Déploiement multiplateforme d'applications multitâche par la modélisation

Wassim El Hajj Chehade

Contribution au Déploiement Multiplateforme d'Applications Multitâches par la Modélisation Comportementale Haut Niveau des Services d'Exécution

DOCTORAT EN INFORMATIQUE

par

Wassim EL HAJJ CHEHADE

Soutenue le 4 Avril 2011

devant la commission d'examen composée de :

Jean-Philippe BABAU,	Rapporteur,	LISyC
Laurent PAUTET,	Rapporteur,	Télécom ParisTech
Ansgar RADERMACHER,	Encadrant,	CEA-LIST
François TERRIER,	Directeur de thèse,	CEA-LIST
Guy VIDAL-NAQUET,	Examinateur,	Université Paris Sud et Supélec

Remerciements

Je tiens tout d'abord à remercier François Terrier mon directeur de thèse pour m'avoir accueilli au sein du Laboratoire d'Ingénierie Dirigée par les Modèles des Systèmes Temps Réel Embarqués (LISE) du CEA Saclay. Francois, qui par sa disponibilité et sa compétence, m'a permis de mener à bien ces trois années de recherche.

Je remercie Ansgar, mon encadrant, pour toutes nos discussions enrichissantes menées dans un cadre communicatif et amical.

Je remercie M. Guy Vidal-Naquet qui m'a fait l'honneur de présider le Jury et pour l'intérêt qu'il a manifesté envers mon travail. Je remercie également, M. Jean Philippe Babau et M. Laurent Pautet d'avoir accepter d'être rapporteurs de cette thèse. Leurs remarques pertinentes m'ont permis d'améliorer la qualité de cette thèse.

Je tiens à remercier aussi Bran Selic pour son soutien, ces conseils et ces encouragements.

J'adresse mes remerciements également à l'ensemble des équipes du LISE, les chercheurs et mes collègues doctorants, pour tous les échanges techniques et scientifiques et les pauses café que nous avons partagés pendant ces trois ans.

Enfin, je remercie ma famille pour leur soutien inestimable, en particulier mon père Issam, ma mère Lamia, mon frère Waêl et mes sœurs Rana et Ghina. Merci à vous pour vos encouragement tout au long de ces trois années de thèse.

Table des matières

CHAPITRE 1

Introduction

1 Contexte

Les systèmes multitâches prennent une place de plus en plus importante dans tous les domaines d'application informatique. Ils sont présents dans les domaines comme la télécommunication, l'automobile, l'aéronautique ou encore l'aérospatial, etc. Les concepteurs de ces systèmes ont tirés profit de l'augmentation régulière de la performance des processeurs pour intégrer de nouvelles fonctionnalités, de plus en plus complexes, à ces systèmes. Ainsi, les logiciels obtenus sont composés des fonctionnalités spécifiques au domaine d'application et des implémentations spécifiques à la plate-forme d'exécution. En addition à cette complexité, des contraintes économiques et concurrentielles très pressantes viennent s'ajouter. Ces contraintes obligent les industries à fournir des produits toujours plus novateurs à moindre prix et dans un temps le plus court possible tout en respectant la qualité du produit.

Pour répondre à ces contraintes, les ingénieurs logiciels sont constamment à la recherche des outils, des méthodologies et des infrastructures permettant de réduire la complexité du développement des logiciels, promouvoir leur réutilisation et faciliter l'évolution. L'innovation dans ce domaine est souvent caractérisée par un gain en pouvoir d'abstraction et une capacité à maintenir une séparation de préoccupations entre les principaux composants du système. Ainsi, les couches logicielles, comme les systèmes d'exploitation, ont permis de s'abstraire des préoccupations liée à la plate-forme matérielle. Par conséquent, les développeurs se sont libérés depuis longtemps des complexités associées au développement des applications avec les langages machines. En plus, en se basant sur ces plates-formes logicielles d'exécution, ils sont devenus capable d'automatiser le portage des applications sur différents support matériels.

Malgré les avantages des plates-formes logicielles pour réaliser les implantations des systèmes concurrents, plusieurs problèmes délicats subsistent. Au cœur de ces problèmes il y a la croissance de la complexité de la plate-forme logicielle. En effet, ces plates-formes offrent des bibliothèques contenant plusieurs milliers de classes et de méthodes qui nécessitent des efforts considérables pour comprendre leur rôles, ainsi que l'ensemble de leurs contraintes

d'utilisation. En outre, étant donné que ces plates-formes évoluent souvent rapidement et que de nouvelles plates-formes apparaissent régulièrement, les développeurs dépensent des efforts considérables afin de porter manuellement le code applicatif sur différentes plates-formes ou sur de nouvelles versions de la même plate-forme.

Pour remédier à ces problèmes, une nouvelle approche basée sur une nouvelle montée en abstraction est proposée. C'est l'ingénierie dirigée par les modèles. Elle permet aux concepteurs des applications multitâches de spécifier la structure, le comportement et les exigences de leur application pour un domaine d'une manière indépendante de la plate-forme et d'automatiser ensuite l'implémentation de l'application à l'aide des transformations de modèles.

2 Problématique

Dans ce contexte, le processus de développement dirigé par les modèles vise à générer l'application implantée sur une plate-forme d'exécution. Ce processus s'appuie sur l'ensemble de modèles et de transformations de modèles. L'approche MDA définit un cadre pour ce type de processus, et plus particulièrement pour la mise en œuvre d'une fonctionnalité sur différentes plates-formes d'exécutions à partir des modèles. Partant de descriptions indépendantes des plates-formes (PIM, Platform Independant Model), elle vise à générer des applicatifs dédiés à une plate-forme donnée. En pratique, cela se concrétise par des transformations de modèles capables de spécialiser les modèles pour des plates-formes cibles. Actuellement, les préoccupations spécifiques à ces plates-formes sont décrites totalement ou partiellement dans les transformations elles même. Par conséquence, ces transformations ne sont pas réutilisables et ne permettent pas de répondre aux besoins hétérogènes et évolutifs qui caractérisent les systèmes multitâches.

La montée en abstraction issue de la séparation des préoccupations entre application et plate-forme a amené un gain important : la réutilisabilité de la transformation pour plusieurs applications et le portage des applications. Toutefois, une nouvelle difficulté de portage est apparue : celle du portage de la transformation lorsque la plate-forme cible évolue.

3 Contributions

L'objectif de cette thèse est d'appliquer le principe de séparation des préoccupations au niveau même de la transformation des modèles, une démarche qui veut garantir la portabilité des modèles et la réutilisabilité des processus de transformation. Pour cela, cette thèse contribue sur les deux points :

- **La proposition d'une approche de modélisation de plates-formes** logicielles d'exécution qui permet d'extraire les informations spécifiques à la plate-forme des transformations de modèles et de les encapsuler dans des modèles détaillés de plates-formes. Cette modélisation s'appuie sur le profil SRM de MARTE.
- **L'expérimentation d'un cadre de développement** basé sur les modèles de plates-formes. L'originalité d'un tel cadre réside dans une véritable séparation des préoccupations entre trois acteurs à savoir le développeur des chaînes de transformation, qui spécifient une transformation de modèle générique, les fournisseurs des plates-

formes qui fournissent des modèles détaillés de leurs plates-formes et le concepteur des applications multitâches qui modélise les fonctionnalités du système.

4 Organisation du document

Le chapitre 2 de ce document délimite le contexte de l'étude. Notamment, il présente les systèmes multitâches et l'ingénierie dirigée par les modèles adoptée pour la mise en œuvre des tels systèmes.

Le chapitre 3 présente un état de l'art sur la modélisation de plates-formes logicielles d'exécution et les différents processus IDM utilisés pour la mise en œuvre des systèmes multitâches. Ces processus sont regroupés en se basant sur la manière dont la plate-forme d'exécution est prise en compte lors de déploiement. A la fin de ce chapitre, les approches présentées sont évaluées sur des critères définis dans ce chapitre.

Le chapitre 4 présente le profil SRM de MARTE qui fournit des concepts utiles pour la modélisation des plates-formes logicielles d'exécution. Ensuite, Après identification des besoins envers des transformations de modèles génériques, des heuristiques de modélisation qui se basent sur les concepts SRM sont proposées. Ces heuristiques permettent de modéliser la structure et le comportement des ressources et des services des plates-formes logicielles d'exécution et, par conséquence, de fournir un modèle détaillé de ces plates-formes. L'intérêt d'une telle modélisation est la mise en œuvre des transformations de modèles génériques qui permettent la génération des applications exécutables.

Le chapitre 5 vise à expérimenter l'intégration des modèles de plates-formes détaillés dans ingénierie générative dirigée par les modèles. Il définit une méthode pour modéliser les systèmes multitâches. Ensuite, il décrit une approche réalisant la correspondance entre les modèles applicatifs et ceux des plates-formes cibles. Enfin, il définit une infrastructure de transformation générique.

Le chapitre 6 propose une évaluation cherchant à valider l'intérêt de notre approche. Il évalue notre approche sur un cas d'étude. Il consiste en l'intégration d'une nouvelle plate-forme d'exécution dans le cadre expérimental proposé et l'évaluation de l'effort requis pour une telle intégration. Cette évaluation vise à confirmer les résultats qui ont motivé le développement de notre cadre expérimental.

Enfin, le chapitre 7 résume les contributions et ouvre de nouvelles perspectives sur le travail réalisé.

CHAPITRE 2

Contexte

Ce chapitre présente les caractérisations des applications mise en œuvre dans cette étude et l'ingénierie utilisée pour leur développement. Pour cela, nous introduisons d'abord les caractéristiques clés des systèmes multitâches exécutés sur des plates-formes logicielles d'exécution et dont le développement doit tenir compte de mécanismes de synchronisation et de communication. Nous présentons ensuite le positionnement de l'ingénierie dirigée par les modèles pour le développement des tels systèmes.

1 Caractérisations des applications mises en œuvre

L'objectif de cette section est de mettre en place le domaine des systèmes visés par notre contribution, c'est-à-dire les systèmes multitâches exécutés sur des plates-formes logicielles d'exécution et d'identifier leurs spécificités et leurs exigences afin de mettre en place les composantes principales de notre contexte de travail.

1.1 La programmation multitâche

La programmation multitâche est le nom donné à une technique de programmation qui permet d'exprimer le parallélisme dans une application logicielle. Elle englobe également des techniques pour assurer la communication et la synchronisation entre des entités parallèles [1]. Cette approche est particulièrement prometteuse en ce qu'elle permet d'augmenter la puissance d'expressivité et de réduire le coût de développement par rapport à une application fonctionnant sans parallélisme.

Les programmes multitâches se retrouvent souvent dans des systèmes réactifs. Un système réactif est un système informatique assujetti à l'évolution dynamique d'un procédé qui lui est connecté et qu'il doit piloter (ou suivre) en réagissant à tous ses changements d'état dans un temps fini et déterminé [11]. Ces systèmes sont intrinsèquement concurrents parce que les procédés qu'ils contrôlent sont constitués des entités plus ou moins indépendantes qui travaillent en parallèle. La mise en œuvre des systèmes multitâches nécessitent alors une coordination entre les actions de chaque entité parallèle, qui coopèrent pour résoudre un problème. Cela se concrétise par des traitements mettant en œuvre des moyens tels que la synchronisation, la communication et le partage des ressources [2].

Outre les mécanismes de coordination, les systèmes multitâches peuvent être soumis à des contraintes de sûreté de fonctionnement [3], des contraintes d'enfouissement [4] ou encore des contraintes temporelles [5]. Selon qu'il faille absolument respecter une contrainte lors de l'exécution du logiciel ou qu'il soit possible de la violer occasionnellement, il est usuel de classer les systèmes en trois catégories : stricte, ferme et souple. Ainsi, les systèmes à contraintes strictes doivent impérativement respecter les contraintes imposées. Les systèmes à contraintes fermes admettent une probabilité des réactions ne satisfaisant pas les contraintes mais en garantissant un certain niveau de qualité de service qui maintient le système dans un état sécuritaire. Enfin, les systèmes à contraintes souples tolèrent une probabilité des contraintes non satisfaites.

Dans cette étude, les systèmes visés sont les systèmes multitâches qui peuvent avoir des contraintes temps réel embarquées souples. Cette étude n'aborde pas les problématiques liées à la sûreté de fonctionnement.

1.2 Mise en œuvre des systèmes multitâches

La mise en œuvre des systèmes multitâches peut être réalisée d'une manière logicielle, matérielle ou mixte. En effet, à partir d'un modèle abstrait du système produit lors d'une phase de conception, ce modèle peut être implanté soit par un programme logiciel s'exécutant sur un processeur, soit par des éléments matériels ou encore, avec une approche mixte (*co-design*) où une partie des fonctionnalités est implémentée à l'aide de circuits

spécialisés (ASICs, FPGAs, etc.) [6].

Dans cette étude, il est admis que les systèmes ou l'ensemble des fonctionnalités sont implémentées de manière logicielle. Nous n'abordons donc pas les problématiques liées à la conception conjointe matérielle logicielle.

La mise en œuvre logicielle des systèmes multitâches nécessite des supports d'exécution temps réel ayant pour but d'offrir une gestion des accès aux processeurs, aux ressources et aux réseaux qui constituent le support matériel. Il peut prendre la forme d'un système d'exploitation éventuellement accompagné d'un middleware ou d'une machine virtuelle s'intercalant entre le système d'exploitation et l'application. Les systèmes d'exploitation peuvent exister sur quatre types d'architectures matérielles différentes : (a) monoprocesseur, (b) multiprocesseur, (c) multicœur et (d) distribuée. Sur une architecture monoprocesseur, il n'y a pas de vrai parallélisme puisqu'il n'y a qu'un processeur (unité centrale) qui décode et exécute les instructions. Les architectures multiprocesseurs ou multicœurs peuvent être mises en places pour paralléliser le traitement des données. Ce parallélisme s'étend géographiquement en utilisant des architectures distribuées.

Cette étude ne s'intéresse qu'aux supports d'exécution logiciels de type monoprocesseur multitâches.

Dans les systèmes multitâches et leurs domaines d'application les plus connus, les systèmes réactifs temps réel, il faut garantir le traitement des différents événements qui interviendront, dans des délais adaptés. Pour cela, deux approches peuvent être envisagées [8] :

- L'approche synchrone : la technique de mise en œuvre synchrone consiste à considérer que le temps de traitement des différentes tâches qui composent l'application multitâche est de durée nulle. En effet, tout événement ou interruption arrivant dans le système, est associé à une tâche. Le traitement de celle-ci nécessite un masquage des interruptions jusqu'à sa terminaison. Ainsi, il est logique de faire l'hypothèse qu'il est inutile d'interrompre le traitement d'une tâche par une autre, jugée plus prioritaire pour la sauvegarde du système, puisque la durée effective de traitement des tâches est négligeable donc considérée comme nulle. Ces systèmes sont donc non préemptifs.
- L'approche asynchrone : la technique de mise en œuvre asynchrone consiste à mettre en place un mécanisme d'écoute des occurrences d'événements même pendant l'exécution d'une tâche sur le processeur. De plus, la tâche en exécution peut être interrompue à tout moment au bénéfice d'une autre tâche jugée plus urgente. Cette approche conduit à la nécessité de disposer d'un système d'exploitation temps réel permettant de gérer l'ensemble des tâches de l'application concurrente. Cette gestion consiste d'une part, à mesurer le temps absolu pour pouvoir ainsi se rendre compte du respect des échéances des tâches et d'autre part, décider de l'ordre d'exécution des tâches pour palier le retard dû à l'asynchronisme.

Nous nous intéressons, dans la suite de cette étude, uniquement à cette approche asynchrone.

1.3 Plates-formes logicielles d'exécution

L'architecture logicielle d'un système multitâche peut être décomposée en deux couches. La première consiste en une application concurrente composée d'un ensemble de tâches. La deuxième, de plus bas niveau, joue le rôle d'une plate-forme logicielle d'exécution qui

propose des mécanismes pour exprimer la concurrence dans la structure des programmes et de mécanismes de communication, de synchronisation entre les entités concurrentes.

Tous ces mécanismes peuvent être fournis par des systèmes d'exploitation multitâches ou à travers des couches d'abstraction logicielles qui peuvent s'intercaler entre les systèmes d'exploitation et l'application telles que les machines virtuelles (machine virtuelle Java temps réel) ou les intergiciels. Pour cela, dans cette étude, toutes ces couches d'abstraction logicielles sont représentées par la notion de la plate-forme logicielle d'exécution.

