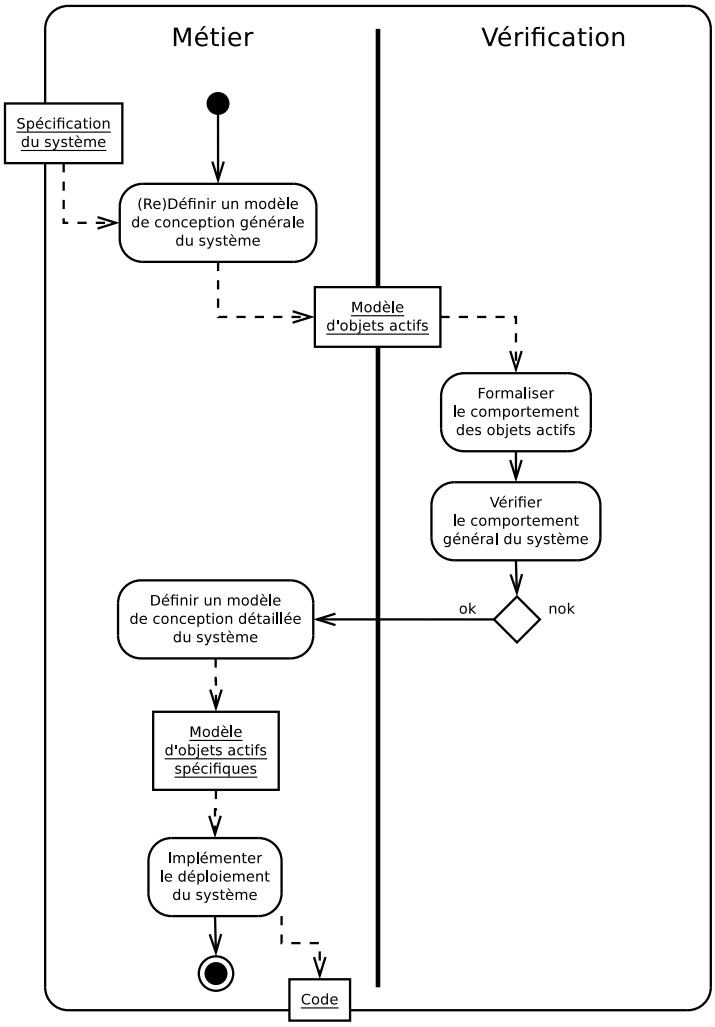


FIGURE 1.3 – Diagramme d'activités de la conception orientée objet : place de la vérification

doivent être envisagés après déploiement. En cas de comportement non attendu durant cette phase, un retour vers la phase de conception détaillée, voire même vers la phase de conception générale serait inévitable.

Hypothèse 5 *Nous présumons par conséquent qu'une recherche est nécessaire pour amener le développement des* SETRS*, vers leur vérification en phase de conception détaillée ; c'est-à-dire en spécifiant une plate-forme particulière. Nous nous appuierons sur une méthodologie orientée objet.*

Cette méthodologie s'oriente vers une approche basée sur les modèles. Dans ce qui suit, l'ingénierie avec laquelle nous envisageons d'intégrer la vérification en phase de conception détaillée est présentée.

1.2 Mise en œuvre dans une ingénierie dirigée par les modèles

Dans cette section, la méthodologie abordée précédemment pour la conception des SETRS va être contextualisée dans une ingénierie dirigée par les modèles (IDM). Cette contextualisation vient soulever les besoins nécessaires à la vérification des SETRS en phase de conception détaillée au sein d'une IDM.

1.2.1 Introduction

L'IDM (ou MDE pour *Model Driven Engineering*en anglais) est une ingénierie générative qui centre son activité de développement sur les modèles. Son but est avant tout technique pour aider les ingénieurs du génie logiciel à la production et à la maintenance de logiciels. Elle permet de traduire différents modèles tout au long du processus de développement, suivant le degré de raffinement recherché.

Modèles & métamodèles

Un modèle représente ainsi le système dans une phase bien déterminée de son développement. Pour une utilisation pratique par le biais d'outils informatiques, un modèle doit être décrit dans un langage particulier. Plusieurs descriptions de modèles sont rendues possible à partir d'un même modèle de langage, appelé métamodèle. Finalement, un métamodèle est une abstraction d'un langage de modélisation. De même, il existe un métamétamodèle qui est une abstraction d'un langage de modélisation de langages de modélisation. Aussi une relation de conformité existe entre un modèle et son métamodèle. Il s'ensuit qu'un modèle se conforme à son métamodèle ; nous adopterons par la suite la notation *c2* pour *conforms to* en anglais.

Transformation de modèles

Comme nous l'avons introduit, la génération d'un modèle escompté peut s'obtenir après une ou plusieurs transformations de modèles pour remplir un rôle particulier du développement. Une transformation est une fonction qui permet la création d'un ensemble de modèles cibles à partir d'un ensemble de modèles sources. Chaque ensemble de modèles se conforme à un même ensemble de métamodèles. Une transformation de modèles est décrite à partir des métamodèles sources et cibles et non pas à partir des modèles eux-mêmes. Cette stratégie privilégie la réutilisation de la transformation quels que soient les modèles qui se conforment aux métamodèles sources et cibles. Dans le cas où les deux ensembles de métamodèles sources et cibles sont identiques, la transformation est dite endogène, sinon elle est dite exogène. Afin d'expliquer le principe de base d'une transformation de modèle, la figure 1.4 suivante expose le cas d'un processus de développement conçu pour modifier un modèle de formes imbriquées (la modélisation est issue d'une autre étude [87]). A travers cet exemple, nous considérons que les ensembles de modèles et les ensembles de métamodèles sont des singletons.

Exemples de transformations

Sur cette figure, le cycle du processus évolue de la gauche vers la droite. Le niveau d'abstraction des modèles est représenté à la verticale. Le métamétamodèle MMM est constitué d'un ensemble d'objets (ou de concepts) permettant de décrire des formes qui peuvent être spécialisées et composées. Deux métamodèles *MM1* et *MM2* se conforment tous deux à MMM pour permettre la description de rectangles dont certains sont des carrés. Une nuance est toutefois faite entre leurs concepts pour que d'un côté, un carré puisse posséder un rectangle (par héritage un carré) ou un rond, et de l'autre côté, un carré puisse posséder un triangle ou une ellipse. Ce sont sur ces concepts que deux transformations sont ici décrites pour accomplir le processus de modification ; l'une (`T1->2`), endogène, vient permuter les ronds avec les carrés, tandis que l'autre (`T2->3`), exogène, vient remplacer les ronds par des triangles et les carrés par des ellipses.

Chaque transformation de modèles est constituée d'un certain nombre de règles qui sont décrites dans un modèle de transformation (ici `M(T1->2)` ou `M(T2->3)`). Ces règles peuvent être décrites de manière ordonnée (impérative) ou non (déclarative) suivant le métamodèle de transformation (ici *MM(T1->2)* ou *MM(T2->3)*) fourni par le langage de transformation de modèles utilisé [5] [38]. Précisons que chaque métamodèle de transformation se conforme aussi à un métamétamodèle qui dans notre cas est différent de MMM. En effet, ce dernier est

5. Notons que le métamodèle de transformation aurait pu être le même si le langage de transformation avait été le même pour les deux transformations

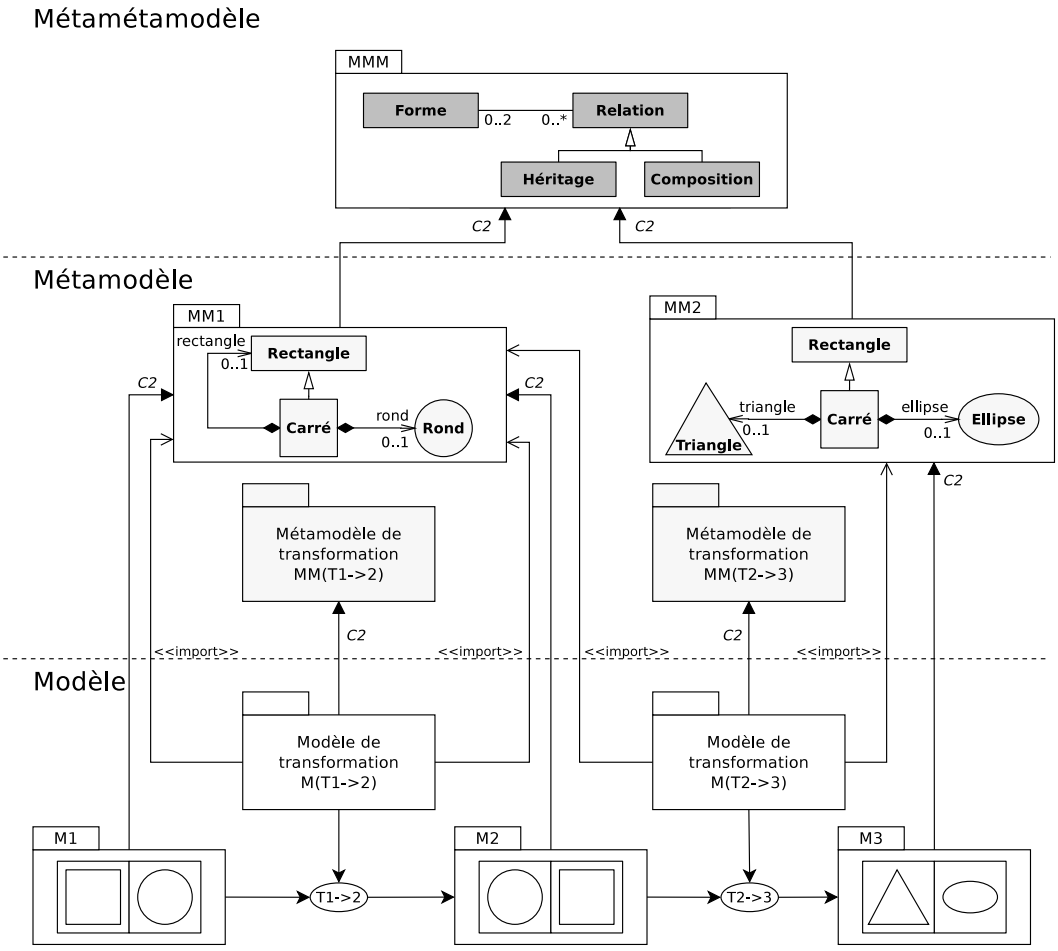


FIGURE 1.4 – Exemple de processus de modification de formes dans une IDM

décrit pour un espace technique qui se restreint à celui des formes géométriques. Pour des raisons de clarté, il n'a pas été représenté ici. L'algorithme 1.1 ci-après montre un exemple de règle qui pourrait être décrite dans M(T2->3) pour remplacer des ronds par des triangles.

Comme illustré ici, chaque règle de transformation effectue génériquement les modifications à l'ensemble des ronds du modèle source, en s'appuyant sur les concepts des métamodèles source et cible.

1.2.2 Processus de développement

Comme nous l'avons abordée précédemment, la conception détaillée des SETRs implique de spécifier chaque objet actif d'un modèle de conception générale en fonction de la plate-forme logicielle d'exécution ciblée pour le déploiement. Cette nécessité laisse entrevoir un enrichissement de ce modèle en intégrant la plate-forme au sein de la conception. En conséquence, l'élaboration d'un processus de développement dans une IDM semble intéressante pour traduire automatiquement ce modèle.

L'approche MDA

L'intégration des plates-formes d'exécution (non uniquement propre au domaine du temps réel) dans un processus de développement vise finalement à générer un modèle de l'application exécutable du système. Pour y parvenir, l'IDM met à disposition un cadre méthodologique pour intégrer progressivement les concepts et les mécanismes spécifiques à la plate-forme visée (e.g., la concurrence, les tâches d'exécution propres à la plate-forme,

ALGORITHME 1.1 – Règle de remplacement de ronds par des triangles

Métamodèle source :
$MM1$: le métamodèle qui décrit des ronds

Métamodèle cible :
$MM2$: le métamodèle qui décrit des triangles

Concept source :
C : le concept *Carré* $\in MM1$

Élement impliqué :
t : un triangle (conforme à *Triangle* $\in MM2$)

début
  si $\exists\, C.rond$ **alors**
    correspondance avec *le concept Carré* $\in MM2$
      $triangle \leftarrow t$
      $ellipse \leftarrow \emptyset$

etc.). L'initiative MDA (*Model Driven Architecture*) [60], promue par l'OMG, a vu le jour au début des années 2000 pour mettre en œuvre de tels processus de développement. Cette architecture distingue les modèles indépendants de toute plate-forme d'exécution (PIM, *Platform Independant Model*), de ceux spécifiques à une plate-forme d'exécution (PSM, *Platform Specific Model*). Le passage d'un modèle à un autre peut se faire au moyen d'une succession de transformations, comme introduit précédemment, selon le niveau d'abstraction requis. Cet enchaînement de transformations peut finalement aboutir au déploiement de l'application dans un espace technique destiné au codage. Le PSM résultant peut dans ce cas être utilisé pour générer du code [6] implémentable sur une plate-forme particulière. La figure 1.5 présente une succession de transformations conduisant à la génération de code. Chaque transformation peut intégrer une description de la plate-forme (PDM, *Platform Description Model*).

Toutefois, la notion de description de plate-forme évoquée à l'instant n'est pas encore très bien définie avec l'initiative MDA. Elle peut soit faire partie intégrante du modèle de transformation, soit être passée en paramètre de la transformation. La notion de PDM est la plupart du temps mise de côté tellement peu de travaux se sont adressés sur son contenu.

Hypothèse 6 *Représenter la plate-forme dans un* PDM *paraît être la piste la plus propice pour intégrer une telle description dans une conception détaillée.*

La conception détaillée dans un contexte MDA

L'implantation de la conception dans un processus de développement IDM optimise le flux des activités que nous avons présenté figure 1.3. La figure 1.6 détaille le processus de développement qui pourrait être globalement mis en œuvre au sein d'une architecture MDA.

Les activités de définition des modèles d'objets sont précisés par des activités automatisées de génération de modèles représentant chacune un processus. Chaque processus peut finalement comporter plusieurs transformations comme annoncé dans l'approche MDA. La vérification apparaît toujours en phase de conception générale. Néanmoins, cette configuration permet d'automatiser la conception détaillée dans un processus duquel le PDM est passé

6. Le générateur de code est en réalité une transformation de modèle – à ceci près que le code généré est un modèle textuel.

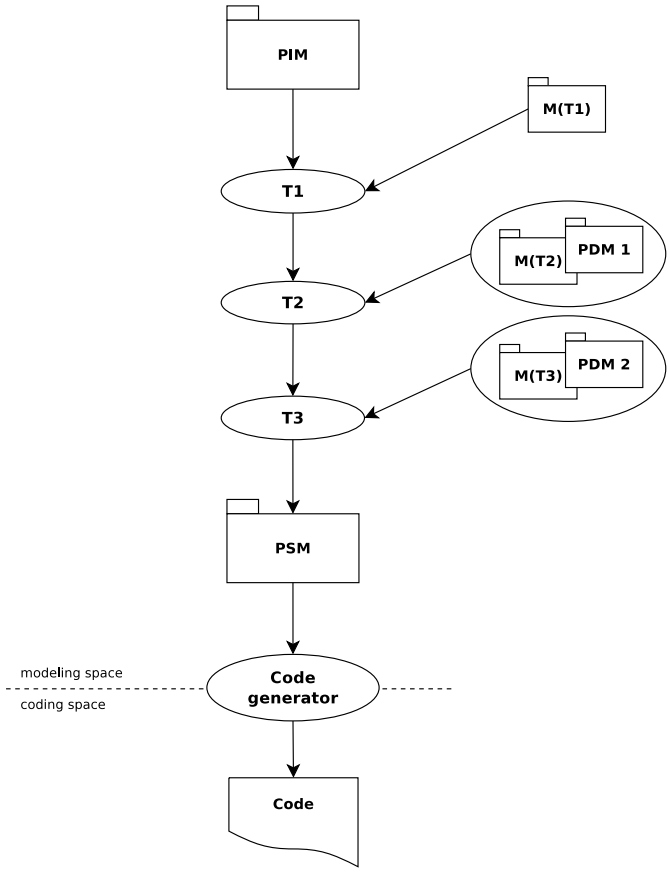


FIGURE 1.5 – Initiative MDA avec intégration de descriptions de plates-formes

en paramètre. Les règles des transformations apparaissent de façon globale en paramètres de l'ensemble des processus sur ce diagramme.

Hypothèse 7 *Cette contextualisation de la conception détaillée n'étant pas suffisante pour y appliquer des activités de vérification, nous ciblerons notre recherche vers un processus de génération de modèles formels d'applications exécutables. Il sera intégré dans le processus de conception détaillée.*

Approches conceptuelles pour la méta-modélisation

Le choix des métamodèles est important dans l'élaboration d'un processus de développement. La généricité des transformations du processus à mettre en œuvre réside en effet dans la généralisation des concepts et des relations qui décrivent la syntaxe abstraite du langage de modélisation. Deux courants sont distingués pour concevoir un métamodèle.

D'un côté, les **GPMLS** *Generic Purpose Modeling Languages* regroupent les langages de modélisation dits "généralistes". UML (*Unified Modeling Language*) [65], standardisé par l'OMG (*Object Management Group*), en est le principal représentant. Les concepts génériques de ces métamodèles tendent à la définition prolifique des concepts de modélisation. La conception de tels langages de modélisation favorise la réutilisation des transformations de part leur généricité, mais aussi de part l'homogénéité des outils employés. Le principal inconvénient réside cependant dans la difficulté de localiser un concept dédié au champ d'application. Comme réponse à ce problème, l'OMG a proposé d'étendre UML en introduisant des profils métiers [35] dans le but de raffiner les concepts visés. A l'instar du domaine du temps réel, le profil UML-MARTE (*Modeling and Analysis of Real-Time and Embedded systems*) [63] est le plus approprié.

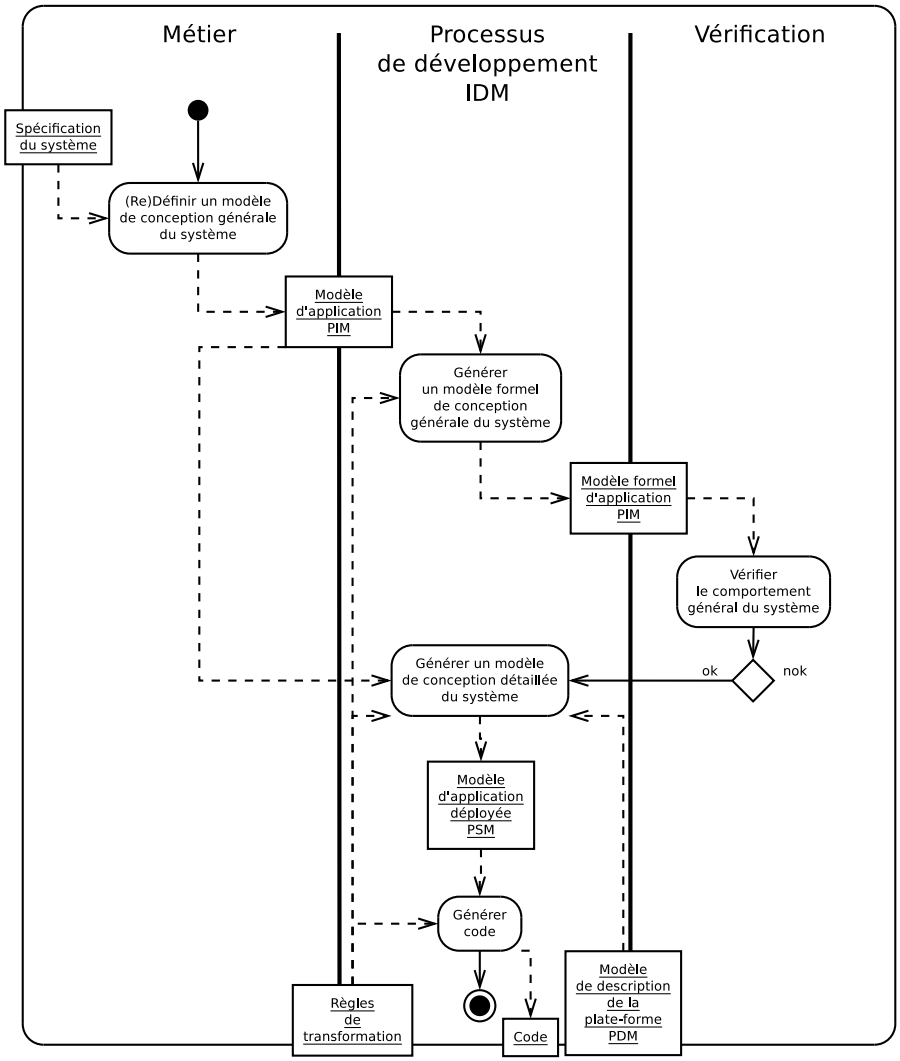


FIGURE 1.6 – La conception orientée objet dans une IDM : place du processus de développement

De l'autre côté, les **DSMLS** *Domain Specific Modeling Languages* regroupent les langages de modélisation dits "dédiés" à un domaine particulier. Les DSMLS, *a contrario* des profils UML, ne se basent pas sur le métamodèle UML. La conception d'un DSML [7] restreint entièrement le langage de modèlisation au domaine visé et allège ainsi la localisation des concepts pour la modélisation. AADL (*Architecture Analysis & Design Language*, standardisé par la SAE (Society of Automotive Engineer) fait parti des DSMLS dédiés au domaine du temps réel. La conception de ces métamodèles et des outils peut s'avérer très coûteuse, puisqu'elle fait appel à des spécialistes du domaine métier. Néanmoins, plusieurs environnements consacrés à la génération automatique d'outils ont vu le jour pour assister de tels développements, à l'image de GME (*Generic Modeling Environment*) [23], EMP (*Eclipse Modeling Project*) [28] ou bien encore TOPCASED [33].

D'un côté comme de l'autre, les GPMLS et les DSMLS garantissent des avantages mais ils présentent aussi des inconvénients. A ce stade, il est difficile d'envisager plus une approche conceptuelle qu'une autre. Le choix se profilera davantage dans la caractérisation de la problématique..

7. Un DSML se conforme en général à des méta-métamodèles tels que le MOF de l'OMG ou Ecore du projet EMF

1.3 Caractérisation de la problématique

Les deux sections précédentes nous ont permis de délimiter le champ d'application de cette étude : élaborer un processus de génération de modèles formels utile à la vérification des propriétés non-fonctionnelles des SETRs en phase de conception détaillée. Ce processus doit permettre la génération de modèles d'applications exécutables qui sont exploitables par des outils de vérification. Pour y parvenir, un cadre méthodologique tel que l'initiative MDA a été présenté pour nous aider à considérer la plate-forme logicielle d'exécution temps réel ciblée dans un PDM. Deux points importants restent cependant en suspens avec l'approche MDA :

- La représentation des plates-formes logicielles d'exécution temps réel dans un PDM : nous devons déterminer quelle métamodélisation adopter pour une telle représentation ;
- La considération des plates-formes logicielles d'exécution temps réel dans une IDM : nous devons déterminer de quelle manière une telle représentation doit être considérée au sein du processus à mettre en œuvre.

1.3.1 Représentation des plates-formes logicielles d'exécution

Une métamodélisation adaptée repose avant tout sur la définition de la syntaxe abstraite d'un langage de modélisation qui répond à nos besoins. Ces besoins sont évoqués ci-après en fonction de ce qui caractérisent les plates-forme d'exécution et de l'approche de représentation des plates-formes d'exécution.

1.3.1.1 Caractérisation des plates-formes

Composantes logicielles temps réel

Les plates-formes logicielles sont considérées comme un "ensemble de sous-systèmes et de technologies" qui fournissent les capabilités nécessaires pour supporter l'exécution d'une application logicielle [8]. La représentation des plates-formes d'exécution logicielles est caractérisée par un ensemble de composantes visibles depuis l'interface de programmation (API, Application Programming Interface) [58] [3]. Une identification plus en adéquation avec l'approche MDA [88] a cependant mis en exergue les besoins structurels et comportementaux de modélisation de cette API et a abouti à la caractérisation des quatre axes suivants :

Les ressources (tâches, sémaphores, boîtes aux lettre, etc.) sont vus comme une représentation structurelle des concepts exécutifs offerts par les plates-formes d'exécution. Cette notion permet par conséquent de décrire les mécanismes des plates-formes à travers des entités aux capacités d'exécution et d'instanciation restreintes.

Les services représentent structurellement les primitives offertes par les ressources pour les utilisateurs de l'API. Nous pouvons noter comme exemple de primitives, la création ou la destruction d'une tâche.

Les règles d'utilisation décrivent les contraintes et les patrons d'utilisation des ressources et des services. Le non respect de ces règles peut entraîner un dysfonctionnement dans l'exécution d'une application interfacée avec la plate-forme. Par exemple, suivant la plate-forme visée, un service appelé dans une routine d'exécution qui est bloquant, ne garantie pas forcément la mise en œuvre d'une application.

Le cycle de vie décrit le comportement observable des ressources et des services. Il représente donc l'aspect comportemental des plates-formes d'exécution depuis l'interface de programmation.

La représentation des règles d'utilisation implique une réflexion en terme de choix décisionnel quant à la plate-forme la mieux adaptée pour l'exécution des applications. Or, la priorité dans cette étude est de permettre la vérification de propriétés non-fonctionnelles des SETRs qui dépendent du cycle de vie des ressources et des services des plates-formes.

Hypothèse 8 *Par conséquent, nous nous focaliserons uniquement sur une représentation structurelle et comportementale des plates-formes d'exécution. Les règles d'utilisation ne seront pas abordées ici.*

1.3.1.2 Approches de représentation

Distinction des approches

Une étude prospective [12] a recensé dans notre équipe de recherche, trois démarches de représentation des plates-formes dans un contexte IDM. Cette prospection a été menée dans les deux approches conceptuelles de métamodélisation. Les approches de représentation sont décrites ici pour les deux approches conceptuelles de métamodélisation. Nous verrons que chaque représentation amène à considérer différemment les plates-formes à travers le processus de développement à mettre en œuvre, et ce quelque soit l'objectif (analyses, déploiements, simulations, etc).

L'approche enfouie offre une représentation des concepts exécutifs des plates-formes, soit par un programme [90], soit par un modèle formaté [85]. Dans le premier cas, le programme est directement implémenté la transformation de modèles. Dans le second cas, le modèle formaté est passé en paramètre de la transformation de modèles. Dans les deux cas, cette approche destine la description de la plate-forme à un traitement unique lié à une seule plate-forme. En effet, les concepts exécutifs n'apparaissent pas dans un métamodèle précis. La transformation ne peut établir de correspondances entre les concepts des métamodèles source et cible, et les concepts exécutifs de la plate-forme. Cette approche est donc délicate puisqu'une modification dans l'une des composantes (i.e., application, plate-forme ciblée, règles de transformation) du processus de développement à mettre en œuvre peut compromettre sa réutilisation.

L'approche implicite offre une autre manière de décrire les plates-formes d'exécution. Dans cette démarche, l'API de la plate-forme d'exécution apparaît dans le métamodèle. Chaque modèle de plate-forme correspond ainsi à un métamodèle différent [46] [81]. Cette approche facilite l'identification des concepts exécutifs au sein des transformations de modèles. Cependant, tout comme l'approche enfouie, la considération implicite ne favorise pas la réutilisation de processus. Le métamodèle étant dédié à une unique plate-forme d'exécution, les règles impliquées dans les transformations ne peuvent que s'appuyer sur les concepts de la plate-forme considérée.

Une alternative à cette approche met en évidence un métamodèle pivot. Ce métamodèle regroupe les concepts exécutifs rencontrés dans la plupart des plates-formes d'exécution [26] [83]. Ce type de métamodèle introduit une première transformation dans laquelle une plate-forme plus abstraite est considérée. Une seconde transformation vient affiner le processus en spécifiant la plate-forme ciblée. Néanmoins, seule une partie du processus de développement (la première transformation) peut être maintenue avec cette approche.

L'approche explicite consiste à décrire chaque plate-forme d'exécution dans un modèle. En conséquence, chaque modèle est conforme au même métamodèle dans lequel les concepts exécutifs génériques sont identifiés de manière abstraite. Contrairement aux approches précédentes, chaque modèle peut être considéré indépendamment des autres sans modifier les transformations puisque le métamodèle impliqué est unique. Avec cette approche, les processus sont donc réutilisables et plus génériques en raison de règles de trans-

formation indépendantes des plates-formes d'exécution ciblées. Cette approche peut permettre aussi de bien séparer les préoccupations métiers et ainsi de rendre chaque intervention indépendante vis-à-vis des autres [8].

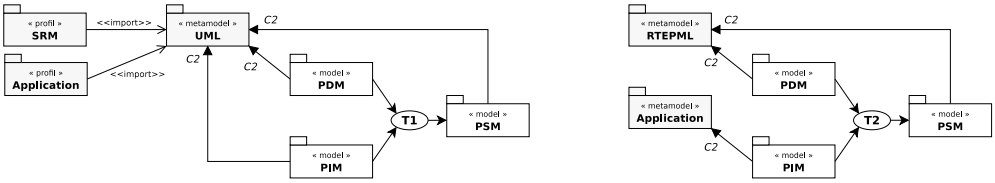
Hypothèse 9 *L'approche explicite semble prédominer les deux autres approches vis-à-vis de la généricité des transformations. Nous nous pencherons donc sur cette approche.*

1.3.2 Considération des plates-formes logicielles d'exécution

Comparaison entre contexte GPML *et contexte* DSML

La même étude prospective [12] que précédemment a permis d'identifier plusieurs approches de considération de plate-forme pour le compte d'un processus de déploiement d'applications temps réel. Dans cette prospection, les considérations sont comparées en fonction de l'approche de représentation et des deux approches conceptuelles de métamodélisation. L'approche explicite nous intéressant plus particulièrement, nous présentons figure 1.7 ici les deux approches de considération qui ont été retenues dans un contexte GPML et dans un contexte DSML. Ces deux approches résultent de contributions effectuées dans notre équipe.

Du côté des GPMLs, le *package* SRM (*Software Resource Modeling*) du profil UML-MARTE [87] permet de modéliser la structure des plates-formes à partir d'un modèle de domaine du logiciel embarqué temps réel. L'implantation d'un motif de conception « RESSOURCE-SERVICE » au sein de ce *package*, permet de définir des concepts exécutifs de la plate-forme ciblée par stéréotypage. La figure 1.7(a) de gauche, nous montre la considération d'une représentation de la plate-forme (PDM) décrite explicitement avec SRM. L'ensemble des modèles se conforment au même métamodèle UML, ce qui offre un contexte unifié entre les domaines de l'application, de la plate-forme et du déploiement de l'application sur la plate-forme. Cette unification existe suite à l'importation, dans UML, du package SRM, considéré comme un profil pour simplifier, et du profil du domaine de l'application. La métamodèlisation unifiée UML apporte une mise en relation interne des concepts de l'application avec ceux de la plate-forme, via des stéréotypes, pour générer un modèle de l'application déployée avec la transformation T1 [9]. Le déploiement est assuré en générant des instances des concepts exécutifs de la plate-forme pour les besoins de l'application.



(a) Contexte UML : profil SRM (b) Contexte DSML : RTEPML

FIGURE 1.7 – Approches de considération du PDM

Dans un contexte DSML, le langage de modélisation RTEPML (*Real-Time Embedded Platform Modeling Language*) est né de l'implantation du modèle de domaine de SRM. Ce langage qui est entièrement dédié au domaine du logiciel embarqué temps réel, aide à la représentation structurelle des concepts exécutifs des plates-formes d'exécution. Il contribue à l'outillage de processus au sein d'une même suite d'outils appelé SEXPISTOOLS (*Software Execution Platform Inside Tools*). La figure 1.7(b) de droite expose cette fois la

8. Dans le chapitre suivant, un état de l'art montre que ce n'est pas toujours le cas
9. Le modèle de transformation n'est pas représenté ici.

considération du PDM décrit avec RTEPML. Les métamodèles de l'application et de la plate-forme (RTEPML) sont dans ce cas disjoints. Contrairement à l'approche précédente, la mise en relation des concepts est effectuée au niveau de RTEPML pour générer un modèle de l'application déployée intégrée dans le modèle de la plate-forme. Les mécanismes utiles à l'exécution de l'application apparaissent dans RTEPML pour permettre une telle représentation dans le modèle de plate-forme. Pour cette raison, ce modèle qui décrit l'application déployée est aussi conforme à RTEPML.

Excepté le fait que la mise en relation des concepts diffère selon le contexte, ces deux approches répondent finalement aux mêmes critères de qualité mentionnés en introduction. L'approche explicite offre tout d'abord une généricité des règles de transformations qui profite à la réutilisabilité des processus de déploiement. Puis la mise en relation des concepts de l'application avec ceux de l'exécutif au niveau de la métamodélisation, de surcroît dans une approche explicite, assure la portabilité des applications vers un grand nombre de plates-formes. Enfin, la séparation des domaines de l'application et de la plate-forme, ajoutée à l'indépendance des processus vis-à-vis des plates-formes d'exécution, facilite la maintenabilité des déploiements en phase de conception détaillée.

Néanmoins, seule une description structurelle des applications déployées sont possibles avec ces deux approches. Les avantages qu'elles offrent nous laisse penser que la considération d'une représentation comportementale des plates-formes doit être menée suivant les deux approches conceptuelles de métamodélisation. Une contribution s'est déjà orientée vers cette considération comportementale avec SRM [30] en se basant sur les services d'exécution pour de la génération de code (se reporter au chapitre 2 à la page 30 pour une synthèse de ces travaux).

Hypothèse 10 *De ce fait, nous considérerons une représentation comportementale des plates-formes logicielles avec* SEXPISTOOLS *pour contribuer aussi dans un contexte* DSML.

1.4 Synthèse

Dans ce chapitre, le champ d'application de cette étude a été ciblé. Il se concentrera sur la génération de modèles formels d'applications temps réel déployées sur des plates-formes logicielles d'exécution. Cette contribution vient renforcer la vérification des SETRs en phase de conception détaillée. L'objectif n'est pas de traiter la vérification mais plutôt d'offrir la possibilité d'appliquer des activités de vérification. Les sytèmes concernés par cette étude sont multitâches, monoprocesseurs, cadencés par les évènements dans une approche asynchrone.

Un processus de génération de modèles formels a été avancé pour être intégré dans un processus existant de déploiement multiplates-formes d'applications temps réel. Dans une mesure plus détaillée que les conceptions actuelles, la considération du comportement des plates-formes logicielles d'exécution temps réel durant le déploiement a été proposée. La suite d'outils SEXPISTOOLS qui est développée dans notre équipe a été choisie pour contribuer à un tel processus et se place dans un cadre méthodologique telle que l'initiative MDA. Au sein de cette suite, le langage de modélisation RTEPML, qui permet actuellement de représenter explicitement la structure des plates-formes dans une approche DSML, a été retenu pour aussi représenter leur comportement. Ce cadre de travail a cependant soulevé deux points importants qui ont été caractérisés :

Nous proposons une représentation explicite du comportement des plates-formes logicielles d'exécution temps réel au sein d'un PDM. La syntaxe abstraite de RTEPML doit

par conséquent être étendue pour définir les concepts comportementaux nécessaires à cette représentation. Nous avons déjà caractérisé le besoin de représenter le cycle de vie des ressources et des services. Toutefois, comment adapter cette représentation à celle proposée par RTEPML ?

Autre point abordé est la considération de ce PDM dans le processus que nous voulons mettre en œuvre. Le processus de déploiement actuel qui est implémenté dans SEXPISTOOLS pour le déploiement multiplates-formes a été élaboré selon une certaine approche de considération. Il intègre l'application déployée dans le modèle de la plate-forme. Néanmoins, la prise en compte du comportement dans la représentation de la plate-forme a-t-elle un impact sur cette approche de considération ? Comment appliquer de façon corollaire le nouveau processus de génération de modèles formels au processus de déploiement existant ?

Pour orienter nos axes de recherche, un état de l'art de SEXPISTOOLS et des contributions en lien avec notre problématique est proposé dans le chapitre suivant.

2
État de l'art

Sommaire

Ce chapitre vise à positionner les contributions de cette étude vis-à-vis des travaux existants. Ce chapitre s'organise en trois sections.

La première section présente notre suite d'outils SExPIsTools qui ne considère actuellement que la structure des plates-formes logicielles d'exécution dans un processus de déploiement d'applications temps réel. La seconde section expose les outils qui ont déjà contribué à la considération du comportement des plates-formes logicielles d'exécution dans une IDM. Enfin la dernière section positionne nos axes de recherche vis-à-vis des outils présentés pour aussi considérer le comportement des plates-formes dans SExPIsTools.

2.1 Présentation de SEXPISTOOLS et de RTEPML

Grâce à l'approche explicite sur laquelle est basée la considération des plates-formes logicielles d'exécution au sein du processus de déploiement implémenté dans SEXPISTOOLS, un grand nombre de plates-formes décrites structurellement peuvent être capitalisées. Nous présentons ici le langage de modélisation RTEPML, ainsi que le processus de déploiement, tels qu'ils sont développés.

2.1.1 SEXPISTOOLS dans un contexte IDM

Placée dans un contexte IDM, l'activité de SEXPISTOOLS (voir figure 2.1) se substitue à la génération de modèles de conception détaillée, déjà évoquée dans la figure 1.6. Le processus actuellement implémenté vient automatiser le déploiement multiplates-formes dès la phase de conception. La plate-forme d'exécution considérée explicitement en paramètre du processus de déploiement est préalablement décrite par un langage de modélisation.

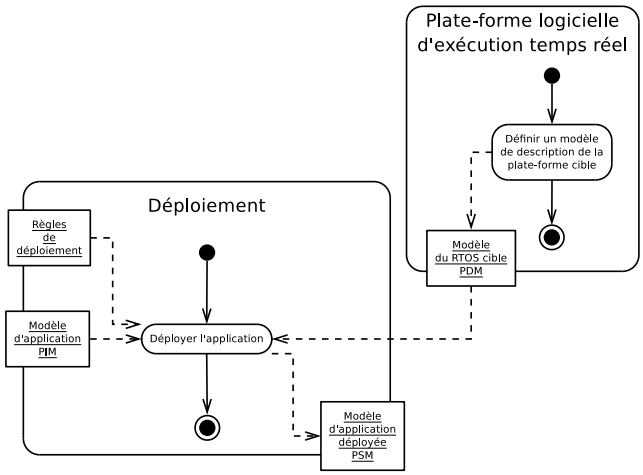


FIGURE 2.1 – Génération d'un modèle de conception détaillée avec SEXPISTOOLS

Comme illustrée sur cette figure, l'activité de SEXPISTOOLS privilégie l'intervention des spécialistes du déploiement et de la plate-forme logicielle d'exécution temps réel visée. L'activité de déploiement se distingue bien de l'activité de description de la plate-forme, ce qui permet à chaque spécialiste d'agir indépendamment.

2.1.2 Représentation explicite des plates-formes avec RTEPML

La description des plates-formes est représentée explicitement dans un PDM grâce au langage de modélisation RTEPML. Sa syntaxe abstraite est une alternative du *package* SRM d'UML-MARTE qui a été abordé dans le chapitre précédent 1 à la page 18. RTEPML a gardé la même taxonomie que SRM pour identifier structurellement les ressources logicielles d'un grand nombre de plates-formes d'exécution temps réel [86]. Ces ressources sont décrites par des concepts exécutifs dont des rôles ont été identifiés pour les repérer dans RTEPML. Les concepts sont catégorisés de la manière suivante [1] :

– **Les ressources concurrentes (*ConcurrentResource*) :** une distinction est faite entre celles qui sont activées par un ordonnanceur (*SchedulableResource*) et celles activées

1. Les rôles sont précisés à chaque fois en italique.

suite à une interruption (*InterruptResource*) [2] ;
- **Les ressources d'intéraction (*InteractionResource*) :** ces ressources sont catégorisées en deux sous-familles :
  - **les ressources de synchronisation (*SynchronizationResource*) :** elles synchronisent les ressources concurrentes. Parmi ces ressources, nous pouvons noter les mécanismes d'exclusion mutuelle (*MutualExclusionResource*) et de notification (*NotificationResource*) qui permettent le contrôle des flots d'exécution ;
  - **les ressources de communication (*CommunicationResource*) :** elles servent de communication de données destinées aux ressources concurrentes. Parmi ce ressources, nous pouvons noter les mécanismes d'échange de données par variable partagée (*SharedDataComResource*) et par message (*MessageComResource*) qui permettent la gestion des flots de données.
- **Les routines (*Routine*) :** les routines servent de points d'entrées à l'exécution des ressources concurrentes ;
- **Les compteurs (*TimerResource*) :** les compteurs comptabilisent les bases de temps servant de référence aux temps d'exécution ;
- **Les partitions mémoires (*MemoryPartition*) :** elles permettent la représentation d'espaces, plus ou moins restreints, d'adressages affectés à des ressources concurrentes.

Précisons que les ressources de gestion n'ont pas été identifiées au sein de RTEPML. Celles-ci couvrent essentiellement l'allocation des ressources matérielles (tels que le processeur ou la mémoire) et la gestion des périphériques [63]. En revanche, RTEPML reconnaît les types de données natifs à une plate-forme d'exécution. Le concept de type de données (*DataType*) a été intégré dans RTEPML pour identifier les différents genres possibles (e.g., booléens, caractères, entiers, flottants, etc.).

D'autres rôles ont aussi été identifiés pour caractériser ces concepts par le biais d'attributs ou de relations vers d'autres concepts. Un extrait de RTEPML est donné figure 2.2 montrant le fondement de la modélisation des ressources et des services des plates-formes logicielles d'exécution temps réel. Un exemple d'assignation de rôles identifiés par référencement est présenté.

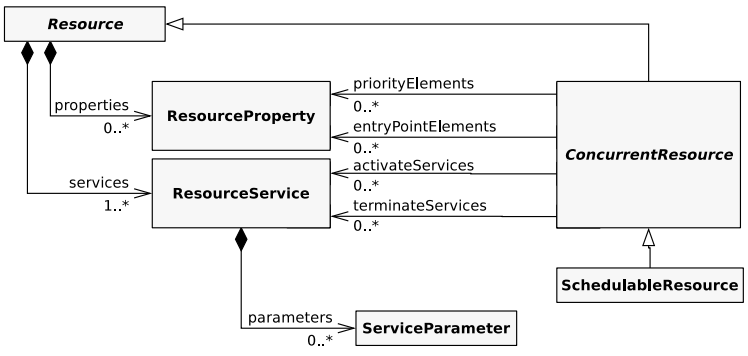


FIGURE 2.2 – Extrait du DSML RTEPML : modélisation structurelle des ressources et des services

Sur cette figure, le concept de ressource *Resource* est un concept abstrait qui généralise l'ensemble des ressources logicielles présentées précédemment. Chaque catégorie de concept de ressource est hiérarchisée grâce à la relation d'héritage : une ressource ordonnançable (*SchedulableResource*) hérite d'une ressource concurrente (*ConcurrentResource*).

2. Les alarmes se distinguent par leur activation sur une valeur de référence par un compteur (*TimerResource*)

Chaque ressource peut ensuite être caractérisée : une ressource est munie de propriétés (*ResourceProperty*) et fournit des services (*ResourceService*) qui sont paramétrables (*ServiceParameter*). Si nous prenons maintenant l'exemple d'une ressource concurrente, certains rôles identifiés servent de références à la caractérisation de propriétés telles que des éléments de priorité (*priorityElements*), de point d'entrée (*entryPointElements*), etc. D'autres aussi permettent la caractérisation de services tels que des services d'activation (*activateServices*), de terminaison (*terminateServices*) [3], etc.

Nous illustrons ci-après deux cas de figure d'utilisation de RTEPML pour représenter un extrait du PDM de la norme OSEK/VDX. Le premier donne un exemple de description de ressource, tandis que le deuxième montre comment décrire un prototype de conception.

2.1.2.1 Description de ressource

La figure 2.3 expose une représentation simple de définition d'une tâche OSEK/VDX dans un PDM conforme à RTEPML.

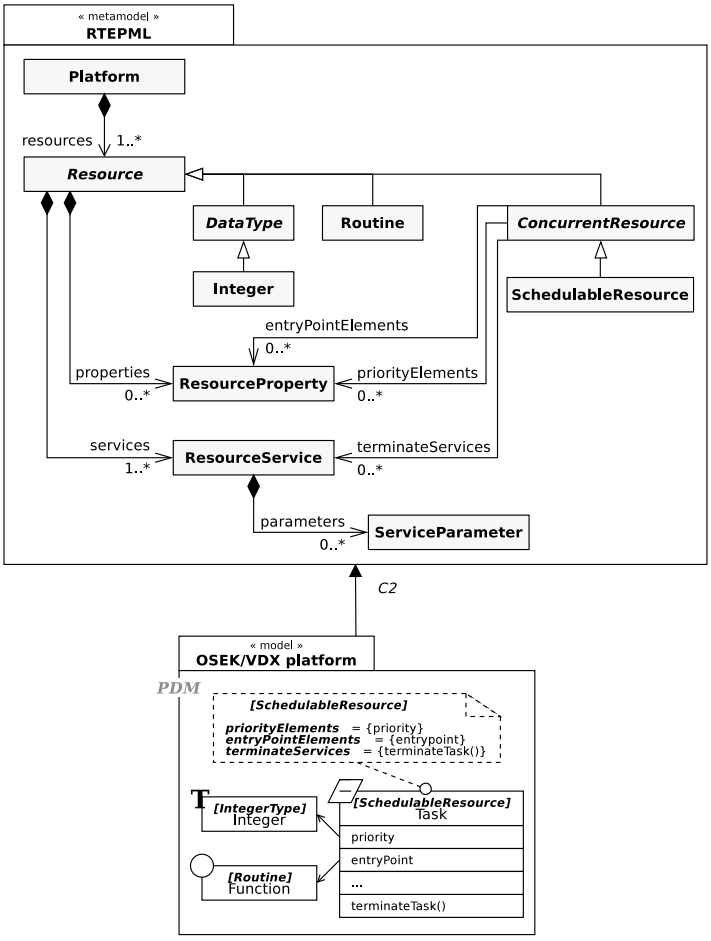


FIGURE 2.3 – Représentation simple d'une plate-forme OSEK/VDX avec RTEPML

Sur cette figure, la tâche (Task) est définie dans le PDM conformément au concept de ressource ordonnançable (*SchedulableResource* [4]). Cette tâche est caractérisée par sa prior-

3. Tous les rôles ne seront bien évidemment pas abordés ici. Pour une plus description plus détaillée, se reporter aux travaux de définition de RTEPML [12]

4. Les rôles de ressources apparaissent en gras entre crochets sur le PDM

ité (priority) qui est définie comme un entier conformément au concept de type entier (*IntegerType*). Cette propriété joue le rôle d'élément de priorité (*priorityElements* [5]) pour la tâche. De même, deux autres rôles permettent de compléter la définition de cette tâche : son point d'entrée (entryPoint) et le service de terminaison (terminateTask()) qu'offre cette tâche pour se terminer à la fin de son exécution [6]. Enfin une fonction (Function) qui est conforme au concept de routine (*Routine*) est aussi définie pour préciser le point d'entrée de la tâche.

Un guide conceptuel d'utilisation de RTEPML est fourni en annexe à la page 131 afin de mieux comprendre la démarche adoptée pour décrire les ressources d'une plate-forme.

2.1.2.2 Description des prototypes de conception

Certaines plates-formes ne fournissent pas de ressources dédiées à l'exécution de traitements particuliers. Néanmoins, un agencement des ressources doit permettre la mise en œuvre de ces ressources dédiées. C'est la raison pour laquelle la notion de prototype de conception a été adoptée dans RTEPML. Pour illustrer l'emploi de cette notion, la figure 2.4 décrit la représentation d'une tâche périodique définie sur une plate-forme OSEK/VDX.

La mise en œuvre de telles ressources est rendue possible dans RTEPML grâce au concept *Prototype*. Chaque ressource dédiée (*Resource*) peut en effet se voir attribuer un prototype qui est constitué d'instances de ressources (*PrototypeResourceInstance*) et d'associations (*PrototypeResourceInstanceAssociation*) pour agencer ces instances. Un certain de nombre de rôles (i.e., *PrototypePeriodElements* et *PrototypePriorityElements* dans notre cas de figure) ont été définis, par référencement, au sein de chaque ressource dédiée pour identifier les instances qui la composent. Ce mécanisme a été adopté pour caractériser les ressources dédiées à partir des rôles (i.e., respectivement *PeriodElements* et *PriorityElements*) qui caractérisent les ressources qui typent (*type*) les instances.

Pour illustrer cet agencement, nous avons décrit une tâche périodique (PeriodicTask) dans un PDM conformément au concept de ressource ordonnançable (*SchedulableResource*). Un prototype a été conçu pour cette tâche périodique qui est composée d'une tâche (Task) et d'une alarme (Alarm). Cette composition est conforme à l'agencement d'une instance de ressource typée par Task (T1) avec celle typée par Alarm (A1). La tâche périodique est finalement caractérisée par sa priorité (T1.priority) et sa période (T1.period) après identification des instances au sein de la tâche périodique et caractérisation des ressources types. Notons qu'il est possible de préciser une instance maîtresse (ici T1) au sein du prototype grâce au rôle *masterInstance*.

Un guide conceptuel d'utilisation de RTEPML est fourni en annexe à la page 131 afin de mieux comprendre la démarche adoptée pour décrire les prototypes de conception d'une plate-forme.

2.1.3 Déploiement sur une plate-forme décrite par RTEPML

La plate-forme d'exécution une fois décrite est considérée à travers le processus de déploiement qui a été implémenté dans SEXPISTOOLS. Par conséquent, le processus a pour but de déployer une description de l'application sur la plate-forme considérée. Sur la figure 2.1, l'activité de déploiement s'appuie sur des règles qui établissent des correspondances entre le modèle de l'application (PIM) et le modèle de la plate-forme d'exécution (PDM) pour générer un modèle de l'application spécifique à la plate-forme considérée (PSM). Plusieurs approches ont été mentionnées pour mettre en relation les éléments du

5. Les rôles de propriétés et de services sont annotés en gras sur le PDM
6. La norme OSEK/VDX impose que ce service soit appelé par la tâche en exécution avant de s'endormir

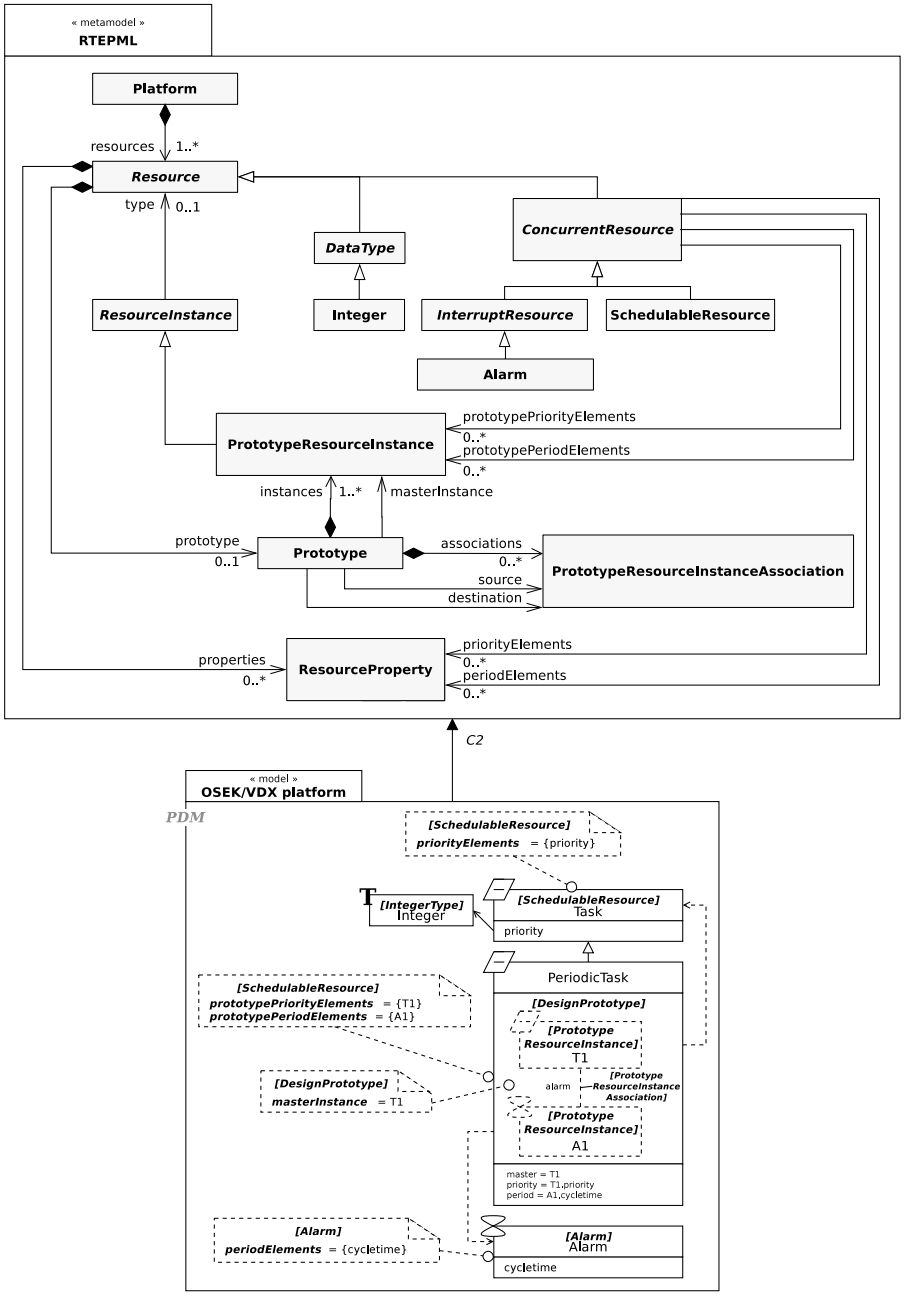


FIGURE 2.4 – Représentation d'une tâche périodique OSEK/VDX avec RTEPML

PIM et du PDM dans la génération d'un PSM [12]. Celle retenue ici est l'intégration de l'application dans la plate-forme.

2.1.3.1 Intégration de l'application

Cette approche introduit un mécanisme générique d'intégration de l'application dans RTEPML. Ce mécanisme est, non pas basé sur les concepts de l'application, mais sur l'instanciation des concepts de ressources mises à contribution pour l'exécution de l'application. Cette instanciation se rapproche de l'artefact de modélisation présenté précédemment sur la figure 2.4 pour décrire les prototypes de conception. Toutefois, contrairement aux instances de prototypes de conception, cette instanciation est liée à l'application et non à

la plate-forme. La figure 2.5 présente par conséquent les concepts de cette instanciation intégrés dans RTEPML. La création de ces instances pour l'application apparaît lors du déploiement ; elle est présentée dans la figure 2.6.

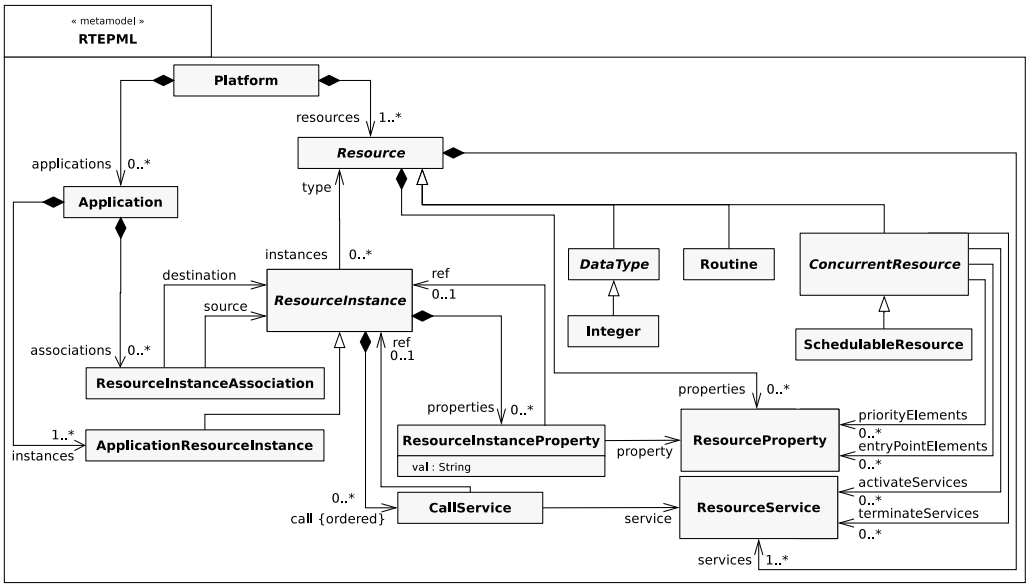


FIGURE 2.5 – Intégration de l'application dans RTEPML

La représentation d'une plate-forme peut intégrer plusieurs applications (*Application*) déployées. Ces applications sont intégrées dans RTEPML de manière simplifiée ; chaque application est en effet constituée d'instances de ressources (*ApplicationResourceInstance*), chacune typée par la ressource qu'elle incarne. Le traitement de l'application telles que les instructions conditionnelles ne sont pas considérées ici.

La plate-forme dispose alors d'une dualité entre les instances qui dépendent de l'application et les ressources qui dépendent de la plate-forme. Une instance peut être enrichie 1) par des propriétés (*ResourceInstanceProperty*) pour exprimer les valeurs des propriétés associées à la ressource type, et 2) par une liste ordonnée d'appels de services (*CallService*) qui caractérisent les ressources qui typent les instances visées par ces appels [7]. Par exemple :

- Pour une instance de ressource concurrente, celle-ci pourrait être enrichie par une propriété qui vient préciser la valeur de la priorité de la ressource concurrente ;
- Pour une instance de routine d'exécution, celle-ci pourrait être enrichie par l'appel du service de terminaison pour exécuter l'arrêt d'une ressource concurrente.

Un enrichissement d'instance résulte du renseignement d'informations spécifiques à l'application. Ce renseignement est effectué par des règles de correspondances durant le déploiement illustré ci-après.

2.1.3.2 Processus de déploiement

La figure 2.6 illustre un cas simple de déploiement par intégration d'application mis en œuvre dans SEXPISTOOLS. L'application (PIM) représente une activité très simple de robotique [8] dont le but serait d'afficher un message "*HelloWorld*" sur un écran. L'applica-

7. Une liste d'appels de paramètres (*CallParameter*) pour chaque service appelé est aussi possible. Mais celle-ci n'est pas représentée ici pour plus de lisibilité

8. Le métamodèle d'application Robot est donné à titre indicatif. En réalité, il ne correspond à une technologie précise ; les règles du processus de déploiement sont générées selon le cas

tion est déployée sur une plate-forme OSEK/VDX dont la description (PDM) est passée en paramètre du processus de déploiement.

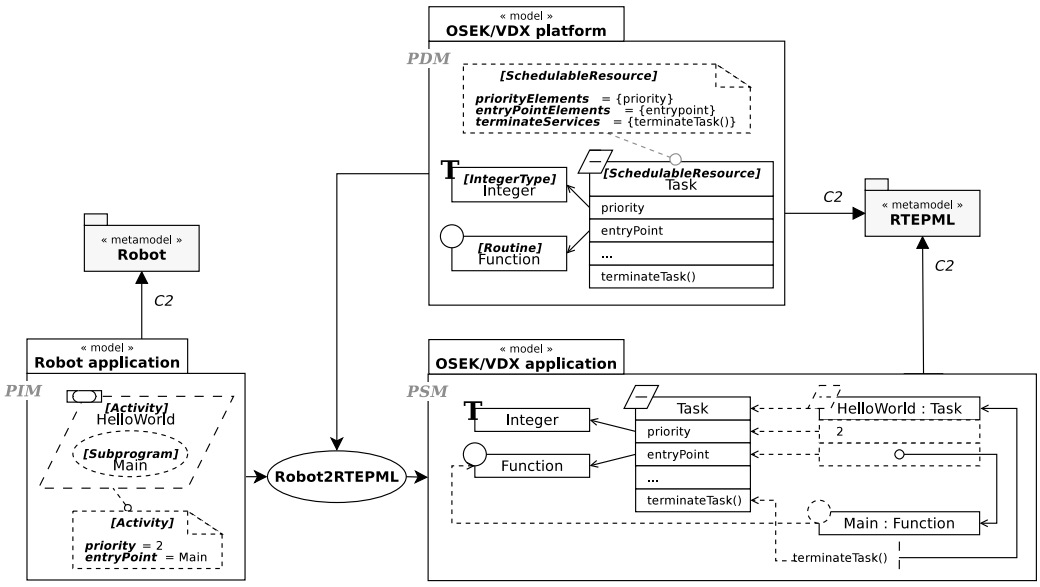


FIGURE 2.6 – Simple déploiement d'application vers une plate-forme OSEK/VDX

Les règles de transformation du processus de déploiement sont définies indépendamment de la plate-forme ciblée [9]. Cette indépendance est permise grâce à la notion de rôle présentée précédemment. Le déploiement est réalisé en établissant une correspondance (ou *mapping* en anglais) entre chaque concept de l'application et celui nécessaire à son exécution sur la plate-forme considérée. Ce *mapping* consiste à localiser le rôle du concept exécutif qui correspond à chaque concept de l'application à travers le PDM. Une fois la correspondance établie, la structure du concept exécutif localisé est instanciée conformément au modèle de l'application intégré dans RTEPML. Chaque instance est par la suite enrichie par la spécification de ses caractéristiques à partir des informations données par les concepts de l'application, en fonction des rôles appropriés. Toutes ces instances spécifiées constituent finalement le modèle de l'application déployée (PSM).

Sur la figure 2.6, une tâche (Task) a été instanciée (HelloWorld : Task) pour permettre l'exécution du sous-programme (Main) de l'activité d'affichage (HelloWorld) de l'application. La tâche est caractérisée par un point d'entrée (*entryPoint*) sur une fonction d'exécution (Function) qui a aussi été instanciée (*Main : Function*). Les deux instances créées ont été enrichies pour spécifier une priorité à la tâche et un appel du service de terminaison depuis la fonction d'exécution pour stopper la tâche.

Un guide conceptuel synthétisé sur l'enchaînement des règles du processus de déploiement est fourni en annexe à la page 131 afin de mieux comprendre l'instanciation des ressources, ainsi que la spécification des propriétés et des appels de service.

9. Pour autant, la généricité de ces règles repose sur leur génération qui provient de *templates* sources et de modèles de déploiement et de décision, tous deux conformes à un métamodèle de déploiement

2.2 Le comportement des plates-formes logicielles d'exécution dans l'IDM

Cette section recense les travaux existants, basés sur une approche IDM, pour améliorer la conception détaillée des SETRs. Ces travaux visent le déploiement des applications temps réel et considèrent le comportement des plates-formes logicielles d'exécution. Cet état de l'art complète ceux des travaux abordés [87] [12] dans le chapitre 1 précédent à la page 18. Deux axes sont privilégiés dans cet état de l'art : l'aide au déploiement, voire à la génération de code et l'aide à la génération de modèles formels d'applications déployées.

2.2.1 Outils d'aide au déploiement

Les outils présentés ici orientent vers la génération de code. Ils s'adonnent à considérer le comportement des plates-formes pour faire face aux exigences hétérogènes des plates-formes. Cette implication demande une grande flexibilité de la part de ces outils.

2.2.1.1 Analyse temporelle après déploiement

Présentation de l'approche

Fujaba Fujaba est un environnement de développement *open source* qui s'appuie sur l'IDM pour concevoir des systèmes logiciels implémentés en Java à partir de modèles UML. Il intègre aussi des outils de transformation pour passer du code à la modélisation (*reverse engineering*). Plusieurs extensions peuvent être apportées, dont l'une s'oriente vers la modélisation et l'analyse temporelle des SETRs critiques. Dans ce sens, une extension des *Statecharts* UML a été apportée à l'outil Fujaba pour considérer les contraintes temps réel : *Real-Time Statecharts* [14]. Les *Real-Time Statecharts* combinent à la fois les avantages qu'offrent les machines à états avec ceux des automates temporisés ; des annotations peuvent être ajoutées par l'utilisateur pour préciser des contraintes temporelles sur les états et les transitions. Ainsi, nous pourrions imaginer la modélisation d'un *thread* périodique avec son temps d'exécution critique (ou *Worst Case Execution Time*). Une telle représentation peut être utilisée pour de l'analyse d'ordonnancement et pour de la génération de code java ou C++.

Représentation et considération de contraintes de temps

La modélisation du comportement du SETR global (PIM) est dans un premier temps décrite en *Real-Time Statechart*. Le PIM intègre par conséquent des contraintes temps réel qui ont été spécifiées au préalable. Ensuite un algorithme de partage et de synthèse vient générer le PSM dans lequel les composants du système (objets actifs) ont été déployés orthogonalement avec des paramètres d'ordonnancement. Le PSM généré est conforme au profil UML-SPT (*Schedubality, Performance and Time*). La réunification des *Real-Time Statecharts* et de SPT au sein d'UML permet ainsi l'annotation des contraintes temporelles sur les objets actifs déployés. Le processus se déroule en deux temps : (1) la génération d'un diagramme de déploiement UML dans lequel des intéractions sont représentées de manière structurelle entre instances des composants du système, puis (2) le *mapping* entre les instances et les objets actifs annotés de *Worst Case Execution Time*. Précisons qu'une description de la plate-forme (*Platform Model*) est considérée en paramètre du processus pour renseigner des informations telles que les contraintes temporelles, le processeur, le système d'exploitation, etc.

Synthèse de l'approche

Cette contribution offre aussi l'avantage d'analyser des propriétés temporelles. A l'image du métamodèle des *Real-Time Statecharts*, un *plugin* UML permet d'intégrer dans Fujaba le métamodèle des automates temporisés hiérarchiques (*Hierarchical Timed Automata* en anglais) [15]. Par conséquent, une double transformation permet de générer un modèle formel qui décrit un automate temporisé qui peut être exploité par un *model-checker*

tel que UPPAAL [6]. Cependant, malgré l'intérêt porté pour contraindre temporellement le système à travers le déploiement, le modèle de plate-forme est considéré de manière enfouie via un modèle formaté. De plus, les composantes logicielles qui caractérisent la plate-forme n'apparaissent pas clairement. La généricité de représentation de la plate-forme permet certes la réutilisabilité du processus de déploiement, mais elle limite le portage vers différentes plates-formes.

2.2.1.2 Spécification du code par configurabilité de règles

Présentation générale de l'approche

TransPI L'outil TransPI [56] a été élaboré pour générer du code C implémentable sur des plates-formes logicielles d'exécution temps réel conformes au standard POSIX. Il peut même être configuré pour cibler d'autres plates-formes non conformes à POSIX. Le code exécutif est généré en deux temps : (1) un premier code indépendant de toute plate-forme est synthétisé à partir d'un modèle comportemental de l'application, puis (2) le code final est obtenu en intégrant les interfaces de programmation de l'application (API, *Application Programming Interface*) spécifiques à la plate-forme visée. Le modèle comportemental de l'application passé en entrée du synthétiseur de code regroupe le flux des données échangées et les APIs génériques à l'ensemble des plates-formes.

Le comportement des plates-formes

La définition de ces APIs génériques provient d'une sous-catégorie de la structure du standard POSIX 1003.1-2004 : le volume des interfaces systèmes. Les APIs identifiées ont été réparties suivant les composants suivants : les gestionnaires de tâches, de mémoire, de fichiers, de réseau, de périphériques, d'alimentation, ainsi que les services du temps réel. Le code synthétisé repose alors sur des patrons qui uniformisent l'usage des APIs génériques.

Dans un second temps, l'outil TransPI est utilisé pour traduire le code précédemment synthétisé en un code spécifique à la plate-forme ciblée. La traduction est basée sur un langage de transformation qui permet la définition d'un ensemble de règles venant affiner les patrons du code synthétisé. Cet affinage vient transformer les APIs génériques en APIs spécifiques, puis vient ajouter des segments de code pour les besoins de la plate-forme ciblée. Le code généré est décrit en langage C et représente le comportement d'une plate-forme spécifique et conforme à POSIX. Afin de s'adapter aux plates-formes non conformes à POSIX, TransPI est configurable pour soit modifier ou soit ajouter des règles pour cibler particulièrement une plate-forme spécifique.

Synthèse de l'approche

L'environnement offre comme principal atout une généricité dans le déploiement du code, puisque la plate-forme visée est conforme au standard POSIX. Toutefois cette approche est qualifiée d'"implicite" avec pivot, car seule la première partie du processus est générique en considérant une représentation abstraite du comportement générique des APIs. La représentation comportementale de la plate-forme visée est ensuite considérée concrètement comme un pivot dans la seconde partie du processus au moyen de règles configurables. Autre constat est qu'avec cette approche, seule une sous-partie de POSIX a été considérée pour la représentation des plates-formes. Cette délimitation nous laisse penser que certaines plates-formes comme ARINC-653-653 [1] qui présente des partitions mémoires, ne peuvent pas être considérées. De plus, cette adaptation coûte en terme de réutilisabilité. À chaque nouvelle plate-forme, l'utilisateur doit en effet intervenir sur l'outil pour configurer de nouvelles règles. Une lacune de cette approche est la non-possibilité d'exercer des activités de vérification, puisqu'il n'y a pas de description formelle des plates-formes. Seule une analyse statique du code généré est applicable après déploiement.

2.2.1.3 Flexibilité par paramétrage du code

Présentation générale de l'approche

Suite d'outils impliquant AADL Cette étude [41] s'est penchée sur la génération de code multiplates-formes. Un processus de déploiement a été réalisé pour traduire un code spécifique à la plate-forme visée. De la même manière que précédemment, le déploiement est assuré en deux temps ; (1) la traduction automatique d'un code indépendant de la plate-forme issu d'un modèle comportemental d'application logicielle commune à toute plate-forme, puis (2) la génération d'un code spécifique à la plate-forme concernée. La traduction nous importe peu ici puisqu'elle ne considère pas une plate-forme particulière. La deuxième transformation est plus spécifique en considérant en paramètres les capabilités de la plate-forme ciblée : d'une part, l'architecture matérielle et logicielle, et d'autre part, des bribes de code de l'API.

Représentation du comportement des plates-formes

Les capabilités structurelles de la plate-forme visée sont spécifiées avec le langage de description d'architecture AADL (*Architecture Analysis & Design language*) [79] standardisé par la SAE (*Society of Automotive Engineer*). AADL permet la description d'architectures du domaine des systèmes temps réel embarqués critiques. Un modèle AADL rassemble un ensemble de composants à la fois matériels (*processors*, *devices* et *busses*) et logiciels (*data*, *threads* et *processes*). Il est possible de compléter la définition des composants en leur annexant un comportement (*Annex behavior_specification*). Ce dernier apporte à chaque composant une extension à un sous-langage qui exprime un système d'états-transitions. Cependant, le comportement est décrit de manière statique dans AADL ; la syntaxe est annexée sans préciser la sémantique d'exécution. Les plates-formes décrites avec ce langage de modélisation sont plutôt vues comme des représentations abstraites. En conséquence, les aspects comportementaux des concepts exécutifs dépendants d'une plate-forme sont décrits sous forme de bribes de code. Chacune représente l'implémentation d'un composant AADL conformément à la plate-forme spécifiée. Chaque bribe est catégorisée selon le composant AADL à implémenter.

Considération du comportement des plates-formes

Le déploiement est réalisé en composant le code indépendant de toute plate-forme avec les bribes de code dépendantes de la plate-forme considérée. Une correspondance est établie en amont entre les composants de l'architecture AADL et les bribes de code pour capturer celles qui sont spécifiques à la plate-forme cible. Les liens sont rendus possibles à travers l'algorithme de transformation par le biais d'une méthode de paramétrage des bribes de code. Chaque bribe est en fait parsemée de fonctions avec des paramètres d'entrée pour permettre à l'algorithme de retourner des morceaux de code relatant des informations externes qui figurent sur le modèle d'architecture AADL. Cette méthode offre une flexibilité au déploiement pour répondre au besoin de migration d'applications sur des plates-formes dont le comportement est hétérogène.

Synthèse de l'approche

Malheureusement, aucune solution n'est proposée à ce jour pour formaliser l'application à un niveau plus détaillé que le PIM dans lequel la plate-forme est considérée de façon abstraite. Comme TransPI, le manque de formalisation du comportement ne permet pas l'emploi d'activités de vérification une fois le déploiement effectué.

2.2.1.4 Modularité par identification du code avec des rôles

Présentation générale de l'approche

SRM Une autre contribution s'est appuyée sur le *package* SRM du profil UML-MARTE [87] pour considérer le comportement des plates-formes dans un contexte GPML [30]. Le *package* SRM déjà abordé dans le chapitre précédent 1 à la page 18 permet la représentation explicite des plates-formes d'un point de vue structurel. Afin de combler le manque de comportement, un heuristique d'identification des comportements liés aux ressources et aux services d'exécution a été intégré dans le *package* SRM. Ensuite, chaque modèle de la plate-

forme détaillé avec SRM et cet heuristique est intégré dans un processus de déploiement multiplates-formes. Ce processus s'appuie sur la mise en relation interne des concepts de l'application avec ceux de la plate-forme (voir à la page 18) en associant les comportements identifiés sur le modèle de la plate-forme aux opérations du modèle de l'application.

Représentation du comportement des plates-formes

Afin de comprendre le principe adopté pour représenter le comportement, nous devons revenir sur la façon dont la structure de la plate-forme est représentée. Elle est similaire à celle permise avec RTEPML puisque ce dernier est fondé sur SRM (page 23). La définition des ressources de la plate-forme visée est permise grâce aux stéréotypes UML du package SRM (sous forme de diagramme de classe UML) ; une tâche ordonnançable est par exemple définissable au moyen du stéréotype «*swSchedulableResource*» selon la plate-forme. Une fois les ressources définies, ces concepts peuvent être enrichis par des attributs et des opérations pour préciser respectivement leurs propriétés et leurs services d'exécution ; une priorité et un service de création ou de destruction dans le cas de notre tâche. L'identification de ces propriétés et de ces services est rendue possible par référencement du stéréotype. Comme avec RTEPML, la notion de "rôle" [87] [12] est employée pour une telle identification ; l'opération de création joue par exemple un rôle de *createServices* au sein du stéréotype «*swSchedulableResource*».

L'heuristique d'identification adopté assigne un comportement sur les opérations (e.g., la création, l'initialisation ou bien encore l'appel à la fonction d'entrée d'une routine d'exécution, etc.) des ressources du modèle de la plate-forme. De ce fait, un comportement est encapsulé dans un UML *OpaqueBehavior* pour chacune de ces opérations. Ce comportement est représenté sous forme de bribe de code pour apporter une sémantique d'exécution indépendamment de la logique de l'application. La syntaxe pour chaque bribe de code correspond au langage de programmation imposé par la plate-forme.

Considération du comportement des plates-formes

Un processus vient ensuite automatiser le déploiement d'application sur des plates-formes Java ou C++/POSIX qui globalisent un grand nombre d'exécutifs temps réel. Ce déploiement s'opère en deux temps : (1) une réification structurelle des concepts du modèle de l'application en établissant des correspondances avec ceux du modèle de la plate-forme, puis (2) une génération des aspects comportementaux en implantant des mécanismes de concurrence et d'intéraction.

La réification s'appuie sur une transformation impliquant un modèle de correspondance dans lequel chaque stéréotype du métamodèle de l'application (sous-profil HLAM de MARTE UML dans cette contribution : *High-Level Application Modeling*) est défini concrètement en corrélation avec un stéréotype de SRM. A l'instar d'une ressource ordonnançable, une entité concurrente dans HLAM est stréotypée par «*RtFeature*». Par conséquent, le concept RtFeature sera annoté par le stéréotype «*swSchedulableResource*» de SRM au sein du modèle de correspondance [10]. Des règles de correspondance viennent par la suite parcourir ce modèle comme un pivot sémantique afin d'établir les liens nécessaires au déploiement. La spécification des éléments de la plate-forme est effectuée à travers ces règles en localisant les rôles joués au sein des stéréotypes concernés.

Les comportements abordés précédemment dans la représentation de la plate-forme ne concernent finalement que des comportements intrinsèques des ressources. Seulement, les comportements liés aux mécanismes de concurrence, de communication et de synchronisation mettent en jeu plusieurs ressources en fonction du comportement de l'application. Pour cette raison, une activité représentant un aspect fonctionnel de l'application peut être assignée à un service d'exécution de la plate-forme (comme par exemple, la protection d'écriture et de lecture d'une donnée partagée par un sémaphore) à travers la transformation

10. A noter qu'il en est de même pour les propriétés et les services.

de modèles. Ce comportement est décrit par le concepteur de l'application (indépendamment de la plate-forme) sous forme de diagramme d'activité dans un patron de conception. Il vient référencer le comportement encapsulé (*OpaqueBehavior*) dans l'opération du service d'exécution appelé pour la création et l'initialisation du mécanisme impliqué (comme celui du sémaphore pour la protection d'une variable partagée).

Une fois le modèle de l'application déployée (PSM) obtenu, des générateurs de code standard UML/Java et UML/C++ traduisent le PSM en code exécutable Java et C++. Ainsi, l'association des comportements encapsulés dans des *OpaqueBehavior* avec ceux des aspects fonctionnels de l'application liés aux mécanismes de la plate-forme apporte une syntaxe bas niveau aux modèles générés qui est proche du code.

Synthèse de l'approche

La démarche proposée ici ne vient pas modifier le package SRM, puisque des concepts tels que *OpaqueBehavior*) et des outils de modélisation tels que les diagrammes d'activités existent déjà dans UML pour représenter le comportement. Elle s'appuie sur une approche "explicite" qui avait été initialement implantée dans SRM [87] pour représenter et considérer génériquement les plates-formes lors de déploiements multiplates-formes. La réutilisabilité et la portabilité est donc assurée. Cependant, la généricité syntaxique et sémantique d'UML bride un peu la bonne séparation des préoccupations dans un contexte comportemental. En effet, le comportement est représenté selon deux points de vue : le comportement exécutif encapsulé dans les *OpaqueBehavior* et les actions haut niveau d'appels de services d'exécution implantés dans des patrons de conception. Bien que, dans les deux cas, tout ait été mis en œuvre pour que le comportement soit représenté indépendamment de l'application, deux styles de représentation sont nécessaires pour implanter l'ensemble des mécanismes des plates-formes. En outre, aucune analyse comportementale est évoquée dans cette étude.

2.2.1.5 Tests de faisabilité de déploiement

Introduction de l'approche

DRIM Le processus DRIM [61] est une approche qui vise à réduire le *gap* entre la phase de conception et celle d'implémentation des applications temps réel sur des plates-formes d'exécution logicielles. Il offre un guide méthodologique aux concepteurs des SETRs pour décider de la plate-forme la mieux appropriée à l'application, dans le respect de contraintes temporelles. Deux étapes sont alors proposées pour, d'une part, appliquer des tests d'implémentabilité, et d'autre part, générer des modèles d'implémentation selon la plate-forme spécifiée.

Représentation de plate-forme abstraite d'analyse

La représentation des plates-formes est rendue explicite grâce à son métamodèle éponyme qui est une extension du profil UML-MARTE. Cette extension offre la possibilité de modéliser les plates-formes de manière abstraite pour l'analyse temporelle. Elle permet aussi d'intégrer un modèle de l'application au sein du modèle de la plate-forme concrète. La représentation du comportement réside par conséquent dans l'association des concepts d'analyse avec les concepts structurels issus du *package* SRM.

Considération de contraintes temporelles

Une batterie de tests est dans un premier temps proposée pour évaluer la faisabilité d'un déploiement d'application. Un modèle de conception plus haut niveau de l'application (i.e., avant déploiement) est comparé avec le modèle abstrait d'analyse pour établir les contraintes temporelles à implémenter. Puis, les tests sont appliqués en corrélation avec la plate-forme visée pour s'assurer du bon déploiement. Lorsque le verdict est révélé positif, une seconde phase de portage de l'application est opérée vers la plate-forme désignée. À noter que dans le cas contraire, une phase de *refactoring* permet de fournir un nouveau modèle de conception initial qui respecte à la fois les contraintes initiales et le portage.

Synthèse de l'approche

Ce processus est une alternative intéressante de considération des plates-formes dans le but de vérifier l'implémentabilité d'une application temps réel. La méthode respecte

l'approche de représentation explicite pour associer des concepts abstraits d'analyse temporelle à ceux de la plate-forme ciblée par le déploiement. DRIM offre aussi un cadre de rétro-ingénierie adapté aux concepteurs des SETRs. Cependant, ce cadre méthodologique ne propose pas de processus de génération de modèles dédiés à la vérification formelle des applications déployées.

2.2.2 Outils d'aide à la génération de modèles en vue de vérification

La considération du comportement des plates-formes d'exécution par la modélisation formelle au cœur de la conception des SETRs permet la vérification de ces systèmes. L'ensemble des contributions qui sont abordées dans cette sous-section se sont focalisées sur la formalisation d'applications en intégrant le déploiement. L'intérêt porté pour une telle formalisation, implique une dépendance de l'application au comportement des plates-formes au sein des suites d'outils qui sont présentés ci-après.

2.2.2.1 Analyse comportementale de systèmes embarqués

Le comportement des plates-formes

GME GME (*Generic Modeling Environment*) [23] contribue à la description structurelle et comportementale des plates-formes d'exécution. Cet environnement intègre le langage de modélisation PML (*Platform Modeling Language*) qui se destine à la description explicite de plates-formes d'exécution et d'analyse. Ceci étant, les plates-formes visées avec ce langage ne sont pas explicitement dédiées au domaine du temps réel. PML permet, entre autres, de modéliser les ressources et les signatures des services qu'offre une plate-forme logicielle ainsi que leurs contraintes d'utilisation. En outre, cet environnement propose des outils d'assistance pour migrer une application d'un système à un autre et permet, de surcroît, d'intégrer des langages de modélisation formelle. Les descriptions rendues possibles peuvent être utilisées par la suite pour des activités de vérification et de validation.

Synthèse de l'approche

Grâce au langage de modélisation PML, GME est adapté pour vérifier des contraintes non-fonctionnelles d'un système déployé puisque les plates-formes sont représentées et considérées de manière détaillée. Malheureusement, la généricité de PML utilisé au sein de GME ne favorise pas le traitement de domaines particuliers tels que celui du temps réel. Autre problème soulevé avec cette approche est la non séparation des règles de transformation exprimées pour le portage d'application vis-à-vis de la modélisation des plates-formes. Les descriptions des règles se confondent avec celles des ressources et des services des plates-formes d'exécution.

Le comportement des plates-formes

Metropolis Metropolis [71] supporte la conception et l'analyse de systèmes embarqués hétérogènes basée sur la méthodologie *Platform-Based Design* [73]. Son métamodèle permet de décrire des concepts de plate-forme d'exécution d'un haut niveau d'abstraction. Le comportement est représenté explicitement par le biais de notions d'activités concurrentes (*Process*) et de communication entre les activités (*Media*). D'un point de vue analytique, cet environnement offre la possiblité d'utiliser des langages formels basés sur la logique LTL (*Linear Temporal Logic*) pour vérifier des propriétés fonctionnelles ou non-fonctionnelles d'applications qui peuvent être déployées sur une large gamme de plates-formes d'exécution. Le déploiement est assuré en établissant des relations (*Mapping*) entre les composants du système et les concepts de la plate-forme visée.

Synthèse de l'approche

Avec son environnement METROII [21], cette approche offre une meilleure séparation des besoins que GME. Pour autant et pour les mêmes raisons que précédemment, cet environnement n'est pas idéalement proposé pour le domaine du temps réel. Aussi d'un point

de vue vérification, celle-ci rendue possible après déploiement porte sur un niveau de précision qui n'est pas suffisamment détaillé en raison d'une considération des plates-formes d'un haut niveau d'abstraction.

2.2.2.2 Synthèse formelle du comportement des systèmes embarqués

Présentation générale de l'approche

ACSR + TROS Dans un soucis de vérification comportementale des SETRs, une approche formelle [42] a défini une synthèse comportementale d'application déployée sur une plate-forme particulière. Pour ce faire, l'application et la plate-forme visée par le déploiement sont modélisées séparément. L'application est, d'un côté, décrite par un statechart étendu (TROS, *Timed and Resource-oriented Statecharts*), tandis que la plate-forme ciblée est, de l'autre côté, formalisée avec l'aide d'un algèbre de processus (ACSR, *Algebra of Communicating and Shared Resources*). La sémantique d'exécution de TROS permet l'expressivité d'annotations temporelles et de contraintes liées aux ressources de la plate-forme sur la modélisation de l'application. Ces annotations permettent, par la suite, au modèle TROS d'intéragir avec l'algèbre ACSR et une composition des deux vient conclure le processus de synthèse du comportement de l'application déployée.

Comportement lié aux ressources et aux contraintes temporelles

Un travail préliminaire avait transposé le problème de considération de la plate-forme dans une IDM pour obtenir un TROS de l'application contrainte par la plate-forme au sens général. Le principe est basé sur une transformation du modèle comportemental de l'application en précisant les ressources logicielles de la plate-forme allouées (i.e., les ressources ordonnançables, l'allocation du processeur, l'assignation des priorités, les ressources partagées) et les contraintes temporelles (i.e., les pires temps d'exécution). Chaque fonctionnalité de l'application est représentée orthogonalement par un *Statechart* UML dans un modèle d'entrée (PIM) de la transformation. Un algorithme vient alors étendre le PIM pour générer un modèle comportemental de l'application (PSM) en TROS. Dans ce PSM, les ressources allouées et les contraintes temporelles sont annotées par des actions temporisées qui viennent compléter les *Statecharts* du PIM. Les ressources ordonnançables assignées d'une priorité nécessaires à l'application sont annotées en nombre limité suivant la plate-forme choisie. Les contraintes temporelles sont annotées en fonction du temps d'exécution alloué pour le traitement de chaque fonction (correspondance avec une ressource ordonnançable) au sein de l'application. Le PSM une fois obtenu, une traduction est nécessaire pour formaliser le TROS en un algèbre de processus ACSR. Pour ce faire, les actions temporisées sont extraites du modèle TROS et ensuite traduites formellement en ACSR. Le comportement temporisé et ordonnancé du système peut ainsi être contrôlé par bi-simulation au moyen de l'outil de vérification VERSA [18].

Comportement lié aux contraintes de l'API d'une plate-forme particulière

Cette transformation a révélé un manque d'exhaustivité dans la considération du comportement de la plate-forme. Le comportement lié aux mécanismes d'intéraction de l'API de la plate-forme considérée n'est pas pris en compte dans le PSM généré. Dans une vision étendue [43], une représentation formelle de ce comportement a été adoptée pour une description plus détaillée de la plate-forme. Par conséquent, l'ensemble des services offerts par la plate-forme pour la gestion des ressources (e.g., l'ordonnancement et la synchronisation des ressources ordonnançables, la communication entre les ressources, etc.) sont représentés indépendamment. Cette représentation est décrite formellement avec l'aide de l'algèbre de processus ACSR. Une synthèse de composition parallèle entre le modèle TROS (PSM) généré de l'application avec les contraintes et l'algèbre de processus ACSR a été formalisée. Cette synthèse permet de vérifier des propriétés non-fonctionnelles sur le modèle composé, toujours avec l'outil VERSA. La figure 2.7 représente un exemple de synchronisation entre une application simple d'affichage d'un message "*HelloWorld*" (repris de l'exemple de la figure 2.6) et une plate-forme OSEK/VDX. L'activité d'affichage de l'application a dans un

premier temps été transformée vers un PSM pour contraindre temporellement son exécution et déployée sur une tâche basique d'OSEK/VDX. Puis le PSM est composé avec un algèbre qui vient décrire le cycle de vie du mécanisme de blocage de l'ordonnanceur dans une plate-forme OSEK/VDX (pour l'exemple, cette description est appelée PDM même si nous ne nous trouvons pas dans un contexte MDA). L'objectif ici est de bloquer l'ordonnanceur pendant l'exécution de la tâche afin d'éviter toute préemption.

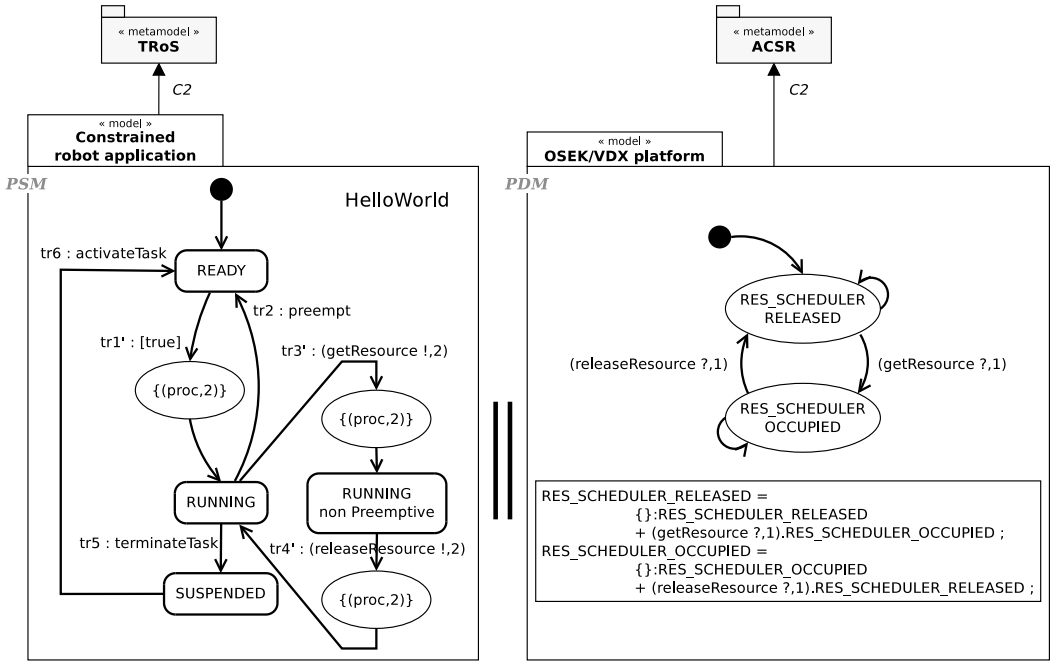


FIGURE 2.7 – Composition parallèle d'une application avec une plate-forme OSEK/VDX avec ACSR + TROS

Sur le PSM généré, à gauche de la figure, l'application est décrite suivant les états de la tâche OSEK/VDX. La transformation est venue contraindre le déclenchement des transitions tr1, tr3 et tr4 en y intercalant des actions temporisées {(proc,2)} (pour simplifier elles ne sont pas temporisées ici). À chaque action temporisée, le processeur proc est alloué et une priorité de 2 est assignée. Notons que deux actions partageant le même processeur pourraient être envisagées simultanément (e.g., deux tâches prêtes avant exécution). Dans ce cas là, l'action la plus prioritaire est exécutée. Enfin sur ce modèle, certains évènements peuvent être diffusés dans le but d'interagir avec les mécanismes de la plate-forme. Ces évènements sont aussi prioritisés et apparaissent entre parenthèses : l'évènement de diffusion (getResource !,2) représente par exemple le service offert pour verrouiller l'ordonnanceur (l'explication est donnée juste après dans la description du PDM). Lorsque un évènement diffusé se synchronise avec un mécanisme de la plate-forme, la transition concernée par cet évènement est déclenchée. Dans l'exemple que nous venons de citer, un changement d'état de la tâche en exécution est effectué une fois l'ordonnanceur verrouillé, afin d'éviter toute préemption (i.e., passage dans l'état RUNNING non Preemptive).

Sur la droite de la figure, le PDM décrit, sur la partie haute, un système de transitions simplifié du comportement du verrouillage de l'ordonnanceur avec la norme OSEK/VDX. Le mécanisme utilisé pour un tel verrouillage consiste dans l'acquisition et le relâchement d'une *ressource* OSEK/VDX partagée [11] pour coordonner les accès concurrents de tâches de priorités différentes. Dans OSEK/VDX, il est possible de consid-

11. Une *ressource* est le nom donné à une ressource partagée dans la norme OSEK/VDX

érer l'ordonnanceur comme une *ressource* grâce au nom prédéfini RES_SCHEDULER. Sur le système de transitions, les états de RES_SCHEDULER sont représentés, ainsi que les services d'acquisition et de relâchement décrits cette fois par des évènements de synchronisation, respectivement pour le verrouillage et le déverrouillage de l'ordonnanceur. Le cycle de vie de ce mécanisme est détaillé via l'algèbre de processus ACSR situé en dessous du système. Les états éponymes du système sont remplacés par des processus (i.e., RES_SCHEDULER_RELEASED et RES_SCHEDULER_OCCUPIED) qui évoluent sous l'activation des évènements de synchronisation pour décrire le passage d'un état à un autre. A l'instar du verrouillage de l'ordonnanceur, celui-ci intervient à chaque occurrence de l'évènement getResource? qui fait passer l'ordonnanceur de l'état RES_SCHEDULER_RELEASED vers l'état RES_SCHEDULER_OCCUPIED.

Synthèse de l'approche

La synthèse des deux modèles est donc respectivement fondée sur l'émission et la réception des évènements de diffusion et de synchronisation. Cette méthodologie est en adéquation avec notre problématique de conception détaillée des SETRs. Elle est sensibilisée par la représentation formalisée du comportement des mécanismes spécifiques des plates-formes logicielles d'exécution temps réel. Une vérification détaillée des SETRs est rendue possible en considérant la plate-forme qui contraint fortement l'application à déployer. Néanmoins, cette étude ne rentre pas dans un cadre de processus de déploiement. Une transformation a pourtant été implémentée dans un contexte MDA [42], mais elle ne considère que des contraintes haut niveau des plates-formes dans une approche implicite sans pivot.

2.2.3 Outils d'aide au déploiement et à la vérification

Les suites d'outils présentées maintenant mutualisent les déploiements la génération de modèles formels. Cet mutualisation offre l'avantage d'appliquer des activités de vérification sur les modèles d'applications déployées utiles à la génération de code.

2.2.3.1 Simulation comportementale après déploiement

Présentation de l'approche

Ptolemy Le projet Ptolemy [47] est consacré à la conception, la modélisation et la simulation hétérogène de sytèmes concurrents. Il permet la description de modèles d'exécution ou MoCs (*Model of Computation*) pour représenter le comportement de ces systèmes. Ces modèles supportent la conception orientée acteur (*actor-oriented design*) axée sur la concurrence et la communication entre les composants d'un SETR. Les composants sont ainsi perçus comme des acteurs. Cette méthodologie est comparée à la conception orientée objet. A l'image des méthodes qui permettent d'agir sur les valeurs des attributs des objets, des ports permettent d'interfacer les acteurs les uns avec les autres pour envoyer et recevoir des données.

Représentation et considération des mécanismes temps réel

Le métamodèle orienté acteur introduit dans son atelier PtolemyII est aussi adapté aux SETRs. Il permet en effet la représentation des SETRs cadencés par des évènements sous formes de MoCs multitâches temporisés (*Timed Multitasking*) [55]. Des composants peuvent ainsi être utilisés pour représenter explicitement des applications temps réel tout en considérant les mécanismes exécutifs liés aux plates-formes logicielles (e.g., les temps d'exécution, la préemption, etc.). Les MoCs peuvent par la suite être traduits en différentes sémantiques, même formelles, pour soit générer du code C/C++ ou Java [75], soit y appliquer des activités de simulation et vérification [72].

Synthèse de l'approche

Ptolemy offre une dualité de déploiement code-formalisation très complète permettant de considérer le comportement des SETRs dans leur ensemble. Cependant, les composants issus de cette approche orientée acteur, se destinent uniquement à la modélisation structurelle des applications temps réel déployées et non à celle des plates-formes. Cette consid-

ération implicite des plates-formes pénalise la séparation des domaines de l'application et des plates-formes d'exécution qui n'apparaît pas distinctement.

2.3 Positionnement

Dans cette section, nous proposons de revenir sur l'ensemble des travaux présentés dans ce chapitre et mis à contribution pour représenter et considérer le comportement des plates-formes logicielles d'exécution temps réel dans une IDM. Les contributions ont été catégorisées de différentes manières dans des tableaux de synthèse. Cette classification permet d'extraire les avantages et les inconvénients portés par chaque contribution vis-à-vis de notre suite d'outils SEXPISTOOLS. Cette comparaison positionne finalement nos axes de recherche en réponse aux questions posées dans le chapitre 1 à la page 19 pour compléter SEXPISTOOLS.

2.3.1 Constatations

Le premier tableau 2.1 présenté ci-après met en évidence les approches adoptées par chaque contribution pour représenter le comportement des plates-formes logicielles d'exécution. Les outils sont différenciés dans ce tableau selon ceux utilisés pour le déploiement, ceux utilisés pour générer des modèles en vue d'activités de vérification et ceux utilisés dans les deux cas.

	Représentation / Outils	Enfouie	Implicite		Explicite
			sans pivot	avec pivot	
Déploiement	TransPI			✓	
	Fujaba	✓			
	impliquant AADL			✓	
	SRM				✓
	DRIM				✓
Vérification	GME				✓(1)
	Metropolis				✓(1)
	ACSR + TROS		✓(2)		
Dépl. et Vérif.	Ptolemy		✓		

(1) Moins adapté au domaine du temps réel.
(2) Représentation formelle du comportement des plates-formes, mais pas dans un contexte MDA.

TABLE 2.1 – Approche de représentation du comportement adoptée par chaque outil

Le premier constat que nous pouvons faire est que seuls quatre outils ont opté pour une approche de représentation explicite. Ils proposent en effet un métamodèle pour décrire les plates-formes logicielles d'exécution. Néanmoins, aucun de ces outils n'est réellement destiné à la vérification formelle. GME et Metropolis qui contribuent pour la génération de modèles formels sont en effet moins adaptés au domaine du temps réel, tandis que SRM

et DRIM sont plutôt orientés génération de code. Néanmoins, DRIM propose des activités d'analyse temporelle.

Nous pouvons aussi retenir la stratégie adoptée pour décrire le comportement des plates-formes avec SRM et ACSR + TROS. Ces travaux se sont assurément focalisés sur la description de chaque ressource, voire même de chaque service avec SRM. D'un côté SRM via UML permet soit d'assigner un comportement exécutif sous forme de bribe de code, à chaque ressource grâce à la notion d'*OpaqueBehavior*, soit de définir un patron de conception sous forme de diagramme d'activités, à chaque service d'exécution. De l'autre côté, la synthèse ACSR + TROS, malgré l'absence de métamodèle pour la représentation des plates-formes, considère formellement le comportement de chaque mécanisme lié à une ressource sous forme d'algèbre de processus.