Les mécanismes, ou "primitives" temps réel, fournis par la plate-forme logicielle d'exécution consistent en des mécanismes de [8] :

Gestion des tâches Ils permettent de gérer l'état des tâches dans le système. Les tâches sont tout d'abord "créées" puis réveillées ce qui les positionne dans l'état "prête". La tâche passe après à l'état "exécutée" par une décision d'un ordonnanceur. De cet état, une tâche peut soit être préemptée par une autre tâche, dans ce cas elle retourne dans l'état "prêt", soit être bloquée par une synchronisation, ce qui la fait passer à l'état "attente", soit enfin elle termine son exécution et passe dans l'état "terminée" avant de disparaître du système.

Gestion des interactions entre tâches Ce sont des services dédiés pour communiquer et synchroniser des contextes d'exécution concurrents. Il existe, alors, des services permettant l'échange des données entre les tâches et des services pour synchroniser les exécutions des tâches. Si ces techniques ne sont pas correctement gérées, ils peuvent engendrer des nouvelles sources d'erreur, comme par exemple, l'interblocage [1].

Gestion de l'ordonnancement des tâches Les plates-formes logicielles d'exécution sont capables d'exécuter plusieurs tâches de manière concurrente, en alternant entre les différentes tâches. Une politique d'ordonnancement détermine alors l'ordre dans lequel les tâches doivent s'exécuter. Cette politique est appliquée par un ordonnanceur.

Gestion du temps La notion du temps intégrée dans une plate-forme logicielle d'exécution est dédiée à la représentation et la manipulation du temps et des échelles de temps.

Gestion des interruptions La gestion des interruptions permet de prendre en compte toutes les sollicitations matérielles et logicielles.

Gestion des erreurs Elle propose des mécanismes pour la gestion des modes de fonctionnement des exceptions logicielles.

Dans cette étude, les mécanismes de gestion des tâches et de gestion des interactions entre les tâches sont supposés suffisants pour la mise en place des applications visées par cette étude.

Ces mécanismes sont fournis par des plates-formes logicielles d'exécution. Chaque plate-forme permet de manipuler et d'utiliser ces mécanismes à travers une API. Elle impose aussi des contraintes d'utilisation de ces mécanismes. Le processus de développement doit donc proposer des artéfacts de modélisation pour pouvoir prendre en compte toutes ces caractéristiques dans une ingénierie générative dirigée par les modèles.

1. Un ensemble de tâches est en situation d'interblocage si chacune attend un événement que seule une autre tâche de l'ensemble peut engendrer. Typiquement, c'est le cas si chaque tâche attend une ressource détenue par l'une des autres tâches

2 Caractérisation de l'ingénierie mise en œuvre

Cette section présente la technologie utilisée pour le développement des systèmes définis dans la section précédente.

2.1 Introduction à l'ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles (IDM) est un paradigme qui vise à placer les modèles au centre du développement logiciel afin de faire face à la complexité croissante de la conception et de la production d'un logiciel.

Le principal avantage de l'IDM est qu'elle permet de capitaliser les savoirs faire dans des modèles plutôt que dans des programmes codés manuellement. Cela ne signifie pas une rupture avec les techniques de développement existante mais, au contraire, une complémentarité [13]. En effet, à partir des modèles et en utilisant des techniques de transformation de modèles, le code des programmes peut être généré automatiquement ou semi-automatiquement.

Dans l'IDM, l'élément central est donc le modèle pour lequel il n'existe pas à ce jour une définition unique. Toutefois, des nombreux travaux tels que [14, 15, 16, 17] et les travaux de l'OMG [2] se sont intéressés à cette nouvelle discipline informatique et ont permis de clarifier ces concepts. Ainsi, à partir de ces travaux, les principaux concepts de l'IDM peuvent être identifiés et définis.

2.1.1 Les niveaux de modélisation

Modèle Un modèle est une abstraction d'un système, décrit sous la forme d'un ensemble de faits construits dans une intention particulière. Un modèle doit pouvoir être utilisé pour répondre à des questions sur le système modélisé. Selon Bran Selic [18], pour que le modèle soit utile et efficace, il doit posséder les caractéristiques suivantes :

- Avoir un certain niveau d'abstraction qui le rend indépendant des technologies de mise en œuvre.
- Être exprimé dans une forme assez compréhensive pour qu'il soit compris de manière intuitive.
- Fournir une représentation précise et fidèle de caractéristiques réelles du système modélisé.
- Être moins coûteux et plus facile à développer que le système qu'il représente.

Pour que ces modèles possèdent ces caractéristiques, il faut disposer des techniques précises de mise en œuvre des tels modèles. Par exemple, dans le domaine du logicielle, des techniques qui permettent à ces modèles d'être interprétables par les machines qui va les exécuter. Pour cela, il faut que les modèles soient conformes à un langage clairement défini. Dans l'IDM, ce langage de modélisation est dit *métamodèle*.

Métamodèle Un métamodèle est une abstraction d'un langage permettant de décrire des modèles. Il définit les éléments que l'on va retrouver dans le langage utilisé pour écrire un modèle ainsi que les relations qu'il existe entre ces éléments. Dans l'IDM, le principe

2. The Object Management Group (OMG), cf. http ://www.omg.org/

de base est “tous est modèle" [13]. Par conséquence, le métamodèle est un modèle. Il doit aussi être conforme à son métamodèle, appelé le métamétamodèle.

Métamétamodèle Un métamétamodèle est un modèle d’un langage de modélisation permettant de décrire des métamodèles. Ce métamétamodèle est aussi un modèle qui doit être conforme à un métamodèle. Pour arrêter ce schémas infiniment récursif le métamétamodèles s’auto défini, c’est à dire qu’il est conforme à lui même. MOF (Meta-Object Facility) [30] standardisé par l’OMG est un tel métamétamodèle, il définit des concepts de base comme classe et relation qui peuvent tout définir y compris eux-mêmes.

Dans l’IDM, deux relations fondamentales peuvent être identifiées entre les différents niveaux de modélisation [13]. D’après la définition d’un modèle, nous pouvons déduire la première relation qui lie le modèle et le système qu’il représente. Cette relation est appelée “Représente" et notée μ. La deuxième relation peut être déduite de la définition du métamodèle présentée ci-dessous. C’est une relation liant le modèle et le langage utilisé pour le construire, elle est appelée “Conforme" à et notée χ.

La figure 2.1 illustre la relation de conformité qui lie un modèle à son métamodèle. On y distingue quatre niveaux de modélisation, M0 correspond au monde réel, qu’on représente en M1 par un modèle contenant une classe Personne avec un attribut nom de type *String*. Ce modèle est donc conforme au métamodèle du niveau M2 qui à son tour est conforme au métamétamodèle du niveau M3. Ce dernier définit le concept de Métaclasse, de Métarelation et les relations source et cible qui les lient. M3 a donc cette capacité de s’auto-définir puisque Métaclasse et Métarelation sont des Métaclasse(s) et les relations source et cible sont des Métarelation(s).

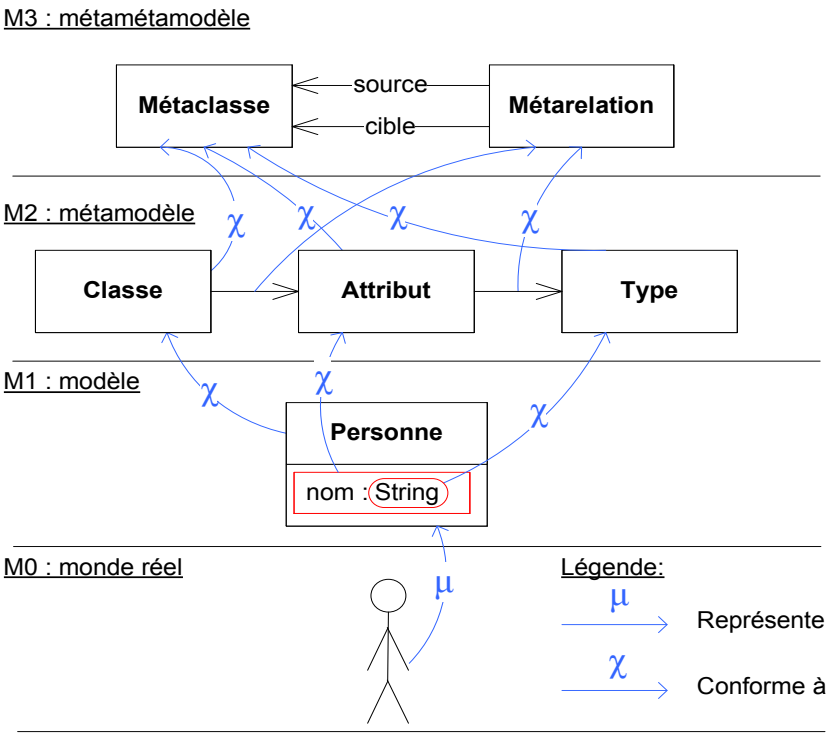

FIGURE 2.1 – Les niveaux de modélisation

2.1.2 Transformation de modèles

La transformation de modèles est une technique qui sert à manipuler un modèle source d'un système pour produire un autre modèle cible de ce même système. Que les métamodèles sources et cibles soient identiques (transformation endogène) ou différents (transformation exogène), l'intérêt de transformer un modèle source en un modèle cible apparaît comme primordial (génération de code, raffinement, refactoring, migration technologique, etc.) [13].

Parmi les différentes manipulations offertes par la transformation de modèles, dans cette étude seule les techniques de transformations de modèles en modèles et de génération de code sont utilisées.

Transformation de modèles en modèles

Ce type de transformation est utilisé pour passer d'un modèle d'un système à un autre modèle du même système. Le but est alors de raffiner les modèles ou d'améliorer leurs qualités [19].

Les langages de transformation de graphes sont utilisés. Ces langages de transformation fournissent des métamodèles de transformation (MMt) à partir desquelles des modèles de transformation (Mt) sont décrits. Ces métamodèles sont composés de règles de transformation manipulant explicitement les éléments des métamodèles source (MMs) et cible (MMc). La figure 2.2 illustre la mise en œuvre des transformations de modèles.

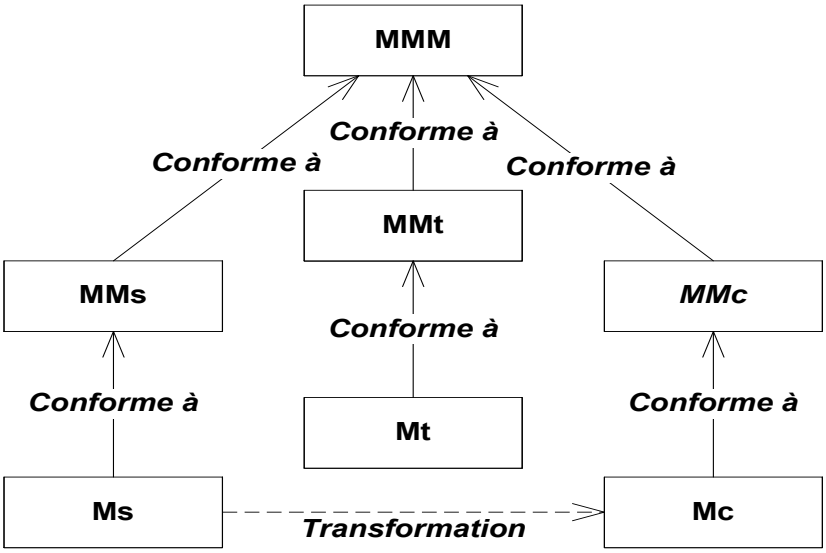

FIGURE 2.2 – Transformation de modèles

Parmi les langages de transformation de graphes, nous pouvons citer QVT (Query/View/Transformation) [20] qui est un standard défini par l'OMG et ATL (Atlas Transformation language) [21]. Les règles de transformation écrites avec ces langages sont exécutées par les moteurs de transformation qui sont associés à ces langages, comme par exemple, la machine virtuelle ATL.

Génération de code

La génération de code au sens large consiste à récupérer des informations contenues dans un modèle, et leur injecter dans différents fichiers textuels après un éventuel traitement qui dépend de la cible. Par exemple, générer de la documentation HTML ou du code source dans un langage de programmation donné.

La génération automatique de code offre de nombreux avantages pour le développement des logiciels, y compris une productivité accrue et une meilleure consistance du code source produit [38]. Elle tend aussi à améliorer la réutilisabilité et la portabilité des modèles, car la relation des modèles avec la plate-forme cible ne dépend plus que de l'existence d'un compilateur du langage de programmation pour la plate-forme cible.

Pour transformer un modèle en une représentation textuelle (du code), plusieurs langages dédiés à la génération de code existent tels que Acceleo [22], XPand [23] ou encore Java Emission Template (JET) [24] (également appelé langage de *templates*). L'utilisateur peut utiliser indifféremment le langage qu'il souhaite pour écrire sont propre générateur de code. Notant qu'il existe des outils de générations de code commerciaux qui permettent la génération automatique de code, comme par exemple, l'outil Rational Rose Technical Developer [25] et l'outil Rhapsody [26].

2.2 Mise en œuvre de l'ingénierie dirigée par les modèles

Différentes approches centrées sur les modèles [27, 28, 29, 31] ont été proposées pour mettre en pratique les principes de l'IDM. Nous revenons ici sur celle que nous avons adopté tout au long de cette étude.

2.2.1 L'approche MDA

Face à la complexité croissante des applications, les modèles développés sont devenus de plus en plus difficiles à réaliser et à maintenir. De plus, la variété et l'évolution rapide des plates-formes d'exécution ont accentué le besoin de réutilisation et d'interopérabilité des modèles. En 2000, l'OMG a lancé son initiative MDA (Model Driven Architecture) [31] afin de promulguer de bonnes pratiques de modélisation et d'exploiter pleinement les avantages des modèles. Cette approche vise à mettre en valeur les qualités intrinsèques des modèles, telles que la portabilité, l'interopérabilité, la maintenance et la réutilisation.

Le principe du MDA consiste à décrire séparément des modèles pour les différentes phases du cycle de développement d'une application. Plus précisément, le MDA préconise l'élaboration de trois modèles, le PIM (*Platform Independent Model*), le PDM (*Platform Description Model*) et le PSM (*Platform Specific Model*). Le PIM décrit la structure de l'application d'une manière indépendante de la plate-forme. Cette structure est ensuite transformée, en utilisant des transformations de modèles, en un modèle d'application spécifique à une plate-forme particulière. Le PDM est quant à lui un modèle de la plate-forme qui permet d'exécuter l'application.

En se basant sur l'approche MDA, la réalisation du portage d'une application d'une plate-forme vers une autre se concrétise par une transformation de modèles qui prend comme entrée la description de l'application (PIM) et sera guidée par le modèle de la

nouvelle plate-forme cible (PDM). Ainsi, il suffit de garder le même modèle PIM et de varier les PDMs pour générer différent PSMs. La figure 2.3 illustre la description d'une telle approche.

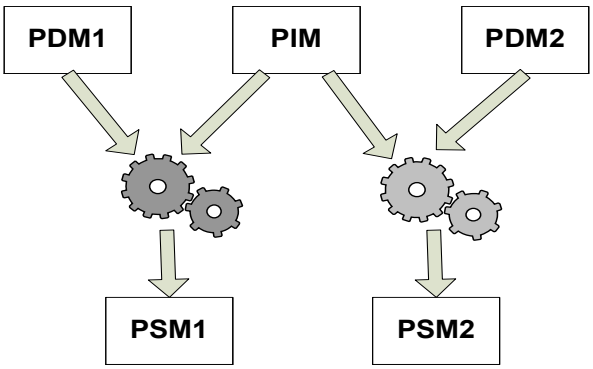

FIGURE 2.3 – Approche MDA

Cependant, comme l'approche MDA utilise le concept de plate-forme sans en donner une définition précise, cela ouvre la possibilité à plusieurs interprétation de la notion de description de la plate-forme (PDM). En effet, lors du passage du PIM vers PSM à l'aide d'une transformation, le PDM peut être décrit soit implicitement dans la transformation et donc le PDM fait partie intégrante de la transformation, soit il est décrit explicitement dans des modèles à part, et il sera alors passé en entrée de la transformation, avec le PIM.

2.2.2 Langages de modélisation

Dans la théorie classique des langages, les grammaires classiques, telles que l'EBNF [32], permettent d'identifier les concepts que l'on va exprimer à travers le langage et les règles de syntaxe de ce langage, c'est à dire la façon correcte de construire les phrases de ce langage. Dans l'IDM, la notion du métamodèle est proche de la notion de grammaire. Ainsi, un métamodèle spécifie la syntaxe abstraite du langage de modélisation. Cela regroupe l'ensemble des concepts du langage et les relations entre ces concepts.