Dans le second tableau 2.2, les outils sont comparés suivant les trois critères de qualité présentés en introduction. Cette comparaison reflète l'impact de l'approche de représentation sur la considération du comportement des plates-formes dans un contexte MDA.

	Critères / Framework	Réutilisabilité	Portabilité	Maintenabilité
Déploiement	TransPI	$-$	$++$	$-$
	Fujaba	$+$	$-$	$-$
	impliquant AADL	$+$	$++$	$+$
	SRM	$++$	$++$	$++^{(1)}$
	DRIM	$++$	$++$	$++$
Vérification	GME	$++$	$-^{(2)}$	$-$
	Metropolis	$++$	$-^{(2)}$	$+$
	ACSR + TROS	$+$	$-^{(3)}$	$-$
Dépl. et Vérif.	Ptolemy	$++$	$+$	$-$

(1) Le comportement est néanmoins considéré de deux manières différentes.
(2) Le comportement est moins bien considéré pour le domaine du temps réel.
(3) Le comportement est uniquement modélisé de manière formelle en réponse à l'hétérogénéité des plates-formes.

Notation : $--$ (Pas adapté), $-$ (Peu adapté), $+$ (Adapté), $++$ (Très adapté) ;

TABLE 2.2 – Évaluation des outils sur la considération du comportement à travers leurs critères de qualité

Un premier point montre le peu d'outils adaptés à la maintenabilité des SETRS. Ce critère tient sur la bonne séparation des domaines de compétence en vue de l'indépendance de chaque intervenant sur la conception de ces systèmes. Or, l'emploi de technologies diverses et variées entraînent parfois certaines dépendances vis-à-vis des méthodologies adoptées, à l'image de GME où la description de la plate-forme est un amalgame de concepts exécutifs et de règles de transformation. Il en est de même pour Fujaba qui offre une approche représentation enfouie des plates-formes. Cette manière de représenter oblige le spécialiste de la transformation à être dépendant de la plate-forme. Autre exemple, cette fois avec Ptolemy qui ne distingue pas explicitement le comportement de la plate-forme à celui de l'application. Un bémol est mis sur SRM qui offre une bonne séparation des préoccupations métiers. L'approche exige néanmoins de considérer le comportement de deux manières différentes, ce qui oblige le spécialiste à une double réflexion dans l'approche de représenta-

tion. DRIM est mieux adapté sur ce point, ce processus distingue aisément les interventions des parties prenantes.

Sur un autre critère, les générateurs de code montrent une meilleure flexibilité que les outils servant à la génération de modèles destinés à la vérification. Ceci est dû aux approches explicite et implicite avec pivot qui obligent à employer des stratégies d'adaptabilité pour considérer les représentations hétérogènes des plates-formes, à travers les processus. Ces stratégies sont mises en œuvre de différentes façons, à savoir par le biais de règles de transformations configurables pour TransPI, ou bien par paramétrage de code pour l'approche impliquant AADL, mais encore via des rôles d'identification pour le *package* SRM. Une nuance est toutefois à noter quant à GME et Metropolis. Malgré les efforts consentis pour représenter explicitement les plates-formes, ces outils ne privilégient pas la considération de plates-formes hétérogènes telles que celles du domaine du temps réel.

2.3.2 Orientations

Les contributions que nous venons de comparer permettent de distinguer plusieurs axes de recherche pour répondre aux questions soulevées dans la problématique détaillée à la page 19. Nous avons donc orienté nos travaux :

Vers une extension de RTEPML

Le but recherché est de dédier aussi notre langage de modélisation à la représentation explicite du comportement des plates-formes logicielles d'exécution temps réel. Cette approche doit faciliter l'indépendance de chaque spécialiste d'une plate-forme particulière contrairement à GME et Metropolis qui, de part leur généricité, ne conviennent pas au domaine du temps réel.

Vers l'intégration de concepts comportementaux dans RTEPML

À l'image du *package* SRM, un certain nombre de concepts doivent être identifiés au sein du métamodèle pour représenter le cycle de vie de chaque ressource et de chaque service d'exécution d'une plate-forme particulière.

Vers l'intégration de concepts formels dans RTEPML

La stratégie recherchée se rapproche de la synthèse formelle mise en oeuvre pour la conception détaillée des SETRs avec ACSR + TROS. La vérification de propriétés non fonctionnelles d'applications déployées exige la considération du comportement de chaque mécanisme lié à la plate-forme considérée. Cette considération réside donc dans la formalisation du cycle de vie de chaque ressource et de chaque service.

Vers la perpétuation de la notion de rôles dans RTEPML

Le processus de génération de modèles formels que nous cherchons à mettre en œuvre doit être flexible comme TransPI, l'environnement qui implique AADL, ou bien encore SRM. Le processus de déploiement implémenté dans SEXPISTOOLS repose déjà sur cette notion de rôles pour localiser les concepts structurels et hétérogènes des plates-formes. Des rôles doivent par conséquent être identifiés d'un point de vue comportemental.

Vers l'intégration du comportement de l'application dans le modèle de la plate-forme

L'approche de considération des plates-formes adoptée dans le processus de déploiement de SEXPISTOOLS doit être respectée. Il est somme toute légitime de s'appuyer sur la méthode qui intègre l'application déployée dans le modèle de la plate-forme, pour aussi y intégrer son comportement.

Vers un processus propre à la génération de modèles formels

L'indépendance des activités de vérification vis-à-vis de la génération de code doit être assurée. Dans un contexte similaire à celui de Fujaba et de Ptolemy, SEXPISTOOLS doit offrir un processus dual pour générer, d'un côté, du code implémentable et générer, de l'autre côté, des modèles formels pour la vérification du comportement non-fonctionnel des applications déployées. Nous ne nous intéresserons pas à la génération de code dans cette thèse.

Deuxième partie

Prise en compte du comportement d’une plate-forme d’exécution

3

Modélisation du comportement avec RTEPML

Sommaire

De part sa syntaxe abstraite, le langage de modélisation RTEPML permet de représenter explicitement les plates-formes logicielles d'exécution temps réel. Ce langage permet la description de PDM en distinguant la plate-forme de l'application qui est intégrée.

Dans ce chapitre, nous proposons une extension de RTEPML afin d'exprimer aussi le comportement des plates-formes d'exécution au sein d'un même PDM. Au même titre que la partie structurelle présentée dans le chapitre précédent, l'objectif ici est d'intégrer un modèle comportemental de l'application déployée en vue du processus de génération de modèles formels. Une approche est donnée ici dans l'optique de composition d'un tel modèle comportemental.

Une première section étend la syntaxe abstraite de RTEPML en vue de représenter le comportement des plates-formes. Puis, une seconde section vient intégrer un modèle comportemental de l'application au sein du modèle de la plate-forme. Ensuite, cette syntaxe est couplée avec un langage de modélisation du comportement en vue d'une traduction formelle. Aussi, la notion de rôles abordée dans le chapitre précédent sera confortée à travers

ce chapitre pour expérimenter la composition du modèle comportemental de l'application déployée. Une synthèse vient conclure ce chapitre dans une dernière section.

3.1 Intégration d'un modèle comportemental de plate-forme

L'extension de RTEPML est fondée sur l'implantation d'un modèle comportemental de la plate-forme dans le modèle structurel introduit dans la présentation de RTEPML au chapitre précédent à la page 23. Dans une vision générale, le modèle comportemental vient compléter le modèle structurel en associant des fragments comportementaux aux ressources et aux services. Cette philosophie s'appuie sur la façon dont les outils de notre état de l'art, notamment SRM et ACSR + TROS, ont cherché à représenter le comportement des composantes logicielles des plates-formes d'exécution temps réel.

3.1.1 Comportement des ressources et des services

L'idée retenue est montrée sur la figure 3.1 [1]. A l'image des ressources qui font partie intégrante du modèle structurel de plate-forme, les fragments comportementaux viennent constituer le modèle comportemental de plate-forme. Ces fragments sont appelées des prototypes comportementaux (*BehavioralPrototype*) pour donner un sens à la définition originale des comportements des ressources qui composent ce modèle comportemental. Ainsi, chaque ressource peut se voir assigner un prototype comportemental pour représenter son cycle de vie au sein de la plate-forme. Une précision est à donner quant à l'attribut *name* de *BehavioralPrototype*. Cette attribut qualifie chaque prototype comme un élément nommé [25] et permet ainsi de les identifier à travers les modèles. Cet attribut de nommage apparaîtra dans les autres concepts que nous introduirons, comme c'était déjà le cas dans RTEPML.

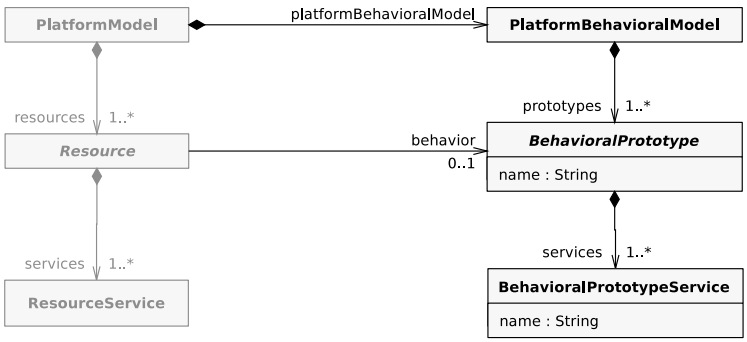


FIGURE 3.1 – Extrait du DSML RTEPML étendu : les prototypes comportementaux

Les prototypes comportementaux ont été classés de façon similaire à la hiérarchisation des ressources logicielles (voir au chapitre précédent à la page 23) initialement proposée dans RTEPML [12]. La figure 3.2 décrit cette hiérarchie. Toutes les familles n'ont pas été représentées afin d'alléger la description. Seules les trois grandes familles qui vont nous intéresser par la suite apparaissent ici. Les partitions mémoires ne seront pas évoquées puisque nous nous sommes orientés vers les contraintes temporelles et non spatiales.

Aussi, les prototypes comportementaux ne concernent pas uniquement les ressources. Les services qui sont offerts par les ressources et visibles depuis les APIs des plates-formes ont un impact direct sur le comportement des ressources. Pour cette raison, sur l'extrait ci-dessus, chaque prototype comportemental est constitué de descriptions comportementales (*BehavioralPrototypeService*) de services. Prenons l'exemple d'une ressource ordonnançable (i.e., une tâche) ; celle-ci peut être impactée par un service d'acquisition d'une

1. Tout au long de ce chapitre, tout concept ou toute relation existant avant l'extension comportementale de RTEPML apparaît en gris dans les extraits présentés

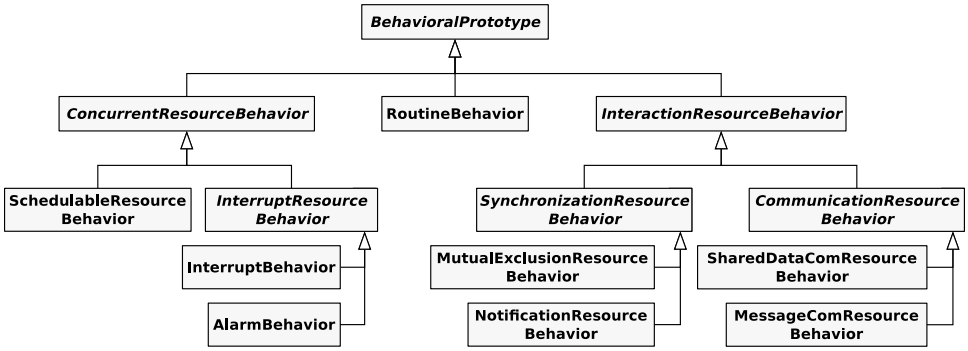


FIGURE 3.2 – Extrait du DSML RTEPML étendu : hiérarchie des prototypes comportementaux

ressource d'exclusion mutuelle (i.e., un sémaphore). Si cette ressource d'exclusion mutuelle n'est pas disponible au moment où la ressource ordonnançable appelle ce service d'acquisition, elle pourrait être mise en attente par l'ordonnanceur et ainsi libérer le processeur pour une autre ressource concurrente en attente d'exécution.

3.1.2 Éléments de comportement

Le sens donné à chaque prototype comportemental tient sur l'agencement des éléments qui vont constituer une telle description. RTEPML doit permettre de décrire génériquement le comportement des plates-formes logicielles à travers ces descriptions. Comme précisé sur la figure 3.3, nous considérons par conséquent que chaque prototype peut être constitué d'un ensemble d'éléments de comportement. Le concept *BehaviorElement* permet la représentation abstraite de ces éléments qui peuvent être soit des éléments de comportement composite, soit des éléments de comportement atomique. Le concept d'élément de comportement composite *Composite* tient de l'application du patron de conception d'objet composite conçu pour la manipulation récursive d'un ensemble d'objets similaires. Il répond ici à la possibilité de définir un élément de comportement pouvant regrouper récursivement des sous-éléments de comportement. Ainsi un élément de comportement composite se différencie d'un élément de comportement atomique (concept *Atomic*) qui est un élément de comportement simple.

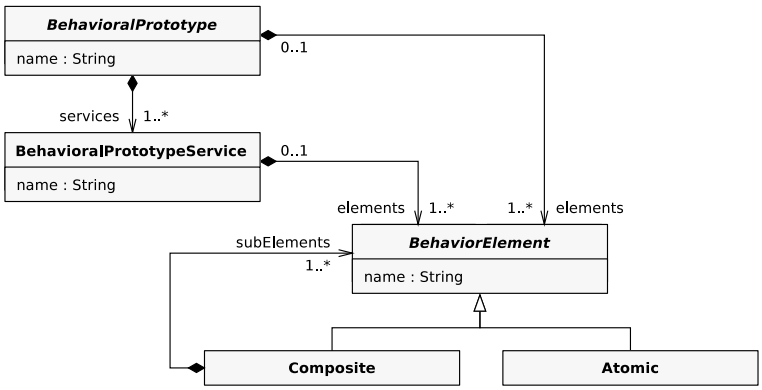


FIGURE 3.3 – Extrait du DSML RTEPML étendu : les éléments de comportement

Cette orientation se vérifie concrètement avec l'exemple de la figure 2.7 du chapitre précédent. Dans cet exemple, le cycle de vie d'une tâche OSEK/VDX est décrit sous forme de machine à états adapté pour le temps réel avec le langage de modélisation TROS. Bien

que cette description apparaît dans un PSM, elle peut très bien s'apparenter à un prototype comportemental d'un PDM constitué de plusieurs éléments. Dans ce cas, certains de ces éléments (e.g., état, transition, etc.) se conformeraient à des éléments de comportement atomique, tandis que d'autres (e.g., action temporisée, évènement diffusé, etc.) se conformeraient à des éléments de comportement composite.

3.1.3 Identification de rôles

Dans le chapitre précédent à la page 23, nous avons introduit la notion de rôles pour repérer des propriétés et des services au sein des ressources. Au même titre que cette identification structurelle, ces rôles ont été dupliqués sur l'aspect comportemental. Ils permettent ainsi de repérer respectivement des éléments de comportement et des prototypes comportementaux de service au sein des prototypes comportementaux de ressources. Contrairement aux services, aucun concept comportemental n'a été associé aux propriétés. Cette raison tient du fait qu'une propriété vient soit quantifier, soit qualifier une ressource. En conséquence, l'impact comportemental qu'arbore une propriété se résume au rôle que joue un élément de comportement au sein d'un prototype comportemental. La figure 3.4 représente un extrait de cette duplication de rôles.

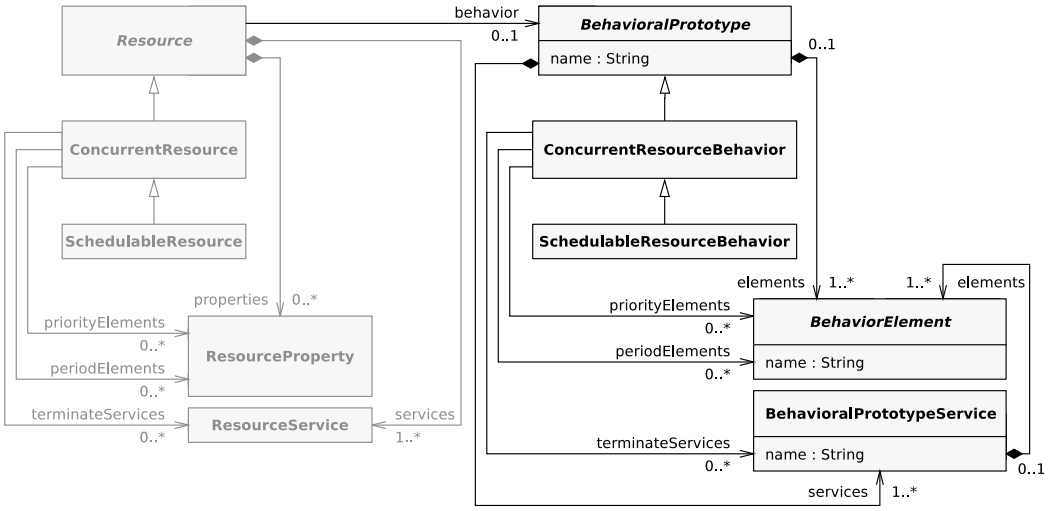


FIGURE 3.4 – Extrait du DSML RTEPML étendu : duplication des rôles

Sur cette figure, les rôles *priorityElements* et *periodElements* ont par exemple été dupliqués pour identifier les éléments de comportement qui jouent les rôles d'éléments de priorité et d'éléments de période au sein du prototype comportemental d'une ressource concurrente. Pour illustrer ce besoin, revenons sur l'exemple de la tâche OSEK/VDX, figure 2.7 décrite avec le langage de modélisation TROS. Au sein de cette description, la priorité de la tâche est annotée au travers d'actions temporisées. Cette annotation se conformerait donc à un élément de comportement atomique. Ainsi, grâce à la duplication du rôle *priorityElements* entre une ressource concurrente et un élément de comportement, l'annotation pourrait être identifiée.

Les services sont aussi identifiables à travers les prototypes comportementaux des ressources à l'image du rôle *terminateServices*. Ce rôle permet l'identification des prototypes comportementaux de service qui jouent les rôles de services de terminaison. Comme nous l'avons mentionné plus haut à la page 48, les services sont décrits par des éléments de comportement qui sont soit atomiques, soit composites. Dans notre exemple de tâche

OSEK/VDX, un appel à un service de terminaison de la tâche est représenté soit par un évènement simple qui se conformerait à un élément de comportement atomique, soit par un évènement diffusé. L'évènement diffusé est rencontré dans le cadre du verrouillage et du déverrouillage de l'ordonnanceur. Dans ce cas de figure, l'évènement diffusé est décrit par un ensemble d'éléments qui contraignent son émission (e.g., allocation du processeur, priorité, etc.). Ces évènements se conformeraient du coup à des éléments de comportement composite.

3.2 Intégration d'un modèle comportemental d'application

Dans RTEPML, l'application est intégrée au sein du modèle de la plate-forme. Cette stratégie répond au besoin de modélisation détaillée de l'application déployée. Ce déploiement se traduit par l'instanciation des concepts de la plate-forme considérée pour l'exécution des concepts de l'application. Dans cette prespective, la notion de concept d'instance de ressource a été introduite dans RTEPML pour enrichir le modèle de plate-forme et mettre en œuvre l'application. Une relation de typage existe déjà entre une instance et la ressource qu'elle incarne au sein de l'application (cf, Intégration de l'application dans RTEPML, page 26). Nous avons étendue cette configuration d'un point de vue comportemental en préservant la notion d'instanciation sur les ressources.

3.2.1 Clonage d'éléments de comportement

De façon similaire au modèle de plate-forme, l'application intègre un modèle comportemental, conformément à la figure 3.5. Ce modèle comportemental (*ApplicationBehavioralModel*) est le recueil des éléments de comportement issus des prototypes comportementaux. Ces éléments se conforment aussi au concept abstrait *BehaviorElement* mais ils diffèrent cependant des éléments contenus dans les prototypes comportementaux. C'est pour cette raison que la multiplicité des cardinalités sur les classes contenantes sont de $0..1$.

L'intégration d'un modèle comportemental de l'application repose donc sur la duplication des éléments qui constituent chaque prototype concerné par l'application. En effet, seuls les prototypes associés aux ressources qui typent les instances de l'application devront être instanciés. De ce fait, chaque ensemble d'éléments instanciés devra subir une duplication dans le but de caractériser chaque élément suivant l'application. Ce mécanisme est l'objet du chapitre. De par son principe qui consiste à dupliquer des éléments instanciés, nous l'appellerons clonage. Les éléments clonés *clonedElements* constituent par conséquent le modèle comportemental de l'application.

Contrairement à la partie structurelle, nous n'avons pas voulu définir de concept pour décrire le comportement des instances. Définir un concept abstrait d'instance d'un prototype comportemental reviendrait à représenter des concepts inutiles sur le modèle de l'application. Ces concepts ne serviraient qu'à une mise en relation intermédiaire entre des éléments de comportement clonés et les instances de ressources (ou les appels de services) à l'origine du clonage. Un référencement est suffisant pour établir une telle mise en relation.

3.2.2 Référencement des éléments de comportement

Nous venons de mettre en évidence le clonage d'éléments de comportement pour le compte de l'application. Ce mécanisme va entraîner la dispersion d'ensemble de ces éléments qui résultent de l'instanciation des prototypes comportementaux. Cette manœuvre soulève d'idée d'assembler les ensembles d'éléments clonés afin de composer un modèle

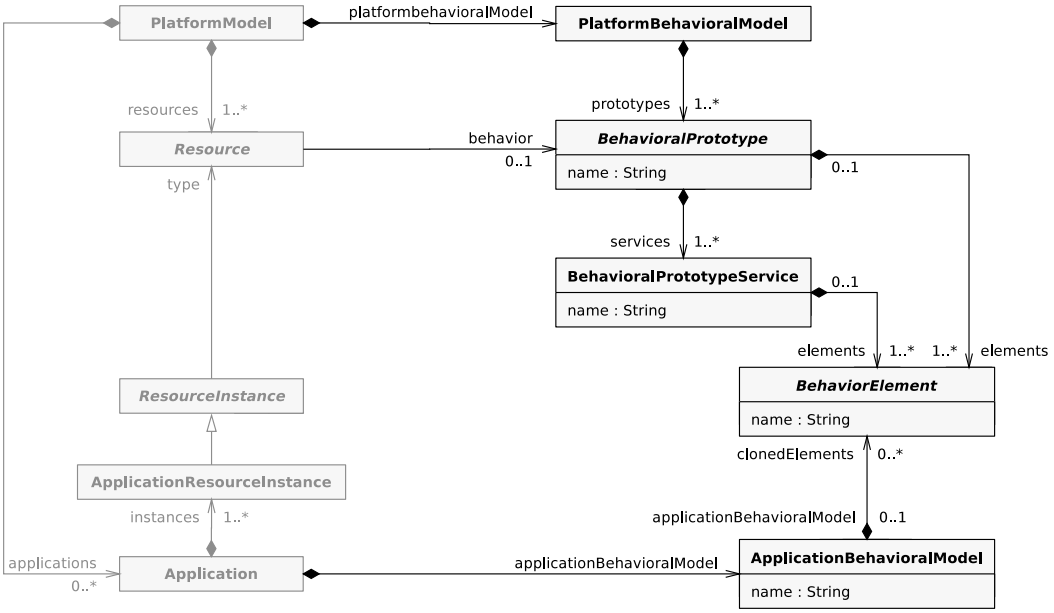


FIGURE 3.5 – Extrait du DSML RTEPML étendu : les éléments de comportement clonés

comportement global de l'application. Or, cette composition découle de la structure de l'application. Les concepts structurels de l'application alors impliqués dans cette composition sont les instances de ressources et les appels de services. La figure 3.6 propose cette anticipation.

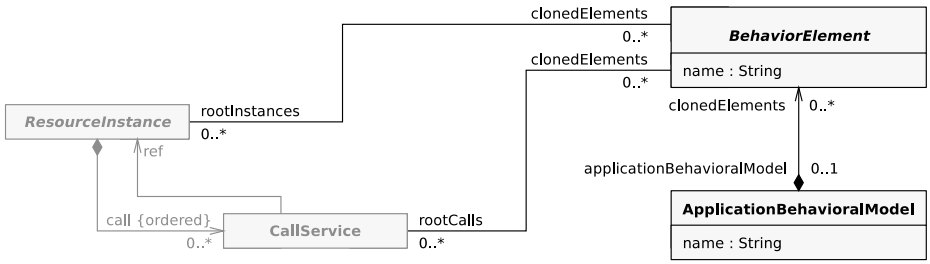


FIGURE 3.6 – Extrait du DSML RTEPML étendu : référencement des éléments clonés

Chaque élément de comportement peut être affecté de plusieurs instances de ressource (*rootInstances*) ou de plusieurs appels de services (*rootCalls*) de l'application. Le terme *root* sera retrouvé par la suite pour désigner les instances et les appels à l'origine du clonage des éléments. En effet, seuls les ressources et les services sont associés à des prototypes comportementaux. La pluralité des cardinalités affectées sur ces affectations est promue par l'assemblage des ensembles d'éléments de comportement issus de prototypes différents. Nous verrons dans le chapitre suivant que cet assemblage demande de fusionner certains éléments qui n'ont pas la même origine.

3.3 Modélisation des prototypes comportementaux

Nous venons de parcourir la nouvelle version abstraite de RTEPML qui a été étendue pour représenter le comportement. Cette syntaxe est volontairement découplée d'un langage spécifique de modélisation du comportement dans le but de garder une généricité de représentation des prototypes comportementaux. Toutefois, un choix doit être fait quant au

langage qui permet d'exprimer ce comportement pour qu'il soit exploitable, dans notre cas, par des outils de vérification. En vue d'appliquer des activités de vérification, le choix d'un langage formel est justifié dans un premier temps. Puis, notre stratégie de représentation du comportement des prototypes dans ce langage est exposé sans remettre en cause la syntaxe de RTEPML.

3.3.1 Justification du choix du langage de modélisation

Le choix d'un langage formel pour modéliser le comportement résulte du compromis entre sa puissance de représentation et sa puissance de validation. C'est le cas des réseaux de Petri (PN, *Petri Net*) [69] qui ont un grand pouvoir d'expression même si sa puissance de représentation s'avère non concise. Ces réseaux ont été développés pour recouvrir la modélisation du comportement de systèmes importants tels que les systèmes de production, les systèmes automatisés, les systèmes informatiques ou bien encore les systèmes de communication, etc. Les PNs permettent de modéliser les systèmes à évènements discrets qui donnent lieu à des phénomènes de synchronisation et de concurrence sans l'aspect temporel. Pour palier ce manque d'expressivité, les réseaux de Petri temporels (TPN, *Time Petri Net*) [59] [10] ont été adoptés. Cette extension des PNs, permet de représenter des contraintes d'échéances à intervalles de temps continu par l'intermédiaire d'horloges.

La construction des modèles comportementaux recherchée doit aboutir à des modèles d'application multitâche de systèmes temps réel. En conséquence, nous avons choisi les TPNs pour traduire le comportement des prototypes comportementaux. Ce langage exprime formellement des modèles avec du synchronisme et du parallélisme. De plus, le temps réel implique des contraintes temporelles. Les TPNs répondent aussi au besoin sémantique de représentation de l'évolution du temps grâce aux horloges.

Dans un TPN, une horloge implicite et un intervalle de temps sont associés à chaque transition du réseau. Cette horloge mesure le temps à partir du moment où la transition est continuellement activée, tandis que la transition est interprétée comme une condition de franchissement : la transition, une fois activée, est seulement franchissable (ou tirable) si la valeur ou la valuation de son horloge appartient à son intervalle de temps.

La définition formelle des TPN va maintenant être donnée. Nous représenterons par $\mathbb{N}$ l'ensemble des nombres naturels, $\mathbb{R}_{\geq 0}$ l'ensemble des nombres réels non négatifs, $\emptyset$ l'ensemble vide, et $\mathbf{0}$ le vecteur nul.

Définition 1 (TPN) *Un TPN $\mathcal{T}$ est un tuple $\langle P, T, \mathsf{Pre}, \mathsf{Post}, m_0, I_s \rangle$ où :*
- *P est un ensemble non vide et fini de* places *;*
- *T est un ensemble non vide et fini de* transitions *;*
- *$\mathsf{Pre} : P \times T \to \mathbb{N}$ est la* fonction d'incidence arrière *;*
- *$\mathsf{Post} : P \times T \to \mathbb{N}$ est la* fonction d'incidence avance *;*
- *m_0 est le* marquage initial *du réseau ;*
- *$I_s : T \to \mathbb{N} \times (\mathbb{N} \cup \{+\infty\})$ assigne un* intervalle de temps *statique à chaque transition.*

Le *marquage* d'un TPN $\mathcal{T}$ est une application de P vers $\mathbb{N}$ donnant pour chaque place le nombre de jetons qu'elle contient. Une transition $t \in T$ est dite *sensibilisée* par un marquage m, désigné par $t \in enabled(m)$, si l'ensemble de ses places d'entrée contiennent "suffisamment" de jetons ; plus formellement, $enabled(m) = \{\, t \in T \mid \forall p \in P,\ m(p) \geq \mathsf{Pre}(p,t) \,\}$. Une transition $t \in T$ est dite *nouvellement sensibilisée* par le franchissement d'une transition t_f à partir d'un marquage m, désigné par $t \in\uparrow enabled\,(m, t_f)$, si elle est sensibilisée par le marquage final m_f défini $\forall p \in P,\ m_f(p) = m(p) - \mathsf{Pre}(p, t_f) + \mathsf{Post}(p, t_f)$ mais

non par le marquage intermédiaire m_i défini $\forall p \in P,\ m_i(p) = m(p) - \mathsf{Pre}(p, t_f)$. Plus formellement, $\uparrow enabled(m, t_f) = enabled(m_f) \bigcap ((T \setminus enabled(m_i)) \cup \{t_f\})$.

Enfin, pour tout intervalle I_s, nous désignons par $I_s{}^{\downarrow}$ le plus petit intervalle gauche fermé que peut contenir I_s avec pour borne minimale 0. Pour chaque transition tr, x_{tr} représente l'horloge associée. Les valuations sur l'ensemble des horloges $\{\ x_{tr} \mid tr \in T\ \}$ sont considérées et par abus de langage, nous écrivons $v(tr)$ au lieu de $v(x_{tr})$ pour désigner la valuation de l'horloge associée avec tr.

Afin de modéliser des comportements tels que des exécutions conditionnelles, des mécanismes de préemption, etc., les TPN ont été étendus avec des *arcs de lecture* (représentés par la suite avec un carré blanc au lieu d'une flèche normale) et des *arcs d'inhibition* (représentés avec un cercle blanc). Ces arcs ont seulement un impact sur les règles de sensibilisation du réseau. Ils n'ont en effet aucun impact sur le marquage obtenu par le franchissement d'une transition : les arcs de lecture testent la présence de jetons dans les places sans les consommer, tandis qu'un arc d'inhibition est utilisé pour stopper l'écoulement du temps sur une transition, aussi longtemps que la place qui se trouve en entrée de l'arc reste marquée par au moins un jeton.

Définition 2 (TPN avec arcs de lecture/inhibition) *Un TPN avec des arcs de lecture et d'inhibition (RI_TPN) est un tuple $\mathcal{T}_{RI} = \langle \mathcal{T}, \mathsf{Read}, \mathsf{Inh} \rangle$ où :*

- $\mathcal{T} = \langle P, T, \mathsf{Pre}, \mathsf{Post}, m_0, I_s \rangle$ *est un TPN,*
- $\mathsf{Read} : P \times T \to \mathbb{N}$ *est la* fonction de lecture *;*
- $\mathsf{Inh} : P \times T \to \mathbb{N} \cup \{+\infty\}$ *est la* fonction d'inhibition [2].

Non formellement, une transition est sensibilisée "s'il existe suffisamment de jetons" dans les places reliées par soit des arcs d'entrée ou soit des arcs de lecture *et* "s'il n'existe pas trop de jetons" dans les places reliées par des arcs d'inhibition. Plus formellement, la définition de l'ensemble des transitions sensibilisées par un marquage m est modifié comme suit :

$$enabled(m) \quad = \quad \{t \in T \mid \forall p \in P, \mathsf{Inh}(p,t) > m(p) \geq \max(\mathsf{Pre}(p,t), \mathsf{Read}(p,t))\}$$

La définition de l'ensemble des transitions nouvellement sensibilisées à partir d'un marquage m par le franchissement d'une transition t_f est semblablement modifiée.

Définition 3 (Sémantique des RI_TPN) *La sémantique opérationnelle d'un TPN avec des arcs de lecture et d'inhibition $\mathcal{T}_{RI}$ défini au-dessus est donnée par le système de transitions temporisé $\mathcal{S} = (Q, q_0 \to)$ tel que :*

- $Q = \mathbb{N}^P \times \mathbb{R}_{\geq 0}^T$ *;*
- $q_0 = (m_0, \mathbf{0})$ *;*
- $\to \in Q \times (T \cup \mathbb{R}_{\geq 0}) \times Q$ *est la* relation de transition *qui est composée des relations de transition discrète et continue suivantes :*
  - *la* relation de transition discrète *est définie :* $\forall t_f \in T$ *par* $(m, v) \xrightarrow{t_f} (m', v')$ *si et seulement si :*
    - $(t_f \in$ *enabled* (m) *;*
    - $v(t_f) \in I_s(t_f)$ *;*
    - $\forall p \in P, m'(p) = m(p) - \mathsf{Pre}(p, t_f) + \mathsf{Post}(p, t_f)$ *;*
    - $\forall t \in T, v'(t) = \begin{cases} 0 \textbf{ if } t \in \uparrow \textit{enabled}\,(m, t_f) \\ v(t) \textbf{ otherwise} \end{cases}$ *;*
  - *la* relation de transition continue *est définie :* $\forall d \in \mathbb{R}_{\geq 0}$ *par* $(m, v) \xrightarrow{d} (m, v')$ *si et seulement si* $\forall t \in enabled(m), \forall \delta \in]0; d], (v(t) + \delta) \in I_s{}^{\downarrow}(t)$.

2. Si aucun arc d'inhibition relie une transition t vers une place p, alors $\mathsf{Inh}(p, t) = +\infty$.

3.3.2 Héritage d'élément de comportement

La modélisation désirée requiert de coupler RTEPML avec le métamodèle des TPNs. Ce couplage repose sur un mécanisme qui pour autant ne doit pas compromettre l'utilisation de RTEPML à travers le processus de génération de modèles formels (en TPN pour ce qui nous concerne). Comme abordé, les règles de transformation que nous voulons mettre en œuvre, doivent uniquement manipuler les concepts de RTEPML et non ceux des TPNs.

Le terme couplage évoque finalement l'introduction de techniques de composition de modèles ou de métamodèles dans une IDM [19] [32]. Ces techniques s'adonnent à l'utilisation d'un mécanisme tel que le *matching* pour établir des correspondances entre les concepts en vue d'une représentation couplée. Cependant ces techniques ne favorisent pas l'idée d'indépendance du concepteur qui chercherait à modéliser formellement une plate-forme. Cette solution oriente en effet la conception vers une modélisation conjointe de la plate-forme entre un utilisateur de RTEPML et un utilisateur des TPNs. De plus, un *mapping* entre les différents concepts serait nécessaire pour établir les correspondances à travers les règles de transformation du processus de génération.

Le couplage doit donc s'opérer dans RTEPML. Une autre solution est d'intégrer le métamodèle des TPNs dans celui de RTEPML. Une possibilité consisterait alors à associer les concepts des deux métamodèles par des relations. Cette manœuvre a pour avantage de préserver la syntaxe de RTEPML, contrairement au mécanisme de composition. Composer des concepts de RTEPML avec ceux des TPNs reviendrait à manipuler les concepts des TPNs au sein des règles de transformation.

La solution que nous proposons couple les concepts des TPNs avec ceux de RTEPML par le biais du mécanisme d'héritage. La figure 3.7 décrit les conséquences de ce choix.

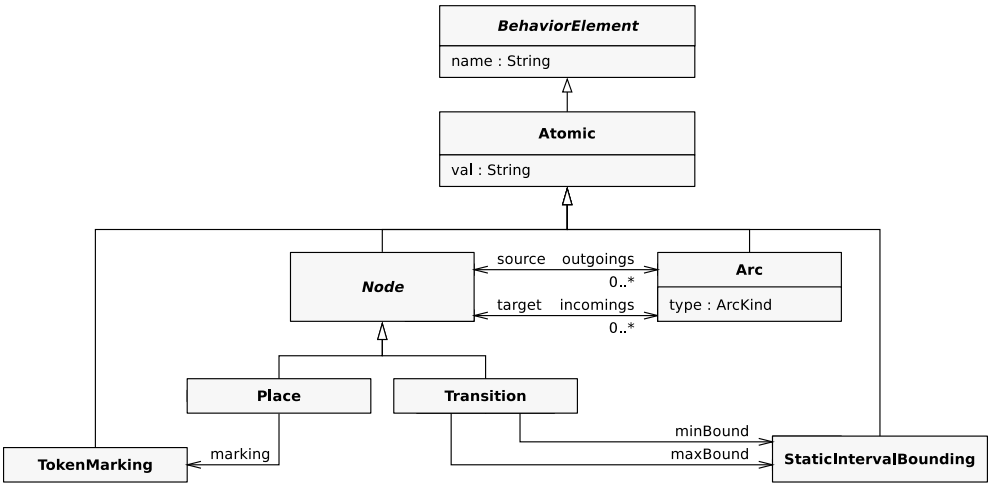


FIGURE 3.7 – Extrait du DSML RTEPML étendu : héritage d'élément de comportement atomique

Dans cet extrait, tous les concepts des TPNs pouvant être associés à des rôles de RTEPML héritent du concept *Atomic*. Cette stratégie permet la représentation formelle de chaque élément de comportement atomique au sein d'un prototype comportemental et ainsi de les identifier. Cette extension est suffisamment simple pour qu'elle ne soit pas détaillée. Toutefois, une précision est à noter quant aux concepts *TokenMarking* et *StaticIntervalBounding*. Ces deux concepts apparaissent ici pour permettre leur quantification, notamment grâce à l'attribut *val*. À l'instar du déclenchement d'une transition que nous chercherions à borner pour contraindre l'exécution périodique d'une ressource concurrente, le bor-

nage de son intervalle statique serait identifiable au moyen du rôle *periodElements* de la figure 3.4.

Précisons aussi que le poids des arcs (résultat des fonctions d'incidence avant et arrière) pourrait être vu comme un élément de comportement atomique. Mais dans la suite de cette étude, de tels éléments ne seront pas associés à des rôles.

3.3.3 Identification de rôles de composition

L'intégration de concepts formels tels que ceux des TPNs dans RTEPML offre la possibilité de formaliser la représentation des prototypes comportementaux. Ainsi chaque élément de comportement correspond soit à une description formelle si il est composite, soit à un élément formel si il est atomique. Comme nous l'avons déjà annoncé à la page 51, ces éléments de comportement sont amenés à être clonés puis composés suivant le modèle comportemental de l'application à déployer. Cette démarche laisse entrevoir la nécessité d'identifier les éléments de comportement qui jouent un rôle de point de connexion à travers les prototypes comportementaux. Elle vient compléter le référencement des éléments de comportement à la page 51.

Des rôles ont donc été identifiés dans ce but. La figure 3.8 met en évidence un ensemble de rôles adaptés pour cibler les points de connexion d'un prototype comportemental d'une ressource concurrente. Ces rôles ont été pensés en fonction des mécanismes et des échanges possibles avec les autres ressources et les services des plates-formes logicielles. Précisons toutefois que des rôles de composition ont aussi été identifiés pour le compte des prototypes comportementaux de service. Néanmoins, ces rôles ont volontairement été confondus avec ceux des prototypes comportementaux des ressources puisqu'ils jouent les mêmes rôles. Ce choix évite ainsi une redondance d'informations entre les rôles joués au sein d'un même prototype d'une ressource et ceux joués au sein des prototypes des services qui caractérisent la ressource.

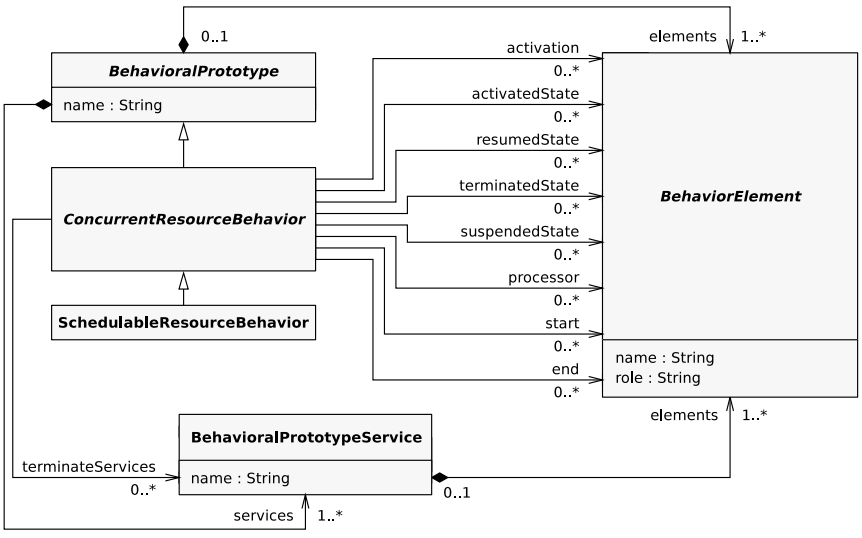


FIGURE 3.8 – Extrait du DSML RTEPML étendu : assignation de rôles de composition

Pour illustrer, imaginons un service de terminaison d'une tâche ordonnançable dont le comportement se conformerait au concept *SchedulableResourceBehavior*. D'un point de vue comportemental, le service de terminaison fait passer la tâche d'un état "en exécution" à un état "terminé". Par conséquent, les rôles de composition *resumedState* et *terminatedState* ont été identifiés à la fois au sein du prototype comportemental de service qui joue

le rôle de *terminateServices* et au sein du prototype comportemental de ressource concurrente *ConcurrentResourceBehavior*. Cette identification multiple de rôles implique de ce fait une pluralité des cardinalités sur l'ensemble des rôles de composition et évite des doublons entre les prototypes des ressources et ceux des services. Les rôles identifiés dans cet exemple ne représentent pas une liste exhaustive. Ceux qui apparaissent ici sont utiles à la représentation qui suit dans ce chapitre, page 58, comme :

- *processor* : l'élément qui joue le rôle de "processeur". Cet élément permet à ces ressources de concurrencer leurs exécutions en se le partageant mutuellement ;
- *start* et *end* : les éléments qui jouent les rôles de "début" et de "fin" d'exécution d'une ressource concurrente.

L'attribut *role* a aussi été ajouté au concept d'élément de comportement. Il permet à chaque élément cloné d'être traçable au sein des ensembles d'éléments clonés référencés (voir page 51) tout au long de la composition.

3.3.4 Identification de rôles de mécanismes spécifiques

D'autres descriptions sont à prendre en considération d'un point de vue comportemental. Elles influent notamment sur les interactions entre les ressources concurrentes et proviennent de mécanismes spécifiques à la plate-forme. Comme mécanisme, nous pouvons retenir le principe d'ordonnancement des ressources concurrentes. Parmi ces spécificités propres aux plates-formes, l'ordonnancement coopératif (*cooperative scheduling*) consiste par exemple à empêcher la mise en exécution d'une ressource concurrente éligible si une autre de priorité supérieure est aussi éligible. Comme tout autre exemple, nous pouvons aussi retenir l'ordonnancement préemptif basé sur les priorités qui cette fois va rendre éligible une ressource concurrente de priorité inférieure en exécution lorsqu'une autre de priorité supérieure se trouve éligible.

Suivant le langage de modélisation utilisé pour exprimer ces mécanismes, il est plus ou moins nécessaire de les identifier au cœur des descriptions comportementales. À l'image du langage TROS déjà abordé dans le chapitre précédent sur la figure 2.7, la préemption apparaît implicitement à travers la modélisation. Les priorités sont à ce sujet précisées de manière atomique sur les actions temporisées. Cependant, certaines traductions à l'instar des TPNs exigent une "mise à plat" de ces mécanismes de part leurs expressivités. Dans le cadre d'un ordonnancement coopératif traduit en TPN, nous pourrions tout simplement imaginer un arc inhibiteur qui viendrait désactiver la mise en exécution d'une ressource concurrente. Dans l'optique de décrire ces mécanismes, des rôles complètent ceux de composition pour identifier des éléments de comportement (voir figure 3.9) nécessaires à de telles descriptions. Ils permettent ainsi une prise en compte explicite de ces mécanismes quelque soit le langage de modélisation choisi pour la traduction.

Dans cet extrait, seuls quelques rôles identifiés pour de tels mécanismes sont montrés. Mais la possibilité d'étendre ces rôles à l'ensemble des prototypes comportementaux est illustrée à l'instar des rôles *mutualAcquisitionScheduling* et *mutualReleasingScheduling* identifié au sein du prototype *MutualExclusionResourceBehavior*. Ces rôles ont particulièrement été identifiés pour repérer les éléments de comportement décrivant un ordonnancement suite à une exclusion mutuelle (e.g., la mise en attente d'une ressource ordonnançable sur la prise d'un *mutex* ou bien encore le fait de la rendre éligible suite au relâchement de ce même *mutex*). La considération de telles descriptions n'est pas sans conséquences au sein du processus que nous cherchons à mettre en place. Elles font en effet l'objet d'un clonage spécifique que nous détaillerons dans le prochain chapitre.

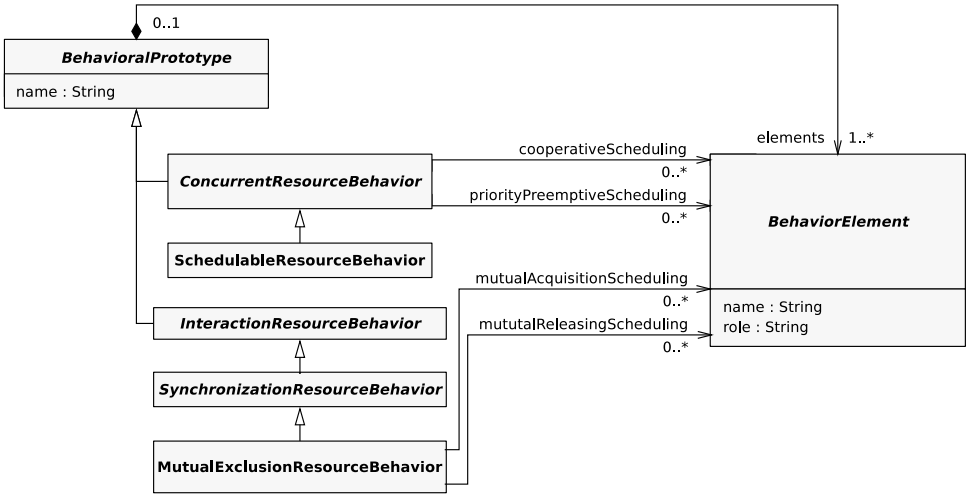


FIGURE 3.9 – Extrait du DSML RTEPML étendu : assignation de rôles de mécanismes spécifiques

3.3.5 Représentation du comportement d'une plate-forme

Une première expérimentation [48] a mis en exergue la représentation du comportement d'une plate-forme OSEK/VDX. Elle a permis d'isoler les prototypes comportementaux de cette norme en fragments de TPN. Cet exercice a permis finalement de reproduire la démarche qu'effectuerait le concepteur pour modéliser sa plate-forme OSEK/VDX en TPN.

La figure 3.10 retranscrit cet exercice en s'appuyant sur le PDM introduit dans le chapitre précédent, sur la figure 2.6. Trois fragments de TPNs ont été définis avec RTEPML étendu et viennent compléter le PDM structurel initialement prévu. Le fragment principal décrit le cycle de vie du prototype comportemental d'une tâche OSEK/VDX (TaskBehavior). Un second fragment décrit le cycle de vie du service de terminaison d'une tâche OSEK/VDX (TerminateTaskBehavior). Enfin, un troisième fragment étend le fragment principal en illustrant le mécanisme d'ordonnancement coopératif [3] qui vient inhiber l'exécution de la tâche (InhibitorAction). Les rôles *behavior*, *terminateServices* et *cooperativeScheduling* identifiés au sein de RTEPML associent respectivement ces fragments de TPN. Conformément à RTEPML, les trois fragments sont représentés par des éléments de comportement atomiques. Toutefois, l'ensemble des atomes de l'action d'inhibition sont considérés sous forme d'élément de comportement composite. Cette configuration émane de la localisation collective des atomes de cette action en TPN à travers le rôle *cooperativeScheduling*.

La configuration de ces fragments de TPN n'a pas été disposée de manière arbitraire. La distinction a certes été faite en fonction des rôles de composition, mais aussi en fonction de la démarche adoptée pour fusionner les éléments de comportement. L'orientation prise se focalise sur la fusion de places. Cette approche pose moins de problèmes qu'une fusion de transitions [68] [84]. La fusion de transitions nécessite une formalisation plus complexe en raison des intervalles statiques des transitions à fusionner qui pourraient être différents. La fusion de places est moins conséquente puisqu'elle n'engage seulement un nouveau marquage de la place fusionnée. Nous verrons dans le chapitre 5 la formalisation adoptée.