Dans l'IDM, il existe deux approches pour concevoir des langages de modélisation. La première approche consiste à créer un métamodèle pour chaque domaine que l'on souhaite modéliser (*Domain Specific Language* ou DSL). Cela veut dire qu'on crée un langage sur mesure collant parfaitement aux concepts du domaine qu'on désire représenter. La seconde approche consiste à créer des langages de modélisation dits généralistes. Le langage UML (Unified Modeling Langage) [37], standardisé par l'OMG, est aujourd'hui le plus représentatif des langages de modélisation généralistes. Chacune de ses deux approches à ses avantages et ses inconvénients [33], et le choix d'une approche plutôt que de l'autre devra se faire en tenant compte des contraintes liées au contexte dans lequel le langage sera utilisé.

Dans cette étude, le langage UML permet de satisfaire l'ensemble de nos besoins de modélisation. Cela, nous évitera des redéfinitions de la structure, la sémantique et la notation du grand nombre des concepts qui sont déjà définis dans UML.

UML est un langage de modélisation généraliste et son métamodèle vise à couvrir tous

les aspects de la modélisation logicielle. Cependant, il arrive que son utilisation telle quelle ne soit pas adaptée pour modéliser les concepts d'un domaine bien particulier. Conscient de ce problème, l'OMG a introduit la possibilité de le spécialiser par le mécanisme de profil. C'est un mécanisme simple et efficace pour étendre UML, à l'aide d'un ensemble de stéréotypes. Un stéréotype permet d'étendre toute métaclasse du langage UML pour enrichir la sémantique (sauf la métaclasse stéréotype elle-même [3]). Il peut posséder des propriétés (*tag definition*) et être sujet à des contraintes OCL (Object Constraint Language) [46]. Ainsi, un éditeur UML évolué sera capable d'interpréter ces contraintes et de signaler une erreur de modélisation.

L'OMG propose deux mécanismes d'extension d'UML. Le mécanisme dit lourd (heavyweight mechanisms) et le mécanisme dit léger (lightweight mechanims).

Mécanisme lourd Il se base sur une copie du métamodèle UML qui sera ensuite manipulé en lecture et en écriture. En d'autre termes, il permet d'importer des éléments du métamodèle UML afin d'ajouter, de supprimer ou de modifier leurs caractéristiques et leurs sémantiques. Pour cela, ce mécanisme utilise le dispositif *merge* du MOF et toutes les modifications sont alors réalisées sur la copie des métaclasses.

Mécanisme léger Il manipule le métamodèle UML en lecture seule. De ce fait, il n'est pas possible de supprimer, modifier ou contredire un de ses éléments.

3 Conclusion

Ce chapitre a présenté le contexte de cette étude. Dans un premier temps, il a spécifié la nature des systèmes qu'on souhaite développer. Ce sont des systèmes multitâches monoprocesseurs s'exécutant sur des plates-formes logicielles d'exécution. Dans un second temps, il a décrit l'ingénierie de développement mise en œuvre, c'est-à-dire l'ingénierie dirigée par les modèles, et plus particulièrement l'approche MDA, dans laquelle les plates-formes logicielles d'exécution doivent être prises en compte (Modélisation UML supportée par des profils).

Le but de cette étude est donc d'utiliser les modèles afin de gérer les activités nécessaires pour le développement des systèmes multitâches. Entre autres, elle préconise l'utilisation de modèles et de transformations afin de :

1. décharger les développeurs d'application des activités de codage fastidieuses
2. libérer les développeurs des chaînes de transformation de modèle de leur dépendance avec les plates-formes logicielles d'exécution cibles.

Pour cela, cette étude doit répondre au deux questions suivantes :

1. Comment avoir un modèle de plate-forme logicielle d'exécution qui décrit la structure et les aspects comportementaux spécifiques aux ressources et services de la plate-forme ?
2. Comment intégrer ces modèles des plates-formes dans une ingénierie générative dirigée par les modèles de type MDA ?

3. En empêchant la spécialisation de la notion de stéréotype, UML s'empêche de modifier son mécanisme de spécialisation, ce qui permet de pouvoir interpréter un profil de manière systématique, mais qui a aussi pour effet de limiter les possibilités de spécialisation.

Le chapitre suivant propose un état de l'art sur les différents travaux existant qui se sont intéressés par cette problématique et positionne les contributions de cette étude vis-à-vis de ces travaux.

CHAPITRE 3

État de l'art

L'objectif de ce chapitre est de positionner les contributions de cette étude face aux travaux existants. Pour cela, ce chapitre est divisé en deux sections. La première définit le concept de la plate-forme dans l'IDM et présente les besoins de modélisation des plates-formes dans une ingénierie générative dirigée par les modèles. Ces besoins sont utilisés comme critères d'évaluation de différentes approches de modélisation des plates-formes logicielles d'exécution multitâche. La deuxième section s'intéresse aux processus de déploiement des applications sur les plates-formes. Tout d'abord, elle introduit les facteurs qui permettent de juger la qualité d'un processus de déploiement des applications multitâches. Ces facteurs sont utilisés comme critère de comparaison. Ensuite, elle présente l'état de l'art sur les différents processus de déploiement des applications multitâches. Elle les décrit succinctement puis les confronte aux critères de comparaisons.

1 Les plates-formes d'exécution

Cette section s'intéresse à la modélisation des plates-formes logicielles d'exécution. Tout d'abord, elle introduit le concept de plate-forme d'exécution dans l'IDM, recense les besoins de modélisation d'une telle plate-forme, et présente les principales approches de modélisation existantes.

1.1 Définition des plates-formes d'exécution

Un des aspects les plus intéressants de l'approche IDM est la séparation des aspects métiers des aspects technologiques. Cette séparation permet de faire évoluer de manière indépendante, mais pas gratuite, chacun de ces aspects. Dans la terminologie de l'ingénierie des modèles, la notion de technologie est généralement désignée sous le terme de plate-forme, dont il n'y a pas une définition précise pour le moment.

Historiquement, le concept de plate-forme a été utilisé pour identifier la structure matérielle permettant d'exécuter le logiciel. L'apparition de couches logicielles a étendu l'usage de ce concept pour décrire de la même façon les supports logiciels. Ainsi, les machines virtuelles, les intergiciels et les exécutifs sont des exemples des plates-formes logicielles d'exécution.

L'OMG propose une définition de la notion de plate-forme qui permet de synthétiser les différentes interprétations. Ainsi, dans la norme *Model Driven Architecture* [31], une définition est proposée dont la traduction peut être :

Une plate-forme est un ensemble de sous-systèmes et de technologies qui fournissent un ensemble cohérent de fonctionnalités au travers d'interfaces et la spécification de patrons d'utilisation que chaque sous-système qui dépend de la plate-forme peut utiliser sans être concerné par les détails et sur la façon dont les fonctionnalités sont implantées.

Cette définition souligne deux caractéristiques principales des plates-formes, les interfaces et les patrons d'utilisation. Par conséquence, toute modélisation de la plate-forme doit prendre en compte ces deux caractéristiques.

Dans ce contexte, Atkinson et Kuhne [15] et Marvie et al. [61] ont identifié les besoins de modélisation de ces plates-formes en tenant compte de ces deux caractéristiques. Leurs travaux soulignent une volonté de modélisation structurelle et comportementale d'une plate-forme d'exécution. Ainsi, dans le respect des besoins spécifiés par ces travaux, Frédéric Thomas [75] identifie quatre axes pour modéliser l'interface de programmation d'une plate-forme logicielle d'exécution :

1. La caractérisation des concepts de la plate-forme tels que les ressources et instances de ressources offertes par la plate-forme.
2. La caractérisation de traitements qui sont les services offertes par les ressources.
3. La caractérisation des règles d'utilisation telles que les patrons d'utilisation des ressources et les comportements des mécanismes et des services observables depuis l'interface de programmation.
4. Les contraintes d'utilisation (les règles imposées par la plate-forme dont le non respect peuvent entraîner une mise en œuvre erronée et non fonctionnelle de l'application).

Dans cette étude, on s'est focalisé sur les trois premiers axes définis. Donc, *nous devons pouvoir définir une approche de modélisation de plate-forme qui permet de décrire la structure de la plate-forme (les ressource et les services) et encore les comportements observables des ressources et des services de cette plate-forme.* Le quatrième axe de modélisation n'est pas abordé dans cette étude. Il sera traité dans les travaux futurs qui succéderont cette thèse.

1.2 Modélisation des plates-formes d'exécution

Plusieurs études se sont intéressées à fournir des concepts destinés à la modélisation explicite des plates-formes logicielles. Ces concepts peuvent être réalisés par des extensions au métamodèle UML, c'est-à-dire des profils UML ou par des langages spécifiques au domaine du temps réel multitaches (Domain Specific Language, DSL).

1.2.1 TUT-Profile

Le profil TUT [68] initialement prévue pour la conception des systèmes temps réel embarqués et la modélisation des plates-formes d'exécution matérielles. Ce profil a été étendu dans [69] pour permettre la modélisation des plates-formes logicielles d'exécution. Ainsi, le *SwPlatform* décrit la plate-forme dans sa globalité, le *SwPlatformLibrary* décrit un package regroupant l'ensemble des constituants de la plate-forme. Ces constituants sont représentés par des classes UML et sont annotés par *SwPlatformComponent*. Les éléments typés par ces constituants sont annotés par *SwPlatformProcess*.

Cette approche de modélisation permet d'identifier explicitement les packages jouant le rôle de la plate-forme et les entités contenues dans cette plate-forme. Cependant, les mécanismes et les services de la plate-forme restent totalement implicites et les aspects comportementaux sont codés dans des bibliothèques logicielles (*run-time library*) indépendamment du modèle de la plate-forme [70]. Ainsi, dans la figure 3.1, TUT-profil est utilisé pour modéliser la plate-forme C++/POSIX, on voit clairement que toutes les ressources, de concurrence et d'interaction, sont annotées par le même stéréotype (SwPlatformComponent), et les services fournis par ces ressources ne sont pas modélisés. Une telle modélisation ne permet pas de capturer explicitement, dans les modèles des ressources, les aspects comportementaux propres à chaque ressource de la plate-forme.

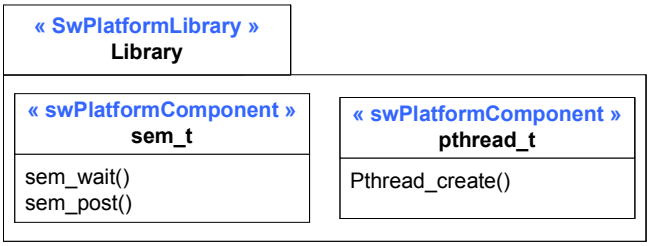

FIGURE 3.1 – Description d'un extrait de C++/POSIX avec TUT-Profile

1.2.2 Platform Modeling Language

PML (Platform Modeling Language) [80] est un métamodèle dédié à la description des modèles de plates-formes. Ce métamodèle définit le concept de *PlatformModel* composé des composants (*componentModel*) et des règles de transformation entre les composants de plates-formes source et cible (*Pattern*, *Match*). Dans cette approche un modèle de la plate-forme est typiquement représenté par un diagramme de classe UML. PML permet, entre autre, de modéliser les ressources et la signature des services, leurs cycles de vie, ainsi que les contraintes d'utilisation.

Dans cette approche, le même métamodèle est utilisé pour décrire l'application, les plates-formes et les transformations entre ces plates-formes. Il est intégré dans l'atelier de modélisation GME [1]. Cependant, dans ce métamodèle, il n'y a aucun concept dédier à l'identification des mécanismes et des services métiers (la concurrence, l'interaction).

1.2.3 AADL

Le langage AADL (*Architecture Analysis and Design Language*) [36] est un standard promu par la SAE [2] afin d'aider à l'analyse et la conception des systèmes temps réel embarqués. AADL offre la possibilité de décrire l'architecture complète, logicielle et matérielle d'un système embarqué. Ainsi, un modèle AADL est un ensemble hiérarchisé de composants, chacun disposant de son propre type ou interface. Le standard propose différents types de composants matériels et logiciels. Les composants *data*, *thread*, *thread group*, *process* sont des composants logiciels. Les composants *processor*, *device* et *bus* sont des composants matériels. La mise en œuvre de ces composants est régie par des règles de traduction de ces entités vers des langages de programmation. Ces règles de traduction, ainsi que les aspects comportementaux de certains composants d'un modèle AADL sont détaillés dans la norme elle-même.

Le langage AADL permet de modéliser explicitement une application logicielle et des plates-formes d'exécution matérielles. Les composants logiciels proposés pour modéliser les applications logicielles possèdent des sémantiques d'exécution particulière, spécifiées dans la norme. AADL peut être considérée comme une plate-forme d'exécution abstraite. Elle ne peut pas être utilisée pour décrire des plates-formes logicielles d'exécution.

1.2.4 Metropolis

Le projet Metropolis [76], fournit une méthodologie et une infrastructure pour la conception des systèmes embraqués. Le point clé de cette approche est un métamodèle qui offre des concepts de haut niveau permettant de décrire à la fois l'application et la plate-forme et les relations entre eux. Le métamodèle Metropolis représente un ensemble de *Process* reliés entre eux par des *Interface* de communication à travers différents *Media*. L'accès au *Media* est contrôlé par un *Quantity Manager*. Tous sont regroupés dans un *Netlist*.

Dans les travaux présentés dans [77], la méthodologie Metropolis a été adaptée à

1. http ://www.isis.vanderbilt.edu/projects/gme/
2. *Society for Automotive Engineers*

l'approche IDM. Dans ces travaux, un profil UML inspiré du métamodèle Metropolis, appelé *Platform profile*, est proposé pour modéliser les plates-formes logicielles et matérielles. Ce profil inclus plusieurs stéréotypes comme par exemple : *Netlist*, *Medium Resource*. Ce dernier est spécialisé par les stéréotypes *Bus*, *Memory* et *Process*. Les services des ressources sont représentés par des opérations Java. Le comportement des ressources et des services sont décrit par ces opérations.

Le métamodèle Metropolis offre des concepts nécessaires à la modélisation explicite de plates-formes logicielle d'exécution de tout domaine. Néanmoins, la généricité des concepts offerts ne facilite pas la description des plates-formes dédiées à un domaine particulier [78]. Ce sont des concepts à un haut niveau d'abstraction qui n'identifient pas, par exemple, les différentes familles de mécanismes et de services d'une plate-forme logicielle temps réel. Cette description de haut niveau limite l'intégration des modèles résultants dans des ingénieries génératives paramétrables et réutilisables.

1.2.5 RTEPML

RTEPML (*Real-time Emdedded Platform Modeling Language*) [79] est un langage de modélisation des plates-formes logicielles d'exécution. Il permet de décrire les ressources qu'offre le système d'exploitation temps réel ainsi que l'API des services associés à ces ressources. Avec ce langage, des prototypes de mise en œuvre des mécanismes exécutifs peuvent être précisés.

Ce langage se base sur le modèle du domaine du package SRM du profil UML-MARTE et fournit une implantation alternative du modèle du domaine SRM dans un contexte DSML. Cette implantation alternative a imposée la description d'un outillage spécifique pour la représentation des modèles de plates-formes et ceci pour avoir les mêmes capacités de modélisation que le profil SRM. Les aspects comportementaux des ressources et des services ne sont pas abordés dans ce langage.

1.2.6 UML-MARTE

Le profil UML-MARTE [47], standardisé par l'OMG, permet de modéliser des plates-formes d'exécution matérielles ou logicielles à différents niveaux d'abstraction. Ainsi, au sein de ce profil, le package GRM (*Generic Resource modeling*) offre des concepts généraux permettant la modélisation des plates-formes. Ces concepts généraux sont ensuite spécialisés dans les packages SRM (*Software Resource Modeling*) et HRM (*Hardware Resource Modeling*) pour permettre la modélisation précise des plates-formes logicielles et matérielles respectivement.

Le package SRM fournit des artéfacts de modélisation pour décrire une plate-forme d'exécution temps réel embarquée. Ces artéfacts spécialisent les concepts généraux de ressources et de services du package GRM (*Generic Resource Modeling*) déjà présents dans le profil SPT (Schedulability, Performence and Time) [53] que MARTE se destine à remplacer. Cette spécialisation a permis de décrire finement trois familles de ressources logicielles : les ressources d'exécution concurrentes, les ressources d'interaction et les ressources de gestion. Par exemple, la modélisation de la plate-forme C/POSIX avec SRM, permet de

capturer toutes les ressources et les services de cette plate-forme. Ainsi, dans la figure 3.2, la ressource concurrente *pthread_t* est décrit par le stéréotype *SwSchedulableResource* et l'opération *pthread_create* qui permet de créer le thread, est référencée par la propriété *createServices* ce stéréotype. De même pour la ressource de synchronisation *sem_t*.

Avec cette description détaillée des services et des ressources de la plate-forme, le profil SRM peut être utilisé pour capturer des aspects comportementaux. Ainsi, dans les travaux de Frédéric Thomas qui ont précédés cette étude [75], des motifs de conception des ressources ont été décrit par des classes structurées annotés par des stéréotypes SRM. Ces classes structurées possèdent des parties (*Part*) et des opérations (*Operation*). Ces opérations sont associées à des activités UML qui décrivent leurs comportements.

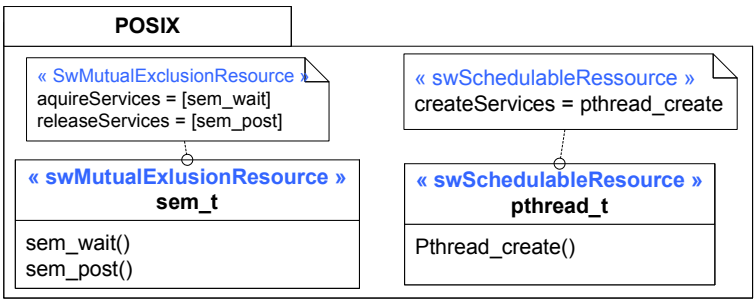

FIGURE 3.2 – Description d'une application avec le profil UML for C

1.3 Synthèse

La figure 3.3 synthétise et compare les différentes approches de modélisation des plates-formes logicielles d'exécution multitâches. Cette comparaison porte sur la capacité de ces approches à modéliser la structure de la plate-forme, en termes de ces ressources et ces services offerts, et notamment la modélisation des ressources de concurrence (Conc.) et les ressources des interactions (Inter.). La comparaison porte aussi sur la capacité des différentes approches à décrire le comportement des ressources et des services de la plate-forme.

Les approches de modélisation, AADL et *Platform profil* (Metropolis) proposent des artéfacts incomplets ou de très haut niveau pour modéliser les plates-formes logicielles d'exécution multitâche. Les modèles résultants sont des modèles de très haut niveau d'abstraction qui ne permettent pas la mise en place des outils automatique d'analyse ou de génération.

Parmi les approches DSL, nous remarquons que le langage RTEMPL est le plus approprié pour la modélisation des plates-formes. Il offre des concepts complets pour la modélisation des ressources et des services. Ceci est dû au fait que ce langage est une implantation alternative du modèle du domaine du profil SRM de MARTE. Ce dernier offre actuellement l'ensemble le plus complet des concepts pour modéliser les plates-formes logicielles d'exécution multitâches.

Par rapport à la modélisation des aspects comportementaux spécifiques aux ressources et services de la plate-forme, le langage PML les décrit dans des règles de transformation et

	Approches	Ressources		Services	Comportement
		Conc.	Inter.		
UML	TUT-Profile	×	×	×	×
UML	Platform profile	✓	✓	×	✓
UML	SRM	✓	✓	✓	✓
DSL	AADL	✓	×	×	×
DSL	PML	×	×	×	✓
DSL	RTEPML	✓	✓	✓	×

FIGURE 3.3 – Comparaison des approches de modélisation des plates-formes logicielles d'exécution multitâche

le langage *Platform Profile* basé sur Metropolis les supporte dans des opérations d'un pseudo code Java. Aucune des approches, à l'exception du SRM, ne permettent de décrire ces aspects comportementaux explicitement dans le modèle de plates-formes. Une telle description doit faciliter l'intégration de ces modèles de plates-formes dans une ingénierie générative permettant la génération d'un code exécutable.

Dans la suite de cette étude, nous allons utiliser le profil SRM comme base pour modéliser la structure de la plate-forme (les ressource et les services) et encore les comportements observables des ressources et des services de cette plate-forme.

2 Prise en compte des plates-formes lors de déploiement des applications

Cette section présente les principales approches de déploiement des applications sur des plates-formes d'exécution. Premièrement, elle définit les critères de comparaison. Ensuite, elle identifie les principales approches de déploiement sur les plates-formes d'exécution. Enfin, elle synthétise les apports et les lacunes des ces approches pour positionner les contributions de cette thèse.

2.1 Définition des critères de comparaison

Le but principal de cette section consiste à définir des critères qui permettent de comparer les différentes approches MDA proposées pour le déploiement des systèmes multitâches.

L'un des points clés le plus important de l'approche MDA est la transformation de modèles [42]. Les avantages espérés de l'utilisation des transformations consistent en une réduction des coûts de développement, une amélioration de la qualité des logiciels, et la facilitation de l'évolution et la migration des logiciels, contribuant ainsi à la limitation des coûts de portabilité et de maintenance des applications multitâches. Ces avantages espérés dépendent fortement de la manière dont cette transformation de modèle est réalisée. Pour cela, les transformations de modèles ne doivent pas être considérées comme des détails

d'implémentation mais au contraire, comme un projet de génie logiciel avec des exigences de qualité claires [39].

Plusieurs travaux se sont intéressés à définir des critères de qualités quant à la réalisation des transformations de modèles. Ces critères permettent d'évaluer et comparer la qualité des différentes transformations de modèles. Ils peuvent donc être utilisés pour évaluer et comparer les transformations de modèles dans les approches dirigées par les modèles dédiées au développement des systèmes multitâches et dont leur mise en œuvre dépend de la manière dont les plates-formes d'exécutions sont considérées.

Verro et Pataricza [40] soulignent que les transformations doivent être considérées comme des modèles et leur réutilisation, maintenabilité, performance et leur compacité doit être envisagée. Van Amstel et al. [41] identifient des critères pour évaluer la qualité des transformations de modèles. Elles sont inspirées des critères proposés pour le développement des logiciels [43]. Ces qualités portent sur la compréhension, la réutilisation, la complétude des résultats et la consistance de la transformation de modèle. Mohagheghi [44] souligne l'importance que le modèle produit à la sortie de la transformation conserve les propriétés du modèle d'entrée, c'est à dire, la transformation produit des modèles cohérents.

En résumé, tous les travaux qui se sont intéressés à la définition des critères de qualité pour la mise en œuvre des transformations de modèles partent de l'hypothèse que le développement des telles transformations est considéré comme une sorte de développement de logiciels. Par conséquence, les exigences de qualité définies pour le développement de logiciels doivent être respectées lors de la construction des transformations de modèles. Ainsi, en analysant les exigences à respecter lors de la construction d'un logiciel, nous pouvons déduire d'où proviennent les critères de qualités proposés par les travaux cités au dessus et par conséquence, déduire nos critères de qualité avec lesquels, nous allons comparer les différentes approches de transformation de modèles.

Dans l'ingénierie des logiciels, les ingénieurs cherchent constamment des technologies de développement et des méthodologies qui permettent de réduire la complexité du logiciel, d'améliorer la compréhension, promouvoir la réutilisation et faciliter l'évolution des applications [60]. Le respect de ces exigences implique des critères spécifiques qu'il faut prendre en compte lors de la mise en œuvre des logiciels et, par conséquence, des transformations de modèles.

Ainsi, en se basant sur le respect de ces exigences, nous pouvons caractériser les critères de qualité qu'il faut adopter lors de la mise en œuvre d'une transformation de modèle par :

- La réduction de la complexité et l'amélioration de la compréhension exigent des mécanismes de décomposition [60]. Cela se concrétise par la séparation de préoccupations lors du développement du logiciels. Dans le cas de la mise en œuvre des transformations de modèles, la décomposition se manifeste en particulier par une séparation entre les informations spécifiques à la plate-forme d'exécution et la réalisation de la transformation de modèles.
- La réutilisation d'une transformation de modèle nécessite l'élaboration de règles de transformation indépendantes du modèle d'entrée et de la plate-forme cible. La réutilisation de ces règles de transformation ne doit pas impacter les propriétés et les comportements des systèmes lors du changement de la plate-forme cible. Dans le contexte d'un système multitâche, la réutilisation doit conserver les propriétés

temporelles et le comportement concurrent de ce système.
– La facilitation de l'évolution d'une transformation de modèles exige des mécanismes pour minimiser l'impact des changements et de substituabilités sur le modèle produit à la sortie de la transformation en garantissant toujours une traçabilité entre les éléments du modèle d'entrée et leurs équivalents dans le modèle de sortie.

En addition à ces critères, la transformation de modèles doit permettre la génération d'un modèle de sortie complet. Cela veut dire, transformer un modèle source en un modèle cible avec toutes les fonctionnalités de mises en œuvre bien définies pour permettre la génération d'un code exécutable. Cela permet de libérer complètement le concepteur du système de toute préoccupation liée au déploiement.

La figure 3.4 présente les critères de comparaison de cette étude.

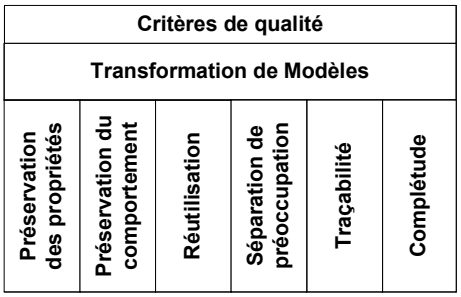

FIGURE 3.4 – Les critères de qualité d'une transformation de modèles

2.2 Description des approches existantes

Le but principal de l'utilisation de l'approche MDA pour la conception d'applications multitâches est de systématiquement séparer les aspects métiers, des aspects spécifiques à la plate-forme, en les décrivant dans des modèles distincts. Le principal point fort de cette approche découle de la possibilité de générer différents modèles spécifique à la plate-forme (PSM) à partir du même modèle indépendant de la plate-forme (PIM), et, par conséquence, d'automatiser partiellement ou totalement le processus de transformation de modèles et la réalisation de l'application multitâches sur différentes plates-formes d'exécution cibles.

Cette section s'intéresse donc à différentes approches proposées pour la mise en œuvre des applications multitâches et qui supportent le principe MDA. Elles seront présentées en fonction de la manière dont la plate-forme d'exécution est prise en compte lors du déploiement de l'application. Cette section ne s'intéresse pas aux approches de développement orientées objet permettant le déploiement sur plusieurs plates-formes, comme par exemple ACE (*Adaptive Communication Environment*) [71]. En effet, ACE peut être considéré comme une plate-forme logicielle d'exécution.

2.2.1 Représentations enfouies dans la transformation

Dans cette catégorie, les approches de développement supportent des outils qui génèrent le code source de l'application via des transformations de type modèles vers texte, dans

lesquelles les informations structurelles et comportementales relatives à la plate-forme sont enfouies.

Parmi ces outils, nous pouvons citer l'outil OCARINA [34], un générateur de code pour AADL [36], qui permet de générer du code Ada et C pour l'intergiciel PolyORB II [35] implémenté en Ada et C respectivement. De même, les générateurs de code à base des *templates* tels que celui de Ptolemy II [62] ou les compilateurs de modèles [64, 65] dans exécutable UML [63], encapsulent les informations structurelles et comportementales relatives aux plates-formes dans des *templates* qui structurent le code source à générer.

Dans ces approches, le concepteur de l'application est responsable de la modélisation des systèmes multitâches indépendamment de la plate-forme. Il utilise ensuite, les générateurs de code pour produire un code exécutable de son application pour une plate-forme spécifique. Le développement, la maintenance et l'optimisation de ces générateurs de code sont réservés aux spécialistes de l'IDM. Ces derniers doivent avoir encore une grande connaissance de la plate-forme cible pour pouvoir écrire les transformations. En effet, dans ces transformations, les informations relatives à l'application et à la plate-forme sont intégrées dans la transformation. Celle-ci est ainsi figée sur les concepts des technologies impliquées (concepts abstraits annotant l'application, la plate-forme cible et le langage de programmation). De plus, le changement de l'une de ces technologies compromet fortement la réutilisation de la transformation.

2.2.2 Représentations implicites

Dans cette catégorie, la prise en compte de plate-forme consiste à raccrocher les concepts de la plate-forme choisie aux concepts du langage UML par le biais de profils. Par conséquence, les informations relatives à la plate-forme sont indissociables des informations capturées dans le profil. Ensuite, ce profil est utilisé pour annoter les éléments du modèle applicatif, telles que les classes et les opérations.

Dans cette catégorie, nous pouvons citer la méthodologie UPES (Unified Process for Embedded Systems) [72]. Dans UPES, la prise en compte de la plate-forme logicielle d'exécution utilise le profil UML for C [73] qui réifie les concepts du langage C et le modèle de tâches de POSIX. La figure 3.5, extraite du profil UML for C, montre comment les concepts du langage C et des tâches POSIX sont raccrochés aux concepts du langage UML. Dans

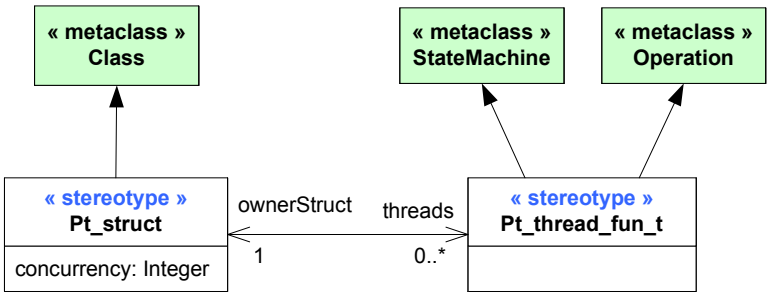

FIGURE 3.5 – Extrait du profil UML for C

cette figure, le stéréotype *Pt_struct* qui étend la métaclasse *Class*, représente une structure en langage C. Ce *pt_struct* peut avoir plusieurs threads d'exécution qui sont représentés par

le stéréotype *Pt_thread_fun_t* qui représente un *pthread*_t de POSIX. Le profil UML for C est ensuite utilisé pour annoter le modèle applicatif, comme le montre l'exemple issue de [73], dans la figure 3.6. A partir de ce modèle applicatif, un générateur de code spécifique au langage C génère le code exécutable de l'application.

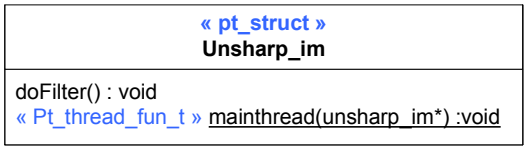

FIGURE 3.6 – Description d'une application avec le profil UML for C

Pour gérer les comportements des threads et les mécanismes de synchronisation, les auteurs de ce profil étendent le langage d'action d'UML pour introduire des actions d'appels relatives à la gestion des threads (*create*, *exit*, *join*, etc.) et aux mécanismes de synchronisation (*lock*, *unlock*, etc.) présents dans POSIX. La mise œuvre des ces appels est ensuite gérée par le générateur de code.

L'intérêt de ces approches est qu'elles sont efficaces pour un ensemble de préoccupations homogènes répétitives et figées. Cependant, ce type d'approche a l'inconvénient de son avantage puisqu'elle reste très dépendante des technologies, ce qui finalement va à l'encontre même des principes de l'IDM.

2.2.3 Représentations explicites

Dans cette catégorie, la représentation explicite des plates-formes, comme par exemple avec le profil SRM ou le langage RTEPML, permet le déploiement des applications selon deux principes, l'allocation et la conformité.

Allocation

L'allocation est utilisée pour mettre en relation le logiciel et le matériel. La méthodologie MoPCoM [74] est un exemple illustrant les approches de cette catégorie. Elle permet au concepteur de modéliser son système et de choisir ensuite un type d'implantation qui explicite la concurrence et la communication décrites dans le modèle : c'est le choix du modèle de calcul (MoC).

Parallèlement, dans MopCom l'ingénierie matérielle conçoit une architecture à gros grain de la plate-forme d'exécution en utilisant les formalismes offerts par le profil MARTE dans le sous-paquetage GRM (*Generic Resource Modeling*). Cette activité consiste à caractériser les différentes ressources de la plate-forme par leurs services en les associant à leurs qualités de service, leurs interfaces et leurs protocoles de communication de haut niveau (*message passing*, rendez-vous, fifo, . . .).

Il est nécessaire ensuite de fournir un choix d'implantation du système à travers un modèle d'allocation qui définit la distribution spatio-temporelle des fonctions, des données et des communications sur la plate-forme. Ce choix d'implantation doit être validé par des scénarios d'analyse mettant en exergue le caractère statistique et probabiliste de la plate-forme. Ces scénarios permettent de générer des modèles exécutables à travers des

transformations, permettant la génération du code.