En conséquence, l'ensemble des places visées par la composition sont annotées au sein du prototype comportemental via les rôles prévus à cet effet. À titre d'exemple, le rôle *resumedState* permet de repérer les places RUNNING_P et RUNNING sur les trois fragments

3. La représentation d'un tel mécanisme n'a d'intérêt que si l'application contient au moins deux activités concurrentes de priorités différentes

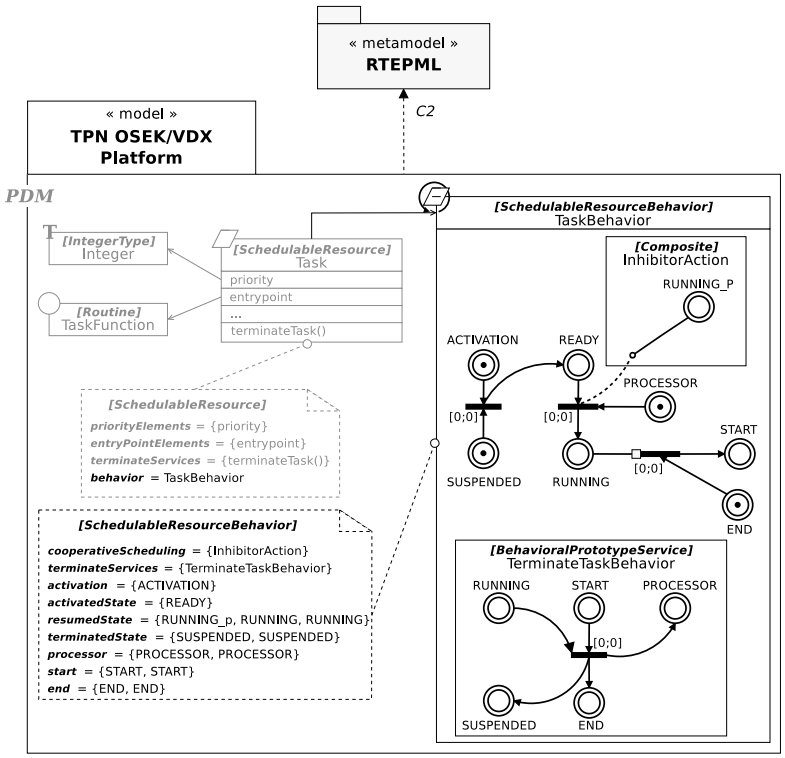


FIGURE 3.10 – Représentation d'un PDM OSEK/VDX en TPN : tâche, service de terminaison et inhibition

de TPN. Pour des raisons de clarté, les places fusionnables seront représentées par la suite avec des doubles ronds comme sur la figure 3.10. Elles se distingueront des places non fusionnables qui seront représentées par des simples ronds.

3.4 Synthèse

Dans ce chapitre, le langage de modélisation de plates-formes logicielles d'exécution temps réel RTEPML a été enrichi dans le but de représenter le comportement qui émane des mécanismes des systèmes d'exploitation temps réel. Cette considération implique de décrire le cycle de vie des concepts exécutifs tels que les ressources et les services offerts par ces systèmes d'exploitation.

Dans ce sens, un modèle comportemental de plate-forme a été intégré dans la syntaxe de RTEPML pour assigner des prototypes comportementaux aux concepts exécutifs. Chaque prototype comportemental est constitué d'éléments de comportement composites ou atomiques traduisibles dans un langage formel tel que les TPNs.

Cette nouvelle syntaxe offre la possibilité de définir un certain nombre de fragments comportementaux en TPN, mais aussi de localiser des points de connexion utiles à la composition de ces fragments. La localisation de ces points de connexion est établie indépendamment du langage formel qui pourrait être différent des TPNs. Par la suite, seuls les TPNs ont été retenus pour traduire le comportement. Les points de connexion sont représentés par des places et oriente la composition vers de la fusion de places.

Une description textuelle plus large de l'extension comportementale de RTEPML est donnée en annexe 139 complétant ainsi la syntaxe abstraite de l'ensemble des ressources et des services qui avait déjà été introduite [12] de manière structurelle. Cette description couvre uniquement une partie des prototypes comportementaux dont les rôles de composition

ont pu être identifiés. La syntaxe proposée est donc susceptible d'être étendue pour couvrir la totalité des concepts comportementaux en ajoutant d'autres rôles de composition.

Cependant, l'identité de ces rôles manquants qui devront être considérés à l'avenir ne doit en aucun cas impacter le processus que nous voulons mettre en œuvre. La composition doit par conséquent se généraliser autour des trois grandes familles de même niveau hiérarchique (i.e., les ressources concurrentes, les ressources d'interaction et les routines) que nous avons figurées au tout début de ce chapitre sur la figure 3.2.

4

Composition du comportement de la plate-forme et de l'applicatif

Sommaire

Le langage de modélisation RTEPML a été étendu au comportement. Le processus de déploiement qui considère explicitement les descriptions de plates-formes d'exécution doit être renforcé pour considérer le comportement.

Nous présentons dans ce chapitre une stratégie pour générer des modèles comportementaux en TPN d'applications déployées sur des plates-formes explicitement considérées. Les règles de transformation sont détaillées indépendamment du langage formel employé pour la traduction du comportement. Ce chapitre s'organise en trois sections. La première section justifie la stratégie adoptée pour une telle génération. La seconde section s'articule autour de deux axes pour composer l'application déployée avec le comportement de la plate-forme considérée. Le premier axe introduit le clonage des prototypes comportementaux, tandis

que le second axe assemble les prototypes clonés. Enfin la dernière section conclue sur la stratégie et le processus proposés.

4.1 Stratégie de considération du comportement

Dans le positionnement du chapitre 2 à la page 41, nous avons promu un processus de génération de modèles formels indépendant de la génération de code. Cette indépendance offre dans ce cas la possibilité de vérifier des propriétés non-fonctionnelles des SETRS à concevoir avant même l'implémentation du code. Toutefois, il reste à définir quelle stratégie adopter pour considérer le comportement des plates-formes logicielles d'exécution vis-à-vis du déploiement des applications temps réel.

Un travail préliminaire a ciblé notre choix pour savoir quelle était la meilleure façon d'établir de nouvelles règles de transformation tout en profitant du processus de déploiement. Deux stratégies sont stipulées.

4.1.1 Intégration dans le processus de déploiement

La première idée a été de modifier le processus de déploiement déjà mis en œuvre [12]. Par conséquent, cette solution propose de considérer explicitement le comportement des plates-formes d'exécution durant le déploiement. La figure 4.1 montre le synoptique de SEXPISTOOLS dans ce contexte. L'enchainement des processus apparaît du haut vers le bas.

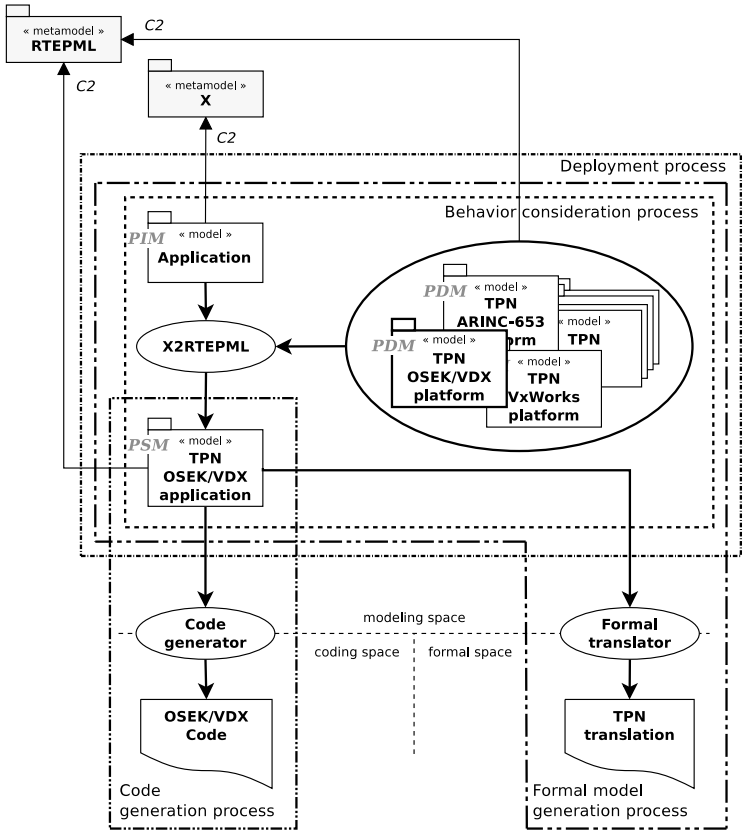


FIGURE 4.1 – Synoptique de SEXPISTOOLS : intégration du processus de considération du comportement

Considérer le comportement de la plate-forme considérée à travers le déploiement implique d'intégrer le processus de considération au sein du processus de déploiement. Dans cet exemple, la plate-forme considérée est choisie parmi un ensemble de PDMs passés en paramètre de la transformation X2RTEPML (e.g., celle considérée pourrait être celle de la

figure 3.10) du chapitre précédent. Le PSM comportemental généré est par conséquent celui d'un TPN composé d'une application intégrée dans une plate-forme OSEK/VDX. Cette représentation peut par la suite servir de point de départ à un générateur de code spécifique à la norme OSEK/VDX ou à un traducteur formel en TPN. D'un côté, le processus de génération de code apparaît juste ici à titre indicatif pour positionner le processus de génération de modèles formels que nous cherchons à mettre en œuvre. De l'autre côté, le traducteur formel est intégré au processus de génération de modèles formels. Une traduction formelle du PSM généré est en effet nécessaire pour exploiter ce dernier avec un outil de vérification.

Bien que cette méthode propose tout de suite un raffinement très détaillé de l'application déployée, elle ne privilégie pas une bonne séparation des processus mis en jeu. Le processus de considération du comportement qui fait partie intégrante du processus de déploiement, doit demeurer aussi partie intégrante du processus de génération de modèles formels. Cette imbrication des processus n'apporte donc pas d'indépendance du déploiement vis-à-vis de la génération de modèles formels.

4.1.2 Adjonction au processus de déploiement

La configuration suivante consiste cette fois à distinguer le processus de déploiement du processus de considération du comportement. La figure 4.2 expose ce cas de figure. Dans un premier temps, le processus de déploiement génère uniquement la structure de l'application déployée sur le PSM 1 qui pourrait être celle obtenue dans l'exemple de la figure 2.6 au chapitre 2.

Pour autant, le comportement de la plate-forme considéré durant le déploiement apparaît dans le PSM 1 puisque le PDM décrit ce comportement. L'introduction du comportement de la plate-forme ne change pas la transformation X2RTEPML puisque les règles non modifiées s'appuient uniquement sur les concepts structurels.

Le processus de considération du comportement engendre dans un second temps la création d'une nouvelle transformation endogène RTEPML2RTEPML. Cette transformation compose uniquement le comportement de l'application déployée à partir du PSM 1 dans lequel le comportement de la plate-forme et la structure de l'application sont déjà intégrés. Le PSM 2 généré représente également le TPN composé d'une application intégrée dans une plate-forme OSEK/VDX.

Avec cette approche, les processus sont bien séparés. Le processus de considération du comportement qui fait partie intégrante du processus de génération de modèles formels n'est pas imbriqué dans le processus de déploiement. Il est finalement adjoint à ce processus.

4.1.3 Synthèse

La stratégie adoptée [49] résulte de la deuxième configuration. Une transformation endogène est nécessaire pour composer un PSM comportemental d'une application déployée en TPN. Une vision plus détaillée est donnée figure 4.3 montrant le résultat escompté de cette transformation.

Sur cette figure, le fruit de la composition est intégré dans la plate-forme à l'image du TPN de l'application au sein du PSM 2 généré. La suite de ce chapitre dispose les règles qui vont permettre une telle représentation en suivant deux grands axes :

– **Clonage** : des éléments de comportement pour les besoins de l'application. Des correspondances doivent être établies entre les concepts structurels de l'application déployée et les concepts comportementaux de la plate-forme visée par le déploiement ;

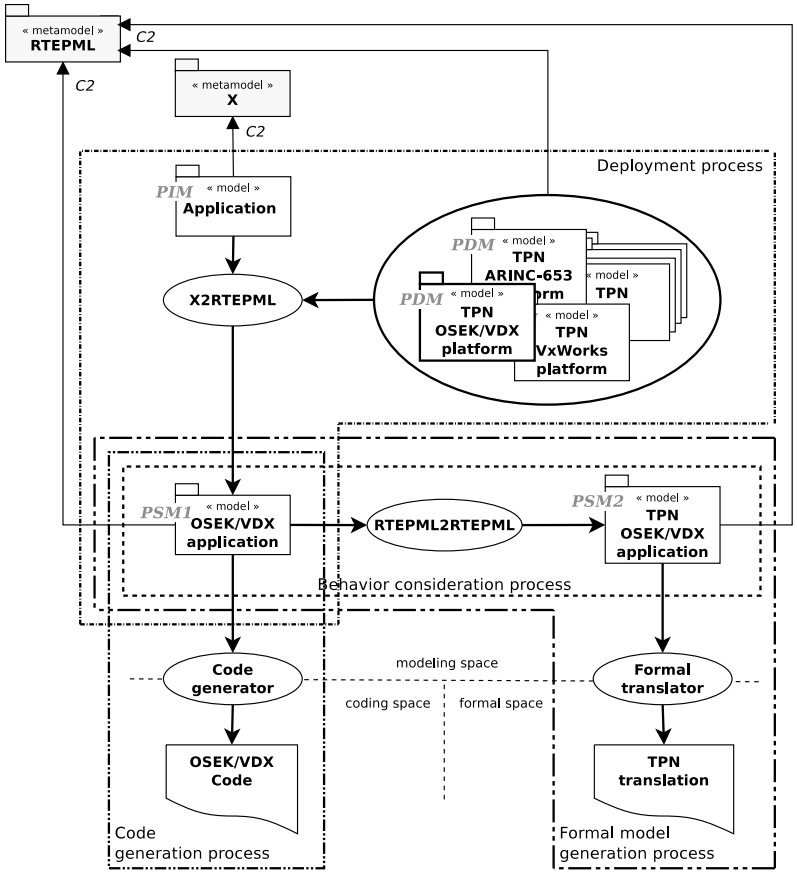


FIGURE 4.2 – Synoptique de SEXPISTOOLS : adjonction du processus de considération du comportement

– **Composition** : des éléments clonés pour la construction du TPN de l'application. Les places à fusionner doivent être localisées à partir des rôles de composition introduits dans le chapitre précédent.

La figure présentée ci-dessus expose finalement le processus de génération de modèles formels. Toutefois, la traduction formelle illustrée aussi sur cette figure n'est pas décrite dans ce chapitre. Elle sera donnée dans le chapitre 6.

4.2 Processus de considération du comportement

Comme mentionné dans le chapitre précédent, l'intégration du comportement de l'application déployée dans le modèle de plate-forme exige de composer un certain nombre de fragments comportementaux. Ces fragments sont issus de prototypes comportementaux qui sont associés aux ressources et aux services de la plate-forme et qui sont au préalable clonés vis-à-vis de l'application déployée.

Dans cette section, les règles de transformation mises en œuvre au sein du processus de composition suivent une hiérarchie impérative. La cohérence d'un tel processus implique un enchainement des règles de façon ordonnée. Les règles suivantes sont donc présentées de manière algorithmique dans l'ordre logique de leurs déclenchements. Elles sont aussi détaillées tout en maintenant une généricité d'implémentation au regard de la plate-forme considérée et du langage de modélisation utilisé pour la traduction des prototypes comportementaux.

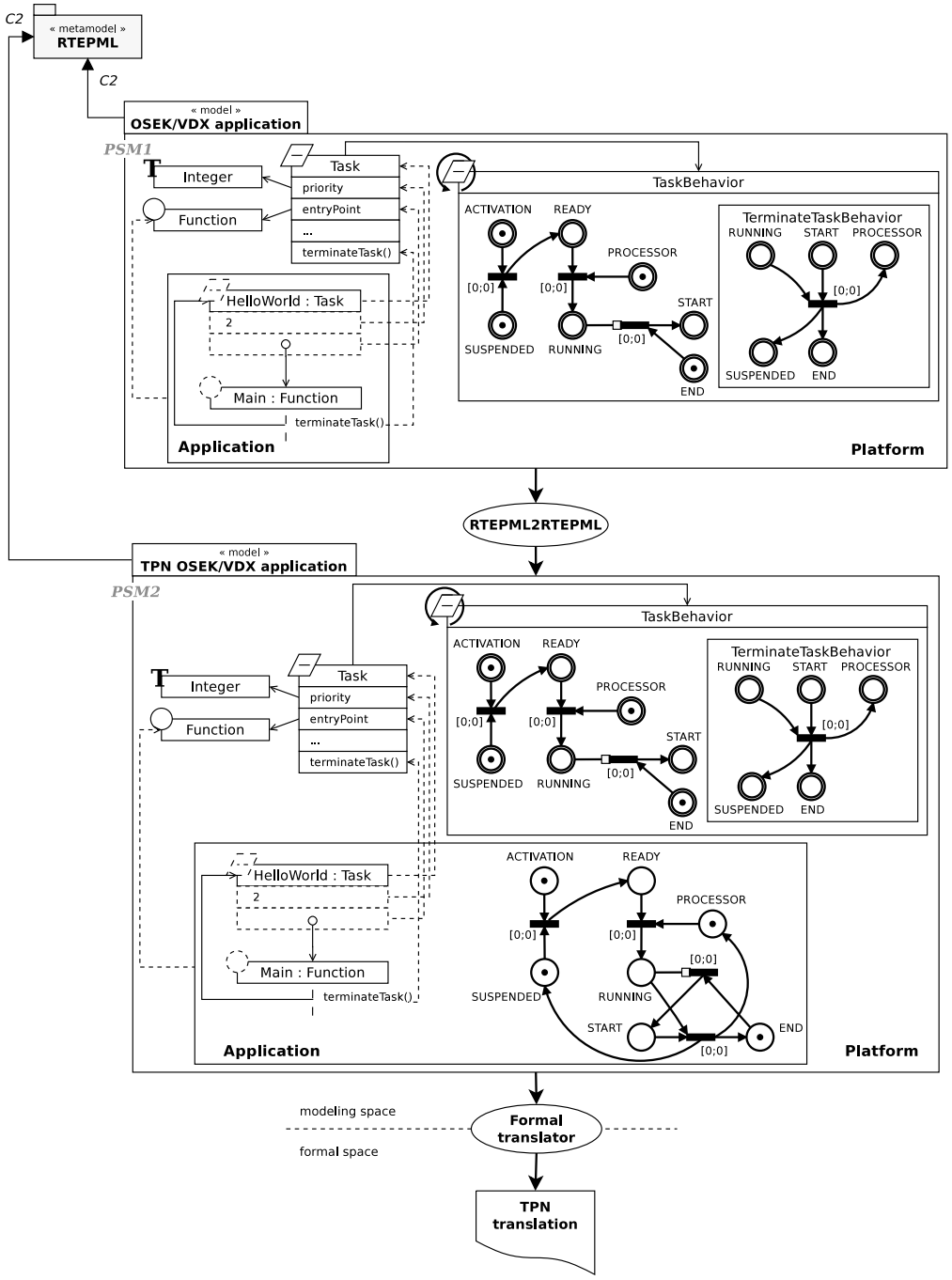


FIGURE 4.3 – Simple composition d'application avec une plate-forme OSEK/VDX

4.2.1 Clonage d'éléments : application sur les instances

Le principe général d'intégration du comportement de l'application réside dans le premier algorithme 4.1 présenté ci-dessous. La première étape nécessite d'intégrer la structure de l'application déployée dans le nouveau modèle de plate-forme à générer. Puisque le modèle source (PSM 1) représente le déploiement, cette structure est déjà intégrée dans la plate-forme explicitement considérée. Par conséquent, une simple correspondance des classes du métamodèle source de plate-forme (MM_p) avec elles-mêmes (identité puisque le métamodèle source est identique au métamodèle cible) apparaît en premier lieu pour

dupliquer la plate-forme dans le nouveau modèle à générer (PSM 2).

ALGORITHME 4.1 – Principe d'intégration d'un modèle comportemental d'application : clonage des prototypes comportementaux

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

début
- // Dupliquer la plate-forme et l'application déployée
- **pour chaque** *classe* $m_p \in MM_p$ **faire**
  - **correspondance avec** m_p // (identité)
- **pour chaque** *instance de ressource* ari χ *ApplicationResourceInstance* $\in MM_p$ **faire**
  - **si** $\exists\ ari.type.behavior$ **alors**
    - // Cloner le prototype comportemental
    - `ruleCloneBehavioralPrototype(`$ari.type.behavior, ari$`)`

Une fois la plate-forme dupliquée, le clonage des prototypes comportementaux est permis pour chaque instance de ressource (conforme à [1] la classe *ApplicationResourceInstance* de MM_p) de l'application déployée. L'appel à la règle de transformation *ruleCloneBehavioralPrototype* est conditionnée par l'existence ou non d'un prototype comportemental associé à la ressource qui type l'instance concernée (localisation par le biais de $ari.type.behavior$ au sein de MM_p). L'algorithme 4.2 en détaille le principe.

ALGORITHME 4.2 – Règle d'application d'un prototype comportemental (*ruleCloneBehavioralPrototype*)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Concept source :
m_{bp} : la classe *BehavioralPrototype* de MM_p

Élement impliqué :
ari : l'instance à l'origine du clonage, conforme à la classe *ApplicationResourceInstance* de MM_p

Notation :
bp : le prototype comportemental à cloner, conforme à m_{bp}

début
- **pour chaque** *élément de comportement* $be \in bp.elements$ **faire**
  - **si** be χ *Atomic* $\in MM_p$ **alors**
    - // Générer un élément atomique à partir d'un prototype comportemental
    - `ruleCreateAtomicElementFromBehavioralPrototype(`be, ari`)`

1. Nous noterons χ pour décrire la relation de conformité tout au long des algorithmes présentés

Chaque prototype comportemental localisé avec la règle précédente est passé en paramètre de cette nouvelle règle puisqu'il se conforme au concept source *BehavioralPrototype*. Il est parcouru ici pour localiser tous les éléments de comportement qui le constituent et qui devront être clonés. L'instance de ressource à l'origine du clonage, est aussi impliquée dans cette règle (conformité avec le concept impliqué *ApplicationResourceInstance*).

Comme relaté dans le chapitre précédent à la page 49, les éléments de comportement parcourus peuvent être décrits soit de manière composite, soit de manière atomique avec RTEPML. Nous nous intéressons pour le moment à la description atomique de ces éléments, puisque le langage de modélisation utilisé pour traduire formellement chaque prototype comportemental est les TPNs. La syntaxe abstraite des TPNs associe effectivement des éléments de comportement atomiques (voir figure 3.7, page 55). Le clonage des éléments de comportement composites sera introduit plus loin dans ce chapitre, notamment pour préciser le comportement de mécanismes spécifiques des plates-formes.

Dans ce contexte, cette règle se poursuit avec le déclenchement de la règle *ruleCreateAtomicElementFromBehavioralPrototype* dont le but est de générer des éléments atomiques clonés issus du prototype comportemental source. L'algorithme 4.3 précise son fonctionnement.

ALGORITHME 4.3 – Génération d'un élément de comportement atomique (*ruleCreateAtomicElementFromBehavioralPrototype*)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Concept source :
m_a : la classe *Atomic* de MM_p

Élément impliqué :
ari : l'instance de ressource à l'origine de la génération, conforme à la classe *ApplicationResourceInstance* de MM_p

Notation :
abe : l'élément de comportement atomique à cloner, conforme à m_a

début
  correspondance avec *Atomic* $\in MM_p$
    $rootInstances \leftarrow \{ari\}$
    $name \leftarrow abe.name\ +'\_'\ + ari.name$

Le concept source est cette fois le concept *Atomic* de RTEPML. Chaque élément de comportement qui s'y conforme sera caractérisé à travers cette règle. Afin de garantir une identité atomique pour chaque élément considéré, l'instance de ressource est une nouvelle fois impliquée servant d'instance *root* à la caractérisation.

Une correspondance est donc établie, au sein de cette règle, entre l'élément de comportement atomique source et un nouvel élément conformément au concept *Atomic*. Le nouvel élément créé contient de nouvelles informations qui proviennent de l'application. L'instance de ressource *root* est assignée au moyen de *rootInstances*. Grâce à la référence opposée à *rootInstances* (cf. Référencement des éléments de composrtement, page 51), l'élément nouvellement créé est ajouté parmi les éléments clonés référencés pour cette instance. Ce niveau de précision aura son importance au moment de la composition à la page 86. En-

fin, cette création incorpore aussi un nommage pour préciser davantage le nouvel élément quant à son identité à travers le modèle à générer.

La création d'éléments atomiques introduite précédemment n'apporte pas un niveau de granularité suffisant de modélisation comportementale des instances de ressource de l'application déployée. Certaines propriétés définies sur les ressources peuvent effectivement influer sur la quantification ou la qualification de certains éléments de comportement. De ce fait, des rôles de propriétés identifiés au sein des ressources de la plate-forme (cf. Identification de rôles, page 50) ont été dupliqués au sein des prototypes comportementaux. L'algorithme suivant 4.4 vient enrichir la règle *ruleCreateAtomicElementFromBehavioralPrototype* en considérant l'impact comportemental que procurent certaines propriétés sur les éléments créés (les modifications apportées à l'ensemble des algorithmes seront par la suite pointées par le symbole ➤). L'impact comportemental engendré par ces propriétés se résume à la valuation des éléments créés.

ALGORITHME 4.4 – Génération d'un élément de comportement atomique (*ruleCreateAtomicElementFromBehavioralPrototype*)(➤propriétés)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Concept source :
m_a : la classe *Atomic* de MM_p

Élement impliqué :
ari : l'instance de ressource à l'origine de la génération, conforme à la classe *ApplicationResourceInstance* de MM_p

Notation :
abe : l'élément de comportement atomique à cloner, conforme à m_a

```
début
    correspondance avec Atomic ∈ MM_p
        ...
➤       pour chaque rôle r ∈ getPropertyRoles(ari.type.behavior) faire
➤           si abe = getAtomicElement(ari.type.behavior, r) alors
➤               pour chaque propriété p ∈ ari.properties faire
➤                   si p.property = getProperty(ari.type, r) alors
                        // Valuer l'élément cloné en fonction d'une propriété
➤                       val ← getPropertyValue(p, r)
```

Dans cet algorithme, un certain nombre de fonctions apparaissent pour retourner des informations. Suivant les besoins, ces informations peuvent provenir soit du modèle de la plate-forme (structure ou comportement) ou soit de l'application intégrée dans le modèle de la plate-forme. Dans tous les cas, ces informations sont localisables au moyen des concepts de RTEPML. À travers les prochains algorithmes, nous retiendrons le terme *helper* de pour désigner ces fonctions. Ce terme est en effet rencontré dans certains langages de transformation tels que ATL [4] pour remplir ces fonctions. Ces *helpers* ne seront pas systématiquement détaillés. Toutefois, leurs principes seront précisés.

L'algorithme précédent se décompose en quatre points visant ainsi la valuation de l'élé-

ment créé :

1. une recherche introspective est dans un premier temps effectuée sur le prototype comportemental de l'instance impliquée dans cette règle, afin de retourner l'ensemble des rôles de propriétés joués. Pour y parvenir, cette recherche est lancée sur le concept de RTEPML auquel se conforme le prototype $ari.type.behavior$, passé en paramètre du *helper getPropertyRoles* ;
2. l'existence d'un rôle de propriété joué par l'élément atomique source abe est ensuite vérifiée au sein du prototype comportemental. Le *helper getAtomicElement* contribue à la localisation du rôle de propriété en localisant l'élément atomique source qui joue ce rôle ;
3. le rôle de propriété une fois localisé, celui-ci va permettre alors de localiser la propriété recherchée au sein de l'instance *root*, dans l'optique de valuer l'élément créé. La localisation est réalisée en retournant la propriété qui joue ce rôle au sein de la ressource type $ari.type$, par l'intermédiaire du *helper getProperty* ;
4. la valeur de la propriété p est finalement retournée par le *helper getPropertyValue*. Cette valeur peut être traduite selon l'expressivité du langage de modélisation employé pour décrire la valeur de l'élément de comportement. Le rôle de propriété est par conséquent passé en paramètre du *helper* pour identifier la traduction à réaliser.

Chaque élément de comportement qui joue un rôle de composition doit aussi être tracé au moment du clonage (cf. Identification de rôles de composition, page 56). L'algorithme 4.5 complète alors la règle *ruleCreateAtomicElementFromBehavioralPrototype* en attribuant à chaque élément créé le rôle joué par l'élément à l'originaire du clonage.

ALGORITHME 4.5 – Génération d'un élément de comportement atomique (*ruleCreateAtomicElementFromBehavioralPrototype*)(➤rôle de composition)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Concept source :
m_a : la classe *Atomic* de MM_p

Élément impliqué :
ari : l'instance de ressource à l'origine de la génération, conforme à la classe *ApplicationResourceInstance* de MM_p

Notation :
abe : l'élément de comportement atomique cloné à générer, conforme à m_a

début
  correspondance avec *Atomic* $\in MM_p$
    ...
➤ **pour chaque** *rôle* $r \in$ `getCompositionRoles(`$ari.type.behavior$`)` **faire**
➤ **si** $abe =$ `getAtomicElement(`$ari.type.behavior, r$`)` **alors**
      // Caractériser l'élément cloné selon le rôle joué pour la composition
➤ $role \leftarrow r.name\ +'\_'\ + ari.name$

De façon similaire à *getPropertyRoles*, une introspection est réalisée sur le prototype comportemental impliqué, pour retourner l'ensemble des rôles de composition. Le *helper*

getAtomicElement est ensuite réutilisé pour localiser cette fois le rôle de composition joué par l'élément source abe. Une fois localisé, le rôle de composition est annoté sur l'élément de comportement créé en l'identifiant selon l'application. Pour ce faire, le nom de l'instance impliquée est ajouté à celui du rôle d'origine, à l'image du nommage des éléments créés.

Une remarque est à préciser quant à l'emploi des deux *helpers getPropertyRoles* et *getCompositionRoles*. Une traduction est effectivement appliquée au sein de ces *helpers* pour distinguer les types de rôles rencontrés (i.e., rôles de propriété ou bien rôles de composition). Ceci étant, cette distinction ne dépend uniquement de RTEPML.

Une mise en application des règles qui viennent d'être présentées est illustrée figure 4.4 pour montrer une première étape d'intégration d'un modèle comportemental d'application. Cet exemple s'appuie sur le clonage du prototype comportemental d'une alarme OSEK/VDX préalablement instanciée sur le PSM 1 après déploiement de l'application. Le modèle comportemental obtenu après exécution des règles apparaît dans la partie Application du PSM 2 généré. Les éléments de TPN qui constituent le prototype comportemental AlarmBehavior ont ainsi été clonés et nommés avec le suffixe Alarm qui correspond au nom de l'instance *root* impliquée.

Sur le TPN obtenu, l'intervalle statique de la transition a aussi été borné suivant la valeur de la propriété cycletime de l'instance. L'application de l'algorithme 4.4 vient successivement :

1. localiser le rôle *periodElements* à travers le concept *AlarmBehavior* de RTEPML ;
2. localiser l'élément period (i.e., les bornes basse et haute de l'intervalle statique) qui joue ce rôle ;
3. localiser la propriété cycletime qui joue ce même rôle, sur la ressource Alarm qui type l'instance éponyme ;
4. localiser la valeur 50 qui renseigne la propriété cycletime, sur l'instance.

Enfin, chaque place clonée dont l'élément source joue un rôle de composition dans le prototype AlarmBehavior a été annotée du rôle localisé. Chaque rôle est suffixé par le nom de l'instance Alarm et apparaît sous forme d'exposant en gras italique, sur chaque place annotée (e.g., ***enable_Alarm*** et ***activation_Alarm***).

Nous venons de voir une première mise en œuvre du processus dont le principe réside dans le clonage d'éléments de comportement. Ce clonage s'applique ici sur les instances de ressources. Le processus a par conséquent été enrichi pour aussi traiter les services.

4.2.2 Clonage d'éléments : application sur les services

Le principe de base d'intégration d'un modèle comportemental d'application a été repris pour couvrir les services offerts par les ressources. Plus exactement, ce sont les appels de service possédés par les instances dans l'application qui sont concernés par cette seconde étape. Dans RTEPML, une séquence ordonnée d'appels de service est accessible depuis toute instance de ressource. Nous verrons par la suite que ces séquences d'appels sont principalement déployées à travers les instances de routine. Cependant, comme l'attribution d'une séquence d'appels est permissive à l'ensemble des instances, les règles qui vont suivre s'appliquent à l'ensemble des instances de ressources [2]. L'algorithme 4.6 étend le principe d'intégration d'un modèle comportemental en appelant la règle *ruleCloneBehavioralPrototypeService* pour chaque service appelé (conforme à la classe *CallService* de MM_p).

2. Précisons toutefois que des contraintes type OCL (dans UML) ont déjà été appliquées sur les règles du processus de déploiement pour se focaliser seulement sur les routines d'exécution

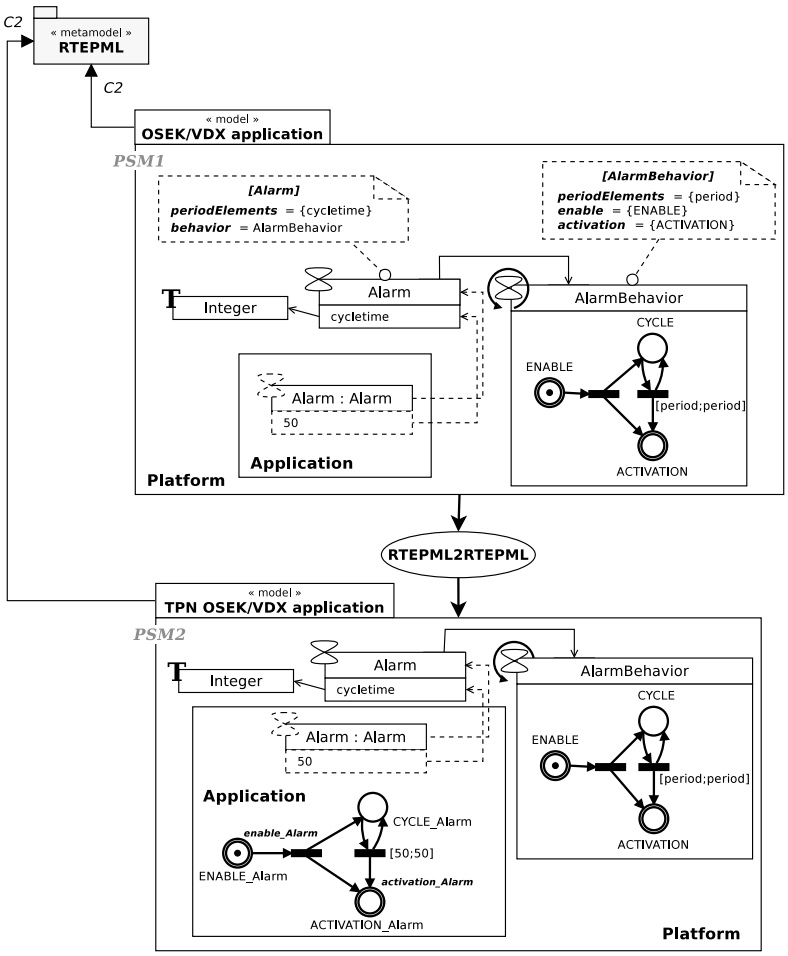


FIGURE 4.4 – Intégration du comportement : clonage d'une alarme OSEK/VDX

Cette fois, pour chaque appel de service, le déclenchement de la règle de clonage est conditionné par l'existence d'un service au sein de la ressource qui type l'instance référencée par cet appel. Si c'est le cas, un prototype comportemental de ce service existe pour cet appel. La démarche est la suivante :

1. une recherche introspective est premièrement appliquée sur la ressource type, dans le but de retourner l'ensemble des rôles de services joués au sein de cette ressource. La recherche est concentrée sur le concept de RTEPML auquel se conforme $cs.ref.type$, passée en paramètre du *helper getServiceRoles* ;
2. l'existence d'un service $cs.service$ (qualifié par l'appel de service) qui joue ce rôle au sein de la ressource type est vérifiée par le biais du *helper getService*.
3. le prototype comportemental du service appelé est finalement localisé à travers le prototype comportemental associé à la ressource type $cs.ref.type.behavior$. Pour ce faire, le *helper getBehavioralPrototypeService* retourne le prototype correspondant au rôle localisé précédemment.

L'algorithme de la règle *ruleCloneBehavioralPrototypeService* n'est pas présenté ici puisqu'il s'appuie sur le même que la règle *ruleCloneBehavioralPrototype* 4.2. Les éléments de comportement qui constituent le prototype comportemental de service sont parcourus afin d'être clonés. De nouveaux éléments de comportement peuvent alors être créés, caractérisés en fonction du service appelé.

ALGORITHME 4.6 – Principe d'intégration d'un modèle comportemental d'application : clonage des prototypes comportementaux(➤services)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

début
  // Dupliquer la plate-forme et chaque application déployée
  pour chaque *classe* $m_p \in MM_p$ **faire**
    correspondance avec m_p // (identité)
  pour chaque *instance de ressource* $ari \chi$ *ApplicationResourceInstance* $\in MM_p$ **faire**
    // Cloner le prototype comportemental
    ...
➤   **pour chaque** *service appelé* $cs \in ari.call$ **faire**
➤   **pour chaque** *rôle* $r \in$ `getServiceRoles(`$cs.ref.type$`)` **faire**
➤   **si** $cs.service =$ `getService(`$cs.ref.type, r$`)` **alors**
          // Cloner le prototype comportemental de service
➤   `ruleCloneBehavioralPrototypeService(`cs,
          `getBehavioralPrototypeService(`$cs.ref.type.behavior, r$`))`

La génération des éléments de comportement clonés issus de prototypes comportementaux de service est cette fois exécutée avec la règle *ruleCreateAtomicElementFromBehavioralPrototypeService*. Cette règle est présentée dans l'algorithme 4.7. Elle est appelée pour chaque élément de comportement conforme au concept source *Atomic* de RTEPML. Une fois encore nous nous intéressons uniquement aux éléments de comportement atomiques.

Une correspondance avec ce même concept source *Atomic* est réalisée pour créer un clone de l'élément atomique source. La caractérisation de l'élément créé est cependant légèrement différente de celle d'un élément cloné issu d'un prototype comportemental de ressource. L'identification est assurée selon l'instance référencée par l'appel de service $cs.ref$ impliqué dans cette règle. Le nom du service contenu dans la ressource type de l'instance référencée par le service appelé $cs.service.name$ est cette fois intercalé dans le suffixe de nommage. Le but est de distinguer les éléments clonés d'une même séquence d'appels. En effet, sans cette distinction, les éléments clonés de deux appels de service qui référencent une même instance de ressource (e.g., deux services de prise et de relâchement d'un sémaphore) auraient la même identité.

La figure 4.5 reprend une partie de l'exemple de l'application HelloWorld pour illustrer cette deuxième phase d'intégration d'un modèle comportemental d'application. Cette fois, les éléments du prototype comportemental du service de terminaison d'une tâche OSEK/VDX TerminateTaskBehavior ont été clonés et nommés avec le suffixe HelloWorld. Les éléments clonés sont identifiés en intercalant le nom du service de terminaison terminateTask au suffixe.

En appliquant l'algorithme 4.6, le prototype comportemental du service appelé a été localisé au moyen du *helper getBehavioralPrototypeService* après avoir successivement :

1. localisé le rôle *terminateServices* à travers le concept *SchedulableResource* de RTEPML ;
2. localisé le service appelé terminateTask qui joue ce rôle ;

ALGORITHME 4.7 – Génération d'un élément de comportement atomique (*ruleCreateAtomicElementFromBehavioralPrototypeService*)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Concept source :
m_a : la classe *Atomic* de MM_p

Élement impliqué :
cs : l'appel de service à l'origine de la génération, conforme à la classe *CallService* de MM_p

Notation :
abe : l'élément de comportement atomique cloné à générer, conforme à m_a

début
- **correspondance avec** *Atomic* $\in MM_p$
  - $rootCalls \leftarrow \{cs\}$
  - $name \leftarrow abe.name\ +'\_'\ + cs.service.name\ +'('\ + cs.ref.name\ +')'$
  - **pour chaque** *rôle* $r \in$ `getPropertyRoles(`$cs.ref.type.behavior$`)` **faire**
    - **si** $abe =$ `getAtomicElement(`$cs.ref.type.behavior, r$`)` **alors**
      - **pour chaque** *propriété* $p \in cs.ref.properties$ **faire**
        - **si** $p.property =$`getProperty(`$cs.ref.type, r$`)` **alors**
          - // Valuer l'élément cloné en fonction d'une propriété
          - $val \leftarrow$ `getPropertyValue(`p, r`)`
  - **pour chaque** *rôle* $r \in$ `getCompositionRoles(`$cs.ref.type.behavior$`)` **faire**
    - **si** $abe =$ `getAtomicElement(`$cs.ref.type.behavior, r$`)` **alors**
      - // Caractériser l'élément cloné selon le rôle joué pour la composition
      - $role \leftarrow r.name\ +'\_'\ + cs.ref.name$

3. localisé le prototype comportemental TerminateTaskBehavior qui joue ce même rôle, sur le prototype comportemental TaskBehavior associé à la ressource type.

Les instances générées sur le modèle de l'application déployée avant intégration du comportement peuvent parfois être associées. Ces associations ont un impact non négligeable d'un point de vue comportemental puisqu'elles incarnent des interactions entre les instances concernées. Nous devons par conséquent considérer ces associations au moment de la création du clonage des éléments de comportement.

4.2.3 Clonage d'éléments : application sur les associations

Les associations apparaissent lors du déploiement de l'application suite au clonage de prototypes de conception. Pour rappel, ces prototypes permettent d'agencer des instances de ressource dans le but de définir des ressources non fournies pas la plate-forme considérée (cf. Prototypes de conception à la page 23). À travers ces agencements, des associations peuvent alors être tissées de manière structurelle entre les instances du prototype. Le processus de déploiement reproduit ainsi ces mêmes associations entre les instances de ressources générées à partir de ces prototypes sur le modèle de l'application déployée.

Le clonage des prototypes comportementaux qui décrivent ces instances de ressources

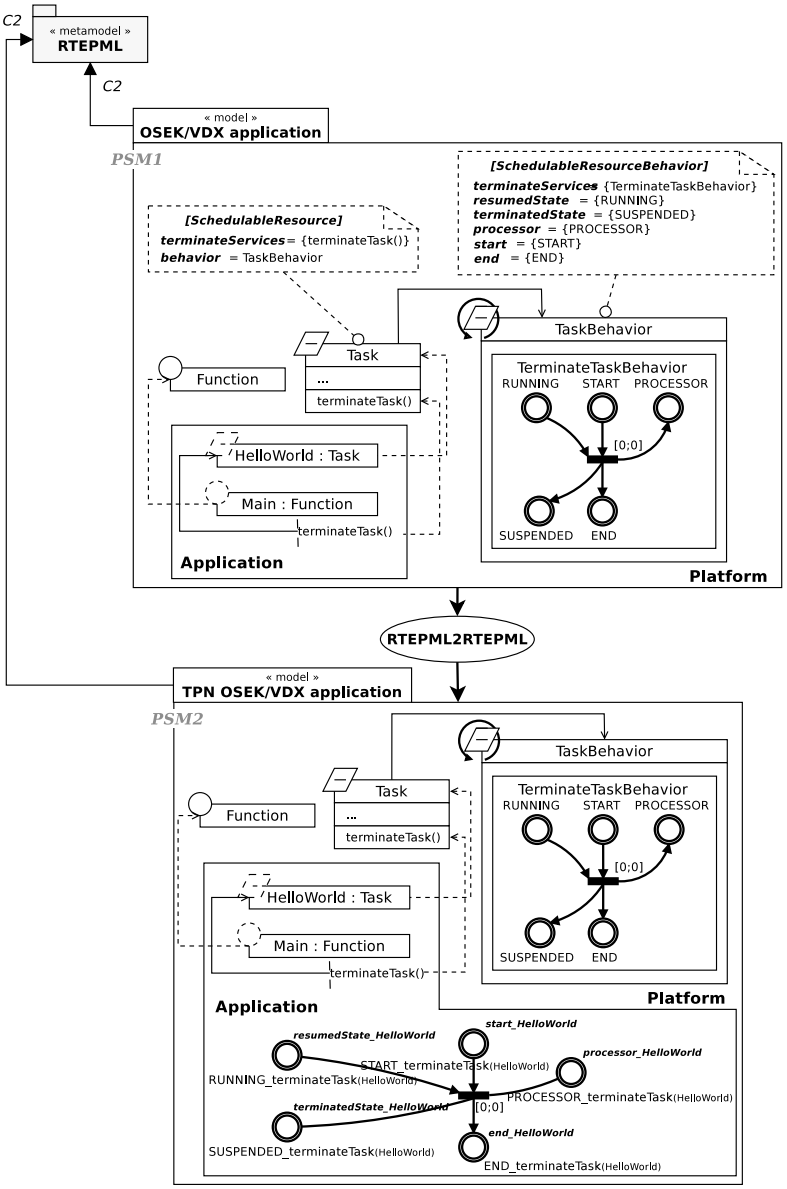


FIGURE 4.5 – Intégration du comportement : clonage d'un service de terminaison d'une tâche OSEK/VDX

associées doit en conséquence marquer les éléments de comportement ciblés par ces associations. Nous avons déjà vu précédemment comment repérer les éléments clonés en vue de la composition de fragments comportementaux. Cette stratégie est préservée ici en annotant des rôles sur les éléments ciblés par ces associations. L'algorithme 4.8 reprend celui présenté pour la création d'éléments de comportement issus de prototypes comportementaux de ressources.

La principale modification apportée se concentre sur l'annotation de l'élément créé avec le rôle de composition qui a été préalablement localisé. Un aiguillage dirige cette annotation selon que l'instance est associée ou non :

1. l'instance impliquée est tout d'abord observée dans le modèle de l'application déployée pour vérifier si elle est reliée avec une autre instance. L'observation se fait au moyen du *helper hasAssociatedInstance* qui passe en revue l'ensemble des associations au sein de l'application. Ces associations se conforment au concept *Resource-*

ALGORITHME 4.8 – Génération d'un élément de comportement atomique
(*ruleCreateAtomicElementFromBehavioralPrototype*)
(➤rôle de composition d'instances associées [prototype de conception])

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Concept source :
m_a : la classe *Atomic* de MM_p

Élément impliqué :
ari : l'instance de ressource à l'origine de la génération, conforme à la classe *ApplicationResourceInstance* de MM_p

Notation :
abe : l'élément de comportement atomique cloné à générer, conforme à m_a

```
début
    correspondance avec Atomic ∈ MM_p
        // Caractériser l'élément cloné selon l'instance
        ...
        pour chaque rôle r ∈ getCompositionRoles(ari.type.behavior) faire
            si abe = getAtomicElement(ari.type.behavior, r) alors
                // Caractériser l'élément cloné selon le rôle joué pour la composition...
➤               si hasAssociatedInstance(ari) alors
➤                   role ← r.name +' _' +
                    getMasterAssociatedInstance(ari).name
➤               sinon
                    role ← r.name +' _' + ari.name
```

InstanceAssociation de RTEPML. Lorsque l'instance passée en paramètre du *helper* est capturée parmi les "images" source et cible d'une association, la valeur *vrai* est retournée.

2. le rôle est ensuite annoté sur l'élément cloné si l'information précédente est vérifiée. Le nom du rôle de composition est alors suffixé par le nom de l'instance maître associée qui est localisée à l'aide du *helper getMasterAssociatedInstance*.

Le choix de privilégier l'instance maître pour le nommage des rôles de composition est légitime. En vue de la composition, les rôles destinés à la localisation des éléments impliqués doivent au maximum être identiques. L'instance maître étant le seul moyen de repérer une instance commune à l'ensemble des instances d'un même prototype de conception, nous l'avons choisie pour suffixer les rôles de composition. L'algorithme 4.9 donne un aperçu du *helper getMasterAssociatedInstance* pour préciser son utilité.

L'instance de ressource passée en paramètre du *helper* étant associée, elle est de ce fait issue d'un prototype de conception. Au sein de ce prototype localisé avec le *helper getDesignPrototype*, l'instance maître $masterInstance$ peut ensuite être retrouvée. L'instance de ressource qui a été déployée à partir de cette instance maître peut alors être repérée grâce au *helper getInstance*. La mise en application de ce dernier *helper* vise à localiser parmi les instances de ressources maîtres déployées, celle qui est associée avec l'instance de ressource ari. Notons tout de même que l'instance de ressource maître peut s'avérer être la même que

ALGORITHME 4.9 – Localisation d'une instance associée maître issue d'un prototype de conception (*getMasterAssociatedInstance*)

Entrées :
ari : une instance de ressource associée, conforme à la classe *ApplicationResourceInstance* de MM_p

Résultat :
l'instance maître associée avec ari et toutes deux issues du même prototype de conception

début
  retourner `getInstance(getDesignPrototype(`ari`).`$masterInstance$`)`

ari.