L'intérêt de cette approche réside dans la véritable séparation des préoccupations qu'on peut constater lors de la conception du modèle applicatif et du modèle de la plate-forme. En plus, l'identification par allocation des correspondances entre les éléments applicatifs et les éléments de la plate-forme cible décrits d'une manière commune, permet de réaliser une transformation de modèle indépendante des concepts de la plate-forme cible. Cependant, les informations comportementales spécifiques à la plate-forme, telles que la création ou l'initialisation des ressources, sont encapsulées dans la transformation. Par conséquent la transformation n'est pas totalement générique.

Conformité

La conformité est utilisée pour identifier par stéréotypage les éléments du modèle source et les éléments correspondants dans le modèle cible. Pour cela, les approches se basent sur un langage pivot pour décrire explicitement les informations relatives à la plate-forme source et cible.

Le premier travail qui utilise ce type de correspondance est celui présenté par Frédéric Thomas dans [75]. Dans ce travail, le profil SRM est utilisé comme un langage pivot pour l'élaboration d'une infrastructure de transformation indépendant des plates-formes source et cible. Cette infrastructure se base sur une description explicite et commune des plates-formes logicielles d'exécution pour la génération d'un modèle spécifique à ces plates-formes. Ainsi, pour chaque instance du modèle de départ dont le type est stéréotypé dans la plate-forme source (la plate-forme sur laquelle l'application est déjà déployée), s'il existe une ressource stéréotypée de la même façon dans la plate-forme cible (la nouvelle plate-forme sur laquelle l'application doit être implantée), cette ressource est instanciée.

Bien que cette approche permette la génération de modèles de la plate-forme spécifique à l'aide de règles de transformation génériques indépendantes du domaine, elle permet une implantation structurelle de l'application. Les aspects comportementaux relatifs aux ressources et services de la plate-forme ne sont pas traités dans l'infrastructure de transformation proposée. Donc le modèle spécifique généré n'est pas complet.

Dans la même vision que l'approche à base de SRM, le processus de déploiement basé sur le langage RTEPML, proposé dans [79], vise à proposer des règles de transformation génériques pour automatiser le déploiement des systèmes multitâches. Il repose sur le mécanisme d'intégration introduit dans leur langage de modélisation de plates-formes RTEPML. Ce processus permet la mise en œuvre de mécanismes applicatifs avec des concepts exécutifs. Il permet également de préciser des décisions qui répondent au choix de déploiement.

Ce processus basé sur des transformations génériques, permet une séparation nette de préoccupations applicatives et technologiques et les transformations sont réutilisables. Cependant, dans cette approche, les modèles spécifiques à la plate-forme ne supportent pas les aspects comportementaux spécifiques aux ressources et services de la plate-forme. Le code généré à partir de ces modèles n'est donc pas exécutable.

2.2.4 Approches alternatives

D'autres approches, ne se basent par sur des modèles de plates-formes, pour supporter le déploiement des systèmes sur plusieurs plates-formes. on en retiendra deux : l'abstraction par machines virtuelles et le tissage d'aspects.

Parmi les approches par machines virtuelles, on peut noter celles qui proposent d'exécuter directement le modèle UML de l'application, en proposant des machines virtuelles UML. Par exemple, Schattkowsky et Muller présentent dans [66] une approche de développement des systèmes embarqués basée sur les modèles. Elle s'appuie sur le diagramme de classes UML et les diagrammes d'états, et sur une plate-forme d'exécution abstraite (AEP). Les diagrammes UML sont convertis par une transformation en des *bytecodes* qui peuvent être exécutés sur cette plate-forme abstraite. Cette approche permet la réutilisation des transformations de modèles UML en *bytecode* AEP. Cependant, la plate-forme d'exécution abstraite doit être re-implémentée, à chaque fois que la plate-forme cible est modifiée.

Les approches IDM orientées aspects, en particulier le travail présenté dans [67], propose de générer à partir d'un modèle indépendant de la plate-forme capturant des contraintes non-fonctionnelles le code exécutable de ce modèle. L'outil de transformation utilisée est appelée *GenERTiCA*. Cet outil permet d'automatiser la génération de code exécutable en réalisant du tissage des aspects (*aspect weaving*) dans l'endroit spécifié dans lors de la modélisation. L'outil de transformation est réutilisable avec les plates-formes logicielles d'exécution supportant le même langage de programmation. Les aspects qui sont des implémentations spécifiques à la plate-forme cible sont représentés par des scripts XML. Bien que cette approche favorise la séparation des préoccupations, elle ne permet pas une description explicite et méthodologique de la plate-forme d'exécution.

2.3 Synthèse et Positionnement

La figure 3.7, synthétise et évalue les différentes approches de déploiements présentées vis-à-vis des critères de qualité définis dans la sous-section précédente. Elle vise concrètement à évaluer si chaque critère de qualité défini pour réaliser une transformation est supporté, peu supporté ou non supporté dans cette transformation.

Les approches, telles que xUML et UPES, utilisent des transformations de modèles spécifiques aux technologies impliquées pour générer des modèles complets. Ces transformations encapsulent les implémentations spécifiques aux plates-formes. De telles transformations demandent une double compétence à la fois d'IDM et des technologies. De plus, le changement de l'une de ces technologies compromet fortement sa réutilisation.

Les approches basées sur la description explicite de plates-formes permet de répondre presque à tous les critères de qualités. Ainsi, parmi ces approches, nous pouvons remarquer que seules les approches à base de SRM et RTEPML supportent toutes les critères de qualité à l'exception de la complétude, c'est-à-dire, que les modèles générés en sortie de ces transformations ne permettent pas la génération d'un code exécutable. Ceci est dû au fait que ces approches ne traitent pas les aspects comportementaux spécifiques aux ressources et services de la plate-forme. Ces informations ne sont pas encapsulées dans les transformations de modèles comme c'est le cas dans le processus de déploiement

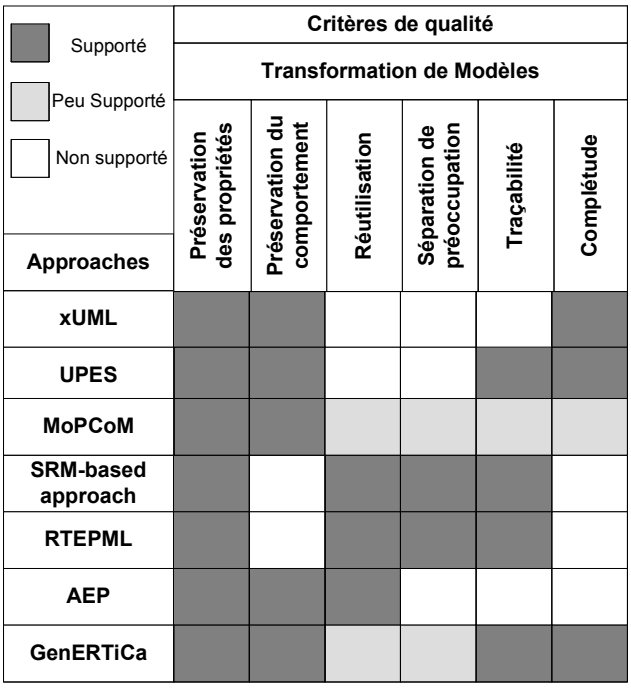

FIGURE 3.7 – Comparaison des approches de déploiement

MoPCoM. Dans ce dernier, les informations comportementales relatives à la plate-forme sont encapsulées dans la transformation pour permettre la génération d'un code exécutable.

Par rapport aux travaux existants, le processus de déploiement visé doit satisfaire tous les critères de qualité. On cherchera donc à permettre la génération des modèles complets tout en garantissant des règles de transformations génériques réutilisables. Les implémentations spécifiques à la plate-forme devront donc être extraites des transformations de modèles. Cela permettra à un expert en IDM de fournir des règles de transformations génériques sans se préoccuper des implantations relatives à la plate-forme. Ces derniers seront pris en compte dans des modèles de plates-formes explicites, comme présenté dans la sous section précédente.

Une telle approche permettra de réaliser une séparation de préoccupation pour la mise en œuvre des systèmes multitâches. Ainsi, le développeur des chaînes de transformation offrira une transformation de modèle générique, les fournisseurs des plates-formes fournisseront des modèles détaillés de leurs plates-formes et le concepteur des applications concurrentes multitâche modélisera le système.

3 Conclusion

Ce chapitre a présenté la modélisation des plates-formes logicielles d'exécution et la prise en compte de ces plates-formes lors de déploiement. Pour chaque sujet des besoins ont été définis et les études existantes ont été analysées. Leurs synthèses et leurs comparaisons positionnent les objectifs de cette thèse.

Ainsi dans cette étude, une modélisation explicite des plates-formes logicielle

d'exécution doit être réalisée. Cette modélisation doit décrire la structure des ces plates-formes mais aussi les comportements observables de ses ressources et services. Une telle modélisation doit permettre la réalisation d'une infrastructure de transformation générique permettant la génération des modèles spécifiques à la plate-forme complet et par conséquence la génération du code exécutable.

Pour cela, le chapitre suivant traite la partie concernant la modélisation de la plate-forme explicite, et le chapitre d'après traite l'intégration de ces modèles de plates-formes dans une ingénierie générative dirigée par les modèles.

CHAPITRE 4

Contribution à la modélisation détaillée des plates-formes logicielles d'exécution

L'objectif de ce chapitre est de proposer des heuristiques de modélisations permettant de décrire la structure des ressources et des services offerts par les plates-formes et décrivent, en plus, les comportements associés à ces ressources et ces services.

Pour cela, ce chapitre est décomposé en trois sections. La première section présente le point de départ : le profil SRM de MARTE qui offre l'essentiel des concepts importante à la description des plates-formes logicielles d'exécution. La deuxième section étudie l'impact de ces plates-formes sur la réalisation de transformations de modèles produisant des modèles spécifiques à la plate-forme à partir des modèles indépendants de la plate-forme. Elle identifie les besoins de modélisation envers des transformations génériques. Enfin, la troisième section définit des heuristiques de modélisation basées sur le profil SRM vis-à-vis de ces besoins.

1 Présentation succincte du profil SRM d'UML-MARTE

Le profil SRM (Software Resource Modeling) de MARTE permet de décrire les ressources et les services de systèmes d'exploitation temps réel (RTOS) d'une manière commune. SRM a été défini après une analyse détaillée des principaux mécanismes et services représentatifs des exécutifs temps réel embarqués [75]. De cette analyse, trois familles de concepts ont été identifiées : les exécutions concurrentes (par exemple, une interruption et une tâche), les interactions entre les entités concurrentes (par exemple, la boîte aux lettres et les mécanismes de sémaphore) et les gestionnaires des entités matérielles et logicielles (par exemple, les ordonnanceurs). La conception du profil SRM est basée sur le motif *Resource-Service*. C'est l'utilisation de ce motif qui lui permet de décrire facilement les APIs des systèmes d'exploitation temps réel. En fait, l'approche adoptée lors de la conception de profil, a été de fournir un support, langage, pour décrire n'importe quelle API, des systèmes d'exploitation temps réel, plutôt que de tenter de décrire et normaliser chacune des APIs des RTOSs existants.

1.1 Présentation du Motif *Resource-Service*

La figure 4.1 illustre le modèle du motif *Resource-Service*. Ce motif est proposé par l'OMG dans le but de modéliser les ressources spécifiques à un certain domaine d'application, y compris les plates-formes logicielles d'exécution. Dans le contexte de la modélisation des plates-formes, les ressources du motif (*Resource*) correspondent aux mécanismes à gérer, fournis par la plate-forme d'exécution. Ces ressources sont caractérisées par des propriétés (*ResourceProperty*) et peuvent être instanciées (*ResourceInstance*). Les services des ressources (*ResourceService*) sont les traitements offerts par la plate-forme afin de gérer et manipuler les ressources. Ces services sont composés de paramètres dont la valeur est précisée lors des appels aux services.

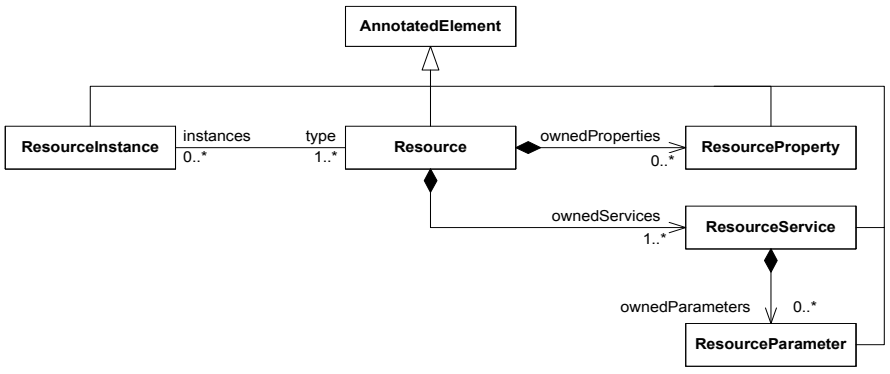

FIGURE 4.1 – Extrait du motif Resource-Service

Les ressources des plates-formes d'exécution sont variées. Elles fournissent des opérations et des propriétés qui accomplissent un certain rôle comme la création et la destruction de la ressource ou encore la spécification de la période d'exécution d'une ressource. Pour permettre au motif Ressource-Service de modéliser les opérations et les propriétés des différentes ressources possédant chacune un rôle spécifique, les rôles joués

par ces derniers sont utilisés. Créer pour chaque concept (propriétés ou opérations des ressources) un métatype qui le modélise aurait imposé la création d'un nombre très important de métatypes pour structurer les concepts du domaine des plates-formes. Pour éviter cela, les concepteurs de SRM ont utilisé des propriétés référencées par des associations comme étant des attributs de la ressource de ce motif dont les caractéristiques seront précisées lors de la modélisation de chaque plate-forme. Dans ce cas, la propriété de la ressource du motif correspond au rôle joué par la propriété ou le service de la ressource de la plate-forme.

Par exemple, dans la figure 4.2, *SchedulableResource* est un métatype qui modélise une ressource concurrente. Cette ressource possède des attributs. Un de ces attributs joue le rôle de priorité, il décrit donc les propriétés de la ressource concurrente qui jouent le rôle d'une priorité. D'autres attributs jouent le rôle des services d'activation ou de suspension de l'exécution, ils décrivent donc les opérations de la ressource concurrente qui permettent d'activer et de suspendre l'exécution de cette ressource.

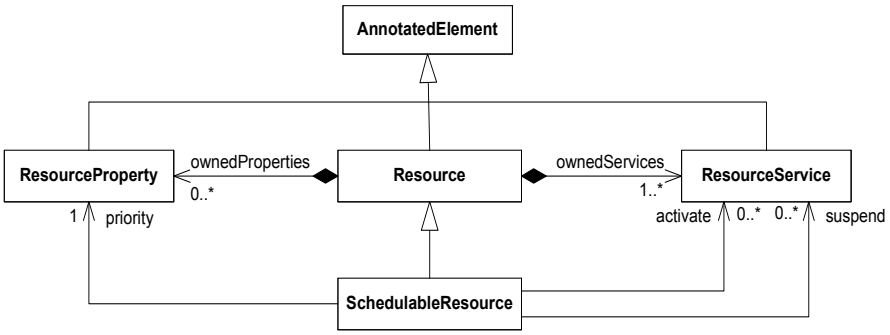

FIGURE 4.2 – Utilisation des associations pour décrire les propriétés et les services

1.2 Les packages du profil SRM

Le profil SRM est structuré en quatre packages. Le premier, *SwResourceCore*, vise à décrire les fondations du profil de SRM. Les trois autres sous packages (*SwConcurrency*, *SwInteraction*, *SwBrokering*) réifient les concepts de *SwResourceCore* pour les entités concurrentes, les interactions entre les entités concurrentes et les ressources de gestion. Cette partie présente succinctement les packages *SwResourceCore*, *SwConcurrency* et *SwInteraction* qui sont utilisés dans la section suivante pour modéliser les plates-formes logicielles d'exécution. Pour une description plus détaillée de ces packages, le lecteur est invité à consulter le document du standard MARTE [47].

1.2.1 Le package Sw_ResourceCore

Dans ce package, le stéréotype *SwResource* étend la métaclasse *Class* et permet de capturer les concepts primitifs communs à toutes les ressources de la plate-forme (figure 4.3). Ces concepts communs sont modélisés par les propriétés de ce stéréotype. En effet, chaque ressource de la plate-forme doit posséder un identificateur qui représente la ressource créée. La propriété *identifierElements* permet de référencer cet identificateur. En plus, tous

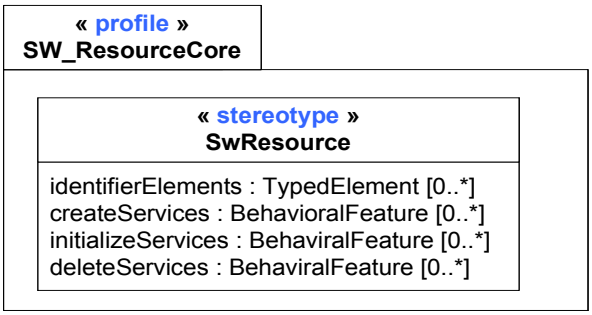

FIGURE 4.3 – Extrait du package *SW_ResourceCore* du profil SRM

les ressources de la plate-forme fournissent des opérations pour les créer, les initialiser et les détruire. Ces services communs peuvent être référencés par les propriétés *createServices*, *initializeServices* et *deleteServices* du stéréotype *SwResource*.

1.2.2 Le package Sw_Concurrency

Ce package vise à fournir des artefacts de modélisation pour décrire des contextes d'exécution logicielle concurrentes. Ces contextes d'exécution peuvent être décrits par le stéréotype *SwConcurrentResource*. Dans le cas où l'exécution peut être activée par un ordonnanceur ou par une autre ressource de la plate-forme, la ressource concurrente est spécialisée par le stéréotype *SwSchedulableResource* de ce package. Si elle est activée suite à une interruption, elle est alors annotée par le stéréotype *InterruptResource*.

Ces deux catégories des ressources concurrentes possèdent des concepts communs. Ils sont capturés par les propriétés du stéréotype *SwConcurrentResource* dont les deux autres stéréotypes héritent. Notamment, ce stéréotype permet de capturer les caractéristiques essentielles suivantes des ressources concurrentes :

- Il permet de préciser la manière dont les séquences d'actions associées à une ressource concurrente peuvent être exécutées : périodique, apériodique ou sporadique.
- il permet d'identifier les opérations permettant d'arrêter, de suspendre et de reprendre l'exécution des séquences d'actions dans une ressource concurrente. Ces opérations peuvent être référencées par les propriétés *resumeServices*, *suspendServices* et *terminateSerivces* de ce stéréotype.
- il permet de spécifier la séquence d'actions que la ressource concurrente doit exécuter une fois qu'elle est activée. Cette séquence est décrite par la propriété *entryPoints* de ce stéréotype.
- Lorsqu'une entité concurrente peut être prioritaire par rapport aux autres entités ou réclame une exécution périodique avec une valeur prédéfinie, ces deux concepts sont identifiés par les propriétés *priorityElements*, *periodElements* de ce stéréotype.
- Il permet d'identifier la zone mémoire dans laquelle une ressource concurrente s'exécute. Ainsi, la valeur de la propriété *addressSpace* du stéréotype *SwConcurrentResource* spécifie la zone mémoire. Le type de la valeur de la propriété *addressSpace* doit être annoté par le stéréotype *MemoryPartition*. Ce dernier permet de capturer les différentes zones mémoires.

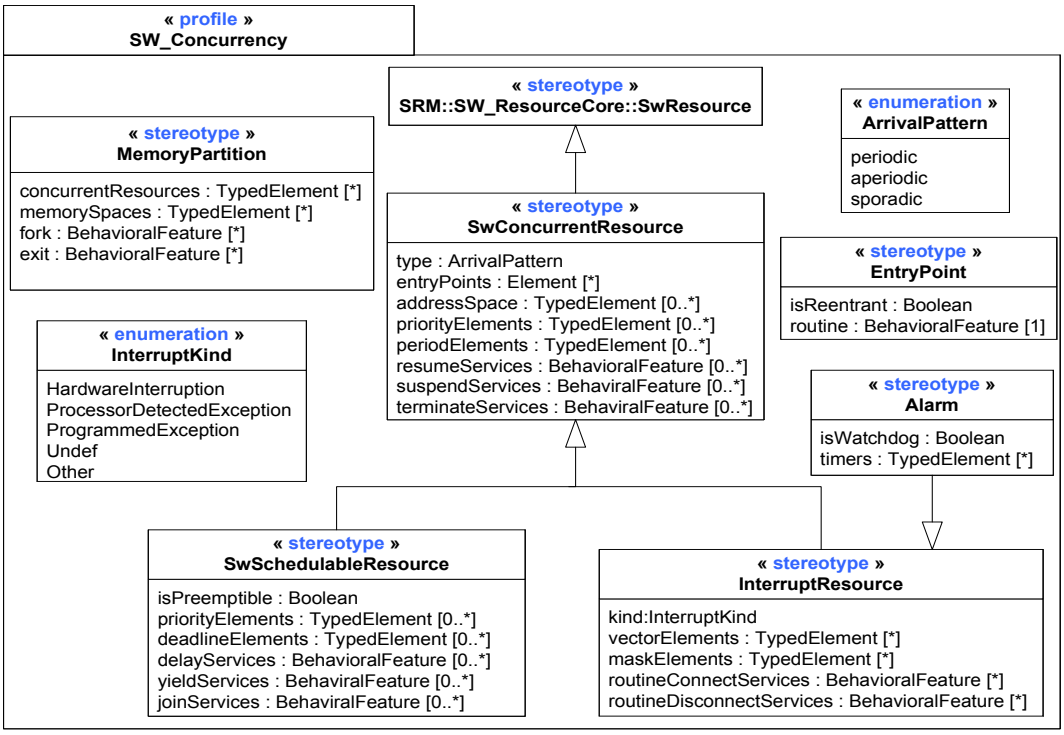

FIGURE 4.4 – Extrait du package *Sw_Concurrency* du profil SRM

1.2.3 Le package SW_Interaction

Ce package offre des stéréotypes qui décrivent les mécanismes de communication et de synchronisation des plates-formes logicielles d'exécution. Les mécanismes de communication sont utilisés par les ressources concurrentes pour échanger des flots de données. Cet échange est divisé en deux catégories :

Échange de message Il est capturé par le stéréotype *MessageComResource* de ce package. Ce stéréotype possède deux propriétés qui référencent les services d'envoi et de réception d'une structure de donnée (*sendServices*, *recieveServices*). Il permet encore de référencer la taille de file d'attente où les messages sont rangés (*messageQueueCapacityElements*) et de spécifier la politique d'ordonnancement des messages à travers la propriété *messageQueuePolicy*. Cette propriété est typée par une *Enumeration* qui spécifie différent politique d'ordonnancement.

Échange par variable partagée Ce type de communication est capturé par le stéréotype *SharedDataComResource*. Il possède deux propriétés qui référencent les services de lecture et d'écriture de la ressource (*readServices*, *writeServices*).

Outre les stéréotypes décrivant les mécanismes de communication, le stéréotype *SwMutualExlusionResource* décrit les ressources d'exclusion mutuelle synchronisant l'accès mutuelle à une même variable partagé. Ces ressources fournissent des opérations permettant de prendre ou de libérer l'accès à la variable partagée. Elles sont référencées par les propriétés *acquireServices* et *releaseServices* du stéréotype. Ce stéréotype permet aussi de spécifier la politique d'accès à cette variable à travers la propriété *concurrentAccessProtocol*. Elle est de type *Enumeration* qui spécifie l'ensemble des protocoles d'accès comme le *mutex*,

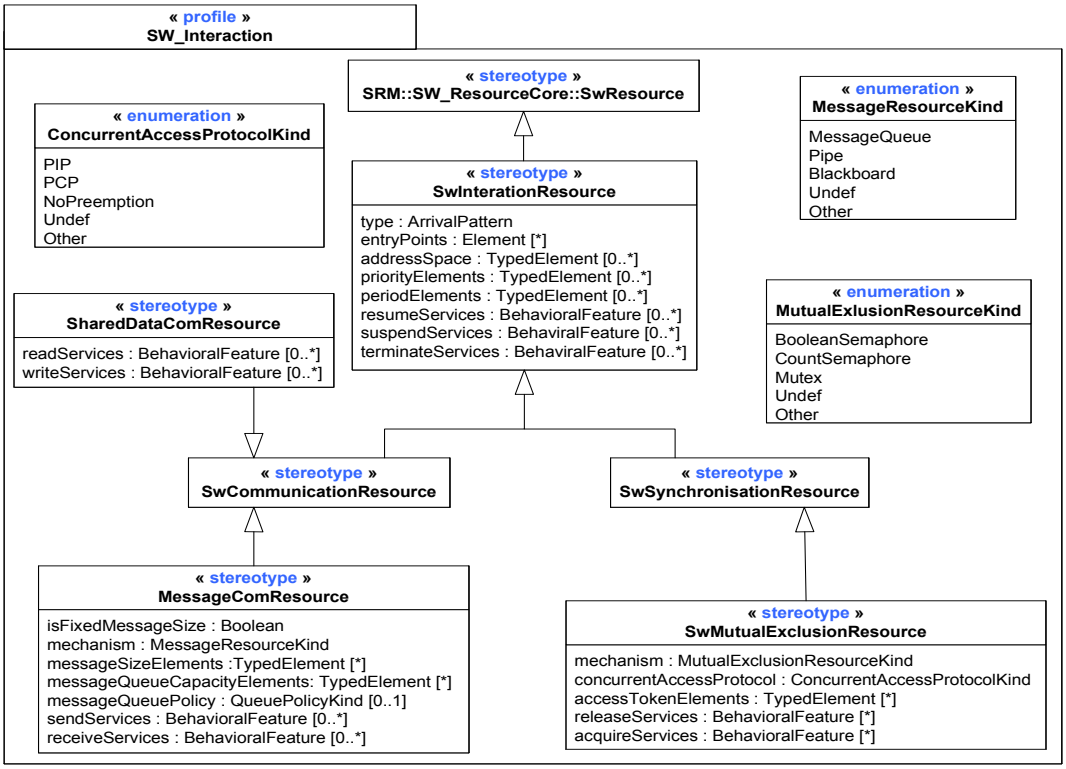

FIGURE 4.5 – Extrait du package *Sw_Interaction* du profil SRM

le sémaphore, etc.

1.3 Bilan

Cette section a présenté premièrement le motif *Resource-Service* qui a été utilisé pour décrire le métamodèle SRM. Deuxièment, elle a présenté le profil SRM. Nous avons vu que ce profil permet de modéliser explicitement les ressources de concurrence et les ressources d'interaction de la plate-forme. Il répond donc notre besoin pour qu'il soit utilisé pour la modélisation des plates-formes logicielles.

Notons que dans le profil SRM, certains propriétés des stéréotypes sont typées par des types primitifs (booléen, entier, énumération) et d'autres sont typées par *TypedElement* et *BehavioralFeature* spécifiés pour référencer des éléments. En effet, une propriété de type primitif permet de caractériser la ressource tandis que une propriété typée par *TypedElement* et *BehavioralFeature* permet d'identifier une caractéristique. Par exemple, dans le stéréotype *SwSchedulableResource*, la propriété *type* spécifie si la ressource modélisée est périodique ou apériodique. La propriété *createServices* de type *BehavioralFeature*[0..*] identifie, par exemple, les opérations fournies par la ressource permettant de la créer. De même, la propriété *priorityElements* de type *TypedElement*[0..*] référence les éléments typés de la ressource jouant le rôle de la priorité de la ressource.

En utilisant le profil SRM comme langage pivot pour décrire les plates-formes d'une manière commune, il est possible d'expérimenter des transformations de modèles basées sur des modèles de plates-formes explicites. La section suivante compare le pouvoir de cette

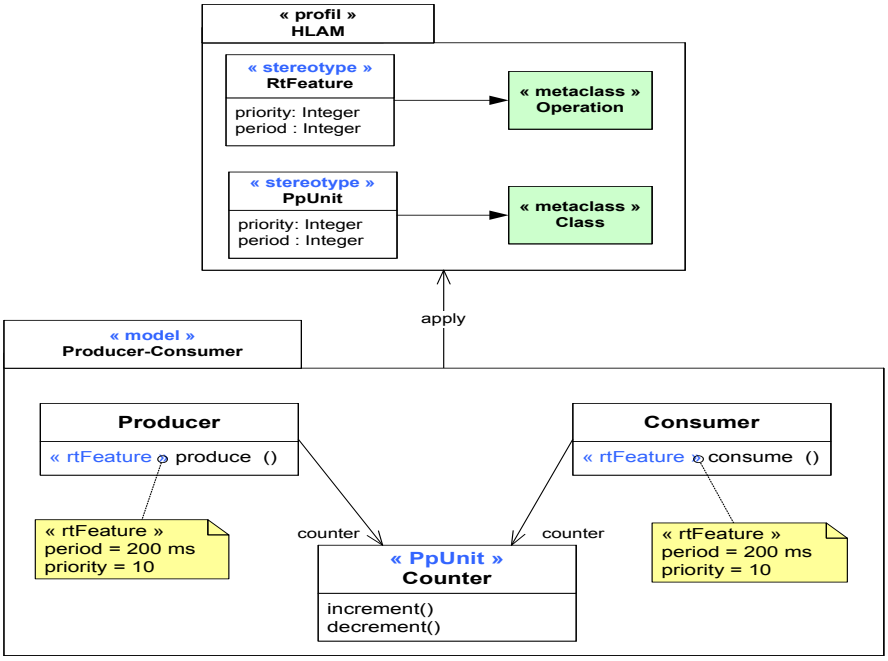

FIGURE 4.6 – Modèle de l'application Producteur/Consommateur

transformation de modèles à générer des modèles applicatifs spécifiques à la plate-forme.

2 Identifications des besoins pour une infrastructure de transformation générique

Afin d'identifier les besoins d'une infrastructure de transformations génériques, il faut étudier les relations entre les modèles applicatifs et les plates-formes d'exécution cibles. Pour illustrer concrètement ces relations, la figure 4.6 représente une application producteur/consommateur. Dans cette application, un producteur effectue des mesures et incrémente un compteur à chaque fois qu'une mesure est effectuée par l'opération *produce* et un consommateur traite les mesures et décrémente le compteur avec chaque mesure traitée par l'opération *consume*. Le compteur est partagé par le producteur et le consommateur, donc, l'accès à cette ressource doit être protégé.

L'analyse de cette application montre la présence des contraintes qualitatives en relation avec la concurrence, la synchronisation et les mécanismes de communication et des contraintes quantitatives spécifiant la valeur de la période d'une exécution périodique. La concurrence se manifeste dans l'exécution de deux opérations *produce* et *consume* qui, dans cette exemple, nécessitent de s'exécuter sur leurs propre threads. Un mécanisme de synchronisation est requis afin de protéger l'accès aux opérations du compteur. Et enfin, un mécanisme de communication doit être mis en œuvre pour assurer les échanges de mesures entre les deux objets concurrents (producteur et consommateur).

Au niveau modèle (figure 4.6), pour modéliser les contraintes qualitatives et quantitatives le concepteur utilise un profil dédié aux applications concurrentes. Dans cet exemple, une version largement simplifiée du sous-profil HLAM de MARTE est utilisé (voir

chapitre 14 du [47]). L'utilisation de ce profil se manifeste par l'annotation des éléments applicatifs par des stéréotypes de ce profil qui lient implicitement le contexte d'exécution de ces élément avec les contextes d'exécution des concepts de la plate-forme. Par conséquence, dans l'application Producteur/Consommateur, les deux opérations *produce* et *consume* sont annotées par le stéréotype *RtFeature* pour capturer le fait que, au moment d'exécution, ces deux opérations doivent s'exécuter sur leur propre thread d'exécution. La classe *Counter* est annotée par *PpUnit* pour spécialiser un accès protégé aux opérations du compteur. Les valeurs de la période et de la priorité des threads d'exécution sont spécifiées par les valeurs affectées aux propriétés *period* et *priority* du stéréotype *RtFeature*.

L'utilisation du profil HLAM et ces concepts abstraits ont permis la modélisation de l'application Producteur/Consommateur d'une manière indépendante de la plate-forme. A ce niveau de modélisation, une première séparation de préoccupation est réalisée. Elle consiste à abstraire l'application des implémentations spécifiques à la plate-forme nécessaires pour réifier les concepts abstraits de modélisation.

Dans l'ingénierie dirigée par les modèles, la réification des concepts abstraits utilisés par le concepteur au niveau modèle se réalise à l'aide d'une transformation de modèle. La transformation prend le modèle applicatif indépendant de la plate-forme comme entrée et génère un modèle spécifique à une plate-forme choisie comme plate-forme cible. Cependant, pour implémenter une transformation de modèle, la majorité des réalisateurs des transformations ont tendance à écrire des règles de transformation spécifiques permettant d'automatiser la production des applications. Ces règles de transformations sont spécifiques parce qu'ils sont dépendantes des spécificités des plates-formes cibles. Afin d'illustrer les impacts que les plates-formes cibles peuvent avoir sur les transformations des modèles et, par conséquence sur les modèles spécifiques à la plate-forme, la section suivante présente la spécialisation du modèle indépendant de la plate-forme de l'application Producteur/Consommateur vers des modèles spécifiques aux deux plates-formes Java et C++/POSIX.