Nous sommes revenus sur la figure 2.4 de la page 23 pour illustrer l'intégration du comportement d'instances issues d'un prototype de tâche périodique OSEK/VDX dans une application. La figure 4.6 expose ce genre de situation dans laquelle une activité périodique a initialement été déployée sur une tâche OSEK/VDX et ainsi provoqué la génération de deux instances associées sur le PSM 1.

Ces deux instances, la tâche HelloWorld et l'alarme Alarm, sont associées par la relation alarm représentant le fait que la tâche est cadencée par une alarme. Les prototypes comportementaux TaskBehavior et AlarmBehavior décrivant les ressources types de ces instances ont été clonés. Les éléments créés ont ainsi été intégrés dans le modèle de l'application déployée sur le PSM 2. Parmi les éléments créés, ceux qui jouent un rôle pour la composition ont été annotés de ce rôle conformément à l'algorithme que nous venons de présenter. Le suffixe ajouté à l'ensemble des éléments annotés correspond au nom de l'instance maître qui joue le rôle de maître dans la relation d'association.

Cette instance maître a été localisée au moyen du *helper getMasterAssociatedInstance* après avoir successivement :

1. vérifier l'existence de l'association alarm à travers le modèle de l'application déployée ;
2. localiser le prototype de conception à l'origine de cette association au sein de la ressource *PeriodicTask* ;
3. localiser l'instance maître dans le prototype localisé avec le rôle *masterInstance*. Dans notre exemple, l'instance désignée est celle typée par la tâche (i.e., T1) ;
4. retourner l'instance de ressource HelloWorld correspondante déployée dans l'application.

Cette démarche a aussi été adoptée pour les services appelés qui ont pour référence l'une des instances associées. L'ensemble des éléments clonés qui jouent un rôle de composition sont en effet annotés par ce rôle suffixé du nom de l'instance maître associée. Dans le cas de la tâche périodique OSEK/VDX, nous pourrions imaginer son déclenchement initial au moyen d'un service offert par l'alarme OSEK/VDX. L'instance de référence à cet appel de service équivaudrait alors à l'alarme. Mais la tâche étant désignée comme l'instance maître dans cette association, les rôles de composition annotés sur les éléments clonés issus du prototype comportemental de ce service devraient alors être suffixés par cette tâche. L'algorithme n'est pas présenté ici puisqu'il est calqué sur l'algorithme 4.8 en remplaçant le paramètre d'instance ari par celui de l'instance référencée par l'appel de service $cs.ref$ dans les *helpers*.

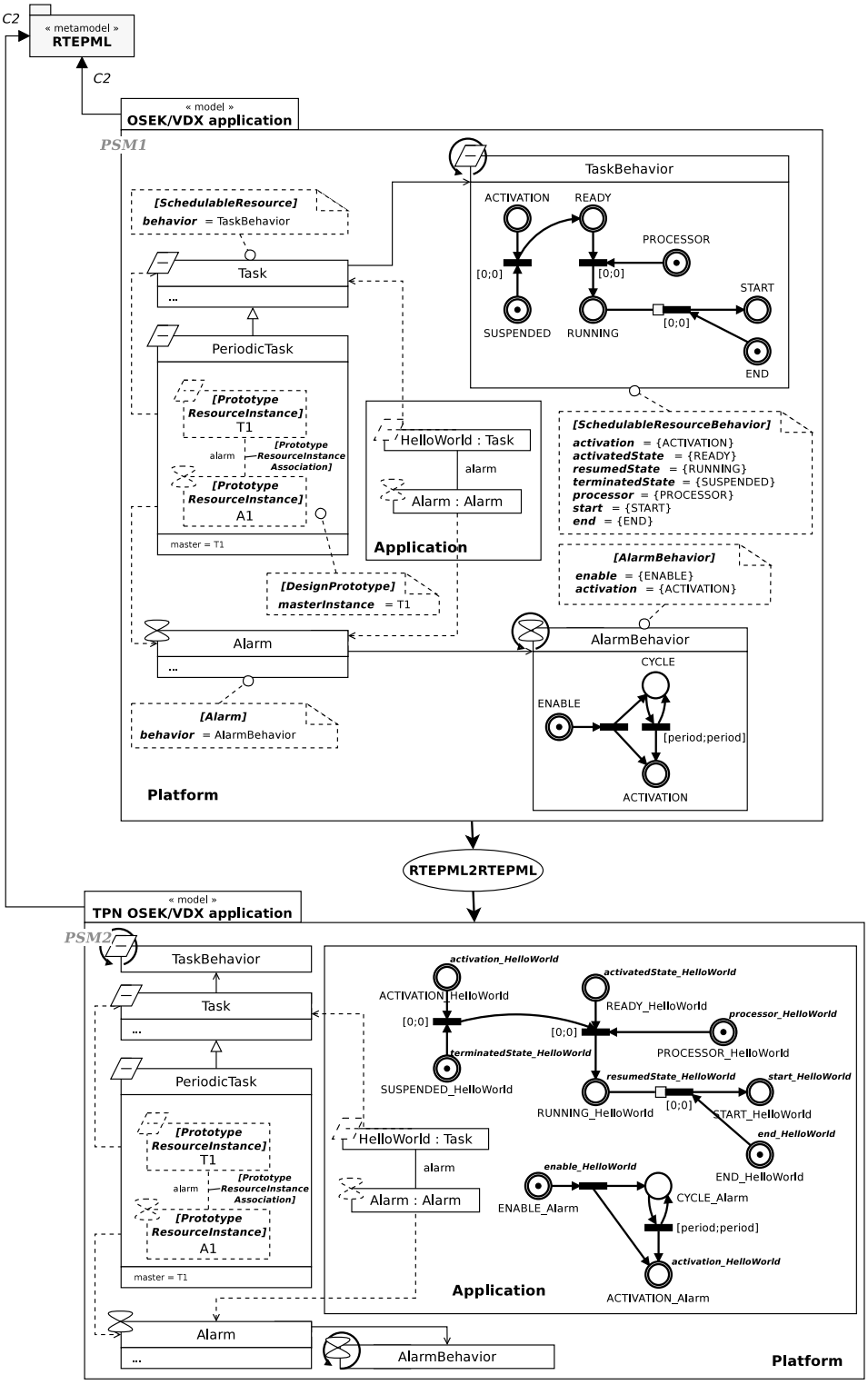


FIGURE 4.6 – Intégration du comportement : clonage d'un prototype de tâche périodique OSEK/VDX

Depuis le début de cette section, le clonage s'appuie sur les éléments de comportement atomiques en raison de l'expressivité des TPNs. Toutefois, nous allons profiter de ce concept d'élément de comportement composite pour traiter le cas des mécanismes spécifiques (cf. Rôles pour les mécanismes spécifiques à la page 57) rencontrés sur les plates-formes

logicielles.

4.2.4 Clonage d'éléments : application sur les mécanismes spécifiques

De base, les éléments de comportement composites sont constitués de sous-éléments de comportement qui peuvent être soit atomiques, soit de nouveau composites. Cette configuration, bien répandue dans la modélisation orientée objet, engage un clonage récursif des sous-élements de comportement. Pour cette raison, l'algorithme 4.10 complète la règle de clonage d'un prototype comportemental en considérant les éléments composites.

ALGORITHME 4.10 – Règle d'application d'un prototype comportemental (*ruleCloneBehavioralPrototype*)(➤élément de comportement composite)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Concept source :
m_{bp} : la classe *BehavioralPrototype* de MM_p

Élement impliqué :
ari : l'instance à l'origine du clonage, conforme à la classe *ApplicationResourceInstance* de MM_p

Notation :
bp : le prototype comportemental à cloner, conforme à m_{bp}

```
début
    pour chaque élément de comportement be ∈ bp.elements faire
        // Condition d'atomicité
        ...
➤       si be χ Composite ∈ MM_p alors
            // Cloner l'élément composite
➤           ruleCloneCompositeBehaviorElement(be, ari)
```

Pour chaque élément rencontré dans le prototype comportemental source qui se conforme à la classe *Composite* de MM_p, une sous-règle de clonage est appelée. La règle *ruleCloneCompositeBehaviorElement* a pour objectif de parcourir les sous-éléments de l'élément sélectionné dans le but de les cloner à leur tour. Cette règle est décrite dans l'algorithme 4.11.

Chaque sous-élément de comportement est tour à tour discerné selon que l'élément est de nature atomique ou composite. Dans le cas où le sous-élément est atomique, la règle *ruleCreateAtomicElementFromBehavioralPrototype* (voir algorithme 4.3) est une nouvelle fois mise à contribution pour créer un clone de cet élément. Dans le cas où le sous-élément s'avère être composite, la règle *ruleCloneCompositeBehaviorElement* est appelée de manière itérative.

Cette introduction au clonage des éléments de comportement composites nous permet de mieux aborder la prise en compte des mécanismes spécifiques. À l'image du mécanisme d'ordonnancement coopératif avancé précédemment sur la figure 3.10 à la page 58, un ensemble d'éléments atomiques (e.g., InhibitorAction) peut être imbriqué dans le prototype comportemental d'une ressource (e.g., respectivement TaskBehavior). Cet arrangement vise

ALGORITHME 4.11 – Règle d'application d'un élément de comportement composite (*ruleCloneCompositeBehaviorElement*)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Concept source :
m_c : la classe *Composite* de MM_p

Élement impliqué :
ari : l'instance de ressource à l'origine du clonage, conforme à la classe *ApplicationResourceInstance* de MM_p

Notation :
cbe : l'élément de comportement composite à cloner, conforme à m_c

début
- **pour chaque** *sous-élément de comportement* $sbe \in cbe.subelements$ **faire**
  - **si** sbe χ *Atomic* $\in MM_p$ **alors**
    - // Générer le sous-élément atomique à partir d'un prototype comportemental
    - `ruleCreateAtomicElementFromBehavioralPrototype(`sbe, ari`)`
  - **si** sbe χ *Composite* $\in MM_p$ **alors**
    - // Cloner le sous-élément composite
    - `ruleCloneCompositeBehaviorElement(`sbe, ari`)`

à spécifier le comportement de la ressource contrainte par ce type de mécanisme. Cette description qui vise à étendre celle du prototype comportemental associé implique de modifier le clonage de ce dernier. L'algorithme 4.12 revient sur la règle *ruleCloneBehavioralPrototype*.

Cet algorithme est étendu pour traiter le clonage des éléments de comportement qui jouent un rôle spécifique dans la description de certains mécanismes. Lorsqu'un de ces éléments est rencontré au sein du prototype comportemental source, cet élément est orienté vers un clonage spécifique suivant sa nature :

1. un test est dans un premier temps appliqué pour vérifier si un rôle de mécanisme est joué par l'élément de comportement sélectionné. Le *helper hasSpecificMechanismRole* retourne la valeur *vrai* si le test est vérifié ;
2. une recherche introspective vient ensuite retourner l'ensemble des rôles de mécanismes spécifiques joués au sein du prototype comportemental. La recherche est focalisée sur le concept de RTEPML auquel se conforme le prototype bp, passé en paramètre du *helper getSpecificMechanismRoles* ;
3. les instances concernées par le mécanisme identifié à travers chaque rôle sont ensuite rassemblées à l'aide du *helper getConcernedInstances*. Le nombre d'instances retournées avec ce *helper* varie en fonction du nombre d'instances qui interagissent avec l'instance ari passée en paramètre, vis-à-vis du mécanisme localisé ;
4. le nombre d'instances concernées impose finalement un clonage multiple de l'élément de comportement. Pour chaque instance concernée, le clonage de l'élément est orienté selon sa nature et conditionné selon le rôle qu'il joue au sein du prototype comportemental source. Les *helpers getAtomicElement* et *getCompositeElement* corrèlent l'élément de comportement à cloner avec le mécanisme joué.

ALGORITHME 4.12 – Règle d'application d'un prototype comportemental (*ruleCloneBehavioralPrototype*)(➤mécanisme spécifique)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Concept source :
m_{bp} : la classe *BehavioralPrototype* de MM_p

Élément impliqué :
ari : l'instance à l'origine du clonage, conforme à la classe *ApplicationResourceInstance* de MM_p

Notation :
bp : le prototype comportemental à cloner, conforme à m_{bp}

```
début
    pour chaque élément de comportement be ∈ bp.elements faire
➤       si hasSpecificMechanismRole(bp, be) alors
➤           pour chaque rôle r ∈ getSpecificMechanismRoles(bp) faire
➤               pour chaque instance concernée ari_c ∈
                getConcernedInstances(ari, r) faire
                    // Générer l'élément atomique à partir d'un mécanisme spécifique
➤                   si beχ Atomic ∈ MM_p et be = getAtomicElement(bp, r) alors
➤                       ruleCreateAtomicElementFromSpecificMechanism(be, ari, ari_c)
                    // Cloner l'élément composite qui décrit un mécanisme spécifique
➤                   si beχ Composite ∈ MM_p et be = getCompositeElement(bp, r)
                    alors
➤                       ruleCloneSpecificCompositeBehaviorElement(be, ari, ari_c)
➤       sinon
            // Générer ou cloner normalement selon la nature de l'élément de comportement
            ...
```

Dans le cas où l'élément de comportement doit être cloné spécifiquement, deux nouvelles règles rentrent en jeu. La première règle *ruleCreateAtomicElementFromSpecificMechanism* adapte le principe de création d'un clone d'élément atomique en considérant l'instance concernée par le mécanisme localisé. Cette règle est décrite dans l'algorithme 4.13.

Le principe de caractérisation d'élément atomique se différencie légèrement de celui vu précédemment dans l'algorithme 4.3. La principale variante réside dans le nommage de l'élément cloné et du rôle de composition annoté. L'instance concernée par le mécanisme localisé dans la règle précédente doit en effet interagir avec cet élément. Ce dernier est de ce fait caractérisé en fonction de cette instance.

La seconde règle *ruleCreateCloneSpecificCompositeBehaviorElement* adapte quant à elle le principe de clonage d'un élément composite toujours en fonction de l'instance concernée par le mécanisme localisé. Cette règle est décrite dans l'algorithme 4.14.

Comme dans l'algorithme 4.12, cette règle vise à appeler les deux règles *ruleCreateAtomicElementFromSpecificMechanism* et *ruleCloneSpecificCompositeBehaviorElement*, mais cette fois quelque soit l'information obtenue avec le *helper hasSpeci-*

ALGORITHME 4.13 – Génération d'un élément de comportement atomique spécifique (*ruleCreateAtomicElementFromSpecificMechanism*)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Concept source :
m_a : la classe *Atomic* de MM_p

Élements impliqués :
ari : l'instance de ressource à l'origine de la génération, conforme à la classe *ApplicationResourceInstance* de MM_p
ari_c : l'instance de ressource concernée par un mécanisme, conforme à la classe *ApplicationResourceInstance* de MM_p

Notation :
abe : l'élément de comportement atomique spécifique cloné à générer, conforme à m_{abe}

début
- **correspondance avec** *Atomic* $\in MM_p$
  - $rootInstances \leftarrow \{ari\}$
  - $name \leftarrow abe.name\ +'\_'\ + ari_c.name$
  - // Condition de valuation de l'élément cloné en fonction d'une propriété
  - ...
  - **pour chaque** *rôle* $r \in$ `getCompositionRoles(`$ari.type.behavior$`)` **faire**
    - **si** $abe =$ `getAtomicElement(`$ari.type.behavior, r$`)` **alors**
      - // Caractériser l'élément cloné selon le rôle joué pour la composition
      - $role \leftarrow r.name\ +'\_'\ + ari_c.name$

ficMechanismRole. Le même principe de création d'élément atomique ou de clonage d'élément composite que précédemment, est appliqué ici. Si un des sous-éléments joue un rôle de mécanisme spécifique (différent de celui impliqué au début de cette règle), ce nouveau mécanisme implique la considération de nouvelles instances. Sinon, le principe est appliqué pour le mécanisme localisé précédemment selon l'instance concernée.

Nous sommes revenus sur le mécanisme d'ordonnancement coopératif pour illustrer la mise œuvre d'un mécanisme spécifique. Pour rappel, ce type de mécanisme permet de prioritiser l'exécution de tâches ordonnançables de façon coopérative. La figure 4.7 représente sur le PSM 1, une application déployée constituée de trois tâches ordonnançables avec des priorités différentes.

Au cœur du prototype comportemental TaskBehavior, l'action inhibiteur InhibitorAction a été définie pour décrire le mécanisme d'ordonnancement coopératif d'une tâche. Elle est représentée sous la forme d'un élément de comportement composite. À la suite de la transformation, le prototype comportemental d'une tâche a été cloné trois fois en raison du nombre d'instances représentées dans l'application. Par mesure de clarté, les éléments clonés générés sur le PSM 2 n'apparaissent pas entièrement pour les instances HelloWorld2 et HelloWorld3. Ceci étant, l'action inhibiteur est bien représentée. Chaque clone de tâche moins prioritaire a été étendu de manière à intégrer autant d'actions inhibiteurs que de tâches plus prioritaires.

En se focalisant sur la tâche la moins prioritaire, le clonage a été appliqué après avoir

ALGORITHME 4.14 – Règle d'application d'un élément de comportement composite spécifique (*ruleCloneSpecificCompositeBehaviorElement*)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Concept source :
m_c : la classe *Composite* de MM_p

Élements impliqués :
ari : l'instance de ressource à l'origine de la génération, conforme à la classe *ApplicationResourceInstance* de MM_p
ari_c : l'instance de ressource concernée par un mécanisme, conforme à la classe *ApplicationResourceInstance* de MM_p

Notation :
cbe : l'élément de comportement composite à cloner, conforme à m_c

début
- **pour chaque** *sous-élément de comportement* $sbe \in cbe.subelements$ **faire**
  - **si** `hasSpecificMechanismRole(`bp, sbe`)` **alors**
    - // Clonage du sous-élément en fonction d'un nouveau mécanisme spécifique
    - ...
  - **sinon**
    - **si** sbe χ *Atomic* $\in MM_p$ **alors**
      - // Générer le sous-élément atomique à partir d'un mécanisme spécifique
      - `ruleCreateAtomicElementFromSpecificMechanism(`sbe, ari, ari_c`)`
    - **si** sbe χ *Composite* $\in MM_p$ **alors**
      - // Cloner le sous-élément composite qui décrit un mécanisme spécifique
      - `ruleCloneSpecificCompositeBehaviorElement(`sbe, ari, ari_c`)`

successivement :

1. vérifier l'existence d'un élément de comportement jouant le rôle de mécanisme *cooperativeScheduling* au sein du prototype comportemental. En l'occurrence, il s'agit ici de l'action InhibitorAction ;
2. repérer le nombre d'instances concernées par ce mécanisme. Le nombre retourné équivaut à deux en raison des deux tâches plus prioritaires. Remarquons ici que pour ce mécanisme, le rôle *priorityElements* a été considéré pour localiser les propriétés de priorité des tâches ;
3. vérifier la nature de l'élément de comportement qui est composite dans notre cas ;
4. cloner chaque atome de cet élément composite en le caractérisant avec l'instance concernée à chaque itération (i.e., HelloWorld2 puis HelloWorld3).

Cette manœuvre adaptée aux mécanismes des plates-formes logicielles concerne aussi les services appelés. Ces derniers peuvent en effet être impliqués dans certains mécanismes. Prenons l'exemple du service de relâchement d'un *mutex*. Dans la situation où ce *mutex* est relâché suite à l'appel de ce service, un certain nombre d'instances de ressources ordonnançables en attente de ce même *mutex* peuvent alors être rendues éligibles par l'ordonnanceur. Le nombre d'instances concernées est dans ce cas déterminé selon les instances susceptibles d'acquérir ce *mutex*, à travers le modèle de l'application déployée.

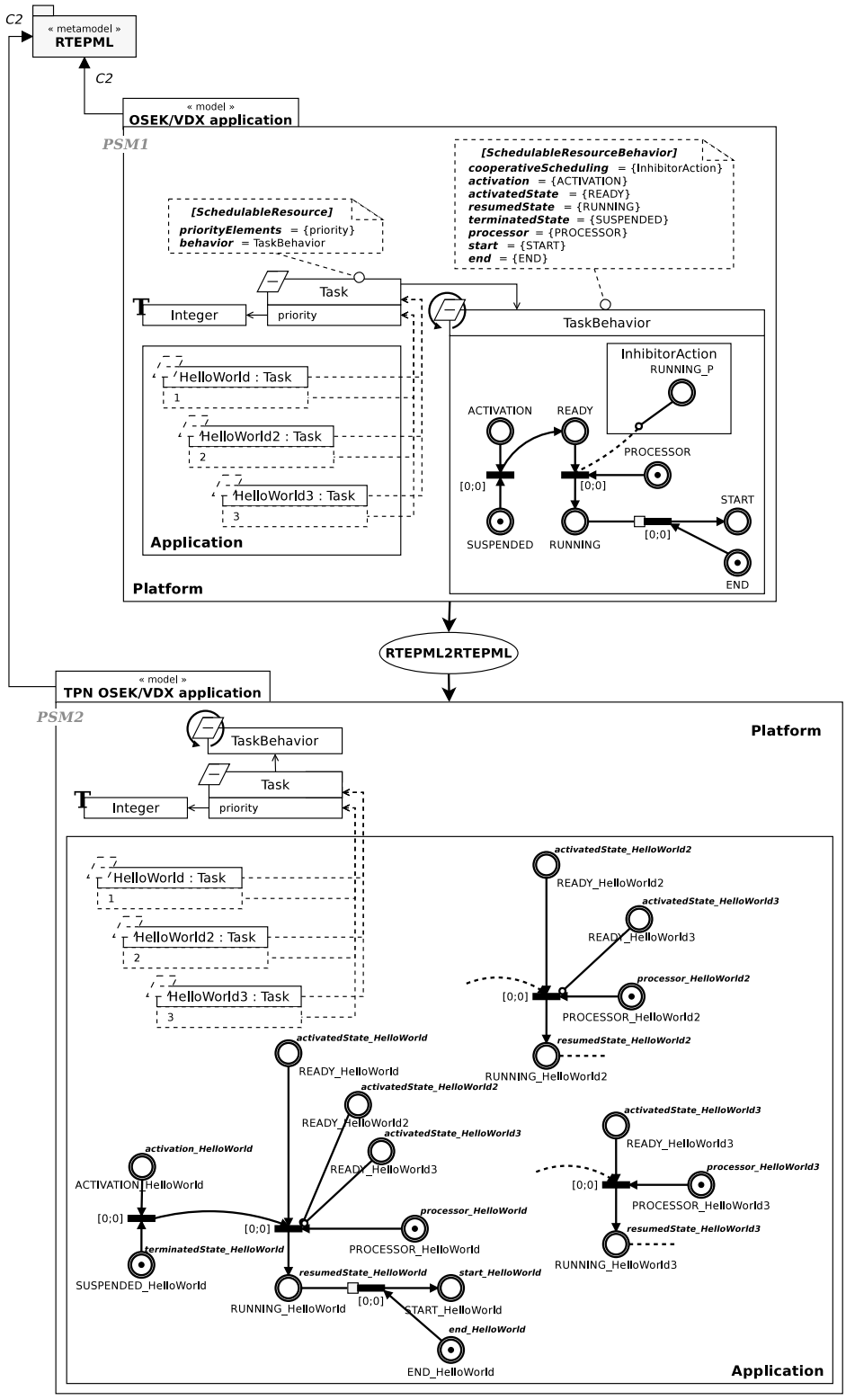


FIGURE 4.7 – Intégration du comportement : clonage de 3 tâches OSEK/VDX avec ordonnancement coopératif

La mise en œuvre de tels mécanismes sur les services est alors calquée sur les règles que nous venons de voir à partir de l'algorithme 4.10. Les règles les concernant n'ont pas été décrites puisque la seule différence réside dans la substitution de l'instance ari par l'appel

de service. Reprenons l'exemple du service de relâchement d'un *mutex*. Un rôle a été identifié dans RTEPML pour localiser le comportement d'une action permettant de rendre éligible une ressource ordonnançable sur un relâchement de ce genre (cf. Identification de rôles de mécanismes spécifiques, page 57). Grâce à ce rôle, une telle action peut être localisée parmi les éléments de comportement qui constituent le prototype comportemental d'un tel service. En conséquence, cette action peut être clonée spécifiquement selon le nombre d'instances concernées.

Nous avons passé en revue l'ensemble des règles retenues pour l'intégration d'éléments de comportement clonés dans un modèle d'application déployée. Ces éléments, une fois générés, sont disposés de manière orthogonale sur le modèle. Cette répartition fragmentaire nous amène à devoir les composer. Afin de localiser les points de connexion à travers ces fragments, les éléments qui remplissent cette fonction ont chacun été annotés d'un rôle de composition au moment du clonage. De plus, chaque ensemble d'éléments a été affecté de l'instance de ressource (ou du service appelé) qui est l'origine de leur clonage (*root*). Tous ces traceurs vont être exploités maintenant à travers la composition des fragments. Les fragments ne peuvent évidemment pas être composés de manière arbitraire. À l'image des routines d'exécution, certaines compositions sont à effectuer individuellement avant d'arriver à un modèle global.

4.2.5 Composition d'éléments : application sur les routines

Le comportement des instances de routines d'exécution constitue la base de notre composition. Les services d'exécution qui sont appelés depuis ces routines sont à l'origine de la concurrence et des interactions entre les ressources. Les fragments comportementaux qui représentent ces appels doivent donc être agencés de façon séquentielle en respectant l'ordre des appels rencontrés dans le modèle d'application déployée. L'algorithme 4.15 revient sur le principe d'intégration du comportement en ajoutant cette première étape de composition.

ALGORITHME 4.15 – Principe d'intégration d'un modèle comportemental d'application : composition des éléments de comportement clonés(➤routines)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Notation :
➤ I_R : l'ensemble des instances de routines $\{ari_R | ari_R.type\chi$ Routine $\in MM_p\}$
début
    // Dupliquer la plate-forme et chaque application déployée
    ...
    // Cloner les prototypes comportementaux
    ...
    // Composer les routines d'exécution
➤   **pour chaque** *instance de routine* $ari_R \in I_R$ **faire**
➤   **pour chaque** *service appelé* $cs_R \in ari_R.call$ **faire**
            // Composer la routine avec le prochain service appelé
➤   `ruleComposeRoutineCalls(`ari_R, cs_R`)`

Dans cette première étape, seules les instances de routines sont impliquées. Nous par-

tons du principe que l'ensemble de ces instances I_R a déjà été listé au sein de l'application. Par conséquent, les services appelés depuis chaque instance de routine ari_R sont atteignables. De ce fait, chaque fragment d'éléments clonés qui représente un service appelé cs_R est assemblé tour à tour, pour former le comportement de l'instance de routine. Pour chaque appel sélectionné, la règle *ruleComposeRoutineCalls* est alors appelée. L'algorithme 4.16 en détaille le principe.

ALGORITHME 4.16 – Règle de composition d'une routine d'exécution (*ruleComposeRoutineCalls*)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Concept source :
m_{ari} : la classe *ApplicationResourceInstance* de MM_p

Élément impliqué :
cs_R : le service appelé à composer, conforme à la classe *CallService* de MM_p
Notation :
ari_R : l'instance de routine à composer, conforme à m_{ari}

début
  pour chaque *élément cloné* $ce \in ari_R.clonedElements$ **faire**
    si $ce\chi$ *Atomic* $\in MM_p$ **alors**
      si `getMatchCallElement(`ce, cs_R`)` $\neq \emptyset$ **alors**
        // Générer un élément atomique fusionné à partir d'une composition
        `ruleCreateAtomicElementFromComposition(getMatchCallElement(`ce, cs_R`)`$, ce$`)`
  $ari_R.clonedElements \leftarrow \{cs_R.clonedElements\}$

Dans cette règle, l'objectif est de fusionner les éléments clonés qui sont annotés de rôles de composition au sein des fragments à composer. Les éléments sont pris en compte uniquement s'ils sont de nature atomique. En effet, les fragments de comportement étant décrits en TPN, nous avons orienté leur composition vers la fusion de places qui sont des éléments atomiques (cf. Représentation du comportement d'une plate-forme, page 58).

L'exécution de cette règle permet *in fine* de référencer les éléments clonés de l'appel impliqué vers ceux de l'instance de routine. Cette règle étant déclenchée à chaque nouvel appel, chaque élément cloné ce qui aura été référencé sur l'instance de routine peut potentiellement être fusionné avec un des éléments référencés sur ce nouvel appel. L'élément correspondant dans cs_R est retourné à l'aide du *helper getMatchCallElement*. Si cette correspondance est établie, une règle de création d'un élément fusionné (*ruleCreateAtomicElementFromComposition*) est appelée avec comme paramètres les éléments localisés.

En aparté, nous devons préciser comment ces correspondances sont établies à travers le *helper getMatchedCallElement*. Lors du clonage des appliqué sur les appels de service, les rôles annotés sur les éléments clonés d'un appel de service ont été suffixés en fonction de l'identité de l'instance référencée par cet appel. Ce parti pris tient de la légitimité de composition entre un fragment de comportement d'un service appelé avec celui qui décrit l'instance référencée (e.g., service de terminaison avec la tâche ciblée). Dans ce cas, les éléments clonés à fusionner sont ceux dont les rôles ont été suffixés de façon homonyme. Néanmoins, cette correspondance ne peut se limiter ici à une telle équivalence entre les

rôles. La figure 4.8 appuie cette idée.

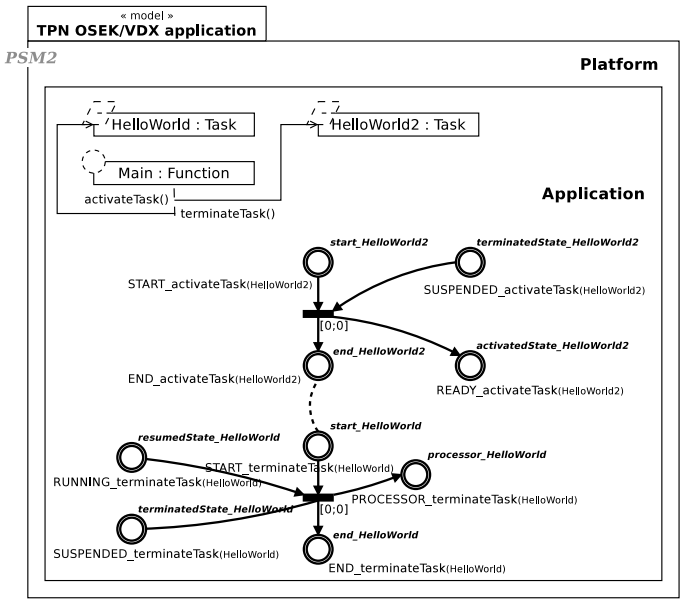


FIGURE 4.8 – Intégration du comportement : composition d'une routine OSEK/VDX avec 2 appels de services

Un extrait de modèle formel en TPN représente ici deux fragments obtenus pour deux services appelés depuis la même routine Main. Ces deux services sont appelés successivement pour 1) activer la tâche HelloWorld2 et 2) terminer la tâche HelloWorld. Ces deux fragments doivent par conséquent être assemblés dans cet ordre pour composer la routine. Cet extrait montre que, dans ce cas précis, seules les places caractérisées par les rôles *end_HelloWorld2* et *start_HelloWorld* sont à fusionner (i.e., celles reliées avec un arc en pointillés). Ce constat soulève l'idée que deux éléments clonés à fusionner ne sont pas obligatoirement annotés avec des rôles équivalents. De ce fait, certaines adaptations doivent s'opérer dans le *helper getMatchCallElement* au cas par cas selon le langage de modélisation employé pour la traduction des éléments. Une attention particulière sera par conséquent nécessaire pour éviter toute ambiguïté de composition. Une formalisation de cette composition est présentée dans le chapitre 5 suivant pour faire état de ces correspondances en TPN.

Ceci sert de transition à l'introduction de l'algorithme 4.17. Ce dernier décrit la règle *ruleCreateAtomicElementFromComposition* appelée lorsqu'une correspondance entre deux éléments clonés à fusionner a été trouvée.

Cette règle est une version générique et simplifiée de fusion entre un élément de comportement atomique source et un élément ciblé. Elle implique la création d'un nouvel élément atomique dans lequel sont centralisées les informations qui caractérisent ces éléments source et cible. La caractérisation de ce nouvel élément débute par l'affectation des instances *root* de l'élément ciblé à celles du nouvel élément. Comme lors de la création des éléments nouvellement clonés, cette affectation est aussi nécessaire pour le traçage des éléments fusionnés qui devront être considérés par la suite dans de nouvelles compositions. La caractérisation se poursuit par une succession de *helpers* appelés *getMergedName*, *getMergedName* et *getMergedRole*. Ces *helpers* retournent respectivement le nom, la valeur et le rôle annoté sur le nouvel élément. Comme pour la mise en correspondance des éléments à fusionner, le renseignement de ces informations doit être adapté sur le nouvel élément fusionné en fonction des informations qui caractérisent les éléments source et cible. La for-

ALGORITHME 4.17 – Génération d'un élément de comportement atomique fusionné (*ruleCreateAtomicElementFromComposition*)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Concept source :
m_{abe} : la classe *Atomic* de MM_p

Élement impliqué :
abe_t : l'élément de comportement atomique cloné cible à fusionner, conforme à la classe *Atomic* de MM_p
Notation :
abe_s : l'élément de comportement atomique cloné source à fusionner, conforme à m_{abe}

début
  correspondance avec *Atomic*
    // Caractériser l'élément atomique fusionné
    $rootInstances \leftarrow \{abe_t.rootInstances\}$
    $name \leftarrow$ `getMergedName(`abe_s, abe_t`)`
    $val \leftarrow$ `getMergedValue(`abe_s, abe_t`)`
    $role \leftarrow$ `getMergedRole(`abe_s, abe_t`)`
  `setMergedAtomicElement(`abe_s, abe_t`)`

malisation décrite dans le chapitre 5 suivant donne une idée plus précise de ces adaptations en TPN.

En dernier lieu, le *helper setMergedAtomicElement* caractérise l'élément fusionné selon d'autres critères qui sont abstraits au niveau de RTEPML. D'un point de vue TPN, ces critères concernent d'une part, les relations d'arcs entrants et d'arcs sortants associées entre les places et les transitions (cf. Héritage d'élément de comportement, page 55) et d'autre part, le marquage des places. Une introspection sur la classe *Atomic* de RTEPML est nécessaire pour parcourir l'ensemble des relations associées sur les classes filles de *Atomic* incarnées par les concepts en TPN. Une définition formelle de la composition en TPN vient fixer cette caractérisation dans le chapitre suivant.

L'étape suivante de composition doit ensuite être appliquée sur les points d'entrées des ressources concurrentes. Le but est de lier le cycle de vie de ces ressources vis-à-vis de leurs exécutions (i.e., les routines d'exécutions).

4.2.6 Composition d'éléments : application sur les points d'entrée

Dans RTEPML, le rôle de point d'entrée est affecté à une propriété de ressource concurrente. Comme nous l'avons déjà évoqué (cf. Représentation explicite des plates-formes, page 2.1.2), un point d'entrée a pour but de pointer le traitement exécutif associé à une ressource concurrente. Ce traitement est caractérisé par une routine d'exécution. Lorsqu'une ressource concurrente a été instanciée durant le déploiement, cette propriété sert à renseigner l'instance de routine qui traitera l'exécution de l'instance concurrente. L'instance de routine concernée est référencée par cette propriété de point d'entrée au sein de l'application déployée.

D'un point de vue comportemental, cette propriété n'implique pas la valuation d'élé-

ment de comportement comme nous l'avons vu dans l'algorithme 4.4. Elle est gérée comme une étape de la composition. Chaque routine composée est ensuite associée avec la ressource concurrente qui pointe sur la routine appropriée. L'algorithme 4.18 étend le principe d'intégration d'un modèle comportemental d'application, en ajoutant la composition des points d'entrée.

ALGORITHME 4.18 – Principe d'intégration d'un modèle comportemental d'application : composition des éléments de comportement clonés(➤points d'entrée)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Notations :
I_R : l'ensemble des instances de routines $\{ari_R | ari_R.type\chi$ Routine $\in MM_p\}$
➤ I_C : l'ensemble des instances de ressources concurrentes $\{ari_C | ari_C.type\chi$ ConcurrentResource $\in MM_p\}$
début
    // Dupliquer la plate-forme et chaque application déployée
    ...
    // Cloner les prototypes comportementaux
    ...
    // Composer les routines d'exécution
    ...
    // Composer les points d'entrée
➤ **pour chaque** *instance de ressource concurrente* $ari_C \in I_C$ **faire**
➤ **pour chaque** *rôle* $r \in$ `getPropertyRoles(`$ari_C.type$`)` **faire**
➤ **si** $r =$ `getEntryPointElementRole(`$ari_C.type$`)` **alors**
➤ **pour chaque** *propriété* $p \in ari_C.properties$ **faire**
➤ **si** $p.property =$`getProperty(`$ari_C.type, r$`)` **alors**
// Composer la ressource concurrente avec son point d'entrée
➤ `ruleComposeInstances(`$p.ref, ari_C$`)`

Cette seconde étape agit pour chaque instance de ressource concurrente ari_C préalablement repérée dans l'ensemble I_C au sein de l'application. Chaque instance de ressource concurrente va être composée avec son point d'entrée localisé. Ainsi, successivement :

1. les rôles de propriété joués au sein de la ressource qui type l'instance concurrente $ari_C.type$ sont en premier lieu localisés à l'aide du emphhelper *getPropertyRoles* déjà présenté ;
2. le rôle de point d'entrée est ensuite isolé parmi les rôles précédemment localisés, grâce au *helper getEntryPointElementRole*. Ce *helper* sert uniquement ici de traduction pour retourner le rôle désiré ;
3. la propriété qui joue ce rôle de point d'entrée est enfin localisée parmi les propriétés de l'instance de ressource concurrente sélectionnée. Nous retrouvons le *helper getProperty* pour cela.

Une fois la propriété localisée, la règle de composition *ruleComposeInstances* est appelée afin de composer l'instance de ressource concurrente ari_C avec son point d'entrée

$p.ref$. Cette règle est détaillée dans l'algorithme 4.19. Elle est générique à l'ensemble des instances dont les fragments d'éléments clonés doivent s'associer.

ALGORITHME 4.19 – Règle de composition de deux instances (*ruleComposeInstances*)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Concept source :
m_{ari} : la classe *ApplicationResourceInstance* de MM_p

Élément impliqué :
ari_t : l'instance cible à composer, conforme à la classe *ApplicationResourceInstance* de MM_p
Notation :
ari_s : l'instance source à composer, conforme à m_{ari}

début
  pour chaque *élément cloné* $ce \in ari_s.clonedElements$ **faire**
    si $ce\chi$ *Atomic* $\in MM_p$ **alors**
      si `getMatchInstanceElement(`ce, ari_t`)`$\neq \emptyset$ **alors**
        // Générer un élément atomique fusionné à partir d'une composition
        `ruleCreateAtomicElementFromComposition(`ce,
        `getMatchInstanceElement(`ce, ari_t`))`
  $ari_t.clonedElements \leftarrow \{ari_s.clonedElements\}$

Le principe de cette règle est finalement le même que celui mentionné dans l'algorithme 4.16 pour composer les appels de service d'une routine. Son exécution permet *in fine* de référencer les éléments clonés de l'instance source vers ceux de l'instance cible impliquée. La mise en correspondance des éléments clonés à fusionner intervient cette fois grâce au *helper getMatchInstanceElement*. Il a pour paramètre cible une instance de ressource et non un appel de service. Nous avons bien affaire à des fragments d'éléments clonés qui représentent des instances.

Afin de donner un aperçu d'une telle composition, la figure 4.9 montre deux fragments en TPN clonés qui décrivent une instance de ressource concurrente et celle d'une routine préalablement composée.

La tâche HelloWorld a pour point d'entrée la routine principale Main. Dans cet extrait, le comportement de la routine est composée d'une séquence arbitraire d'appels de service. Nous pouvons constater ici la non connaissance des rôles annotés sur les places de début et de fin de cette routine. Ces rôles dépendent bien évidemment des services appelés. Les places devant être fusionnées sont celles de début et de fin de la routine d'exécution qui sont reliées par des arcs en pointillés. Cette fois, l'adaptation est faite au cœur du *helper getMatchInstanceElement* pour localiser les éléments annotés des rôles *start_HelloWorld* et *end_HelloWorld*) de la tâche.

Le comportement des ressources concurrentes associées avec leurs routines d'exécution est maintenant formé. L'étape suivante se concentre sur l'ensemble des ressources concurrentes, y compris celles qui n'ont pas de points d'entrée.

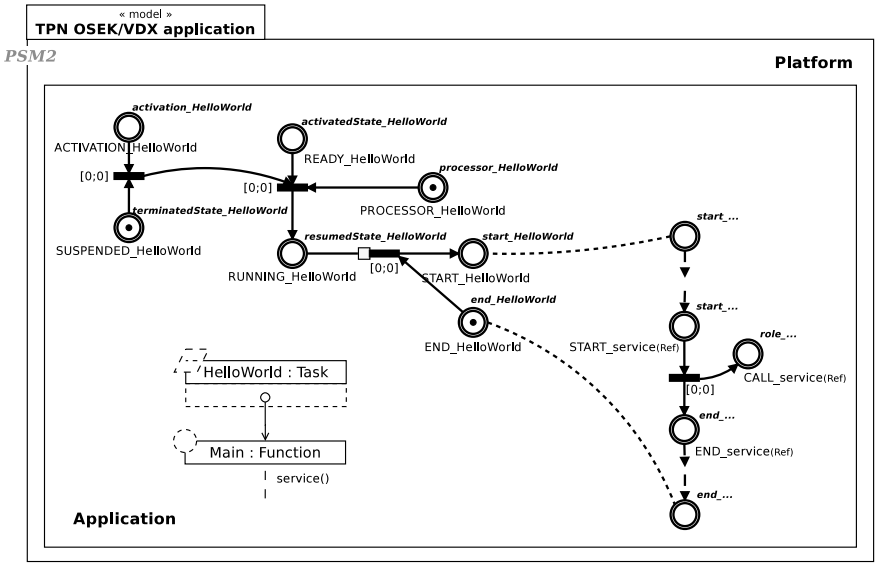


FIGURE 4.9 – Intégration du comportement : composition d'une tâche OSEK/VDX avec son point d'entrée

4.2.7 Composition d'éléments : application sur les ressources concurrentes

La mise en concurrence de telles ressources repose avant tout sur le partage d'une ressource commune : le processeur. Ce dernier peut aussi être représenté comme une ressource d'interaction. Ceci étant, nous le verrons dans la sous-section suivante, page 94, les ressources concurrentes interagissent entre elles au moyen des services appelés. Or, bien souvent les plates-formes n'offrent pas de services pour directement manipuler le processeur. Pour cette raison, un rôle *processor* a été identifié au sein du prototype comportemental associé au concept de *ConcurrentResource* dans RTEPML. Les éléments clonés qui jouent ce rôle doivent en l'occurrence être fusionnés à ce stade de la composition.

Dans la première partie présentée, nous avons annoncé le clonage d'éléments issus de mécanismes spécifiques. Selon le mécanisme, un certain nombre de ressources concurrentes peuvent être impliquées et avoir un impact sur l'ordonnancement de la ressource concurrente à l'origine du clonage. En conséquence, ces mécanismes sont aussi mis en jeu en fusionnant ces éléments à ce moment de la composition. Les éléments issus des associations entre instances de prototypes de conception sont aussi concernés par cette étape de la composition. Si les instances sont typées par des ressources concurrentes, les éléments engagés seront fusionnés ici.

Un enrichissement du principe d'intégration de modèle comportemental d'application est par conséquent présenté dans l'algorithme 4.20. Il tient compte de la composition des éléments clonés pour chaque instance de ressource concurrente sélectionnée à travers le modèle de l'application déployée.

La mise en œuvre d'une telle composition consiste à assembler tour à tour les éléments clonés de chaque instance de ressource concurrente. Chaque instance listée dans I_C est passée en paramètre de la règle de composition *ruleComposeInstances* comme l'instance source. Elle est composée avec l'instance située juste après dans la liste qui, quant à elle, est considérée comme l'instance cible. Comme pour les appels de service, un *helper* nommé *getNextConcurrentResource* vient retourner l'instance de ressource concurrente cible.

Une simplification du résultat escompté est illustrée figure 4.10.

Dans cet extrait, deux tâches HelloWorld et HelloWorld2 qui doivent être mises en concurrence sont décrites suivant leurs TPNs respectifs. Leurs routines ne sont pas représentées ici

ALGORITHME 4.20 – Principe d'intégration d'un modèle comportemental d'application : composition des éléments de comportement clonés(➤ressources concurrentes)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Notations :
I_R : l'ensemble des instances de routines $\{ari_R | ari_R.type\chi$ Routine $\in MM_p\}$
➤ I_C : l'ensemble *ordonné* des instances de ressources concurrentes $\{ari_C | ari_C.type\chi$ ConcurrentResource $\in MM_p\}$
début
  // Dupliquer la plate-forme et chaque application déployée
  ...
  // Cloner les prototypes comportementaux
  ...
  // Composer les routines d'exécution
  ...
  // Composer les points d'entrée
  ...
  // Composer les ressources concurrentes
➤ **pour chaque** *instance de ressource concurrente* $ari_C \in I_C$ **faire**
➤ **si** `getNextConcurrentResource(`ari_C`)`$\neq \emptyset$ **alors**
      // Composer la ressource concurrente avec la prochaine listée
➤ `ruleComposeInstances(`ari_C`,`
      `getNextConcurrentResource(`ari_C`))`

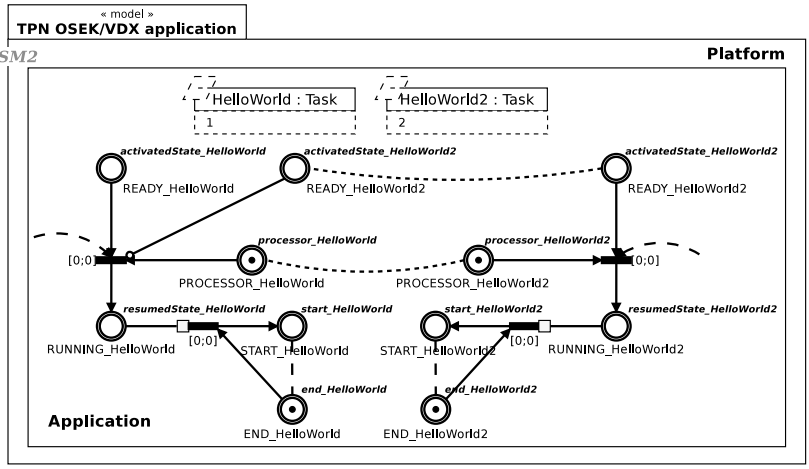


FIGURE 4.10 – Intégration du comportement : composition de deux tâches OSEK/VDX en concurrence

pour plus de clarté. Dans la norme OSEK/VDX, aucun service n'offre la possibilité de mettre une tâche en exécution. De fait, le processeur est décrit à travers le comportement d'une tâche OSEK/VDX (i.e., PROCESSOR_HelloWorld). En admettant alors que HelloWorld soit la première instance listée, l'assemblage est réalisé après localisation de l'élément annoté du rôle *processor_HelloWorld2* au sein du TPN qui décrit HelloWorld2.

Cette composition intègre aussi le mécanisme d'ordonnancement coopératif déjà men-

tionné à la page 80. La mise en jeu de ce mécanisme est rendue possible grâce à la fusion des éléments READY_HelloWorld2. Pour parvenir à la localisation de ces éléments, la mise en correspondance des rôles homonymes *activatedState_HelloWorld2* a été établie. Ainsi, la tâche HelloWorld2 est rendue plus prioritaire.

Cette configuration met en exergue le problème que nous avons introduit au sujet du marquage des places fusionnées. Dans notre exemple, les places PROCESSOR_HelloWorld et PROCESSOR_HelloWorld2 sont toutes deux marquées d'un jeton. Dans ce cas précis, un seul jeton doit apparaître au moment de la fusion puisque cette place décrit le processeur. Nous devons donc bien déterminer un marquage générique afin d'éviter toute ambiguïté.

Les ressources concurrentes peuvent directement interagir entre elles (e.g., activation de tâche, etc.), mais bien souvent elles interagissent aussi par le biais des ressources d'interaction (e.g., synchronisation à l'aide d'un évènement, message échangé au moyen d'une boîte aux lettres, etc.). Le comportement de ces ressources d'interaction a un rôle important dans la composition finale du modèle comportemental à générer.

4.2.8 Composition d'éléments : application sur les ressources d'interaction

Cette phase complète le TPN formé précédemment lors de la mise en concurrence des ressources concurrentes. L'objectif ici est de composer chaque ressource d'interaction avec l'ensemble des ressources concurrentes assemblées précédemment. Comme pour les ressources concurrentes, les ressources d'interaction offrent des services pour qu'elles puissent être manipulées. L'assemblage de ces ressources d'interaction passe donc par ces service qui ont déjà été inclus dans les routines d'exécution.

L'algorithme 4.21 termine le travail de composition en sélectionnant une à une chaque instance de ressource d'interaction dans une liste ordonnée I_I.

Le principe reste le même qu'avec les ressources concurrentes, à ceci près que l'instance cible de composition est représentée cette fois par la dernière instance de ressource concurrente ari_C. Cette dernière instance référence en effet l'ensemble des éléments clonés assemblés lors des précédentes compositions. Chaque instance d'interaction ari_I listée dans l'ensemble I_I est ainsi composée à travers la règle *ruleComposeInstances*, comme instance source.