2.1 Implantation de l'application

2.1.1 Implémentation à travers des transformations spécifiques

La spécialisation de l'application Producteur/Consommateur pour les plates-formes Java et C++/POSIX est illustrée dans les figures 4.7 et 4.8. Ces deux modèles sont générés à travers deux transformations de modèles spécifiques. Les deux modèles générés contiennent tous les aspects comportementaux nécessaires à l'exécution de l'application. L'analyse des modifications entre le modèle applicatif indépendant de la plate-forme et les modèles générés spécifiques à la plate-forme permet d'identifier les implémentations spécifiques qui étaient encapsulées dans les transformations de modèles.

Dans les classes *Producer* et *Consumer* de la figure 4.6, le stéréotype *RtFeature* est appliqué sur les opérations *produce* et *consume*. En Java, ce concept est associé à la ressource *Thread* et à la ressource *Pthread_t* en POSIX. La manipulation d'un thread diffère d'une plate-forme à une autre.

Java Afin de manipuler un *Thread*, la plate-forme Java impose un patron d'utilisation. Il consiste à définir une classe qui étend la classe *Thread* ou qui implémente l'interface

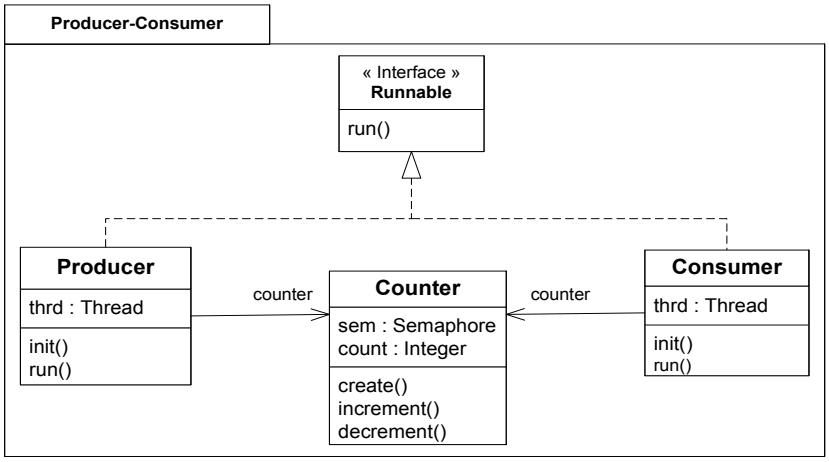

FIGURE 4.7 – Modèle de l'application Producteur/Consommateur spécifique à Java

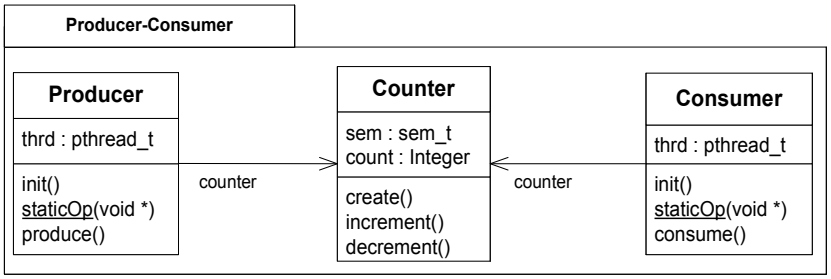

FIGURE 4.8 – Modèle de l'application Producteur/Consommateur spécifique à C++/POSIX

Runnable. Dans les deux cas, la classe doit fournir une implémentation de la méthode *run*. Cette méthode sera exécutée par le thread qui vient de démarrer après sa création. La création et le démarrage du thread Java est une implémentation spécifique réalisée par un constructeur qui prend le *runnable* (instance de la classe qui implémente l'interface *Runnable*) comme paramètre, ensuite par un appel à l'opération *start*. Ce patron d'utilisation est reflété dans les classes *Producer* et *Consumer* spécifiques à la plate-forme Java de la figure 4.7. Ainsi, les méthodes *createProducer* et *createConsumer* permettent de créer et démarrer le *Thread* et la méthode *run* encapsule le code métier qui doit être exécuté.

POSIX la réification du stéréotype *RtFeature* consiste à créer premièrement un thread POSIX en utilisant la fonction *pthread_create* qui prend quatre paramètres dont l'opération à exécuter une fois que la thread est activée. En utilisant C++ comme langage de programmation avec POSIX, l'opération passée en paramètre doit être une opération statique. Par conséquence, dans les deux classes *Producer* et *Consumer* générées de la figure 4.8, l'opération statique *staticOp* est ajoutée.

La valeur de la propriété *period* du stéréotype *RtFeature* spécifie la période des exécutions des opérations *produce* et *consume*. Cependant, Java et C++/POSIX n'offrent pas nativement la notion de ressource concurrente périodique. Par conséquence, un patron de conception est adopté pour simuler une exécution périodique. Il consiste à modifier respectivement le comportement des opérations *run* et *staticOp* afin de simuler des exécutions périodiques. La figure 4.9 présente le comportement périodique de la méthode *run* de la plate-forme Java. Dans la boucle *while*, le code métier est exécuté au début, ensuite l'opération *sleep* est appelée. Elle permet d'interrompre temporairement le thread et de le reprendre après une période.

```
public void run() {
            while (true) {
                  try {
                        Data data = getData();
                        counter.increment();
                        queue.send(data);
                        Thread.sleep(200);
                  } catch (InterruptedException i) {
                  }
            }
      }
```

FIGURE 4.9 – Comportement périodique de la Thread Java

La classe Counter est annoté avec *PpUnit*, l'accès à ses opérations doit donc être protégé. En Java, ce concept est associé à la ressource *Semaphore* et en POSIX, à la ressource *sem_t*. Pour réaliser la protection des appels des opérations *increment* et *decrement*, un sémaphore doit être créé et le comportement des ces opérations doit être encapsulé entre un appel vers le service *acquire* et un appel vers le service *release* des ressources *Semaphore* et *sem_t*. Cette implémentation spécifique est appliqué dans les classes *Counter* générées spécifiques aux plates-formes Java et C++/POSIX. L'opération *createCounter* permet de créer le sémaphore et le comportement des opérations *increment* et *decrement* est modifié comme capturé dans l'activité de la figure 4.10.

L'analyse du modèle Producteur/Consommateur permet d'identifier quelques implémentations et informations spécifiques à la plate-forme et qui sont nécessaires à l'exécution de l'application :

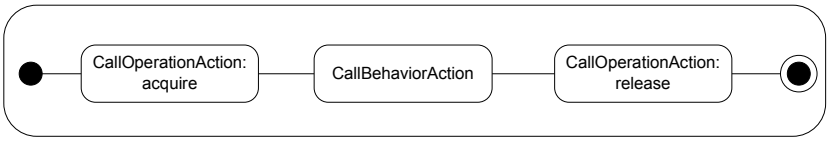

FIGURE 4.10 – Implémentation spécifique pour la protection

Transformation spécifique pour Java:

```
<%if (getStereotypeValue("RtFeature","period") != null) {%>
<%genModifier%>run(<%genMethodParam%>)<%genException%> {
while(true){
<%genMethodContent%>
Thread.sleep(<%getStereotypeValue("RtFeature","period")%>);
}
<%}%>
```

Transformation spécifique pour C++/POSIX:

```
<%if (getStereotypeValue("RtFeature","period") != null) {%>
static void *staticOp(void *arg) {
while(1){
<%genMethodContent%>
sleep(<%getStereotypeValue("RtFeature","period")%>);
}
<%}%>
```

Légende:

Implémentation spécifique à la plate-forme

Concept abstrait

FIGURE 4.11 – Transformations spécifiques pour générer les opérations *run* et *staticOp*

– La création et l'initialisation des ressources.
– La réalisation d'un comportement périodique d'un thread.
– La création des appels aux services *acquire* et *release* de la plate-forme.

Ces implémentations sont réalisées implicitement par la transformation de modèle spécifique. La transformation spécifique se base sur une forte connaissance de la plate-forme cible et crée une grande dépendance de la transformation à cette plate-forme. Elle identifie premièrement le concept abstrait utilisé au niveau applicatif et génère une implémentation spécifique par défaut décrit dans une règle de transformation.

Par exemple, la figure 4.11 présente un extrait des deux règles de transformation spécifiques (formulées ici dans le même langage, Acceleo) nécessaires pour réifier le stéréotype *RtFeature* et générer les opérations *run* et *staticOp*. Dans ces transformations, les implémentations spécifiques à la plate-forme (cf. figure 4.11, l'opération *run* et l'appel à l'opération *sleep* de l'API Java, et l'opération *staticOp* et l'appel à l'opération *sleep* de l'API POSIX, dans le code Acceleo) sont mélangées avec les concepts du langage de transformation utilisé et des concepts abstraits du langage de modélisation. De telles transformations sont difficiles à porter en cas de changement de plate-forme et nécessite une double compétence pour les maintenir surtout pour une transformation à une échelle industrielle avec un grand nombre ligne de code. Une alternative consiste à réaliser des transformations de modèle plus

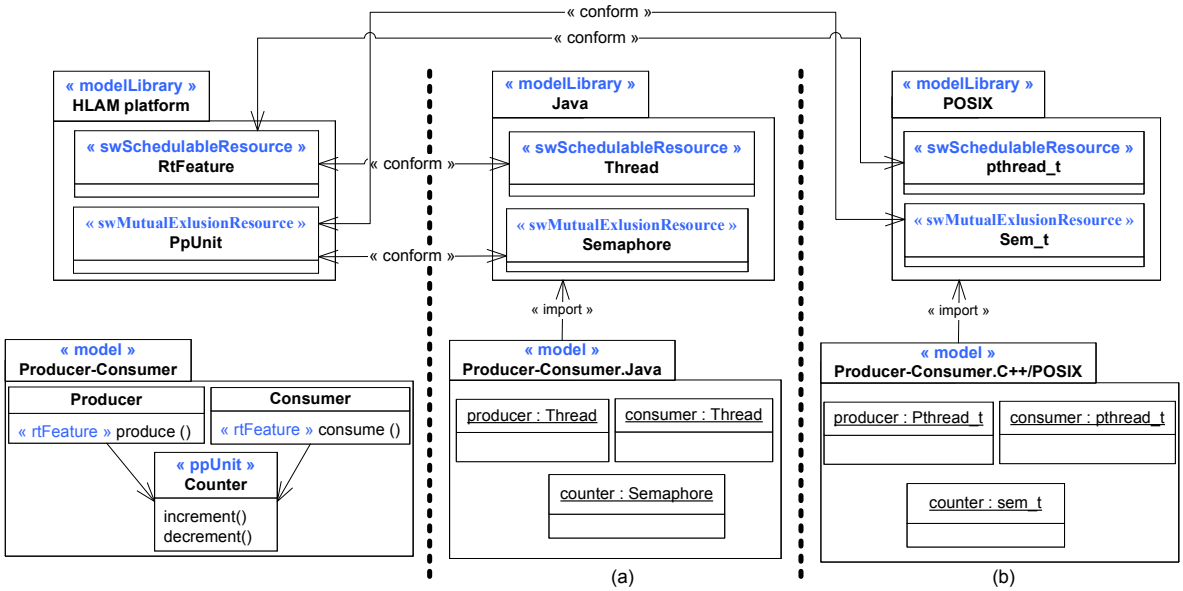

FIGURE 4.12 – Modèles Producteur/Consommateur spécifiques générés avec une transformation générique

génériques où les règles de transformation peuvent être capitalisées.

2.1.2 Implémentation à travers des transformations génériques

Afin de permettre une capitalisation des règles de transformations, on propose d'utiliser un métamodèle pivot. Ce métamodèle doit permettre une modélisation commune des plates-formes d'exécutions pour pouvoir guider les transformations de modèles. Dans ce contexte, le profil SRM de MARTE est conçu pour jouer ce rôle comme présenté dans la section 1. En plus, les travaux présentés dans [75] propose une infrastructure de transformation générique basée sur ce pivot.

Cette infrastructure de transformation générique basée sur SRM vise à porter une application décrite avec un profil dédié aux applications concurrentes vers une ou plusieurs plates-formes cibles. Les concepts du profil, les ressources et les services des plates-formes cibles sont décrits par le même métamodèle de plates-formes (SRM).

Afin d'évaluer la capacité des transformations génériques à générer des modèles spécifiques aux différents plates-formes, l'infrastructure de transformation proposée dans [75] est utilisée. Le but sera d'évaluer la complétude des modèles générés face à ceux générés avec des transformations spécifiques. Pour cela, cette infrastructure est utilisée pour générer les modèles de l'application Producteur/Consommateur spécifiques aux plates-formes Java et C++/POSIX. Ces transformations ne sont pas contraintes par les langages de conception de haut niveau, ni les ressources de la plate-forme cible. Elles sont basées sur un algorithme de conformité entre les concepts de haut niveau et les ressources de la plate-forme cible (voir algorithme 7.2 dans [75]). Son principe consiste à générer un déploiement en faisant une correspondance entre les ressources offertes par les différentes plates-formes, via leurs stéréotypes.

La figure 4.12.a présente le modèle spécifique à la plate-forme Java généré avec une transformation générique basée sur SRM. Les opérations *produce* et *consume* des deux classes *Producer* et *Consumer* sont annotées par *RtFeature* qui spécifie une exécution concurrente. Dans SRM, une source de concurrence correspond au stéréotype *SwSchedulableResource*. De plus, le stéréotype *SwSchedulableResource* correspond à la ressource *Thread* en Java. Par conséquent une relation de conformité peut être déduite entre le stéréotype *RtFeature* et la ressource Java. De même, La classe *Counter* annotée avec *PpUnit* correspond au stéréotype *SwMutualExclusionResource* de SRM. Ce dernier correspond à la ressource *Semaphore* de la plate-forme Java. A l'aide de ces relations de conformités basées sur les stéréotypes SRM, la transformation de modèle générique génère le modèle spécifique à la plate-forme Java de la figure 4.12.a. Dans cet exemple, les classes *Producer* et *Consumer* encapsulant les opérations annotés avec *RtFeature* sont transformés en des instances des Thread Java. La classe *Counter* est transformée en une instance d'une *Semaphore*.

La transformation de modèle basée sur une relation de conformité peut être utilisée avec des autres plates-formes cibles. Par exemple, la Figure 4.12.b présente le modèle spécifique à la plate-forme C++/POSIX généré avec la même transformation utilisée pour la plate-forme Java. Le seul effort requis pour permettre cette réutilisation est la création des correspondances entre les ressources de la plate-forme POSIX et les stéréotypes du profil SRM. Cet effort consiste à identifier la ressource concurrente dans POSIX et l'annoter avec *SwSchedulableResource*, et identifier la ressource d'exclusion mutuelle et l'annotée avec *SwMutualExlusionResource*. Par conséquent, la ressource *pthread_t* est annotée par *SwSchedulableResource*, et la ressource *sem_t* est annotée avec *SwMutualExlusionResource*. Cette description de la plate-forme avec SRM permet donc à la transformation générique de créer deux instances de la ressource *pthread_t* correspondant aux classes *Producer* et *Consumer* et une instance de la ressource *sem_t* correspondant à la classe *Counter*, comme le montre la figure 4.12.b.

2.2 Apport et limites de l'approche offerte par SRM

L'utilisation d'une infrastructure de transformation générique basée sur des modèles de plates-formes d'exécution explicites a apporté un degré de flexibilité supplémentaire par rapport aux infrastructures de transformation spécifique. Pour une même transformation et pour un même modèle applicatif (Producteur/Consommateur), plusieurs modèles applicatifs cibles sont générés. Les modèles de plates-formes paramètrent les exécutions des transformations et donc capitalisent les descriptions de ces transformations.

Cependant, la comparaison entre les modèles générés par une transformation spécifique et ceux générés par une transformation générique reflète que les transformations génériques n'ont permis qu'une simple transition structurelle du modèle applicatif indépendant de la plate-forme vers les modèles spécifiques aux plates-formes. Des tels modèles sont insuffisants pour la génération automatique d'un code applicatif complet et exécutable. Dans cet objectif, nous devons affiner l'analyse afin d'obtenir une synthèse des supplémentaires. Le but sera de pouvoir générer des modèles applicatifs cibles complet[1] permettant la génération du code exécutable. Ces modèles générés doivent être identique en niveau de

1. Un modèle spécifique à la plate-forme est complet si le code généré à partir de ce modèle ne nécessite aucune intervention humaine pour l'exécuter

complétude à ceux générés par des transformations spécifiques.