La figure 4.11 offre un aperçu du résultat attendu après une telle composition. Elle s'appuie sur l'exemple de la ressource allouée dans OSEK/VDX pour bloquer l'ordonnanceur (cf. Composition parallèle avec ACSR et TROS, figure 2.7) et ainsi éviter toute préemption.

Cette configuration impliquerait alors de considérer un mécanisme spécifique pour inhiber la préemption. Dans le cas présenté, nous nous abstrayons volontairement de ce mécanisme afin de se concentrer uniquement sur la composition des ressources d'interaction. Les deux mêmes tâches que précédemment se partagent cette fois la ressource d'interaction Res_Scheduler. Chacune pointe sur une routine d'exécution. Dans la première routine, un service est appelé pour relâcher Res_Scheduler dans le but de rendre la tâche HelloWorld préemptable. Puis dans la seconde routine, un service est appelé à son tour pour acquérir Res_Scheduler dans le but d'inhiber HelloWorld2 de toute préemption. Les deux tâches qui ont précédemment été composées via le processeur sont une nouvelle fois reliées par la ressource d'interaction Res_Scheduler. Les éléments FREE_Res_Scheduler et BUSY_Res_Scheduler qui décrivent cette ressource sont fusionnés avec les éléments localisés grâce aux rôles homonymes *countState_Res_Scheduler* et *discountState_Res_Scheduler*.

ALGORITHME 4.21 – Principe d'intégration d'un modèle comportemental d'application : composition des éléments de comportement clonés(➤ressources d'interaction)

Métamodèle source :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Métamodèle cible :
MM_p : le métamodèle de plate-forme d'exécution (RTEPML)

Notations :
I_R : l'ensemble des instances de routines $\{ari_R | ari_R.type\chi$ Routine $\in MM_p\}$
I_C : l'ensemble *ordonné* des instances de ressources concurrentes $\{ari_C | ari_C.type\chi$ ConcurrentResource $\in MM_p\}$
➤ I_I : l'ensemble *ordonné* des instances de ressources d'interaction $\{ari_I | ari_I.type\chi$ InteractionResource $\in MM_p\}$

```
début
    // Dupliquer la plate-forme et chaque application déployée
    ...
    // Cloner les prototypes comportementaux
    ...
    // Composer les routines d'exécution
    ...
    // Composer les points d'entrée
    ...
    // Composer les ressources concurrentes
    pour chaque instance de ressource concurrente ari_C ∈ I_C faire
        si getNextConcurrentResource(ari_C) ≠ ∅ alors
            // Composer la ressource concurrente avec la prochaine listée
            ruleComposeInstances(ari_C,
            getNextConcurrentResource(ari_C))
➤       sinon
            // Poursuivre la composition avec la prochaine ressource d'interaction
➤           pour chaque instance de ressource d'interaction ari_I ∈ I_I faire
                // Composer la ressource d'interaction avec l'ensemble
➤               ruleComposeInstances(ari_I, ari_C)
```

4.3 Synthèse

Une stratégie a été proposée dans ce chapitre pour considérer le comportement des plates-formes d'exécution logicielles après déploiement d'une application temps réel. Un processus a donc été avancé dans le but de générer un modèle global composé à la fois de l'application déployée et du comportement de la plate-forme considérée. Un certain nombre de règles de transformation ont aussi été définies de manière impérative dans l'optique de séquentialiser le processus de génération. Ces règles ont séparées en deux principaux axes : 1) le clonage des prototypes comportementaux associés aux ressources et aux services offerts par la plate-forme considérée pour les besoins de l'application déployée (i.e., suivant le nombre d'instances de ressources et d'appels de service) et 2) la composition des prototypes clonés pour former le modèle global du comportement de l'application déployée.

Le premier axe est réparti en deux parties disjointes comme en témoigne la figure 4.12. Toutefois dans les deux cas, la transformation aboutit à la création d'éléments atomiques clonés issus de prototypes comportementaux. La divergence des deux orientations prises

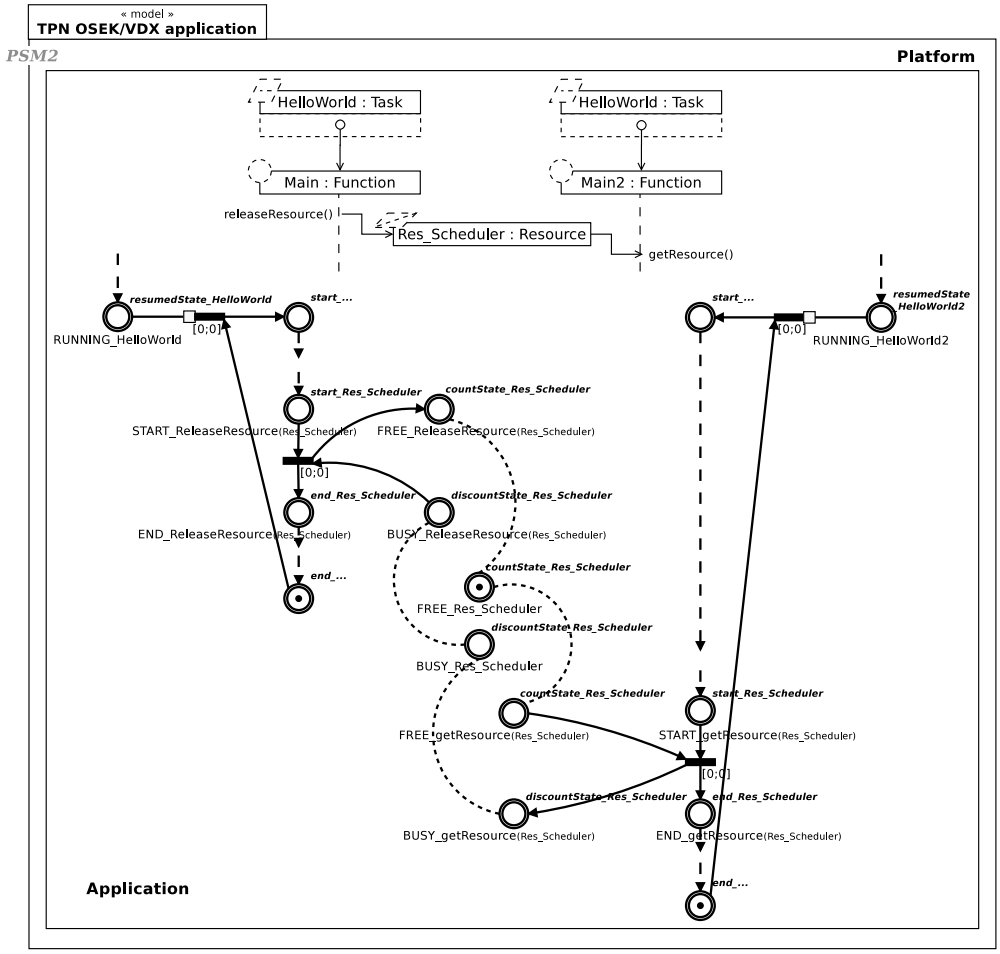


FIGURE 4.11 – intégration du comportement : composition avec une ressource d'intéraction OSEK/VDX

aux travers des règles de clonage provient de la spécificité de certains mécanismes des plates-formes considérées. Cette orientation (partie droite de la figure) peut générer une multiplicité des prototypes clonés selon l'application. En réalité, deux autres axes compléteraient cette hiérarchie. Il s'agit des services qui sont supportés de la même façon que les ressources. La séquentialité des règles pour les services étant identique à celle des ressources, ces deux axes ne sont pas représentés sur cette figure.

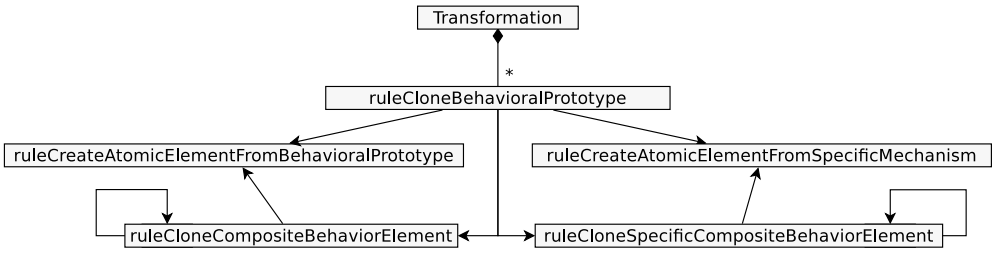


FIGURE 4.12 – Hiérarchie des règles de clonage mises à contribution dans le processus

Le second axe apparaît sur la figure 4.13. Il aboutit au même point, celui de créer des éléments atomiques fusionnés issus de la composition des prototypes comportementaux clonés. Le chemin parcouru est néanmoins divisé en deux en raison de la nature des fragments de comportement à composer. D'un côté, les instances de routine d'exécution sont

constituées au regard des services appelés depuis chacune d'entre elles. De l'autre côté, le modèle global est constitué au fur et à mesure en fonction des instances de routine, de ressource concurrente et de ressource d'interaction.

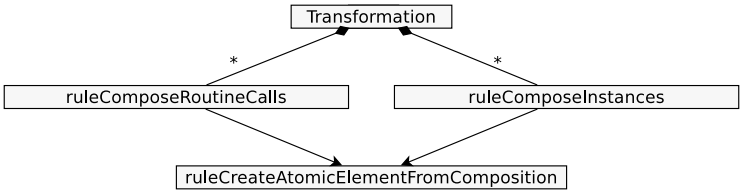


FIGURE 4.13 – Hiérarchie des règles de composition mises à contribution dans le processus

Ces règles ont été réalisées indépendamment de la plate-forme considérée, mais aussi indépendamment du langage de modélisation adopté pour la traduction des prototypes comportementaux. Elles s'appuient en effet sur les concepts de RTEPML et non sur ceux des TPNs. La notion de rôles que nous avons abordée dans la présentation de RTEPML permet cette transparence au moment de cloner et de composer. Toutefois, nous avons rencontré certaines dépendances vis-à-vis des TPNs au moment de développer ces règles. Le tableau 4.1 expose ces dépendances au travers des fonctions (ou *helpers*) de localisation et de traduction rencontrées tout au long du processus. Les dépendances au sein de RTEPML sont aussi catégorisées selon la structure, le comportement et l'application intégrée. Ces dernières restent cependant génériques quelque soit la plate-forme considérée.

Les fonctions qui dépendent de l'expressivité du langage de modélisation du comportement comme les TPNs, s'articulent autour de deux points importants :

- **la mise en correspondance des éléments clonés à fusionner** dans l'optique de localiser les points de connexion des fragments de comportement à composer. Même si la localisation est indépendante des TPNs grâce aux rôles de composition, la synchronisation des fragments de comportement en dépend néanmoins ;
- **la caractérisation des éléments clonés fusionnés** afin d'éviter toute ambiguïté de composition. Cette caractérisation est aussi générique, mais la synchronisation évoquée à l'instant nécessite de préciser quelles informations sont à prendre en compte au moment de la fusion. De plus, les relations entre les places sous-jacentes aux éléments clonés en fusion et les autres éléments doivent être ajustées sur les places fusionnées.

Devant de telles précisions, le chapitre suivant complète ce processus en apportant une formalisation de la composition en TPN.

		RTEPML			TPN
		Structure	Comportement	Application	
Fonctions (ou *helpers*)	$getPropertyRoles$		✓		
	$getAtomicElement$		✓		
	$getProperty$	✓			
	$getPropertyValue$		✓	✓	
	$getCompositionRoles$		✓		
	$getServiceRoles$	✓			
	$getService$	✓			
	$getBehavioralPrototypeService$		✓		
	$hasAssociatedInstance$			✓	
	$getMasterAssociatedInstance$			✓	
	$hasSpecificMechanismRole$		✓		
	$getSpecificMechanismRoles$		✓		
	$getConcernedInstances$			✓	
	$getCompositeElement$		✓		
	$getMatchCallElement$		✓	✓	✓
	$getMergedName$		✓	✓	✓
	$getMergedValue$		✓	✓	✓
	$getMergedRole$		✓	✓	✓
	$setMergedAtomicElement$		✓	✓	✓
	$getEntryPointElementRole$	✓			
	$getMatchInstanceElement$		✓	✓	✓
	$getNextConcurrentResource$			✓	

TABLE 4.1 – Fonctions mises à contribution et dépendances dans le processus

5

Formalisation de la composition

Sommaire

Dans ce chapitre, une formalisation de la composition qui vient d'être présentée est proposée. Cette contribution qui a déjà fait l'objet de publications [50][51] cherche à vérifier la faisabilité d'une telle démarche de composition en TPN. Elle répond aussi à certaines adaptations de composition évoquées au moment de fusionner des éléments telles que les places.

Ce chapitre s'organise en trois sections. La première section introduit les fondamentaux des opérations de clonage et de composition de TPNs basée sur la notion de rôle. La section suivante s'appuie sur ces opérateurs pour illustrer la composition d'un modèle comportemental en TPN d'une application déployée sur une plate-forme. Enfin, la dernière section donne une synthèse de la formalisation proposée.

5.1 Composition de TPNs basée sur des rôles

Cette section donne une manière d'aborder la construction d'un modèle global d'une application déployée à partir de fragments comportementaux qui vont être interconnectés. Ces fragments représentent les prototypes comportementaux en TPN de la plate-forme ciblée, sur lesquels des rôles ont été projetés. Les prototypes subissent un clonage en fonction de l'application. Les clones ainsi obtenus sont synchronisés suivant les rôles attribués pour former le modèle global.

5.1.1 Les *Time Petri Nets* avec rôles

Dans RTEPML, l'attribut *role* a été ajouté au concept d'élément de comportement dans le but d'annoter les éléments clonés d'un rôle en vue de la composition (cf. Identification de rôles de composition, page 56). Ensuite, nous avons convenu que dans le cadre d'une traduction des prototypes comportementaux en TPN, les éléments clonés qui sont amenés à être fusionnés au fil de la composition ne pouvaient être que des places (cf. Représentation du comportement d'une plate-forme, page 58). Ces hypothèses nous amènent à étendre la définition des TPNs avec la notion de rôles. Cette définition vient étendre celle déjà abordée dans le chapitre 3 (cf. Justification du choix du langage de modélisation formelle, page 53) pour la modélisation des TPNs avec arcs de lecture et d'inhibition.

Définition 4 (RI_TPN avec rôles) *Un RI_TPN avec des rôles est un tuple* $\mathcal{N} = \langle \mathcal{T}_{\mathcal{RI}}, R, \lambda \rangle$ *où :*

- $\mathcal{T}_{\mathcal{RI}}$ *est un RI_TPN,*
- R *est un ensemble fini de* rôles,
- $\lambda : P \rightarrow R \cup \{\perp\}$ *est la* fonction d'assignation *d'un rôle à une place avec* $\perp$ *précisant qu'aucun rôle n'est assigné à une place. Ci-après, quelques notations et quelques propriétés de cette fonction sont énumérées :*
  1. $P_\lambda = \{p \in P \mid \lambda(p) \neq \perp\}$ *est l'ensemble des places avec un rôle ;*
  2. $\lambda_{\setminus P_\lambda} : P_\lambda \rightarrow R$ *est une fonction* injective d'assignation *d'un rôle ;*
  3. $\lambda^{-1} : R \cup \{\perp\} \rightarrow P \cup \{\emptyset\}$ *tel que*
  $$\begin{cases} \forall r \in R, \lambda^{-1}(r) = \begin{cases} p \text{ si } \lambda(p) = r \\ \emptyset \text{ sinon} \end{cases} \\ \lambda^{-1}(\perp) = \emptyset \end{cases}$$

Notons que la sémantique opérationnelle d'un RI_TPN avec rôles $\mathcal{N} = \langle \mathcal{T}_{\mathcal{RI}}, R, \lambda \rangle$ est la même que celle d'un RI_TPN. En effet, l'utilisation de rôles au sein de la définition d'un RI_TPN n'impacte pas sur sa sémantique.

5.1.2 Clonage de TPNs avec rôles

Dans le chapitre précédent, le clonage des éléments de comportement (cf. Clonage d'éléments, page 67) est présenté comme une étape intermédiaire du processus de considération du comportement. Une même instance de prototype comportemental peut en effet être dupliquée plusieurs fois selon le nombre d'instances de ressources et d'appels de services présents sur le modèle de l'application déployée. En conséquence, un opérateur de clonage a été défini formellement pour les fragments comportementaux en RI_TPN avec rôles, conformément à la caractérisation des éléments clonés donnée à travers les algorithmes de clonage.

Seuls les éléments de comportement atomiques tels que les places, les transitions, mais aussi les rôles annotés sont identifiés pour cet opérateur [1]. La caractérisation s'appuie principalement sur le nom des instances de ressources auxquelles les éléments de comportement clonés sont liés. Toutefois, les éléments de comportement clonés qui sont liés aux appels de service doivent être caractérisés selon le nom des instances référencées pour ces appels. Cette double caractérisation est à prendre en considération si la nature du RI_TPN à composer représente un service appelé. Dans ce dernier cas, le nommage des rôles est appliqué distinctement de celui des places et des transitions.

Soit $\mathcal{N} = \langle P, T, \mathsf{Pre}, \mathsf{Post}, m_0, I_s, \mathsf{Read}, \mathsf{Inh}, R, \lambda \rangle$, un RI_TPN avec rôles qui représente une instance de prototype comportemental (de ressource ou de service) à cloner. Les arguments suivants $root$ et ref renseignent respectivement :

- le suffixe retenu pour nommer les places et les transitions suivant l'identité de l'instance de ressource ou de l'appel de service à l'origine du clonage (cf. Revenir au clonage d'éléments pour avoir connaissance de ce suffixe selon la nature du prototype comportemental) ;
- le suffixe retenu pour nommer les rôles suivant l'identité d'une instance de ressource référencée dans le cas où c'est un appel de service qui est à l'origine du clonage. Si c'est une instance de ressource qui est à l'origine du clonage, alors cette instance sera considérée comme l'instance de ressource référencée.

La fonction de nommage $\rightharpoonup$ est une fonction bijective de Set vers Set' où $Set \in \{P, T, R\}$.

Définition 5 (Clonage de RI_TPN avec rôles) *Le clonage d'un RI_TPN $\mathcal{N}$ avec des rôles est noté par $\mathcal{N}_{clone} = Clone(\mathcal{N}, root, ref) = \langle P_{root}, T_{root}, \mathsf{Pre}_{root}, \mathsf{Post}_{root}, m_{0-root}, I_{s-root}, \mathsf{Read}_{root}, \mathsf{Inh}_{root}, R_{ref}, \lambda_{root} \rangle$. Sa définition est la suivante :*

$$
\begin{aligned}
\mathcal{N}_{clone} &= Clone(\mathcal{N}, root, ref) \\
&= \mathcal{N} \Big| \begin{array}{l}
P_{root} = \{p_{root} \textit{ tel que } \forall p \in P \textit{ et } p \rightharpoonup p_{root}\}, \\
T_{root} = \{t_{root} \textit{ tel que } \forall t \in T \textit{ et } t \rightharpoonup t_{root}\}, \\
R_{ref} = \{r_{ref} \textit{ tel que } \forall r \in R \textit{ et } r \rightharpoonup r_{ref}\}, \\
\forall p \in P, \forall t \in T, \forall r \in R, \textit{ tel que } p \rightharpoonup p_{root}, t \rightharpoonup t_{root}, \\
\textit{ et } r \rightharpoonup r_{ref} \textit{ nous avons :} \\
\mathsf{Pre}_{root}(p_{root}, t_{root}) = \mathsf{Pre}(p, t), \\
\mathsf{Post}_{root}(p_{root}, t_{root}) = \mathsf{Post}(p, t), \\
\mathsf{Read}_{root}(p, t) = \mathsf{Read}_{root}(p_{root}, t_{root}), \\
\mathsf{Inh}_{root}(p, t) = \mathsf{Inh}_{root}(p_{root}, t_{root}), \\
\lambda_{root}(p) = r \textit{ ssi } \lambda_{root}(p_{root}) = r_{ref} \\
\lambda_{root}(p) = \perp \textit{ ssi } \lambda_{root}(p_{root}) = \perp
\end{array}
\end{aligned}
$$

5.1.3 Clonage spécifique de TPNs avec rôles

Nous avons soulevé précédemment le besoin de décrire partiellement certains mécanismes spécifiques rencontrés sur les plates-formes d'exécution logicielles. Un mécanisme peut entraîner un clonage spécifique d'un fragment comportemental en RI_TPN selon un certain nombre d'instances de ressources concernées (cf. Clonage des mécanismes spécifiques, page 80). Du fait de la spécificité de chaque mécanisme, une formalisation est nécessaire et doit être appliquée au cas par cas. Évidemment, nous ne verrons pas l'ensemble de

1. La caractérisation qui va suivre ne concerne que le nommage d'éléments clonés. Pour cette raison, les autres éléments de comportement atomiques tels que les arcs, le marquage des places, etc., ne sont pas identifiés ici car leur nommage n'impactera pas sur la composition.

ces mécanismes dans ce chapitre. Pour cette raison, nous nous sommes basés sur un exemple simple de mécanisme rencontré sur toutes les plates-formes. Le mécanisme proposé est celui de l'ordonnancement coopératif basé sur des priorités fixes.

Ce mécanisme est rencontré dans les applications temps réel au moment où des ressources concurrentes telles que des tâches ordonnançables deviennent éligibles en attente du processeur. Dans ce contexte, les tâches de priorités inférieures doivent être inhibées par celles de priorités supérieures. L'ordonnancement coopératif intervient en partie à travers le mécanisme de préemption. Néanmoins, ce dernier étant plus conséquent à mettre en application avec les TPNs, il ne sera pas formalisé ici. La préemption implique en effet de prévoir à la fois de stopper l'exécution des tâches de priorités inférieures vis-à-vis des tâches éligibles de priorités supérieures et d'inhiber les services appelés depuis les routines d'exécution pointées par les tâches préemptées.

Un cas d'ordonnancement coopératif a été illustré dans le chapitre précédent sur la figure 4.7. Le RI_TPN $\mathcal{N}_{HelloWorld} = Clone(\mathcal{N}, root, ref)$ représenté sur cette figure a dans un premier temps été obtenu suite au clonage du prototype comportemental TaskBehavior avec $ref = root = HelloWorld$. Puis, le RI_TPN qui décrit l'action d'inhibition InhibitorAction a quant à lui été cloné deux fois puisque les deux tâches HelloWorld2 et HelloWorld3 sont de priorités supérieures.

Cette illustration nous montre que des arcs d'inhibitions ont dû être connectés à la transition qui permet de passer de l'état READY_HelloWorld à l'état RUNNING_HelloWorld, mais aussi qu'un ensemble de places ont été clonées spécifiquement en fonction des tâches de priorités supérieures. Formellement, ce cas particulier nous amène à définir un opérateur adapté pour étendre le clonage d'un prototype comportemental de ressource concurrente traduit en RI_TPN avec rôles, dans un contexte d'ordonnancement coopératif.

Soit $\mathcal{N}_{clone} = \langle P_{root}, T_{root}, \ldots, \lambda_{root} \rangle$ un RI_TPN avec rôles qui a été préalablement cloné comme vu précédemment, et qui représente une instance de ressource concurrente. $\mathcal{N}^{cpsch}_{clone} = \langle P^{cpsch}_{root}, T^{cpsch}_{root}, \ldots, \lambda^{cpsch}_{root} \rangle$ représente l'extension apportée à $\mathcal{N}_{clone}$ selon un ensemble de n instances de ressources concurrentes concernées par ce mécanisme d'ordonnancement coopératif. Cet ensemble est décrit par $INS = \{ins_1, ins_2, \ldots, ins_n\}$ et représente des instances avec des priorités supérieures.

Définition 6 (Clonage spécifique de RI_TPN avec rôles) *Ordonnancement coopératif de ressources concurrentes : L'extension de $\mathcal{N}_{clone}$ en concurrence avec n instances dans un contexte d'ordonnancement coopératif est noté par :*

$$\mathcal{N}^{cpsch}_{clone} = CoopSched(\mathcal{N}_{clone}, INS)$$

avec $\forall t \in T_{root}$, $\exists \mathsf{Pre}(\lambda^{-1}(activatedState_{ref}), t) \in \mathsf{Pre}_{root}$ *et* $\exists \mathsf{Post}(\lambda^{-1}(resumedState_{ref}), t) \in \mathsf{Post}_{root}$

Formellement, cette définition donne :

- $R^{cpsch}_{ref} = R_{ref} \cup R_{INS}$ *avec* $R_{INS} = \bigcup_{\forall i \in [1,n]} \{r_{ins_i}\}$;
- $P^{cpsch}_{root} = P_{root} \cup P_{INS}$ *avec* $P_{INS} = \bigcup_{\forall i \in [1,n]} \{p_{ins_i}\}$;
- $T^{cpsch}_{root} = T_{root}$;
- $\lambda^{cpsch}_{root} : P^{cpsch}_{root} \rightarrow R^{cpsch}_{ref}$ *est définie par :*
  - $\forall p \in P^{cpsch}_{root} \setminus P_{INS}$, $\lambda^{cpsch}_{root}(p) = \lambda_{root}(p)$
  - $\forall p \in P_{INS}$ *et* $\forall i \in [1, n]$, $\lambda^{cpsch}_{root}(p) = r_{ins_i}$ *avec* $r_{ins_i} \in R_{INS}$
- $\mathsf{Pre}^{cpsch}_{root} : P^{cpsch}_{root} \times T^{cpsch}_{root} \rightarrow \mathbb{N}$ *est définie* $\forall p \in P^{cpsch}_{root}$ *et* $\forall t \in T^{cpsch}_{root}$ *par* $\mathsf{Pre}^{cpsch}_{root}(p, t) = \mathsf{Pre}_{root}(p, t)$;

– $\mathsf{Post}_{root}^{cpsch} : P_{root}^{cpsch} \times T_{root}^{cpsch} \rightarrow \mathbb{N}$ *est définie* $\forall p \in P_{root}^{cpsch}$ *et* $\forall t \in T_{root}^{cpsch}$ *par* $\mathsf{Post}_{root}^{cpsch}(p,t) = \mathsf{Post}_{root}(p,t)$;

– $m_{0_{root}}^{cpsch} : P_{root}^{cpsch} \rightarrow \mathbb{N}$ *est défini* $\forall p \in P_{root}^{cpsch}$ *par* $m_{0_{root}}^{cpsch}(p) = \begin{cases} m_{0_{root}}(p) \text{ si } p \in P_{root}^{cpsch} \setminus P_{INS} \\ m_{0_{INS}}(p) \text{ si } p \in P_{INS} \end{cases}$ *avec* $m_{0_{INS}}$ *défini par* $m_{0_{INS}} : P_{INS} \rightarrow \mathbb{N}$;

– $I_{s_{root}}^{cpsch} : T_{root}^{cpsch} \rightarrow \mathcal{I}$ *est défini* $\forall t \in T_{root}^{cpsch}$ *par* $I_{s_{root}}^{cpsch}(t) = I_{s_{root}}(t)$;

– $\mathsf{Read}_{root}^{cpsch} : P_{root}^{cpsch} \times T_{root}^{cpsch} \rightarrow \mathbb{N}$ *est définie* $\forall p \in P_{root}^{cpsch}$ *et* $\forall t \in T_{root}^{cpsch}$ *par* $\mathsf{Read}_{root}^{cpsch}(p,t) = \mathsf{Read}_{root}(p,t)$;

– $\mathsf{Inh}_{root}^{cpsch} : P_{root}^{cpsch} \times T_{root}^{cpsch} \rightarrow \mathbb{N}$ *est définie* $\forall p \in P_{root}^{cpsch}$ *et* $\forall t \in T_{root}^{cpsch}$ *par*

$$\mathsf{Inh}_{root}^{cpsch}(p,t) = \begin{cases} \mathsf{Inh}_{root}(p,t) \text{ si } p \in P_{root}^{cpsch} \setminus P_{INS} \\ 1 \text{ si } \begin{cases} p \in P_{INS} \\ t \in T_{root}^{cpsch}, \\ \exists \mathsf{Pre}(\lambda^{-1}(activatedState_{ref}), t) \in \mathsf{Pre}_{root} \\ \textit{et} \\ \exists \mathsf{Post}(\lambda^{-1}(resumedState_{ref}), t) \in \mathsf{Post}_{root} \end{cases} \end{cases}$$

Cette formalisation soulève un problème d'adaptabilité du processus de considération du comportement. La connexion des arcs d'inhibition sur la transition qui permet de rendre la ressource concurrente exécutable est traduite par la formalisation de la fonction d'inhibition $\mathsf{Inh}_{root}^{cpsch}$. À travers cette définition formelle, nous avons admis que tous les arcs d'inhibition clonés issus de ce mécanisme d'ordonnancement coopératif devaient être connectés sur cette même transition (i.e., celle entourée par les places annotées des rôles $activatedState_{ref}$ et $resumedState_{ref}$). Dans la réalité ce sera le cas, mais imaginons une toute autre définition comportementale de ce mécanisme dans laquelle d'autres arcs entreraient en jeu ; il nous serait alors impossible de localiser les arcs qui doivent être connectés. Des rôles n'ont en effet pas été identifiés pour ce genre de situation qui présente un cas particulier de représentation dépendant de l'expressivité des TPNs.

5.1.4 Synchronisation de TPNs basée sur les rôles

Afin de synchroniser les fragments clonés en RI_TPN, nous devons clarifier la définition d'une composition de RI_TPN qui sera basée sur les rôles annotés sur les places.

Soit un ensemble $\mathcal{N}_1, \ldots, \mathcal{N}_n$ de n RI_TPNs $\mathcal{N}_i = \langle P_i, T_i, \mathsf{Pre}_i, \mathsf{Post}_i, m_{0_i}, I_{s_i}, Read_i, Inh_i, R_i, \lambda_i \rangle$ avec rôles que nous cherchons à composer tel que $\forall k \neq k' \in [1,n] \implies T_k \cap T_{k'} = \emptyset$ et $P_k \cap P_{k'} = \emptyset$. Le résultat de la composition $\mathcal{N} = \langle P, T, \mathsf{Pre}, \mathsf{Post}, m_0, I_s, R, \lambda \rangle$ des précédents RI_TPNs avec rôles sera alors précisé par $\mathcal{N} = \mathcal{N}_1 || \mathcal{N}_2 || \ldots || \mathcal{N}_n$. Pour compléter cette composition, nous définissons une fonction de fusion de places dont les rôles assignés et impliqués dans la composition seront pris en compte en paramètres.

La fonction de fusion $\hookrightarrow$ est une fonction partielle de $(R_1 \cup \{\bullet\}) \times (R_2 \cup \{\bullet\}) \times \cdots \times (R_n \cup \{\bullet\}) \rightarrow P \times R$ dans laquelle $\bullet$ représente un symbole particulier visant à préciser qu'un RI_TPN n'est pas concerné par une fusion particulière. Nous étendons alors la définition de la fonction d'assignation inverse avec $\lambda^{-1}(\bullet) = \emptyset$.

La composition de n RI_TPNs avec m fusions est notée par

$$\Big(\mathcal{N}_1 || \ldots || \mathcal{N}_n\Big) \Big| \begin{array}{l} (r_1^1, \ldots, r_n^1) \hookrightarrow (p^1, r^1) \\ \ldots \\ (r_1^m, \ldots, r_n^m) \hookrightarrow (p^m, r^m) \end{array}$$

avec $\forall i \in [1,n], \forall j \in [1,m]$, $r_i^j \in R_i$, $r^j \in R$ et $p^j \in P$, et $\forall k \in [1,m], k \neq j \Rightarrow r_i^k \neq r_i^j$

Nous utiliserons par la suite les notations suivantes :

- Soit $P_i^{merged} \subseteq P_i$ l'ensemble des places du réseau $\mathcal{N}_i$ fusionnées par la composition. Formellement, $P_i^{merged} = \bigcup_{\forall j \in [1,m]} \{\lambda_i^{-1}(r_i^j)\}$
- Soit $P^{\hookrightarrow} \subseteq P$ l'ensemble des places du réseau $\mathcal{N}$ obtenues par la fusion. Formellement, $P^{\hookrightarrow} = \bigcup_{\forall j \in [1,m]} \{p^j\}$

Définition 7 (Composition de RI_TPN avec rôles) *La composition de n RI_TPNs $\mathcal{N}_i$ avec la fonction de fusion $\hookrightarrow$ notée :*

$$\mathcal{N} = \Big(\mathcal{N}_1 || \dots || \mathcal{N}_n\Big) \Big| \begin{array}{l} (r_1^1, \dots, r_n^1) \hookrightarrow (p^1, r^1) \\ \dots \\ (r_1^m, \dots, r_n^m) \hookrightarrow (p^m, r^m) \end{array}$$

est définie par :

- $R = \left(\bigcup_{\forall i \in [1,n]} \left(R_i \setminus \bigcup_{\forall j \in [1,m]} \{r_i^j\}\right)\right) \cup \left(\bigcup_{\forall j \in [1,m]} \left\{r^j\right\}\right)$;
- $P = \left(\bigcup_{\forall i \in [1,n]} P_i \setminus P_i^{merged} \right) \cup P^{\hookrightarrow}$;
- $T = \bigcup_{\forall i \in [1,n]} T_i$;
- $\lambda : P \to R$ *est définie par :*
  - $\forall p \in P \setminus P^{\hookrightarrow}$ *signifiant que* $\exists i$ *tel que* $p \in P_i$ *alors* $\lambda(p) = \lambda_i(p)$
  - $\forall p^j \in P^{\hookrightarrow}$*, signifiant que* p *est le résultat d'une fusion,* $\lambda(p^j) = r^j$
- $\mathsf{Pre} : P \times T \to \mathbb{N}$ *est définie* $\forall p \in P$ *et* $\forall t \in T_i \subseteq T$ *par* $\mathsf{Pre}(p,t) =$
$$\begin{cases} \mathsf{Pre}_i(p,t) \text{ si } p \in P \setminus P^{\hookrightarrow} \text{ et } p \in P_i \\ \mathsf{Pre}_i(p',t), \text{ si } \begin{cases} p \in P^{\hookrightarrow} \text{ et } p' \in P_i \\ (\dots, r_i^k, \dots) \hookrightarrow (p, \lambda(p)) \\ \lambda_i(p') = r_i^k \end{cases} \\ 0 \text{ sinon.} \end{cases}$$
- $\mathsf{Post} : P \times T \to \mathbb{N}$ *est définie* $\forall p \in P$ *et* $\forall t \in T_i \subseteq T$ *par* $\mathsf{Post}(p,t) =$
$$\begin{cases} \mathsf{Post}_i(p,t) \text{ si } p \in P \setminus P^{\hookrightarrow} \text{ et } p \in P_i \\ \mathsf{Post}_i(p',t), \text{ si } \begin{cases} p \in P^{\hookrightarrow} \text{ et } p' \in P_i \\ (\dots, r_i^k, \dots) \hookrightarrow (p, \lambda(p)) \\ \lambda_i(p') = r_i^k \end{cases} \\ 0 \text{ sinon.} \end{cases}$$
- $m_0 : P \to \mathbb{N}$ *est définie* $\forall p \in P$ *par :* $m_0(p) =$
$$\begin{cases} m_{0i}(p) \text{ si } p \in P \setminus P^{\hookrightarrow} \text{ et } p \in P_i \\ \sum_{i=1}^{n} m_{0i}\big(\lambda^{-1}(r_i^k)\big) \text{ si } \begin{cases} p \in P^{\hookrightarrow} \\ (r_1^k, \dots, r_n^k) \hookrightarrow (p, \lambda(p)) \end{cases} \end{cases}$$
- $I_s : T \to \mathcal{I}$ *est défini* $\forall t \in T$ *par :* $I_s(t) = I_{s_i}(t)$ **if** $t \in T_i$;
- $\mathsf{Read} : P \times T \to \mathbb{N}$ *est définie* $\forall p \in P$ *et* $\forall t \in T_i \subseteq T$ *comme* $\mathsf{Pre}(p,t)$;
- $\mathsf{Inh} : P \times T \to \mathbb{N}$ *est définie* $\forall p \in P$ *et* $\forall t \in T_i \subseteq T$ *comme* $\mathsf{Pre}(p,t)$

À titre d'exemple, soit $\mathcal{N} = \Big(\mathcal{N}_1 || \mathcal{N}_2 || \mathcal{N}_3\Big) \Big| \ (r_1, r_2, \bullet) \hookrightarrow (p, r)$ la composition parallèle de 3 TPNs, i.e., $\mathcal{N}_1$, $\mathcal{N}_2$ et $\mathcal{N}_3$, où la place $p_1 \in P_1$ tel que $\lambda_1(p_1) = r_1$ et la place

$p_2 \in P_2$ tel que $\lambda_2(p_2) = r_2$ sont fusionnées. Le nom de la place obtenue par la fusion dans $\mathcal{N}$ est $p \in P$ et son rôle est $\lambda(p) = r \in R$.

Propriété 1 (Associativité) *La composition de RI_TPNs avec rôles est associative suivant :*

$$\left(\mathcal{N}_1||\mathcal{N}_2||\mathcal{N}_3\right)\Big|_{(r_1,r_2,r_3)\hookrightarrow(p,r)} =$$

$$\left(\left(\mathcal{N}_1||\mathcal{N}_2\right)\Big|_{(r_1,r_2)\hookrightarrow(p_{12},r_{12})}||\mathcal{N}_3\right)\Big|_{(r_{12},r_3)\hookrightarrow(p,r)} =$$

$$\left(\mathcal{N}_1||\left(\mathcal{N}_2||\mathcal{N}_3\right)\Big|_{(r_2,r_3)\hookrightarrow(p_{23},r_{23})}\right)\Big|_{(r_1,r_{23})\hookrightarrow(p,r)}$$

Propriété 2 (Commutativité) *La composition de RI_TPNs avec rôles est aussi commutative :*

$$\left(\mathcal{N}_1||\mathcal{N}_2\right)\Big|_{\substack{(r_1^1,r_2^1)\hookrightarrow(p^1,r^1)\\ \dots\\ (r_1^k,r_2^k)\hookrightarrow(p^k,r^k)}} = \left(\mathcal{N}_2||\mathcal{N}_1\right)\Big|_{\substack{(r_2^1,r_1^1)\hookrightarrow(p^1,r^1)\\ \dots\\ (r_2^k,r_1^k)\hookrightarrow(p^k,r^k)}}$$

5.2 Composition appliquée au comportement de plate-forme

La composition de RI_TPNs avec rôles ayant été formalisée, cette section expose de manière générique la construction d'un modèle comportemental d'application déployée. Afin de couvrir l'ensemble des règles de composition décrites dans le chapitre précédent, cette construction s'appuie sur une illustration mettant en application deux ressources concurrentes, périodiques et de priorités différentes, pointant chacune sur une routine d'exécution. Les ressources interagissent entre elles par le bais d'une ressource d'exclusion mutuelle. Le mécanisme d'ordonnancement retenu pour cette application est coopératif. Les ressources concurrentes ne sont donc pas préemptables.

Nous avons dû faire un choix de plate-forme pour l'exécution de l'application. Comme déjà abordé, l'acquisition d'une ressource d'exclusion mutuelle non disponible nécessite de réordonnancer les ressources concurrentes. Ce mécanisme n'ayant pas été présenté, nous avons décidé de nous en abstraire. La plate-forme OSEK/VDX qui nous a servi de support de présentation tout au long de ce mémoire n'utilise pas ce mécanisme. Toutefois, le partage de ressource d'exclusion mutuelle avec cette norme implique le mécanisme de priorité plafond PCP pour éviter les inversions de priorités et les *deadlocks* [66]. Ce mécanisme est plus compliqué à mettre en œuvre et exige une formalisation particulière puisque la gestion des priorités est dynamique. En conséquence, nous avons choisi une plate-forme quelconque qui, sur le principe se comporte comme une pseudo norme OSEK/VDX. Elle ne réordonnance pas les ressources concurrentes lorsque la ressource d'exclusion mutuelle n'est pas disponible.

La construction reprend les fragments comportementaux clonés en RI_TPN pour les assembler. À chaque étape de composition, les fragments concernés sont synchronisés avec l'opérateur de composition précédemment défini. Pour chaque opération, les rôles assignés sur les places fusionnables sont partiellement mis en correspondance en paramètres des

fonctions de fusion. Pour des raisons de clarté et conformément aux règles de composition, la construction est scindée ici en quatre étapes de composition, à partir des équations (5.1) à (5.4). La première équation (5.1) décrit la composition d'une routine d'exécution ($\mathcal{N}_R$). L'équation (5.2) suivante décrit la composition entre une ressource concurrente et son point d'entrée ($\mathcal{N}_{EP}$). Puis, l'équation (5.3) décrit la composition de l'ensemble des ressources concurrentes ($\mathcal{N}_{CR}$). Enfin, l'équation (5.4) finalise la composition en intégrant les ressources d'interactions ($\mathcal{N}_{IR}$).

La figure 5.1 illustre la construction au regard de l'application qui a été introduite. Sur cette illustration, les fragments comportementaux en RI_TPN préalablement clonés et prêts pour la composition sont représentés dans des encadrés en pointillés. Les ressources concurrentes sont représentées par deux tâches ordonnançables : Task1 et Task2, Task2 étant la plus prioritaire. Ces tâches sont respectivement cadencées par deux alarmes : Alarm1 et Alarm2 pour assurer leurs périodicités. Elles interagissent au moyen d'une ressource d'exclusion mutuelle représentée par Mutex. Chaque tâche pointe sur une routine d'exécution dans laquelle trois services sont appelés dans l'ordre suivant :

1. une acquisition de Mutex : get$_\phi$(Mutex) avec $\phi \in [1, 2]$
2. un relâchement de Mutex : release$_\phi$(Mutex)
3. une terminaison de la tâche exécutée : terminate$_\phi$(Taskϕ)

Enfin, comme nous nous trouvons dans un contexte d'application monoprocesseur, le processeur Proc apparaît aussi dans ce modèle, comme une ressource d'interaction. Il permet ainsi de mettre en concurrence les tâches avant leurs mises en exécution.

Les places fusionnables sont représentées en double cercle et celles prêtes à être fusionnées sont reliées par des arcs crochets (pour symboliser la fonction de fusion) en pointillés, en gras et repérés d'une lettre correspondant à l'étape de composition. Les rôles restent indiqués en haut à droite des places.

5.2.1 Composition des routines d'exécution

Le principe de composition qui suit est applicable à l'ensemble des routines d'exécution et généralise formellement la composition des services appelés. Il répond au besoin soulevé dans l'algorithme 4.16 du chapitre précédent qui consistait à mettre en correspondance les rôles de composition utiles à la localisation des places fusionnables.

Soit une séquence de n services appelés depuis une même routine d'exécution : $\{\mathcal{N}_{S1}, \mathcal{N}_{S2}, \ldots, \mathcal{N}_{Sn}\}$ tels que $\forall i \in [1, n], \mathcal{N}_{S_i} = Clone(\mathcal{N}_S, S_i, ref\_S_i)$ avec $\mathcal{N}_S$ l'instance du prototype comportemental en RI_TPN du service à cloner, S_i le suffixe de nommage des places et des transitions du RI_TPN cloné correspondant au service appelé et ref_S_i le suffixe de nommage des rôles du RI_TPN cloné correspondant à l'instance de ressource référencée par l'appel du service. La construction d'une routine implique alors $n - 1$ compositions, chacune pouvant avoir m_j fusions de places avec $j \in [1, n - 1]$. Le RI_TPN avec rôles résultant $\mathcal{N}_R$ est donné par l'équation (5.1) en précisant les rôles concernés à chaque opération de composition.

Illustration 1 (Voir figure 5.1) $\forall \phi \in [1, 2]$, *en appliquant l'équation 5.1 aux ensembles* $\{\mathcal{N}_{get_\phi(Mutex)}, \mathcal{N}_{release_\phi(Mutex)}, \mathcal{N}_{terminate_\phi(Task\phi)}\}$*, nous obtenons les RI_TPNs* $\mathcal{N}_{Task\phi Body}$ *qui représentent les routines d'exécution destinées aux tâches Task1 et Task2. Les places qui sont fusionnées lors des deux opérations de composition sont reliées par les arcs crochets repérés par l'indice (a).*

$$\mathcal{N}_R = \left(\left(\left(\mathcal{N}_{S_1}||\mathcal{N}_{S_2}\right)\middle|\begin{array}{l}(end_{ref\_S_1}, start_{ref\_S_2}) \hookrightarrow (S_{S_1\to S_2}, \perp)\\ (r^2_{S_1}, r^2_{S_2}) \hookrightarrow (p^2_{S_2}, r^2_{S_2})\\ \dots\\ (r^{m_1}_{S_1}, r^{m_1}_{S_2}) \hookrightarrow (p^{m_1}_{S_2}, r^{m_1}_{S_2})\end{array}||\mathcal{N}_{S_3}\right)\middle|\begin{array}{l}(end_{ref\_S_2}, start_{ref\_S_3}) \hookrightarrow (S_{S_1S_2\to S_3}, \perp)\\ (r^2_{S_1S_2}, r^2_{S_3}) \hookrightarrow (p^2_{S_3}, r^2_{S_3})\\ \dots\\ (r^{m_2}_{S_1S_2}, r^{m_2}_{S_3}) \hookrightarrow (p^{m_2}_{S_3}, r^{m_2}_{S_3})\end{array}$$

$$\dots||\mathcal{N}_{S_n}\Big)\Bigg|\begin{array}{l}(end_{ref\_S_{n-1}}, start_{ref\_S_n}) \hookrightarrow (S_{S_1S_2\dots S_{n-1}\to S_n}, \perp)\\ (r^2_{S_1S_2\dots S_{n-1}}, r^2_{S_n}) \hookrightarrow (p^2_{S_n}, r^2_{S_n})\\ \dots\\ (r^{m_{n-1}}_{S_1S_2\dots S_{n-1}}, r^{m_{n-1}}_{S_n}) \hookrightarrow (p^{m_{n-1}}_{S_n}, r^{m_{n-1}}_{S_n})\end{array} \quad (5.1)$$

avec $\forall k \in [1, m_j]$ et $n \geq 2$ si $k \geq 2$ alors $r^k_{S_1\dots S_j} = r^k_{S_{j+1}}$

$$\mathcal{N}_{EP} = \Big(\mathcal{N}_{C_\tau}||\mathcal{N}_R\Big)\Bigg|\begin{array}{l}(start_{C_\tau}, start_{ref\_S_1}) \hookrightarrow (S, \perp)\\ (end_{C_\tau}, end_{ref\_S_n}) \hookrightarrow (E, \perp)\\ (r^3_{C_\tau}, r^3_R) \hookrightarrow (p^3_{C_\tau}, r^3_R)\\ \dots\\ (r^m_{C_\tau}, r^m_R) \hookrightarrow (p^m_{C_\tau}, r^m_R)\end{array} \quad (5.2)$$

avec $\forall k \in [1, m]$ si $k \geq 3$ alors $r^k_{C_\tau} = r^k_R$

$$\mathcal{N}_{CR} = \left(\Big(\mathcal{N}_{EP_1}||\mathcal{N}_{EP_2}\Big)\middle|\begin{array}{l}(processor_{EP_1}, processor_{EP_2}) \hookrightarrow (P_{EP_1\to EP_2}, processor_{Proc})\\ (r^2_{EP_1}, r^2_{EP_2}) \hookrightarrow (p^2_{EP_2}, r^2_{EP_2})\\ \dots\\ (r^{m_1}_{EP_1}, r^{m_1}_{EP_2}) \hookrightarrow (p^{m_1}_{EP_2}, r^{m_1}_{EP_2})\end{array}\right.$$

$$\left.\dots||\mathcal{N}_{EP_{q_C}}\right)\Bigg|\begin{array}{l}(processor_{Proc}, processor_{EP_{q_C}}) \hookrightarrow (P_{EP_1\dots EP_{q_C-1}\to EP_{q_C}}, processor_{Proc})\\ (r^2_{EP_1\dots EP_{q_C-1}}, r^2_{EP_{q_C}}) \hookrightarrow (p^2_{EP_{q_C}}, r^2_{EP_{q_C}})\\ \dots\\ (r^{m_{q_C-1}}_{EP_1\dots EP_{q_C-1}}, r^{m_{q_C-1}}_{EP_{q_C}}) \hookrightarrow (p^{m_{q_C-1}}_{EP_{q_C}}, r^{m_{q_C-1}}_{EP_{q_C}})\end{array} \quad (5.3)$$

avec $\forall k_C \in [1, m_{j_C}]$ et $q_C \geq 2$ si $k_C \geq 2$ alors $r^{k_C}_{EP_1\dots EP_{j_C}} = r^{k_C}_{EP_{j_C+1}}$

$$\mathcal{N}_{IR} = \left(\Big(\mathcal{N}_{CR}||\mathcal{N}_{I_1}\Big)\middle|\begin{array}{l}(r^1_P, r^1_{I_1}) \hookrightarrow (p^1_{I_1}, r^1_{I_1})\\ \dots\\ (r^{m_1}_P, r^{m_1}_{I_1}) \hookrightarrow (p^{m_1}_{I_1}, r^{m_1}_{I_1})\end{array}\right.\quad \left.\dots||\mathcal{N}_{I_{q_I}}\right)\Bigg|\begin{array}{l}(r^1_{PI_1\dots I_{q_I-1}}, r^1_{I_{q_I}}) \hookrightarrow (p^1_{I_{q_I}}, r^1_{I_{q_I}})\\ \dots\\ (r^{m_{q_I}}_{PI_1\dots I_{q_I-1}}, r^{m_{q_I}}_{I_{q_I}}) \hookrightarrow (p^{m_{q_I}}_{I_{q_I}}, r^{m_{q_I}}_{I_{q_I}})\end{array} \quad (5.4)$$

avec $\forall k_I \in [1, m_{j_I}]$ et $q_I \geq 1$, $r^{k_I}_{PI_{j_I-1}} = r^{k_I}_{I_{j_I}}$

5.2.2 Composition des points d'entrée

Le principe de composition suivant s'applique à l'ensemble des ressources concurrentes pointant sur une routine d'exécution. Il établit cette fois les rôles qui doivent être mis en correspondance entre les instances de ressources et leurs points d'entrée (voir algorithme 4.19 du chapitre précédent).

Soit une routine $\mathcal{N}_R$ préalablement composée et pointée par une ressource concurrente décrite par : $\mathcal{N}_{C\tau} = Clone(\mathcal{N}_C, C_\tau, C_\tau)$ avec $\mathcal{N}_C$ l'instance du prototype comportemental en RI_TPN de la ressource concurrente à cloner et C_τ le suffixe de nommage des places, des transitions et des rôles du RI_TPN cloné correspondant à l'instance de la ressource concurrente. La construction d'une ressource concurrente avec son point d'entrée implique alors 1 seule composition pouvant avoir m fusions de places. Le RI_TPN avec rôles résultant $\mathcal{N}_{EP}$ est donné par l'équation (5.2) en précisant les rôles concernés à chaque opération de composition. Nous admettons ici que les éventuelles extensions de $\mathcal{N}_{C\tau}$ dues au clonage de mécanismes spécifiques aient déjà été appliquées dans cette équation.

Illustration 2 (Voir figure 5.1) $\forall\phi \in [1,2]$, *en appliquant l'équation 5.2 aux ensembles* $\{\mathcal{N}_{Task\phi}, \mathcal{N}_{Task\phi Body}\}$*, nous obtenons les RI_TPNs* $\mathcal{N}_{Task\phi\_withBody}$ *qui représentent les tâches* Task1 *et* Task2 *composées de leurs corps exécutifs. Les places qui sont fusionnées lors des deux opérations de composition sont reliées par les arcs crochets repérés par l'indice (b). Avant de composer, du fait de sa priorité inférieure,* $\mathcal{N}_{Task1}$ *a été étendue en appliquant l'opérateur de clonage spécifique à l'ordonnancement coopératif tel que* $CoopSched(\mathcal{N}_{Task1}, \{Task2\})$.

5.2.3 Composition des ressources concurrentes

Les ressources concurrentes étant composées de leurs routines d'exécution, elles peuvent maintenant être composées entre elles. La principale raison évoquée pour cette étape est la mise en concurrence des ressources vis-à-vis du processeur. Mais elle permet aussi de composer les ressources en fonction des mécanismes spécifiques (e.g., Task1 inhibée par Task2 pour l'ordonnancement coopératif) et des associations issues de prototypes de conception (i.e., Task1 cadencée périodiquement par Alarm1).

Soit un ensemble constitué de q_C ressources concurrentes préalablement composées avec leurs corps exécutifs et décrites $\forall i_C \in [1, q_C]$ par : $\mathcal{N}_{EP_{i_C}}$. La construction de l'ensemble des ressources concurrentes implique alors $q_C - 1$ compositions, chacune pouvant avoir m_{j_C} fusions de places avec $j_C \in [1, q_C - 1]$. Le RI_TPN avec rôles résultant $\mathcal{N}_{CR}$ est donné par l'équation (5.3) en précisant les rôles concernés à chaque opération de composition.

Illustration 3 (Voir figure 5.1) $\forall\phi \in [1,2]$, *en appliquant l'équation 5.3 à l'ensemble* $\{\mathcal{N}_{Task\phi\_withBody}, \mathcal{N}_{Alarm\phi}\}$*, nous obtenons le RI_TPN* $\mathcal{N}_{DeployedApplication}$ *qui représente l'application déployée avec l'ensemble des ressources concurrentes sans les ressources d'interactions. Les places qui sont fusionnées lors de cette opération de composition sont reliées par les arcs crochets repérés par l'indice (c).*

5.2.4 Composition des ressources d'interactions

Les ressources d'interactions représentent la dernière étape de composition. Dans une composition en RI_TPN, les ressources d'interactions viennent finalement compléter la composition des ressources concurrentes. Le travail avait déjà été amorcé dans la composition des routines d'exécution puisque les ressources d'interaction concernées par l'application sont référencées par les appels de service.

Soit un ensemble constitué de q_I ressources d'interactions décrites $\forall i_I \in [1, q_I]$ par : $\mathcal{N}_{I_{i_I}} = Clone(\mathcal{N}_I, I_{i_I}, I_{i_I})$ avec $\mathcal{N}_I$ l'instance du prototype comportemental en RI_TPN de la ressource d'interaction à cloner, I_{i_I} le suffixe de nommage des places, des transitions et des rôles du RI_TPN cloné correspondant aux instances de ressource d'interactions. La construction de l'ensemble des ressources d'interactions avec l'ensemble des ressources concurrentes $\mathcal{N}_{CR}$ préalablement composé implique alors q_I compositions, chacune pouvant avoir m_{j_I} fusions de places avec $j_I \in [1, q_I]$. Le RI_TPN avec rôles résultant $\mathcal{N}_{IR}$ est donné par l'équation (5.4) en précisant les rôles concernés à chaque opération de composition.

Illustration 4 (Voir figure 5.1) *En appliquant l'équation 5.4 à l'ensemble* $\{\mathcal{N}_{DeployedApplication}, \mathcal{N}_{Mutex}, \mathcal{N}_{Proc}\}$*, nous obtenons le RI_TPN* $\mathcal{N}_{DeployedApplication}$ *qui représente l'application globale déployée et complétée avec les ressources d'interactions. Les places qui sont fusionnées lors de cette opération de composition sont reliées par les arcs crochets repérés par l'indice (d).*

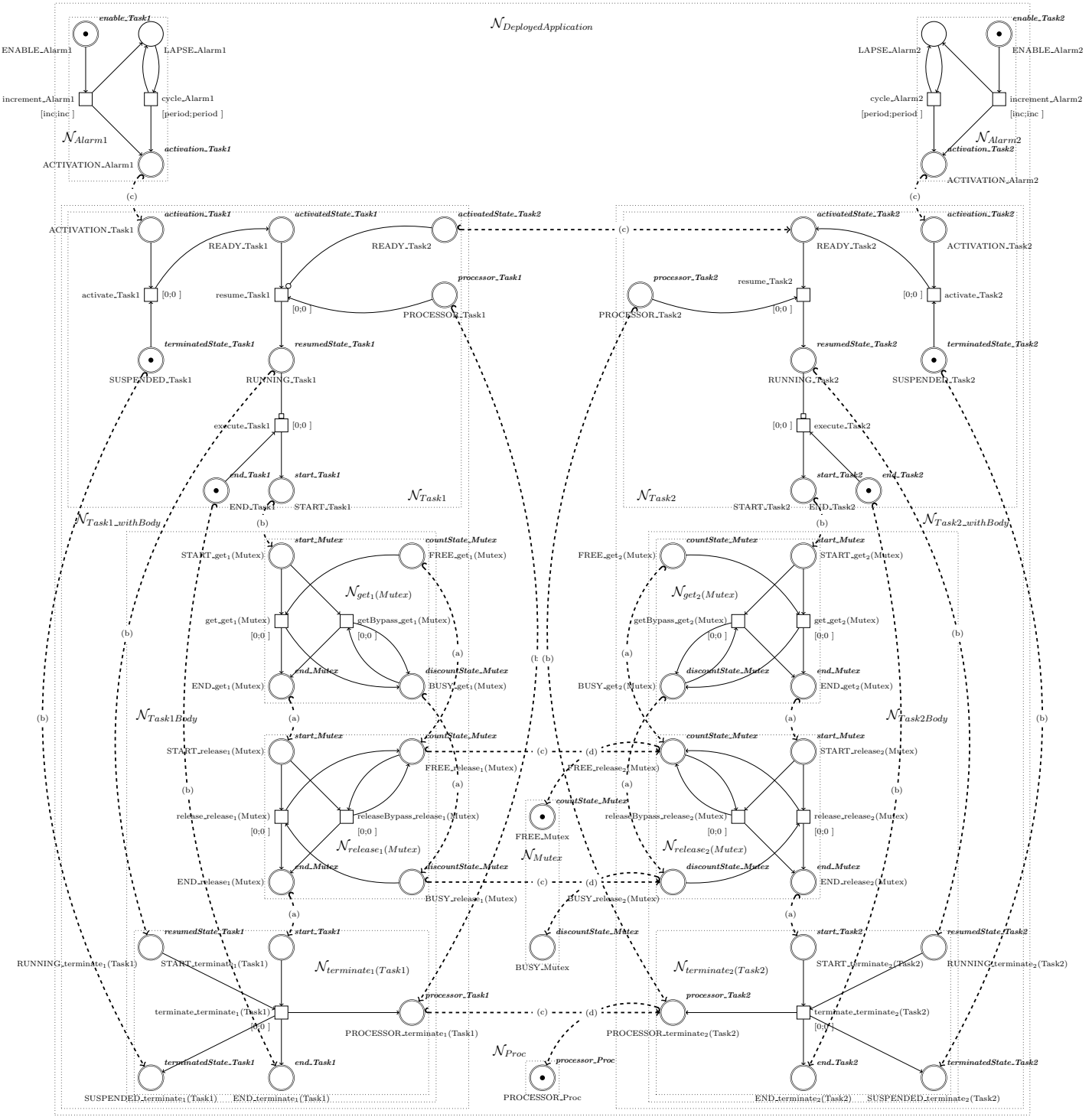


FIGURE 5.1 – Modèle d'application déployée avec partage de ressource d'exclusion mutuelle composé en RI_TPN

5.3 Synthèse

La formalisation qui vient d'être présentée avait deux objectifs : 1) s'assurer du bon enchaînement des règles du processus de considération du comportement et 2) compléter les règles de composition qui avaient révélé un besoin de s'adapter au langage de traduction du comportement. De manière à décrire formellement la composition en RI_TPN, les opéra-

teurs de clonage et de composition ont été définis dans la première section. Puis, dans la seconde section, ces opérateurs ont été utilisés pour formaliser la composition de modèles de SETRS (i.e., application avec le comportement de la plate-forme visée pour le déploiement) en RI_TPN.

La formalisation a démontré que les règles étaient en adéquation avec la composition en RI_TPN. L'enchaînement a été respecté et les informations nécessaires à l'adaptation des règles avec les RI_TPN ont pu être identifiées de manière générique. Les rôles servent en effet à mettre en correspondance et à localiser les places à fusionner. Les fonctions de fusion qui apparaissent sur chaque opération de composition renseignent ces informations.

La définition formelle de l'opérateur de composition a aussi permis de fixer la caractérisation des places fusionnées vis-à-vis des relations d'arcs entrants et d'arcs sortants. Elle vient aussi préciser le nouveau marquage qui résulte de la somme des marquages source et cible. Cette définition de nouveau marquage oblige cependant à bien définir les marquages initiaux des RI_TPNs à assembler. Le marquage doit être défini à un seul endroit pour éviter toute accumulation de jetons au fil de la composition. Nous avons par conséquent adopté l'idée de ne marquer que les prototypes comportementaux des ressources, plutôt que ceux des services. À travers l'application déployée, ces derniers peuvent effectivement être appelés conjointement pour une même instance de ressource référencée. À l'instar du mutex dans notre illustration, le marquage est porté sur le RI_TPN $\mathcal{N}_{Mutex}$ et non sur les RI_TPNs $\mathcal{N}_{get_\phi(Mutex)}$ et $\mathcal{N}_{release_\phi(Mutex)}$ qui décrivent les services appelés pour acquérir et relâcher le mutex.

Enfin un dernier point est à souligner. Comme abordé dans ce chapitre, le niveau d'expressivité des TPNs est si exhaustif qu'il peut limiter le clonage générique des mécanismes spécifiques. À l'image de l'ordonnancement coopératif, l'opérateur qui a été défini dépend fortement de la représentation du RI_TPN à cloner spécifiquement. L'assignation de rôles étant réservée aux places, la connexion d'éléments autres que les places peut se révéler ambigüe.

6

Validation expérimentale

Sommaire

Ce chapitre montre un exemple de notre processus de génération de modèles formels en TPN. Il traite un cas d'étude théorique visant deux noyaux temps réel : OSEK/VDX et VXWORKS.

Ce chapitre est composé de quatre sections. Une implémentation de notre prototype est tout d'abord présentée dans la première section. Une autre modélisation en TPN d'un autre noyau temps réel est ensuite proposée. Un exemple d'application est alors déployé pour générer deux modèles comportementaux visant deux plates-formes (OSEK/VDX et VXWORKS). Un bilan de cette mise en application est finalement donné dans la dernière section.

6.1 Prototype

La stratégie de considération du comportement des plates-formes que nous avons adoptée (cf. Adjonction au processus de déploiement, page 65) aboutit à la composition de modèles d'application déployée avec la représentation explicite de la plate-forme considérée. Une première implémentation de ce processus a été développée dans une transformation. Cette transformation est présentée en premier lieu. Toutefois, le modèle généré avec cette transformation doit être traduit dans un format exploitable par un outil de vérification. Un exemple d'extraction est décrit dans un deuxième temps.

6.1.1 Implémentation

Nous avons choisi le langage de transformation Kermeta [91] [27] car il semblait correspondre à nos besoins d'implémentation. Kermeta est adapté à la métamodélisation et plus particulièrement à l'exécutabilité des métamodèles. C'est un projet *open source* directement inspiré du MOF [64] (*Meta Object Facility*) qui est un standard de l'OMG pour décrire structurellement des métamodèles dans un contexte MDA. Il peut être vu comme un métamétamodèle auquel est ajouté un langage permettant la définition d'actions sur les métamodèles, pour offrir cette exécutabilité.

À l'image d'Ecore du projet EMF [29] (*Eclipse Modeling Framework*), Kermeta est utilisé comme un langage orienté objet. Élaborer une transformation de modèles avec Kermeta revient finalement à développer un métaprogramme objet qui vient lire ou écrire dans les métaclasses des métamodèles impliqués. L'exécutabilité repose sur le tissage d'aspects qui viennent enrichir les métaclasses sans en modifier le contenu. Cette idée nous intéressait particulièrement pour suivre une logique impérative dans l'exécution de nos règles de transformation à travers les métaclasses.

Néanmoins, malgré l'indépendance de notre processus de génération vis-à-vis du langage de transformation, certaines propriétés d'implémentation devaient être respectées. Kermeta répond à ces contraintes, puisqu'il permet :

- d'établir des règles impératives (notamment pour la composition) ;
- d'appeler des règles récursivement (précisément pour le clonage des éléments composites) ;
- de contraindre l'exécution de règles (pour restreindre leur applicabilité à certains éléments) ;
- de facilement adapter certains traitements (comme la localisation ou la traduction d'informations à travers divers modèles) sur les règles ;

L'ensemble des règles qui ont été décrites dans le chapitre 4 ont été implémentées en Kermeta. La transformation prend par conséquent en entrée RTEPML qui a été étendu au comportement. Grâce à l'approche par aspect, les métaclasses de RTEPML ont été enrichies d'opérations permettant l'exécution de nos règles. Le *listing* 6.1 donne un exemple d'implémentation en Kermeta d'une des règles du processus de considération du comportement. Il reprend la règle *ruleCreateAtomicElementFromBehavioralPrototype* décrite à travers les algorithmes 4.3 à 4.5. Dans cet exemple, la métaclasse *Atomic* de RTEPML, qui représente le concept source de la règle, a bien été "aspectée" et une opération a été ajoutée pour exécuter la caractérisation d'un élément atomique cloné.

```
1  aspect class Atomic
2  {
3       /**
4        * @operation : createAtomicElementFromBehavioralPrototype
5        * @param     : ari The instance described by the behavioral prototype
```

```
6         *        owning atomic element to clone
7         */
8       operation createAtomicElementFromBehavioralPrototype(
9            ari : ApplicationResourceInstance): Void is do
10           var abe : Atomic init self
11           // Correspondance with the cloned atomic element
12           self.rootInstances.add(ari)
13           self.name := abe.name + "_" + ari.name
14           ari.type.behavior.getPropertyRoles.each{ r |
15                if abe = ari.type.behavior.getAtomicElement(r) then
16                     ari.properties.each{ p |
17                          if p.~property = ari.type.getProperty(r) then
18                               self.val = p.getPropertyValue(r)
19                          end
20                     }
21                end
22           }
23           ari.type.behavior.getCompositionRoles.each{ r |
24                if abe = ari.type.behavior.getAtomicElement(r) then
25                     self.role = r.name + "_" + ari.name
26                end
27           }
28      end
29 }
```

Listing 6.1 – Règle de création d'élément atomique à partir d'un prototype comportemental

La version proposée n'est qu'en phase de prototypage. Toutefois, un ensemble d'exemples servant de tests unitaires ont pu être mis en place, ainsi que des cas d'étude [49] [50] ; l'un d'entre est ensuite présenté dans ce chapitre. Il restera à développer des suites de tests plus exhaustives pour augmenter la confiance dans ce prototype. Néanmoins, ce dernier nous permet une première validation expérimentale de nos règles de transformation, ainsi que du processus.

6.1.2 Extraction

La mise en application d'activités de vérification exige de traduire les modèles comportementaux générés au format imposé par l'outil de vérification. Cette traduction est de ce fait spécifique. Elle reste cependant nécessaire à l'intégrité du processus de génération de modèles formels (cf. Adjonction au processus de déploiement, page 65). La figure 6.1 reprend ce processus qui figurait sur la figure 4.2 dans laquelle la stratégie de considération du comportement avait été exposée. Comme annoncé, l'implémentation que nous proposons pour ce processus est réalisée à l'aide de l'outil Kermeta.

Dans cette configuration, le PSM 2 est finalement interprété comme un modèle pivot à partir duquel n'importe quelle traduction pourrait être appliquée. Dans notre cas, nous avons choisi d'implémenter un traducteur formel dédié à transcrire le modèle TPN d'application déployée en une description textuelle exploitable par l'outil de vérification Roméo [53]. Nous avons donc élaboré une transformation RTEPML2Romeo qui transcrit les éléments de comportement du modèle source (objets TPN) en un arbre textuel au format XML. La mise en œuvre de cette transformation en Kermeta s'appuie sur l'approche par aspects pour extraire de ces éléments, via les métaclasses TPN de RTEPML, les données nécessaires au balisage du fichier XML.

6.2 Modélisation d'une autre plate-forme

Afin de mettre en évidence la réutilisabilité et la portabilité de notre approche, nous avons utilisé notre processus pour générer un modèle formel d'application sur une autre

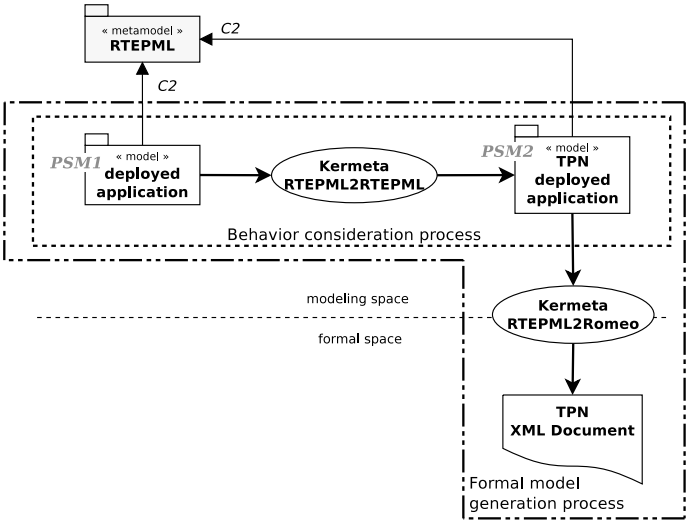


FIGURE 6.1 – Processus de génération de modèles formels en RI_TPN vers Roméo

plate-forme. Cette plate-forme est le noyau VXWORKS [93] qui est aussi beaucoup utilisé dans le secteur industriel. Nous avons choisi cette plate-forme dans l'optique du cas d'étude qui va être illustré dans la section suivante. Nous aurons en effet besoin de représenter une tâche périodique. Similairement à OSEK/VDX, le noyau VXWORKS ne contient pas intrinsèquement de ressource dédiée à la périodicité d'une tâche. Nous devons donc modéliser ce concept par l'intermédiaire d'un prototype de conception (cf. Description des prototypes de conception, page 23).

Une tâche VXWORKS périodique peut être représentée en associant une tâche, avec un *watchdog* et avec un sémaphore binaire. La tâche est dans ce cas cadencée par le *watchdog* qui libère cycliquement le sémaphore. Lorsque le sémaphore est libéré, la tâche qui est en attente acquiert le sémaphore et réentre dans sa routine d'exécution. La figure 6.2 reprend le cheminement intégral pour générer un modèle formel en TPN d'une application déployée sur une plate-forme VXWORKS, traduit pour Roméo. Sur cette figure, le PDM de cette plate-forme qui a été représenté explicitement avec RTEPML est considéré en paramètre du processus de déploiement. Un zoom sur ce PDM est fourni pour distinguer les ressources impliquées dans le prototype de conception d'une tâche périodique (PeriodicTask), ainsi que les prototypes comportementaux qui ont été attribués.

Les prototypes comportementaux représentés sont les suivants :

- TaskBehavior : le comportement de la tâche dans lequel le mécanisme d'ordonnancement coopératif est décrit ;
- SemaphoreBehavior : le comportement du sémaphore qui contient aussi des prototypes comportementaux de services pour son acquisition (SemTakeBehavior) et son relâchement (SemGiveBehavior) ;
- WatchdogBehavior : le comportement du *watchdog* qui contient aussi un prototype comportemental de service pour son activation (WdStartBehavior) ;

À travers ce PDM, les rôles utiles à la localisation des éléments lors du déploiement et lors de la composition des fragments de comportement en RI_TPN clonés ont aussi été assignés. Une remarque est à préciser au sujet des services d'acquisition et de relâchement du sémaphore. Les prototypes comportementaux de ces services intègrent respectivement les mécanismes de mise en attente et d'éligibilité de la tâche, puisque le concept de tâche est la seule ressource concernée par ces mécanismes. Notons aussi que le marquage initial des prototypes a seulement été effectué sur les ressources et non sur les services comme

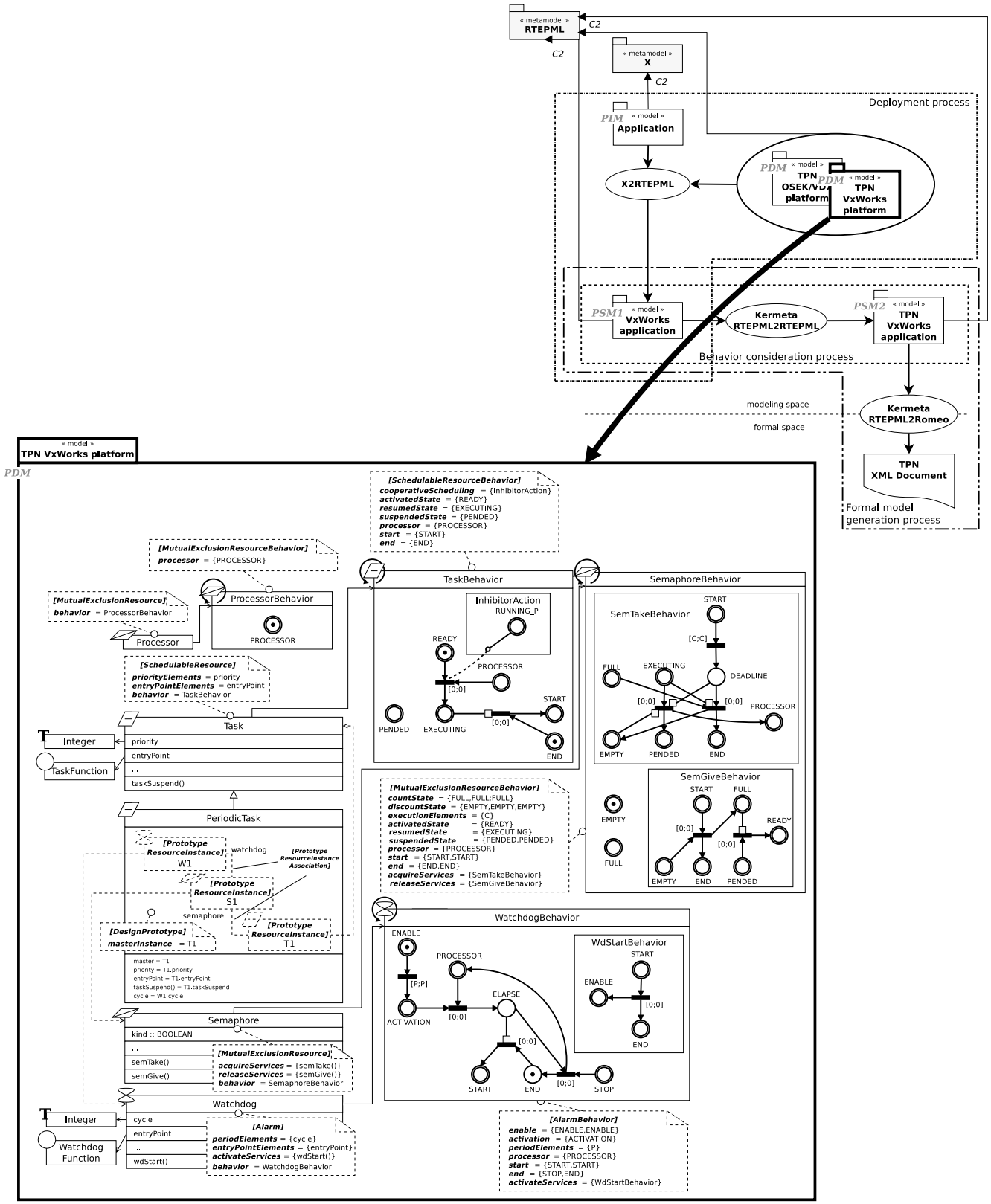


FIGURE 6.2 – Processus de génération de modèle formel en TPN avec considération d'une plate-forme VXWORKS

mentionné à la fin du chapitre précédent. En ce qui concerne le marquage du sémaphore

(SemaphoreBehavior), celui-ci est initialisé sur la place EMPTY pour que la tâche puisse être mise en attente dès la fin de sa première exécution. Ce marquage est certes adapté pour l'application qui va être présentée juste après. Mais, avec RTEPML, il est facile de définir plusieurs ressources du même type (e.g., sémaphore vide ou sémaphore plein) auxquelles un prototype comportemental est attaché.

Cette modélisation nous permet d'être optimiste quant à la généricité de représentation des noyaux temps réel. La notion de rôles compte évidemment beaucoup dans l'interprétation des choix de représentation. Finalement, les descriptions restent assez proches d'un noyau à l'autre, ce qui nous promet de passer rapidement d'un portage à un autre quelque soit l'application.

6.3 Application sur un exemple

Nous mettons ici en application notre processus de génération de modèles formels dans un contexte de déploiement multiplates-formes. L'exemple est adapté d'un cas d'étude d'une application multitâches [13] mise dans un contexte d'ordonnancement coopératif. Ce cas d'étude avait déjà été présenté dans une de nos contributions [51]. L'expérimentation qui repose sur ce cas a pour objectif de valider notre processus en s'appuyant sur deux plates-formes aux comportements et aux APIs différents. Ces plates-formes sont la norme OSEK/VDX qui a servi d'illustration tout au long de ce mémoire et le noyau VXWORKS [93] que nous venons d'aborder dans la section précédente.

6.3.1 Description du cas d'étude

L'application comporte trois activités concurrentes temps réel qui sont déployées sur trois tâches ordonnançables Task1, Task2 et Task3. La concurrence de ces trois tâches est mise en œuvre suivant une politique d'ordonnancement non préemptive basée sur des propriétés fixes. Leurs caractéristiques sont les suivantes :

- Task1 est périodique de période $P1 = a$ avec $a \in [0, +\infty[$ et un temps d'exécution $C1 \in [10, 20]$.
- Task2 est sporadique avec seulement un délai minimal de $P2 = 2a$ unités de temps entre deux activations. Le temps d'exécution de Task2 est $C2 \in [18, 28]$.
- Enfin, Task3 est périodique de période $P3 = 3a$ unités de temps et a un temps d'exécution $C3 \in [20, 28]$.

Ces trois tâches sont définies dans cet ordre de priorités : Task1 $>$ Task2 $>$ Task3. La période a de Task1 est un paramètre qui détermine la condition limite d'ordonnançabilité des tâches. L'application est ordonnançable si chaque activité a au moins une instance en exécution. La condition suffisante d'ordonnançabilité en mode non préemptif exige alors une charge processeur U telle que :

$$U = \sum_{i=1}^{n} (C_i/P_i) \leq 1 \tag{6.1}$$

avec n représentant le nombre de tâches, C_i précisant le pire temps d'exécution de chaque tâche et P_i indiquant la période (respectivement le délai minimal) de chaque tâche périodique (respectivement tâche sporadique) Task$i$.

À titre d'exemple, nous avons cherché à composer cette application déployée avec le comportement de la plate-forme considérée, à l'aide de notre processus. Puis, la propriété d'ordonnançabilité que nous venons d'évoquer a été vérifiée sur les deux modèles générés dans le but de retrouver le domaine de validité théorique (i.e., avant déploiement)

du paramètre a. Après calcul et conformément à une autre étude [40], le domaine attendu est $a \geq 44$.

6.3.2 Composition en TPN

Comme nous avons pu le voir dans le chapitre précédent, la composition d'un RI_TPN peut très vite exploser d'un point de vue combinatoire. Pour cette raison et pour simplifier, les figures 6.3 et 6.4 donnent les compositions établies en RI_TPN, uniquement sur la tâche Task3 ($\mathcal{N}_{Task3}$) après avoir respectivement considérer le comportement de la norme OSEK/VDX et du noyau VXWORKS.

Task3 a été préférée puisqu'elle présente le cas le plus complexe de composition due à sa priorité la plus basse. La localisation d'un mécanisme d'ordonnancement coopératif dans les PDMs a provoqué la création d'arcs d'inhibition sur la transition resume_Task3. En conséquence, le RI_TPN $\mathcal{N}_{Task3}$ cloné a subi l'extension suivante $CoopSched(\mathcal{N}_{Task3}, \{Task1, Task2\})$. Task3, comme les deux autres tâches, pointe sur une routine très simplifiée qui consiste juste à s'exécuter dans le temps imparti. Sur les deux modèles, la routine est constituée soit d'un service de terminaison ($\mathcal{N}_{terminate_3(Task3)}$) pour la norme OSEK/VDX, soit d'un service de prise du sémaphore ($\mathcal{N}_{semTake_3(Sem3)}$) avec mise en attente de la tâche pour le noyau VXWORKS. Le temps d'exécution $C3$ a aussi été intégré dans la description de ces services appelés, suite à la composition. Il est en effet possible, sur chaque prototype comportemental de service, d'assigner un rôle d'exécution (cf. rôle *executionElements*, figure 6.2) à un élément de comportement pour valuer ce temps d'exécution. Dans notre exemple, ce rôle est assigné sur le bornage de l'intervalle statique de la transition située entre les places START_terminate$_3$(Task3) (respectivement START_semTake$_3$(Sem3)) et DEADLINE_terminate$_3$(Task3) (respectivement DEADLINE_semTake$_3$(Sem3)) du RI_TPN $\mathcal{N}_{terminate_3(Task3)}$ (respectivement $\mathcal{N}_{semTake_3(Sem3)}$).

Rappelons aussi que les rôles apparaissent en gras, en haut à doite des places fusionnables, pour mettre en évidence les points de connexion utiles à la composition des RI_TPNs. Ces places fusionnables, toujours connectées par des arcs crochets, représentent celles localisées lors de la composition de Task3 avec sa routine d'exécution conformément à l'équation (5.2) [1]. Les fragments associés issus des prototypes de conception sont ensuite composés en appliquant les équations (5.3) et 5.4. Le même raisonnement est évidemment appliqué sur Task1 et Task2 avant d'être elles-mêmes composer avec Task3, toujours en respectant les équations (5.3) et (5.4).

Une fois les modèles composés, ils ont pu être vérifiés en utilisant Roméo. Les systèmes composés sont ordonnançables si les deux RI_TPNs sont *saufs* ; c'est-à-dire il existe au plus un jeton dans chaque place. Cette propriété doit par conséquent être satisfaite en appliquant la logique temporelle suivante : $AG_{[0,\infty]}bounded(1)$ qui dénote que pour tous les chemins du réseau parcourus, le marquage des places doit être, au maximum, borné à 1, dans un temps infini [74] [62]. Les résultats obtenus sur les deux RI_TPNs avec Roméo sont les suivants :

———-

Checking property AG[0,inf]bounded(1) on TPN :

"/home/clelionnais/TPN/OSEKVDX_NonPreemptiveApplication.xml" Waiting for response...

Result :

{a >= 44

}

———-

1. Il en est de même pour Watchdog3 qui présente aussi une routine d'exécution

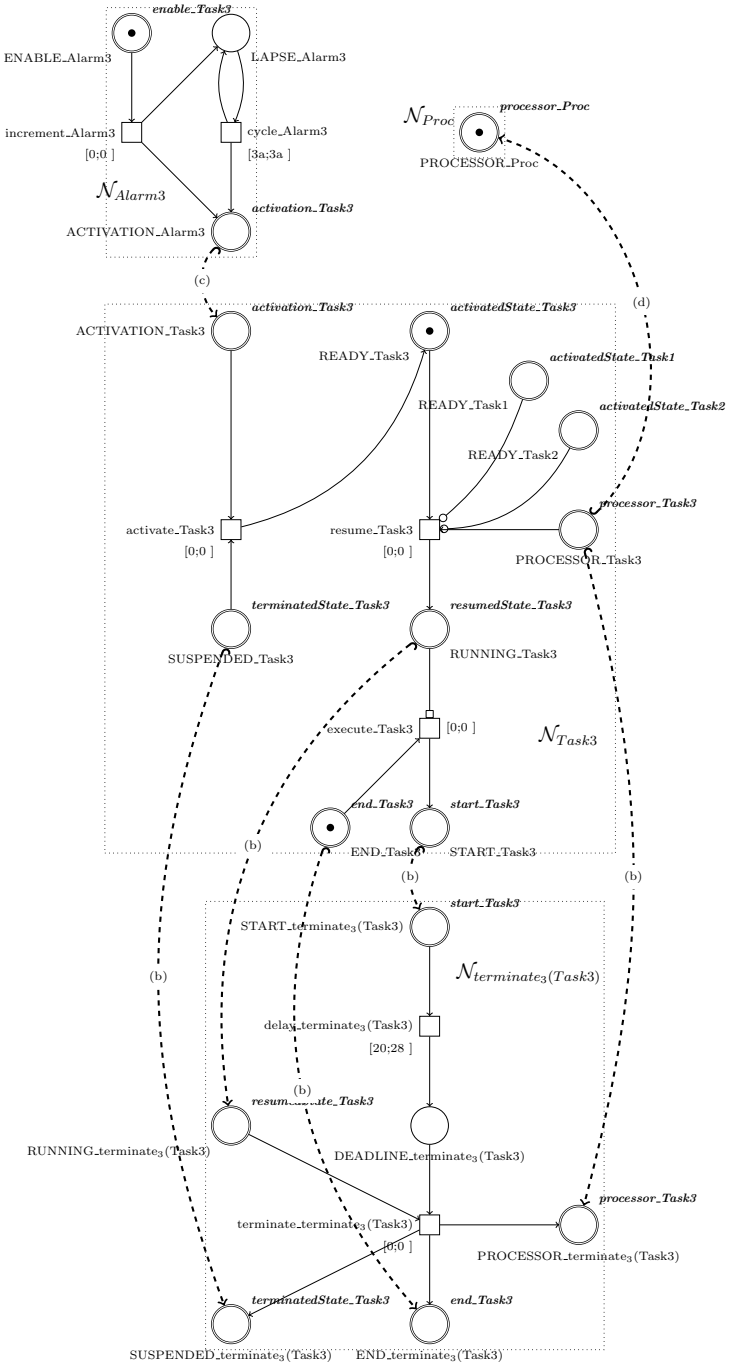


FIGURE 6.3 – Description en RI_TPN d'une tâche périodique déployée sur une plate-forme OSEK/VDX

Checking property AG[0,inf]bounded(1) on TPN :
"/home/clelionnais/TPN/VxWorks_NonPreemptiveApplication.xml" Waiting for response...
Result :
{a >= 44
}

Les résultats correspondent bien à la valeur théorique mentionnée plus haut. Dans cet exemple, la vérification nous donne satisfaction puisque la valeur théorique du domaine de validité annoncée avant déploiement est respectée après déploiement, et ce dans les deux cas

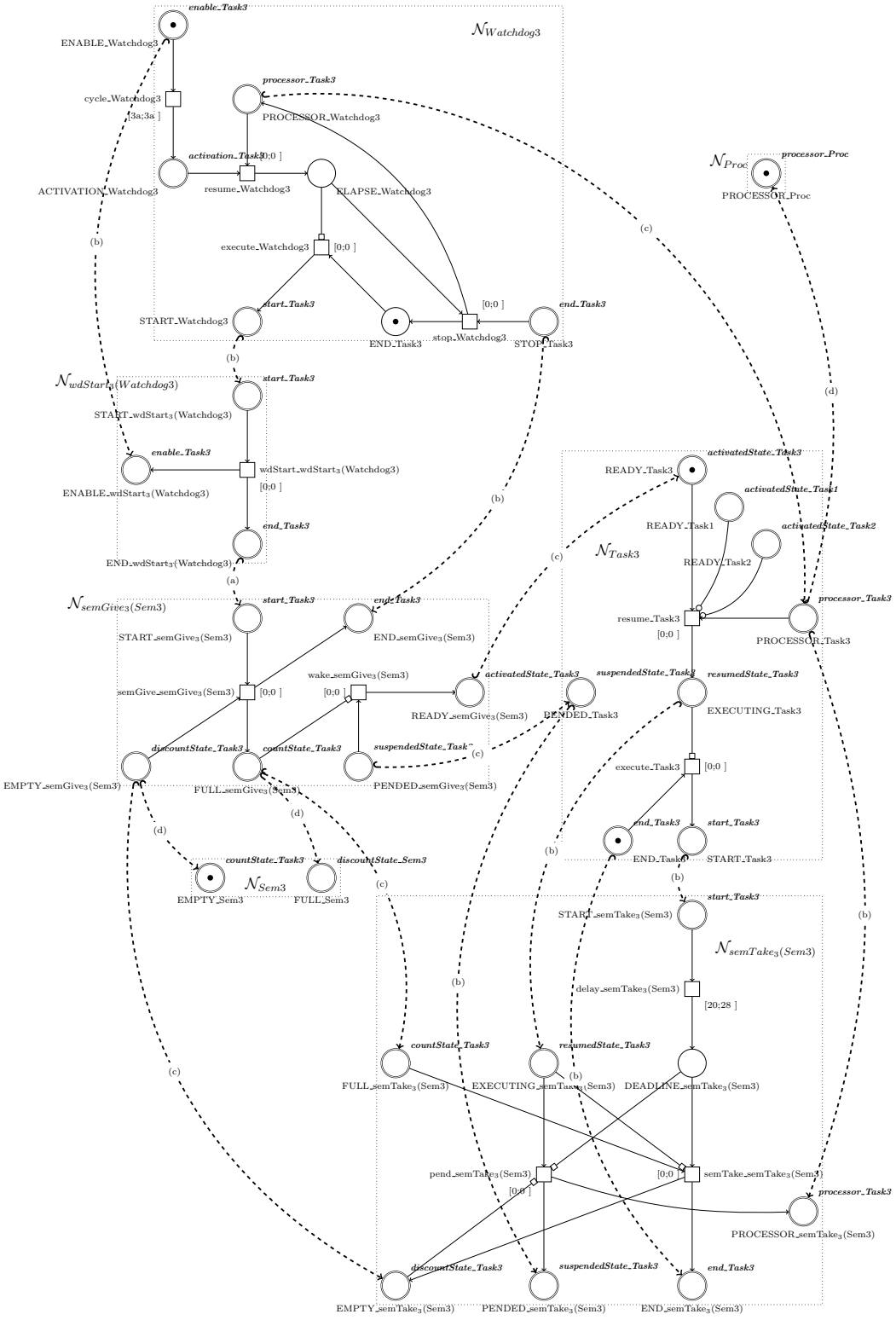


FIGURE 6.4 – Description en RI_TPN d'une tâche périodique déployée sur une plate-forme VXWORKS

proposés. Cette propriété de sécurité est ici nécessaire, mais bien évidemment, elle est loin d'être suffisante. D'autres propriétés pourraient être vérifiées montrant qu'un mauvais choix de plate-forme ou bien même de mécanisme au sein d'une plate-forme compromettrait le

déploiement.

Alternativement, nous aurions pu représenter le comportement de la périodicité autrement qu'avec une alarme. Avec le noyau VxWORKS par exemple, l'application d'un délai permet de différer la suite de l'exécution d'une tâche pendant une durée spécifiée. En pratique, cette mise en œuvre pour le compte d'une tâche périodique peut engendrer une dérive temporelle qui biaiserait le domaine de validité théorique.

6.4 Bilan

Ce chapitre avait pour objectif de montrer une mise en application de notre processus de génération de modèles formels en TPN. Un cas d'étude d'une application déployée sur deux plates-formes aux comportements différents (OSEK/VDX et VxWORKS) a été proposé. Deux modèles comportementaux ont ainsi pu être composés indépendamment de la plate-forme considérée pour le déploiement. Une vérification simple a de surcroît été menée sur les modèles générés, pour s'assurer d'une limite d'ordonnançabilité théorique d'un système après déploiement.

Cette approche contribue dans son intégralité aux trois points évoqués dans l'introduction, pour améliorer la productivité des SETRS :

- **La portabilité** : La facilité de traduction des prototypes comportementaux grâce à RTEPML et à l'expressivité des TPNs permet de représenter le comportement de différentes plates-formes logicielles d'exécution temps réel. En choisissant la plate-forme qui nous intéresse en paramètre de notre processus de génération, nous pouvons rapidement porter une application et ainsi désigner la plate-forme la plus appropriée à la conception d'un système ;
- **La réutilisabilité** : La notion de rôles que nous avons adoptée pour localiser les éléments à travers les modèles, a permis d'établir des règles de transformation génériques pour la composition. Cette valeur ajoutée apporte précisément plus de flexibilité dans la réutilisation de notre processus de génération ;
- **La maintenabilité** : Enfin, l'indépendance des intervenants est favorisée puisque les préoccupations métiers ont été séparées. Le concepteur du SETR peut définir son type d'application sans se soucier du déploiement. Il peut ensuite intégrer son application dans une plate-forme particulière qui a été préalablement définie par un spécialiste du RTOS. Ce dernier peut indépendamment modéliser sa plate-forme soit d'un point de vue structurel, soit d'un point de vue comportemental. Le concepteur peut ensuite utiliser notre processus pour successivement déployer son application et la formaliser une fois déployée. La formalisation obtenue offre la possibilité d'appliquer des activités de vérification. Un expert dans ce domaine peut par conséquent retourner un verdict au concepteur après avoir vérifié les propriétés fonctionnelles et non fonctionnelles qui lui ont été spécifiées.

Comme nous l'avons déjà mentionné, cette mise en application n'avait pas pour vocation d'établir un recueil de propriétés à vérifier. Cette expérimentation de vérification que nous avons ajoutée a seulement permis de valider notre stratégie de considération du comportement basée sur les rôles. Par la suite, d'autres plates-formes devront néanmoins être considérées pour renforcer la généricité de notre approche.

Conclusion

Bilan

L'objectif principal de cette thèse était de fournir une approche de considération du comportement des plates-formes d'exécution logicielles temps réels dans un cadre de déploiement d'application multiplates-formes. Elle s'inscrit dans une ingénierie générative dirigée par les modèles telle que l'IDM, afin d'intégrer des activités de vérification après déploiement d'applications logicielles. La genèse de cette approche relève du manque d'application de ces activités à un niveau de conception détaillé des SETRs.

En première partie, un état de l'art (chapitre 2) a introduit notre suite d'outils SEXPISTOOLS (pour *Software Execution Platform Inside Tools*). Cette suite intègre le langage de modélisation RTEPML (*Real Time Embedded Platform Modeling Language*) pour représenter explicitement les plates-formes. Aussi, un processus de génération y avait été ajouté pour considérer ces représentations à travers un processus de déploiement. Grâce à l'approche de représentation explicite, le processus avait alors été développé indépendamment de la plate-forme visée, ce qui apporte plus de généricité. Cependant, dans l'état, le comportement des plates-formes n'était pas représentable avec RTEPML. De plus, aucune vérification ne pouvait être menée avec cette suite. Un positionnement a donc été établi vis-à-vis d'autres suites d'outils qui ont contribué à la considération du comportement des plates-formes dans une IDM. Deux grandes familles ont pu être isolées avec d'un côté, les suites qui se sont orientées vers la génération de code et de l'autre côté, celles qui se sont tournées vers la génération de modèles formels. Seulement dans les deux situations et ce malgré l'intérêt porté pour considérer le comportement, rares sont celles qui participent à la fois au déploiement et à la vérification. Lorsque c'est le cas, elles ne favorisent pas l'amélioration de la productivité des SETRs.

Dans une deuxième partie, nos contributions ont été détaillées. En premier lieu, des concepts comportementaux ont été ajoutés dans RTEPML (chapitre 3) pour aussi représenter explicitement le comportement des plates-formes. Une traduction a de plus été intégrée pour représenter formellement ce comportement en TPN. Des rôles ont été identifiés pour localiser des éléments utiles à la composition d'un modèle d'application déployée. Cette stratégie a ensuite permis de rédiger des règles de transformation indépendantes du langage de modélisation du comportement, pour le compte d'un processus de considération explicite du comportement des plates-formes (chapitre 4). Tout au long du processus, des fragments comportementaux sont assemblés pour obtenir un modèle global. Les fragments ayant été définis en TPN, une formalisation de cet assemblage (chapitre 5) a aussi été posée pour nous assurer du bon enchaînement des règles de transformation. Un cas d'étude simplifié a finalement été donné pour composer deux modèles comportementaux d'une même application déployée sur deux plates-formes différentes (chapitre 6).

Perspectives

Les perspectives sont nombreuses et nous les présentons ici selon leur ordre logique de traitement.

Approfondir l'approche

La version du processus implémenté est pour le moment un prototype qui permet de générer des modèles formels en TPN exploitables par l'outil de vérification Roméo. Un certain nombre de rôles ont pu être identifiés pour la composition de tels modèles. Toutefois, d'autres rôles devront certainement être ajoutés pour prendre en compte d'autres mécanismes de plates-formes d'exécution logicielles. Nous prévoyons donc de représenter d'autres plates-formes avec RTEPML, à l'image de la norme ARINC-653 [1] qui présente des contraintes spatiales. De plus, nous avons employé l'outil de transformation Kermeta pour implémenter le processus de considération du comportement. Le processus n'étant qu'un prototype, nous programmons d'uniformiser l'implémentation avec le langage de transformation ATL qui a été utilisé pour le processus de déploiement.

La prise en compte de mécanismes d'ordonnancement plus spécifiques est aussi prévue. Dans cette thèse, notre implication s'est focalisée sur la représentation de priorités fixes. D'autres politiques d'ordonnancement [20] demandent aussi de considérer les priorités dynamiques. Les mécanismes liés à ces politiques devront aussi être considérés dans notre approche. Une approche de rétro-ingénierie [61] s'est récemment focalisée sur l'analyse d'ordonnançabilité en considérant explicitement les politiques d'ordonnancement. Cette démarche peut nous aider à mieux aborder cette prise en compte.

La formalisation proposée a soulevé certaines limites de considération des mécanismes spécifiques avec les TPNs. À l'instar du mécanisme d'ordonnancement coopératif présenté, le clonage itératif de fragments comportementaux oblige à connecter des éléments qui ne sont pas localisables avec des rôles. L'utilisation de fonctions a solutionné certaines adaptations de composition dans le processus, mais elles se basent quand même sur ces rôles. Nous devons par conséquent trouver une stratégie plus générique de composition. Nous envisageons d'utiliser des langages de traduction formelle plus haut niveau que les TPNs, à l'instar des *Scheduling* TPNs [52] qui intègrent le mécanisme de préemption.

Étendre l'approche

Ce besoin d'utiliser d'autres classes de modélisation du comportement nous oblige à étendre notre stratégie d'intégration des concepts formels au sein de RTEPML. De manière similaire à notre approche de représentation explicite du comportement des plates-formes, nous projetons d'apporter à RTEPML une adaptabilité de traduction du comportement. Offrir la possibilité aux concepteurs de modéliser le comportement des plates-formes avec le langage de leur choix apporterait plus de souplesse à notre approche de considération.

Cette orientation amènerait en outre à ouvrir nos recherches dans un contexte GPML avec le profil UML-MARTE. En effet, le comportement dans UML peut être représenté par l'intermédiaire de langages tels que les *Statecharts* UML. Adapter une traduction du comportement des plates-formes dans RTEPML avec ces langages permettrait de réinjecter dans UML-MARTE les descriptions obtenues.

Troisième partie

Annexes

A

Guide conceptuel avec SExPIsTools

Cette annexe propose un ensemble de diagrammes d'activités de SExPIsTools. En premier lieu, un guide conceptuel sur l'utilisation du langage de modélisation RTEPML est fourni pour savoir comment définir un modèle structurel de plate-forme d'exécution logicielle temps réel. Puis, un second guide vient présenter comment le processus de déploiement s'exécute pour générer un modèle structurel global d'application déployée en considérant une représentation explicite de la plate-forme ciblée. La partie comportementale n'apparaît pas ici puisque l'extension de RTEPML et les règles de considération du comportement sont respectivement détaillées dans les chapitres 3 et 4.

Définition d'un modèle structurel de plate-forme avec RTEPML

La figure A.1 permet de visualiser le flux d'activités qui mène à la définition d'un PDM structurel. L'implantation du motif «Ressource-Service» dans RTEPML oriente le concepteur à définir lui-même les concepts exécutifs et les services offerts par la plate-forme à représenter.

La figure A.2 permet de visualiser le flux d'activités qui mène à la définition de prototypes de conception au sein d'un PDM structurel. Le concepteur peut ainsi définir certains concepts exécutifs qui n'existent pas sur la plate-forme à représenter, mais qui peuvent être mis en œuvre par agencement de concepts existants dans un prototype de conception.

Génération d'un modèle structurel d'application déployée

La figure A.3 permet de visualiser le flux d'activités du processus de déploiement qui mène à la génération d'un modèle d'application déployée sur une plate-forme définie (guide présenté précédemment). Le diagramme illustre l'enchaînement des règles de correspondances entre les concepts applicatifs et les ressources logicielles avec ou sans prototype de conception. La création d'instances des ressources définies et de propriétés d'instances pour les besoins de l'application est précisée. Toutefois, la génération de ces règles de correspondances, comme abordée dans le chapitre 2 page 26, n'est pas précisée ici.

Pour plus de clarté, la figure A.4 apporte plus de précisions quant à l'activité de création des appels de services. Cette activité est en effet présentée sous forme de macro dans la figure A.3 ci-dessus. Pour cette raison, le diagramme ci-après illustre l'enchaînement des règles de création des appels de service au sein des instances de ressources.

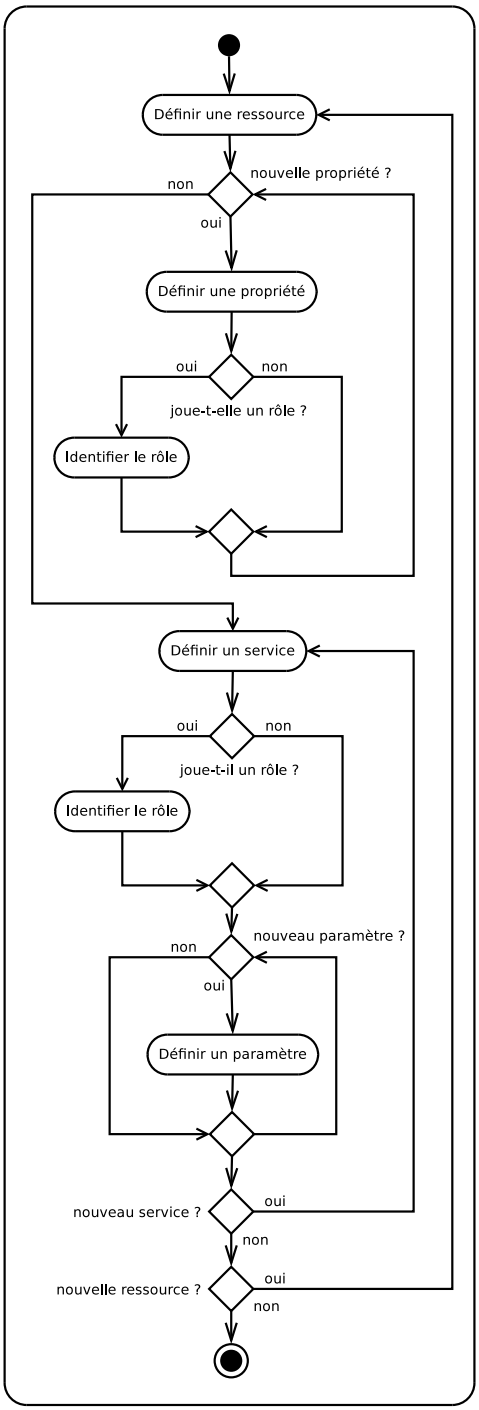


FIGURE A.1 – Guide conceptuel pour la définition des ressources et des services avec RTEPML

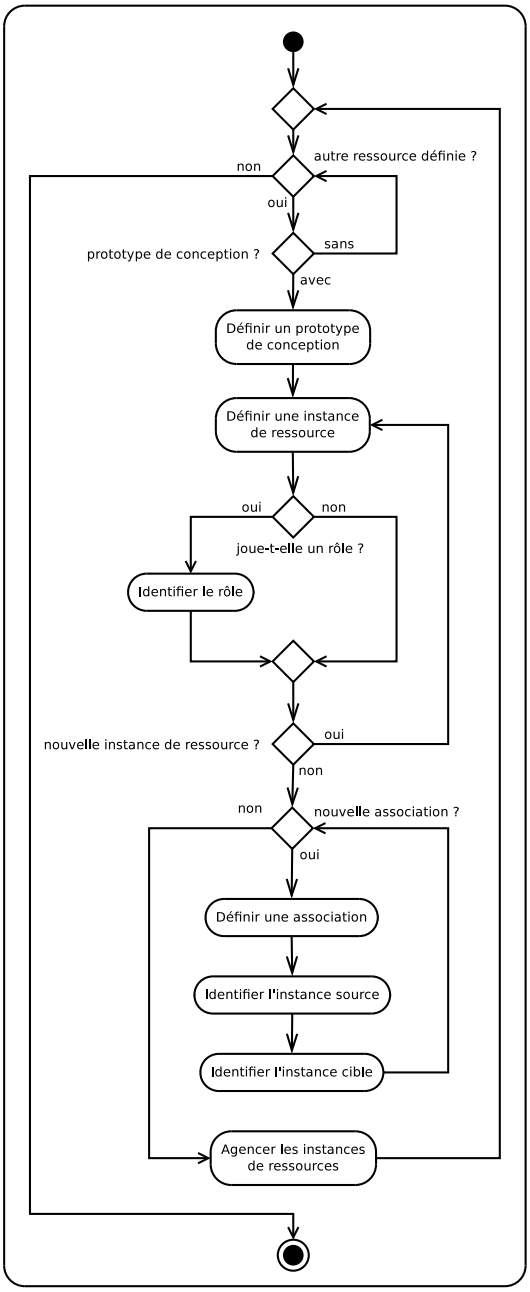


FIGURE A.2 – Guide conceptuel pour la définition de prototypes de conception avec RTEPML

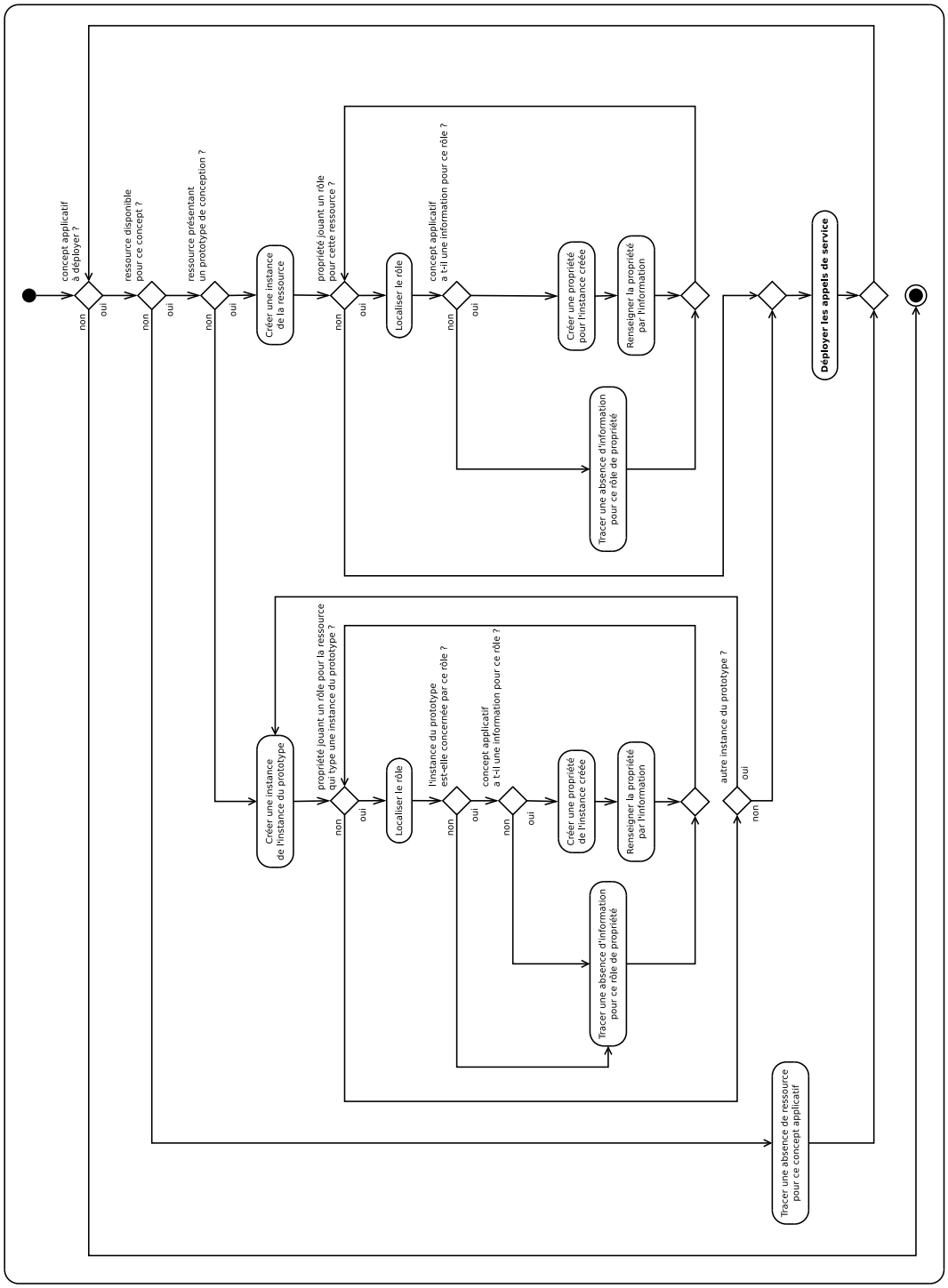


FIGURE A.3 – Guide du processus de déploiement avec création d'instances de ressources

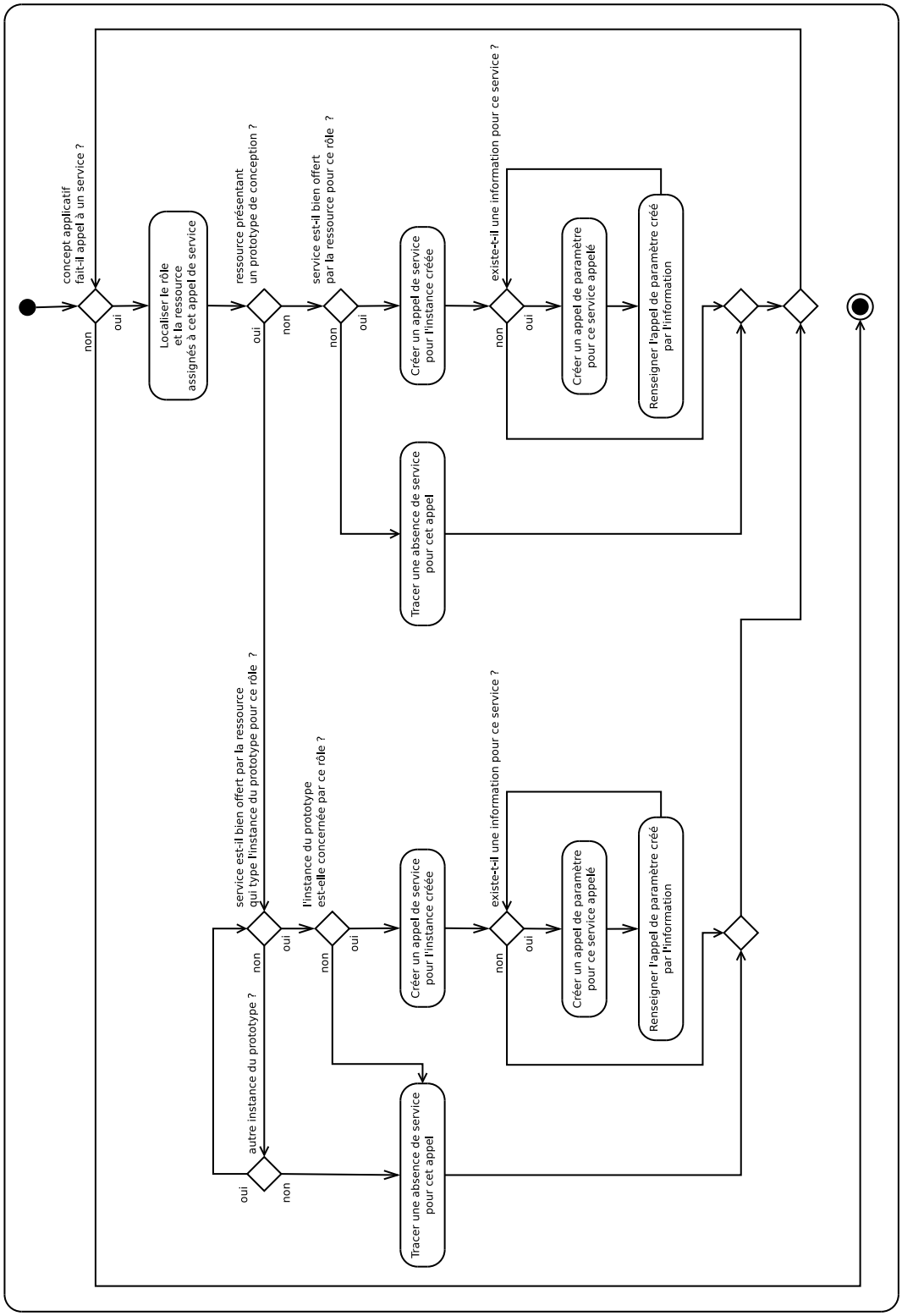


FIGURE A.4 – Guide du processus de déploiement avec création d'appels de services

B

Métamodèle de RTEPML

Le métamodèle du langage RTEPML a été introduit dans le chapitre 2, page 23. Il a été ensuite étendu dans le chapitre 3. Le listing B.1 suivant revient sur le métamodèle structurel de RTEPML [12] en ne présentant ici que l'aspect comportemental.

Listing B.1 – Métamodèle du langage RTEPML (en KM3 [39])

```
1  -- @name              RTEPML
2  -- @version           1.0
3  -- @authors           cedrick.lelionnais@eseo.fr (@irccyn.ec-nantes.fr)
4  -- @description       Real Time Embedded Platform Modeling Language
5  -- @nsURI RTEPML
6  -- @nsPrefix RTEPML
7
8  package RTEPML {
9
10     ------------------------------------------------------------------
11     -- + -- Located and Named Elements
12     ------------------------------------------------------------------
13
14     abstract class LocatedElement {
15         -- already present in RTEPML
16         attribute location[0-1] : String;
17         attribute commentsBefore[*] ordered : String;
18         attribute commentsAfter[*] ordered : String;
19     }
20
21     abstract class NamedElement extends LocatedElement {
22         -- already present in RTEPML
23         attribute name : String;
24     }
25
26     ------------------------------------------------------------------
27     -- 1 -- Platform Model and Platform Behavioral Model
28     ------------------------------------------------------------------
29
30     class PlatformModel extends NamedElement {
31         -- add platform behavioral model concept
32         reference platformBehavioralModel container : PlatformBehavioralModel;
33     }
34
35     class PlatformBehavioralModel extends NamedElement {
36         -- add behavioral prototype concept
37         reference prototypes[*] container : BehavioralPrototype;
38     }
39
40     ------------------------------------------------------------------
41     -- 1.1 -- Resource and Behavioral Prototype
42     ------------------------------------------------------------------
43
44     abstract class Resource extends NamedElement {
45         -- add behavioral prototype concept
46         reference behavior[0-1] : BehavioralPrototype;
47     }
48
49     abstract class BehavioralPrototype extends NamedElement {
50         -- add both behavioral prototype service and behavior element concepts
51         reference services[*] container : BehavioralPrototypeService;
52         reference elements[*] container : BehaviorElement;
53         -- add execution role concept
54         reference executionElements[*] : BehaviorElement;
55     }
56
57     ------------------------------------------------------------------
58     -- 1.2 -- Resource Instance and Resource Instance Property
59     ------------------------------------------------------------------
60
61     abstract class ResourceInstance extends NamedElement {
62         reference clonedElements[*] : BehaviorElement oppositeOf refInstances;
63     }
```

```
64
65      ------------------------------------------------------------
66      -- 1.3 -- Call Service and Behavioral Prototype Service
67      ------------------------------------------------------------
68
69      class CallService extends LocatedElement {
70          reference clonedElements[*] : BehaviorElement oppositeOf refCalls;
71      }
72
73      class BehavioralPrototypeService extends NamedElement {
74          -- add behavior element concept
75          reference elements[*] : container : BehaviorElement;
76      }
77
78      ------------------------------------------------------------
79      -- 1.4 -- Data Type
80      ------------------------------------------------------------
81
82      ------------------------------------------------------------
83      -- 1.5 -- Routine and Routine Behavior
84      ------------------------------------------------------------
85
86      class RoutineBehavior extends BehavioralPrototype {}
87
88      ------------------------------------------------------------
89      -- 1.6 -- Behavior Element
90      ------------------------------------------------------------
91
92      abstract class BehaviorElement extends NamedElement {
93          -- add an assigned role concept
94          attribute role : String;
95          reference [0-1] : BehavioralPrototype oppositeOf elements;
96          reference [0-1] : BehavioralPrototypeService oppositeOf elements;
97          reference applicationBehavioralModel[0-1] :
                ApplicationBehavioralModel oppositeOf clonedElements;
98          reference refInstances[*] : ResourceInstance oppositeOf
                clonedElements;
99          reference refCalls[*] : CallService oppositeOf clonedElements;
100     }
101
102     class Composite extends BehaviorElement {
103         -- add sub behavior element concept
104         reference subElements[*] container : BehaviorElement;
105     }
106
107     class Atomic extends BehaviorElement {
108         attribute val : String;
109     }
110
111     ------------------------------------------------------------
112     -- 2 -- Artefacts
113     ------------------------------------------------------------
114
115     ------------------------------------------------------------
116     -- 2.1 -- Prototype
117     ------------------------------------------------------------
118
119     ------------------------------------------------------------
120     -- 2.2 -- Application and Application Behavioral Model
121     ------------------------------------------------------------
122
123     class Application extends NamedElement {
124         -- add application behavioral model concept
125         reference applicationBehavioralModel container :
                ApplicationBehavioralModel;
126     }
127
128     class ApplicationBehavioralModel extends NamedElement {
129         -- add cloned behavior element concept
130         reference clonedElements[*] container : BehaviorElement oppositeOf
                applicationBehavioralModel;
```

```
131     }
132
133     ------------------------------------------------------------
134     -- 3 -- Behavioral Prototypes
135     ------------------------------------------------------------
136
137     ------------------------------------------------------------
138     -- 3.1 -- Concurrent Resource Behavior
139     ------------------------------------------------------------
140
141     abstract class ConcurrentResourceBehavior extends BehavioralPrototype {
142         -- add property role concepts
143         reference activationCapacityElements[*] : BehaviorElement;
144         reference periodElements[*] : BehaviorElement;
145         reference priorityElements[*] : BehaviorElement;
146         reference stackSizeElements[*] : BehaviorElement;
147         reference entryPointElements[*] : ResourceProperty;
148
149         -- add service role concepts
150         reference activateServices[*] : BehavioralPrototypeService;
151         reference resumeServices[*] : BehavioralPrototypeService;
152         reference suspendServices[*] : BehavioralPrototypeService;
153         reference terminateServices[*] : BehavioralPrototypeService;
154         reference enableConcurrencyServices[*] : BehavioralPrototypeService;
155         reference disableConcurrencyServices[*] : BehavioralPrototypeService;
156
157         -- add composition role concepts
158         reference activation[*] : BehaviorElement;
159         reference activatedState[*] : BehaviorElement;
160         reference resumedState[*] : BehaviorElement;
161         reference terminatedState[*] : BehaviorElement;
162         reference suspendedState[*] : BehaviorElement;
163         reference priorityActivatedState[*] : BehaviorElement;
164         reference priorityResumedState[*] : BehaviorElement;
165         reference processor[*] : BehaviorElement;
166         reference start[*] : BehaviorElement;
167         reference end[*] : BehaviorElement;
168
169         -- add specific mecanism role concepts
170         reference cooperativeScheduling[*] : BehaviorElement;
171         reference priorityPreemptiveScheduling[*] : BehaviorElement;
172         reference fifoScheduling[*] : BehaviorElement;
173         reference lifoScheduling[*] : BehaviorElement;
174 }
175
176     ------------------------------------------------------------
177     -- 3.1.1 -- Interrupt Resource Behavior
178     ------------------------------------------------------------
179
180     abstract class InterruptResourceBehavior extends
            ConcurrentResourceBehavior {
181         -- add property role concepts
182         reference vectorElements[*] : ResourceProperty;
183         reference maskElements[*] : ResourceProperty;
184
185         -- add service role concepts
186         reference routineConnectServices[*] : ResourceService;
187         reference routineDisconnectServices[*] : ResourceService;
188     }
189
190     class InterruptBehavior extends InterruptResourceBehavior {
191         -- add composition role concepts
192         reference enable[*] : BehaviorElement;
193     }
194
195     class AlarmBehavior extends InterruptResourceBehavior {}
196
197     ------------------------------------------------------------
198     -- 3.1.2 -- Schedulable Resource Behavior
199     ------------------------------------------------------------
200
```

```
201     class SchedulableResourceBehavior extends ConcurrentResourceBehavior {
202         -- add property role concepts
203         reference deadlineElements[*] : ResourceProperty;
204         reference deadlineTypeElements[*] : ResourceProperty;
205         reference timeSliceElements[*] : ResourceProperty;
206 
207         -- add service role concepts
208         reference joinServices[*] : ResourceService;
209         reference yieldServices[*] : ResourceService;
210         reference delayService[*] : ResourceService;
211 
212         -- add other scheduling mecanisms role concepts
213     }
214 
215     ------------------------------------------------------------
216     -- 3.2 -- Memory Partition Behavior
217     ------------------------------------------------------------
218 
219     class MemoryPartitionBehavior extends BehavioralPrototype {
220         -- add property role concepts
221         reference memorySpaceElements[*] : ResourceProperty;
222 
223         -- add service role concepts
224         reference forkService : ResourceService;
225         reference exitService : ResourceService;
226     }
227 
228     ------------------------------------------------------------
229     -- 3.3 -- Interaction Resource Behavior
230     ------------------------------------------------------------
231 
232     abstract class InteractionResourceBehavior extends BehavioralPrototype {
233         -- add property role concepts
234         reference waitingPolicyElements[*] : ResourceProperty;
235 
236         -- add composition role concepts
237         reference activatedState[*] : BehaviorElement;
238         reference priorityActivatedState[*] : BehaviorElement;
239         reference resumedState[*] : BehaviorElement;
240         reference priorityResumedState[*] : BehaviorElement;
241         reference suspendedState[*] : BehaviorElement;
242         reference terminatedState[*] : BehaviorElement;
243         reference processor[*] : BehaviorElement;
244         reference start[*] : BehaviorElement;
245         reference end[*] : BehaviorElement;
246     }
247 
248     ------------------------------------------------------------
249     -- 3.3.1 -- Synchronisation Resource Behavior
250     ------------------------------------------------------------
251 
252     abstract class SynchronisationResourceBehavior extends
            InteractionResourceBehavior {}
253 
254     ------------------------------------------------------------
255     -- 3.3.1.1 -- Mutual Exclusion Resource Behavior
256     ------------------------------------------------------------
257 
258     class MutualExclusionResourceBehavior extends
            SynchronisationResourceBehavior {
259         -- add property role concepts
260         reference accessTokenElements[*] : ResourceProperty;
261 
262         -- add service role concepts
263         reference acquireServices[*] : ResourceService;
264         reference releaseServices[*] : ResourceService;
265 
266         -- add composition role concepts
267         reference countState[*] : BehaviorElement;
268         reference discountState[*] : BehaviorElement;
269 
```

```
270         -- add specific mechanism role concepts
271         reference mutualAcquisitionScheduling[*] : BehaviorElement;
272         reference mutualReleasingScheduling[*] : BehaviorElement;
273     }
274
275     ------------------------------------------------------------
276     -- 3.3.1.2 -- Notification Resource Behavior
277     ------------------------------------------------------------
278
279     class NotificationResourceBehavior extends
            SynchronisationResourceBehavior {
280         -- add property role concepts
281         reference occurenceCountElements[*] : ResourceProperty;
282         reference maskElements[*] : ResourceProperty;
283
284         -- add service role concepts
285         reference flushServices[*] : ResourceService;
286         reference signalServices[*] : ResourceService;
287         reference waitServices[*] : ResourceService;
288         reference clearServices[*] : ResourceService;
289
290         -- add composition role concepts
291         reference clearedState[*] : BehaviorElement;
292         reference signaledState[*] : BehaviorElement;
293         reference memorizedState[*] : BehaviorElement;
294
295         -- add specific mechanism role concepts
296         reference waitingScheduling[*] : BehaviorElement;
297         reference signalingScheduling[*] : BehaviorElement;
298 }
299     ------------------------------------------------------------
300     -- 3.3.2 -- Communication Resource Behavior
301     ------------------------------------------------------------
302
303     abstract class CommunicationResourceBehavior extends
            InteractionResourceBehavior {}
304
305     ------------------------------------------------------------
306     -- 3.3.2.1 -- Message Com Resource Behavior
307     ------------------------------------------------------------
308
309     class MessageComResourceBehavior extends CommunicationResourceBehavior {
310         -- add property role concepts
311         reference messageSizeElements[*] : ResourceProperty;
312         reference messageQueueCapacityElements[*] : ResourceProperty;
313
314         -- add service role concepts
315         reference sendServices[*] : ResourceService;
316         reference receiveServices[*] : ResourceService;
317
318         -- add composition role concepts
319         reference pendedState[*] : BehaviorElement;
320         reference postedState[*] : BehaviorElement;
321
322         -- add specific mechanism role concepts
323         reference sendingScheduling[*] : BehaviorElement;
324         reference receivingScheduling[*] : BehaviorElement;
325     }
326
327     ------------------------------------------------------------
328     -- 3.3.2.2 -- Shared Data Com Resource Behavior
329     ------------------------------------------------------------
330
331     class SharedDataComResourceBehavior extends CommunicationResourceBehavior
            {
332         -- add service role concepts
333         reference readServices[*] : ResourceService;
334         reference writeServices[*] : ResourceService;
335     }
336
337     ------------------------------------------------------------
```

```
338     -- 4 -- Timer Resource Behavior
339     ------------------------------------------------------------------
340
341     class TimerResourceBehavior extends BehavioralPrototype {
342         -- add property role concepts
343         reference durationElements[*] : ResourceProperty;
344
345         -- add composition role concepts
346         reference enable[*] : BehaviorElement;
347     }
348 }
```

Table des figures

Liste des tableaux

Liste des Algorithmes

Bibliographie

[1] Airlines Electronic Engineering Committee. *Avionics Application Software Standard Interface, ARINC Specification 653-1*, October 2003. Aeronautical radio INC., Annapolis, Maryland, USA.

[2] Ahmad Badreddin Alkhodre. *Développement formel de systèmes temps réel à l'aide de SDL et IF*. PhD thesis, INSAL, France, September 2004.

[3] Colin Atkinson and Thomas Kühne. A generalized notion of platforms for model-driven development. pages 119–136. 2005.

[4] ATLAS group. ATL *:* ATLAS *Transformation Language,* ATL *User Manual, version 0.7*. LINA & INRIA, Nantes, France, February 2006.

[5] Jean-Philippe Babau. *Formalisation et structuration des architectures opérationnelles pour les systèmes embarqués temps réel*. Hdr, INSA de Lyon, December 2005.

[6] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In M. Bernardo and F. Corradini, editors, *International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–237. Springer Verlag, 2004.

[7] A. Benzina. *Analyse et Conception des Systèmes Temps Réels : Translation d'une Approche Fonctionnelle à une Approche Orientée Objet*. PhD thesis, Université Paul Sabatier, Toulouse, France, Décembre 1997.

[8] S. Beydeda, M. Book, and V. Gruhn. *Model-driven software development*. Springer eBooks collection : Computer science. Springer, 2005.

[9] G. Booch. Object-oriented development. *Software Engineering, IEEE Transactions on*, SE-12(2) :211–221, Feb 1986.

[10] Marc Boyer and Olivier H. Roux. On the compared expressiveness of arc, place and transition time Petri nets. *Fundamenta Informaticae*, 88(3) :225–249, 2008.

[11] Manfred Broy. Challenges in Automotive Software Engineering. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 33–42, New York, NY, USA, 2006. ACM.

[12] Matthias Brun. *Contribution à la considération explicite des plates-formes d'exécution logicielles lors d'un processus de déploiement d'application*. PhD thesis, École Centrale de Nantes, Nantes, France, Octobre 2010.

[13] G. Bucci, A. Fedeli, L. Sassoli, and E. Vicario. Timed State Space Analysis of Real-Time Preemptive Systems. *Software Engineering, IEEE Transactions on*, 30(2) :97–111, February 2004.

[14] S. Burmester and H. Giese. The Fujaba Real-Time Statechart Plugin. In *Proc. of the first International Fujaba Days 2003, Kassel, Germany*, 2003.

[15] S. Burmester, H. Giese, M. Hirsch, and D. Schilling. Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In *Proc. of the International Workshop on Specification and Validation of UML Models for Real Time and*

Embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004, 2004.

[16] Paul Caspi, Grégoire Hamon, and Marc Pouzet. *Systèmes Temps-réel : Techniques de Description et de Vérification – Théorie et Outils*, volume 1, chapter Lucid Synchrone, un langage de programmation des systèmes réactifs, pages 217–260. Hermes, 2006.

[17] Damien Chabrol. *Etude, conception et mise en oeuvre d'un protocole de communication synchrone tolérant aux fautes et prédictible sur des composants réseaux standards*. PhD thesis, Université Paris Sud, Paris, France, Juin 2006.

[18] Duncan Clarke, Insup Lee, and Hong liang Xie. Versa : A tool for the specification and analysis of resource-bound real-time systems. *Journal of Computer and Software Engineering*, 3, 1995.

[19] Mickaël Clavreul. *Composition de Modèles et de Métamodèles : Séparation des Correspondances et des Interprétations pour Unifier les Approches de Composition Existantes*. These, Université Rennes 1, December 2011. final draft.

[20] Francis Cottet and Anne-Marie Déplanche. *Encylcopédie de l'informatique et des systèmes d'information*, chapter Ordonnancement temps réel et ordonnançabilité d'une application, pages 740–760. Vuibert, 2006.

[21] Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Guang Yang, Haibo Zeng, and Qi Zhu. A next-generation design framework for platform-based design. In *DVCon 2007*, February 2007.

[22] Vincent. David, Christophe. Aussaguès, Stéphane. Louise, Philippe Hilsenkopf, Bertrand. Ortolo, and Christophe. Hessler. The OASIS based qualified display system. In *Lecture Notes in Computer Science, 17th International Conf. on Computer Safety, Reliability and Security Fourth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Controls and Human-Machine Interface Technologies (NPIC&HMIT 2004), Columbus, Ohio. September, 2004.*, September 2004.

[23] James Davis. Gme : The generic modeling environment. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 82–83, New York, NY, USA, 2003. ACM.

[24] Jérôme Delatour. *Contribution á la spécification des systèmes temps réel : l'approche* UML/PNO. PhD thesis, Université Paul Sabatier, Toulouse, France, Septembre 2003.

[25] Jérôme Delatour, Raphaël Marvie, and Guillaume Savaton. Workshop on Metamodel Pattern, MP2006. In *2èmes Journées sur l'Ingénierie Dirigée par les Modèles*, Lille, Juin 2006.

[26] Jérôme Delatour, Frédéric Thomas, Guillaume Savaton, and Sébastien Faucou. Modèle de plate-forme pour l'embarqué : première expérimentation sur les noyaux temps réel. In *Actes des 1ère Journées sur l'Ingénierie Dirirgée par les Modèles, IDM05*, pages 209–216, Paris, France, juin 2005.

[27] Zoé Drey, Cyril Faucher, Franck Fleurey, Vincent Mahé, and Didier Vojtisek. *Kermeta language reference manual*. Triskell group - IRISA/INRIA.

[28] Eclipse. Eclipse Modeling Project. http ://www.eclipse.org/modeling/.

[29] Eclispe Modeling Framework. EMF Core. http ://www.eclipse.org/modeling/emf/.

[30] Wassim El Hajj Chehade. *Contribution au Déploiement Multiplateforme d'Applications Multitâches par la Modélisation Comportementale Haut Niveau des Services d'Exécution*. PhD thesis, Laboratoire d'Ingénierie Dirigée par les Modèles des Systèmes Temps Réels Embarqués (LISE) - CEA Saclay, 2011.

[31] Jean-Pierre Elloy. *Le temps réel*. Rapport établi par le Groupe de réflexion du CNRS-SPI, TSI 7(5), Hermes, 1988.

[32] Jean-Rémy Falleri, Marianne Huchard, Mathieu Lafourcade, and Clémentine Nebut. Metamodel Matching for Automatic Model Transformation Generation. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 326–340. Springer Berlin Heidelberg, 2008.

[33] Pierre Farail and Patrick Gaufillet. TOPCASED : un environnement de développement open source pour les systèmes embarqués. In *Journée de travail* NEPTUNE, *numéro 2, Paris , France*, pages 16–20. Génie logiciel, GL & IS, Meudon, France, Mai 2005.

[34] B. Fontan. *Méthodologie de conception de systèmes temps réel et distribués en contexte UML/SysML*. These, Université Paul Sabatier - Toulouse III, January 2008. 08047 08047.

[35] L. Fuentes-Fernández and A. Vallecillo-Moreno. An introduction to uml profiles. UPGRADE, *European Journal for the Informatics Professional*, 5(2) :5–13, April 2004.

[36] S. Gérard, C. Mraidha, F. Terrier, and B. Baudry. A UML-Based Concept for High Concurrency : The Real-Time Object. In *ISORC*, pages 64–67. IEEE Computer Society, 2004.

[37] D. J. Hatley and I. A. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House Publishing, New York, NY, USA, 1987.

[38] Frédéric Jouault. *Contribution à l'étude des langages de transformation de modèles*. PhD thesis, Université de Nantes, Nantes, France, 2006.

[39] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS : a DSL for the specification of textual concrete sysntaxes in model engineering. In *In GPCE'06 : Proceedings of the 5th international conference on Generative programming and component engineering*, pages 249–254, New York, NY, USA, 2006. ACM.

[40] A. Jovanović, D. Lime, and O. H. Roux. Integer Parameter Synthesis for Timed Automata. In Nir Piterman and Scott Smolka, editors, *19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013)*, volume 7795 of *Lecture Notes in Computer Science*, pages 401–415, Roma, Italy, March 2013. Springer.

[41] B. Kim, I. Lee, L. T. X. Phan, and O. Sokolsky. Platform dependent code generation of real-time embedded software. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ICCPS '13, pages 246–246, New York, NY, USA, 2013. ACM.

[42] Jin Hyun Kim, Jin-Young Choi, Inhye Kang, and Insup Lee. Uml behavior models of real-time embedded software for model-driven architecture. *J. UCS*, 16(17) :2415–2434, 2010.

[43] Jinhyun Kim, Inhye Kang, Jin-Young Choi, Insup Lee, and Sungwon Kang. Formal synthesis of application and platform behaviors of embedded software systems. *Software & Systems Modeling*, pages 1–21, 2013.

[44] Hermann Kopetz. The complexity challenge in embedded system design. In *11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008)*, pages 3–12, Orlando, Florida, USA, May 2008. IEEE Computer Society.

[45] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed fault-tolerant real-time systems : The MARS approach. *IEEE Micro*, 9(1) :25–40, 1989.

[46] P. Kukkala, J. Riihimâki, M. Hännikäinen, T. D. Hämäläinen, and K. Kronlöf. UML 2.0 Profile for Embedded System Design. In *Design, Automation and Test in Europe (DATE'05)*, Vol.2, pages 710–715., Los Alamitos, CA, USA, March 2005. IEEE Computer Society.

[47] Edward A. Lee. Overview of the Ptolemy project. Technical Report UCB/ERL M03/25, EECS Department, University of California, Berkeley, July 2003.

[48] C. Lelionnais and J. Delatour. Towards a Behavioral Modeling of Real-Time Kernel in a Model-Driven Development Approach. In *The 4th Junior Researcher Workshop on Real-Time Computing (JRWRTC 2010) in conjunction with the 18th International Conference on Real-Time and Network Systems (RTNS 2010)*, Toulouse, France, November 2010.

[49] Cédrick Lelionnais, Matthias Brun, Jérôme Delatour, Olivier H. Roux, and Charlotte Seidner. Formal Behavioral Modeling of Real-time Operating Systems. In Leszek A. Maciaszek, Alfredo Cuzzocrea, and José Cordeiro, editors, *In 14th International Conference Enterprise Information Systems - Model Driven Development for Information Systems (MDDIS 2012)*, pages 407–414. SciTePress, June 2012.

[50] Cédrick Lelionnais, Matthias Brun, Jérôme Delatour, Olivier Henri Roux, and Charlotte Seidner. Formal composition based on roles within a model driven engineering approach. In *In 5th International Conference on Advances in System Testing and Validation Lifecycle (VALID 2013)*, pages 27–32, Venice, Italie, November 2013. IARA.

[51] Cédrick Lelionnais, Matthias Brun, Jérôme Delatour, Olivier Henri Roux, and Charlotte Seidner. Formal Synthesis of Real-Time System Models in a MDE Approach. *International Journal on Advances in Systems and Measurement*, 7(1&2) :À venir, 2014.

[52] Didier Lime and Olivier H. Roux. Formal verification of real-time systems with preemptive scheduling. *Real-Time Systems*, 41(2) :118–151, 2009.

[53] Didier Lime, Olivier H. Roux, Charlotte Seidner, and Louis-Marie Traonouez. Romeo : A parametric model-checker for Petri nets with stopwatches. In Stefan Kowalewski and Anna Philippou, editors, *15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, volume 5505 of *Lecture Notes in Computer Science*, pages 54–57, York, United Kingdom, March 2009. Springer.

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

[55] J Liu and E.A. Lee. Timed Multitasking for Real-Time Embedded Software. *Control Systems, IEEE*, 23(1) :65–75, Feb 2003.

[56] J. C. Maeng, D. Na, Y. Lee, and M. Ryu. Model-driven development of rtos-based embedded software. In *Proceedings of the 21st International Conference on Computer and Information Sciences*, ISCIS'06, pages 687–696, Berlin, Heidelberg, 2006. Springer-Verlag.

[57] D. A. Marca and C. L. McGowan. *SADT : Structured Analysis and Design Technique*. McGraw-Hill, Inc., New York, NY, USA, 1987.

[58] Raphaël Marvie, Laurence Duchien, and Mireille Blay-Fornarino. *Les plates-formes d'exécution et l'*IDM, chapter 4, pages 71–86. Number ISBN : 2-7462-1213-7. Hermes, January 2006.

[59] P. M. Merlin. *A Study of the Recoverability of Computing Systems*. PhD thesis, 1974. AAI7511026.

[60] J. Miller and J. Mukerji. Model Driven Architecture (MDA) Guide, version 1.0.1. Technical report, June 2003.

[61] Rania Mzid. *Rétro-ingénierie des plateformes pour le déploiement des applications temps réel*. PhD thesis, Université de Bretagne Occidentale, Brest, France, Mai 2014.

[62] Nicolas Navet. *Systèmes temps réel 1 - Techniques de description et de vérification*. Hermes - Lavoisier, 2006.

[63] Object Management Group (OMG). UML Profile for Modeling and Analysis of Real Time and Embbeded Systems (MARTE), version 1.1. Technical report, June 2011.

[64] Object Management Group (OMG). *Meta Object Facility (*MOF*) Core Specification, version 2.0*, January 2006. http ://www.omg.org/mof/.

[65] Object Management Group (OMG). *Unified Modeling Language (*UML*) : Superstructure, version 2.1.2*, November 2007. http ://www.omg.org/mda/.

[66] OSEK/VDX Group. OSEK/VDX Operating System Specification, version 2.2.3. Technical report, February 2005. http ://www.osek-vdx.org/.

[67] M. Paludetto. *Sur la Commande de Procédés Industriels : une Méthodologie Basée Objets et Réseaux de Petri*. PhD thesis, Université Paul Sabatier, Toulouse, France, Décembre 1991.

[68] F. Peres, B. Berthomieu, and F. Vernadat. On the composition of time petri nets. *Discrete Event Dynamic Systems*, 21(3) :395–424, September 2011.

[69] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Bonn, 1962.

[70] T. H. Phan, S. Gérard, and F. Terrier. Languages for System Specification. chapter Real-time System Modeling with ACCORD/UML Methodology : Illustration Through an Automotive Case Study, pages 51–70. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[71] A. Pinto. Metropolis Design Guidelines. Technical report, University of California, Berkeley, USA, November 2004.

[72] Stefan Resmerita, Patricia Derler, and Edward A. Lee. Modeling and simulation of legacy embedded systems. Technical Report EECS-2010-38, UC Berkeley, April 2010.

[73] Alberto Sangiovanni-Vincentelli and Grant Martin. Platform-based design and software design methodology for embedded systems. *IEEE Des. Test*, 18(6) :23–33, 2001.

[74] Philippe Schnoebelen, Béatrice Bérard, Michel Bidoit, François Laroussinie, and Antoine Petit. *Vérification de logiciels : techniques et outils du model-checking*. Vuibert, April 1999.

[75] Martin Schoeberl, Christopher Brooks, and Edward A. Lee. Code generation for embedded java with ptolemy. In *Proceedings of the 8th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2010)*, October 2010.

[76] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[77] Sally Shlaer and Stephen J. Mellor. *Object-oriented Systems Analysis : Modeling the World in Data*. Yourdon Press, Upper Saddle River, NJ, USA, 1988.

[78] Françoise Simonot-Lion and Yvon Trinquet. *Encyclopédie de l'informatique et des systèmes d'information*, chapter Exemple de systèmes temps réel et choix d'implémentation, pages 723–732. Vuibert, 2006.

[79] Society of Automotive Engineer (SAE). *Architecture Analysis & Design Language (*AADL*) AS5506, version 1.0*, 2004.

[80] Syntec Informatique, RNT Logiciel. *Livre Blanc des premières Assises Françaises du Logiciel Embarqué*. Number 6. Collection ThémaTIC, March 2007.

[81] Tivadar Szemethy and Gabor Karsai. Pml : a transformation language for platform modeling. *ECEASST*, 4, 2006.

[82] S. Taha. *Modélisation Conjointe Logiciel/Matériel de Systèmes Temps Réel*. PhD thesis, Université de Lille 1, May 2008.

[83] S. Taha, A. Radermacher, S. Gérard, and J-L. Dekeyzer. An Open Framework for Hardware Detailed Modeling. In *IEEE proceedings SIES'2007*, number 1-4244-0840-7, pages 118–125, Lisbon, Portugal, July 2007. IEEE.

[84] F. Taïani, M. Paludetto, and J. Delatour. Composing Real-Time Objects : A Case for Petri Nets and Girard's Linear Logic. In *Object-Oriented Real-Time Distributed Computing, 2001. ISORC-2001. Proceedings. Fourth IEEE International Symposium on*, pages 298–305. IEEE, 2001.

[85] The MathWorks. *Real-Time Workshop User's Guide*. The MathWorks Inc., Natick, MA, USA, September 2007.

[86] F. Thomas, J. Delatour, F. Terrier, and S. Gerard. Towards a Framework for Explicit Platform-Based Transformations. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 211–218, May 2008.

[87] Frédéric Thomas. *Contribution à la prise en compte des plates-formes logicielles d'exécution dans une ingénierie générative dirigée par les modéles*. PhD thesis, Université d'Evry, Sarclay, France, Novembre 2008.

[88] Frédéric Thomas, Jérôme Delatour, François Terrier, Matthias Brun, and Sébastien Gérard. Contribution á la modélisation explicite des plates-formes d'exécution pour l'IDM. *L'Objet*, 13(4) :9–32, 2007.

[89] Yvon Trinquet and Jean-Pierre Elloy. Systèmes d'exploitation temps réel. *Techniques de l'ingénieur, Traité Contrôle et Mesures*, Mars 1999.

[90] Thomas Vergnaud and Bechir Zalila. Ocarina, a compiler for the AADL. Technical report, Paris, France, 2006. http ://ocarina.enst.fr.

[91] Didier Vojtisek, François Tanguy, and Cyril Faucher. *Kermeta user interface guide*. Triskell group - IRISA/INRIA.

[92] P. T. Ward and S. J. Mellor. *Structured Development for Real-Time Systems*. Prentice Hall Professional Technical Reference, 1991.

[93] WindRiver. *Vx*WORKS *Programmer's Guide, version 6.9.*, February 2011.

[94] Wayne Wolf and Jorgen Staunstrup. *Hardware/Software CO-Design : Principles and Practice*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

Thèse de Doctorat

Cédrick LELIONNAIS

Titre : Contribution à la considération du comportement des plates-formes d'exécution logicielles temps réel

Title: Contribution to the Consideration of the Behavior of Real-Time Software Execution Platforms

Résumé

La conception et le développement des systèmes temps réels embarqués (SETRS) sont très complexes du fait d'un grand nombre de plates-formes d'exécution. Ces systèmes doivent aussi répondre à des exigences non-fonctionnelles telles que les contraintes temporelles. L'Ingénierie Dirigée par les Modèles (IDM) répond à la diversité de mise en oeuvre de tels systèmes. Plusieurs suites de développement tirent ainsi profit de l'IDM pour générer automatiquement du code spécifique à une plate-forme à partir de descriptions de conception haut niveau des SETRS. De tels outils considèrent alors des exigences non-fonctionnelles en générant des modèles formels pour intégrer des activités de vérification. Cependant, la considération du comportement des plates-formes d'exécution logicielles demeure un problème majeur. D'un côté, les générateurs de code manquent bien souvent de généricité de considération, de l'autre, la génération de modèles formels s'abstrait en partie des mécanismes de ces plates-formes. Est-il alors possible de considérer à la fois génériquement et formellement ce comportement ? Ce travail s'intègre dans une suite d'outils durant une phase de déploiement multiplates-formes. La stratégie adoptée repose sur la composition d'un modèle d'application déployée avec celui du comportement de la plate-forme visée, dans une IDM. Le comportement est traduit formellement avec les classiques réseaux de Petri temporels (ou *Time Petri Net* (TPNs)). Une formalisation est alors proposée pour composer un TPN global indépendamment de la plate-forme considérée. À titre d'exemple, la génération de TPNs d'une application exécutée sur deux plates-formes différentes (OSEK/VDX et VXWORKS) est enfin donnée.

Mots clés

Conception, Systèmes d'exploitation temps réel, Ingénierie Dirigée par les Modèles (IDM), Langage de modélisation dédié, Comportement de plate-forme, Réseaux de Petri Temporels, Déploiement, Modèle formel, Modèle de description de plate-forme.

Abstract

Both design and development of Real-Time Embedded Systems (RTES) are very complex because of the wide range of execution platforms. These systems must also meet non-functional requirements such as time constraints. Model Driven Engineering (MDE) is particularly suitable for handling the diversity of implementation targets. Therefore, several RTES development suites leverage MDE by automatically generating platform-specific code from high-level design models. Such tools may also take non-functional requirements into account by integrating verification activities. These activities typically rely on the generation of formal models from the same high-level design descriptions used for code generation. However, the consideration of the behavior of software execution platforms stays a major problem. On the one hand, code generators often reveal a lack of consideration genericity, on the other hand, the generation of formal models partly overlooks mecanisms of these platforms. How can the behavior both generically and formally be considered ? The presented work is part of a tool suite during a multiplatform deployment phase. The adopted strategy relies on the composition of a deployed application model with a model of the behavior of the targeted platform, within a MDE. Behavior is formally translated with classical Time Petri Nets (TPNs). A formalization is then proposed to compose a global TPN regardless of the considered platform. As an illustration, we finally give a generation example of TPNs describing an application deployed on two different platforms (OSEK/VDX and VXWORKS).

Key Words

Design, Real-Time Operating Systems (RTOSS), Model Driven Engineering (MDE), Domain-specific modeling Language (DSML), Platform behavior, Time Petri Nets (TPNs), Deployment, Formal model, Platform Description Model (PDM).