2.2.1 Analyse

Afin de pouvoir obtenir des modèles spécifiques à la plate-forme complet et permettant la génération du code exécutable à partir d'une transformation générique, on peut identifier les problématiques suivantes :

Implémentations spécifiques à la plate-forme :

Une implémentation spécifique à la plate-forme est un code spécifique au langage de programmation de la plate-forme cible. Il sert à réaliser un service ou un comportement spécifique à une ressource de la plate-forme ou encore un patron de conception imposé par la plate-forme. Par exemple, dans les modèles générés par des transformations spécifiques (figures 4.7 et 4.8), plusieurs implémentations spécifiques à la plate-forme peuvent être identifiées. Les opérations *init* et *create* sont des implémentations spécifiques qui servent à créer les threads et les sémaphores. Les opérations *run* et *staticOp* et leurs comportements périodiques associés sont caractéristiques de l'impact des patrons de conception imposés par les plates-formes Java et C++/POSIX respectivement.

Appels des services offerts par les ressources des plates-formes :

Les appels des services des ressources de la plate-forme peuvent être divisés en deux types.

1. Le premier type concerne les appels qui sont émis depuis le code métier de l'application. C'est le concepteur de l'application qui précise le service qu'il faut appeler pour avoir le comportement désiré. Par exemple, dans l'application Producteur/Consommateur, si le consommateur décide d'arrêter la production après un certain nombre de mesures, alors l'appel d'un service permettant d'arrêter le thread du producteur doit être effectué.
2. Le deuxième type des appels concerne les appels qui doivent absolument être effectué pour réaliser un certain mécanisme de communication ou de synchronisation dans l'application. Par exemple, pour réaliser le mécanisme de synchronisation des appels des opérations de l'objet protégé *counter*, les services *acquire* (met l'appelant en attente jusqu'à ce que la ressource soit disponible) et *release* (rend une ressource disponible à nouveau après que l'appelant a terminé de l'utiliser) de la ressource sémaphores doivent être appelés, comme indiqué dans la figure 4.10.

Ces deux besoins représentent les limites principales envers l'adoption des transformations génériques. Une approche visant à adopter des transformations réutilisables et portables doit premièrement offrir des heuristiques de modélisation pertinentes et efficaces permettant d'extraire toutes les implémentations spécifiques à la plate-forme des transformations de modèles. Deuxièmement, elle doit offrir la possibilité de gérer les appels vers les services des ressources de la plate-forme d'une manière indépendante

de la plate-forme. Une fois que les solutions sont trouvées, la spécification d'un cadre méthodologique et technologique pour le développement des systèmes multitâches à base des transformations génériques réutilisables et portables sera possible.

On laisse volontairement de coté, dans cette étude, les appels des services d'une manière indépendante de la plate-forme depuis le code métier de l'application.

2.3 Bilan

Dans cette section, une comparaison a été effectué entre un modèle d'une application généré avec des transformations de modèles spécifiques et un autre modèle pour de la même application généré avec l'infrastructure de transformation générique à base de SRM proposé dans [75]. A l'issue de cette comparaison, nous avons identifié deux problématiques pour lesquelles il faut trouver une solution afin de pouvoir utiliser les transformations génériques pour le déploiement des systèmes multitâches et, par conséquence la génération du code exécutable.

La section suivante traite la première problématique, elle propose ainsi des heuristiques de modélisation permettant d'extraire les implémentations spécifiques à la plate-forme des transformations et de les supporter dans un modèle de cette plate-forme. La deuxième problématique sera traitée dans le chapitre suivant.

3 Gestion des implémentations spécifiques à la plate-forme

Afin de fournir une gestion modulaire des implémentations spécifiques à la plate-forme, une approche étendue de modélisation des plates-formes est proposée. L'idée principale de cette approche consiste à décrire, comme proposé dans [75], avec SRM, la structure des ressources et des services offerts par les plates-formes d'exécution et, en plus, à décrire les comportements (ou sémantiques d'exécution) de ces ressources et ces services. Le but sera, d'une part, d'abstraire les concepteurs des transformations des implémentations spécifiques à la plate-forme, et d'autre part, d'offrir aux fournisseurs de plates-formes ou aux experts de plates-formes, une approche de modélisation à suivre pour fournir des modèles détaillés de leurs plates-formes encapsulant toutes les implémentations spécifiques.

L'approche de modélisation proposée est une approche itérative et chaque itération permet de modéliser entièrement une ressource de la plate-forme logicielle. Cela comprend aussi la modélisation des services, des propriétés et des patrons d'utilisation de la ressource. Cette approche de modélisation emploie principalement le diagramme de classe d'UML car il permet de modéliser explicitement les ressources, les services, les propriétés et les implémentations spécifiques notamment à l'aide des classes (*Class*) constituées des opérations (*Operation*), des propriétés (*Property*) et des comportements opaques (*OpaqueBehavior*).

En considérant que la modélisation en UML avec application de profils n'est pas une pratique courante chez les fournisseurs de plates-formes, des heuristiques de modélisations sont proposées pour les guider dans la modélisation de leurs plates-formes logicielles.

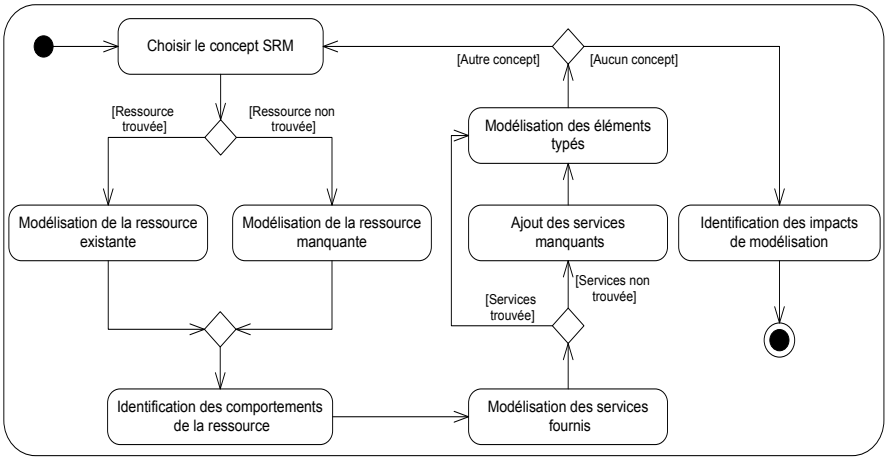

FIGURE 4.13 – Séquencement des étapes de modélisation

3.1 Heuristiques de modélisation de la plate-forme

La figure 4.13 illustre, par un diagramme d'activité, le processus de modélisation, c'est-à-dire, le séquencement des heuristiques de modélisation des plates-formes logicielles. Ce séquencement est constitué de huit actions de modélisation et de trois nœuds de décision. Les huit actions permettent de modéliser des ressources de la plate-forme en les annotant avec le stéréotype correspondant du profil SRM. Elles permettent aussi de référencer les services et les propriétés offertes par une ressource aux propriétés du stéréotype annotant la ressource. Enfin, ces actions permettent de capturer les patrons d'utilisations des ressources modélisées. Les trois nœuds de décision servent à tester respectivement la présence des ressources, des services et des concepts SRM pour la modélisation.

Dans la suite, les heuristiques de modélisations sont détaillées étapes par étapes. Et pour illustrer chacun de ces heuristiques, des exemples de modélisation des deux plates-formes logicielles Java et C++/POSIX sont présentés.

3.1.1 Identification du concept

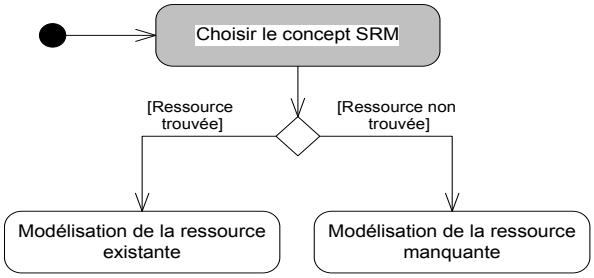

FIGURE 4.14 – Première action : choix du concept SRM

Le profil SRM est composé d'un ensemble de stéréotypes. Un stéréotype avec les valeurs de ses propriétés spécifie une sémantique d'un concept. Par conséquence, un concept du profil SRM correspond à un stéréotype avec des valeurs bien définies de ces propriétés.

Par exemple, le stéréotype *SwSchedulableResource*, qui identifie les ressources concurrentes, possède les deux propriétés *type* et *isPreemtible*. La valeur de la propriété *type* peut être fixée à *periodic*, *aperiodic* ou *sporadic*. En plus, la valeur de la propriété *isPreemtible* peut être *true* ou *false*. Une ressource concurrente périodique et préemptible correspond au stéréotype *SwSchedulableResource* avec la valeur de la propriété *type* fixée à *periodic* et la valeur de la propriété *isPremptible* fixé à *true*. Le stéréotype SRM avec ses valeurs des propriétés constitue un concept de SRM.

La première étape de modélisation de la plate-forme consiste donc à identifier un concept SRM qui permet de capturer la spécificité et les contraintes d'utilisation de la ressource de la plate-forme. Ce concept doit annoter une ressource de la plate-forme. La sémantique de la ressource doit être conforme à la sémantique du concept SRM. Pour cela, une fois que le concept est identifié, le modélisateur doit vérifier au début si une ressource de la plate-forme correspond à ce concept. Si la ressource est fournie nativement par la plate-forme, alors il procède vers l'étape de la modélisation de la ressource existante, sinon, il passe à l'étape de la modélisation de la ressource manquante.

3.1.2 Modélisation d'une ressource existante dans la plate-forme

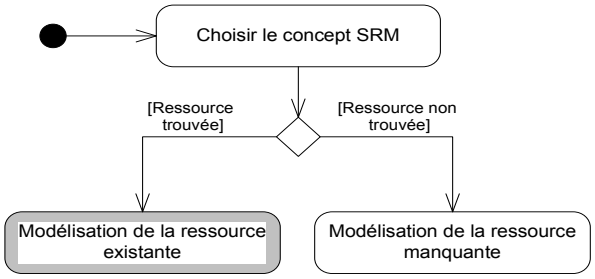

FIGURE 4.15 – Deuxième action : modélisation de la ressource existante

La ressource fournie nativement par la plate-forme logicielle et qui correspond à un stéréotype SRM est modélisée par une classe UML portant le même nom. Ensuite, cette classe est ensuite annotée par le stéréotype SRM.

Par exemple, les plates-formes Java et C++/POSIX offrent respectivement la ressource *Thread* et la ressource *pthread_t*. Ces deux ressources possèdent la même sémantique du stéréotype *SwSchedulableResource* avec la valeur de la propriété type fixé à *aperiodic* et la propriété *isPreemtible* fixé à *true*. La figure 4.16 montre les modèles des ressources *Thread* et *pthread_t* modélisées comme décrit précédemment.

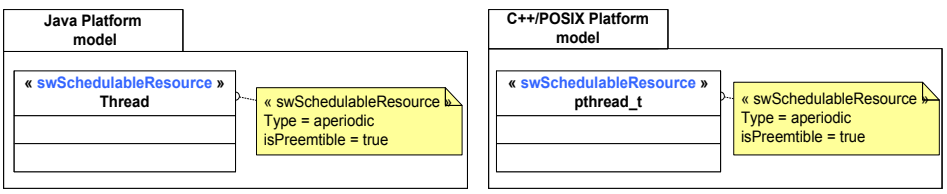

FIGURE 4.16 – Modélisation des ressources existantes

3.1.3 Modélisation d'une ressource manquante dans la plate-forme

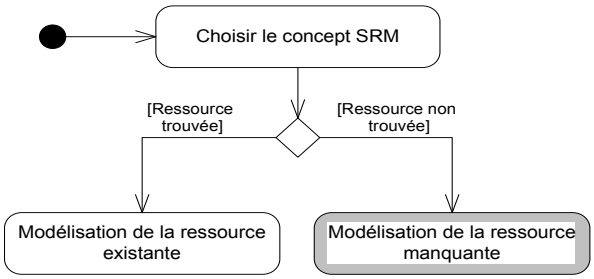

FIGURE 4.17 – Troisième action : modélisation de la ressource manquante

Ce cas se présente lorsque la plate-forme ne fournit aucune ressource qui correspond à un concept dans SRM. Dans ce cas, il est nécessaire de réaliser un patron de conception permettant d'implémenter une telle ressource. La sémantique globale de la ressource implémentée correspond alors à la sémantique du concept SRM.

Dans cette approche, la ressource manquante est remplacée par une nouvelle classe UML portant un nom approprié. Ensuite, le concepteur doit définir une relation d'héritage de cette classe, dans le cas où elle spécialise une ressource déjà modélisées. La figure 4.18 illustre ce cas de modélisation. Le concept d'un thread périodique n'est pas fourni nativement par les plates-formes Java et C++/POSIX. Par conséquence, les classes *PeriodicThread* et *PeriodicPthread_t* sont ajoutées dans le modèle de la plate-forme Java et C++/POSIX. Ces deux classes sont annotées par le stéréotype *SwSchedulableResource* avec sa propriété *type* est fixée à *periodic*. En plus, vue que ces deux classes sont une spécialisation des deux ressources *Thread* et *pthread_t* déjà modélisées (comportement apériodique d'une thread), elles sont spécifiées par la relation d'héritage comme des classes héritières.

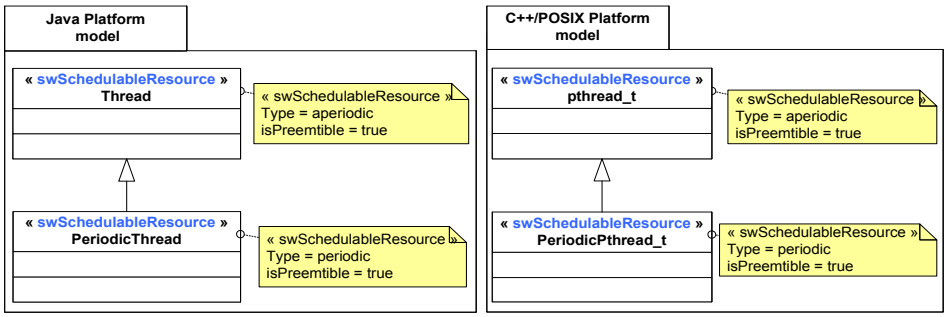

FIGURE 4.18 – Modélisation des ressources manquantes

3.1.4 Identifications des comportements de la ressource

Pour être utilisée, la ressource de la plate-forme doit être premièrement créée et initialisée. En plus, la plate-forme d'exécution peut imposer l'application d'un certain patron d'utilisation pour pouvoir utiliser la ressource. Dans le contexte de cette étude, un patron d'utilisation propose une solution architecturale et comportementale à une famille de problèmes de concurrence. Il peut être mis en œuvre dans la classe UML de la ressource

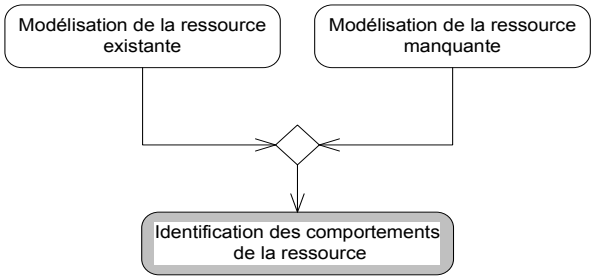

FIGURE 4.19 – Quatrième action : comportements de la ressource

avec des opérations, des propriétés et des comportements opaques.

Création et initialisation des ressources

La création et l'initialisation de la ressource nécessitent des appels à des opérations fournies par la plate-forme. Ceci se réalise à travers un comportement codé avec le langage de programmation supporté par la plate-forme. Ce comportement est indépendant de la logique de l'application. Par conséquence, une opération avec un comportement permettant la création et l'initialisation de la ressource peut être codé et appelé à chaque fois qu'une instance de cette ressource doit être créée. L'idée est d'ajouter cette opération dans le modèle de la ressource et d'encapsuler son comportement dans un UML *OpaqueBehavior*. L'opération ajoutée est alors référencée par la propriété *createServices* du stéréotype SRM qui annote la ressource. L'instance de la ressource créée est modélisée par une propriété ajoutée dans le modèle de la ressource et typée par la ressource. Par exemple, dans la figure 4.20, les deux opérations *init* et les deux propriétés *thrd* sont ajoutées dans les modèles des ressources du *Thread* Java et du *pthread_t* POSIX. Les opérations *init* sont référencées par la propriété *createServices* du stéréotype *SwSchedulableResource*.

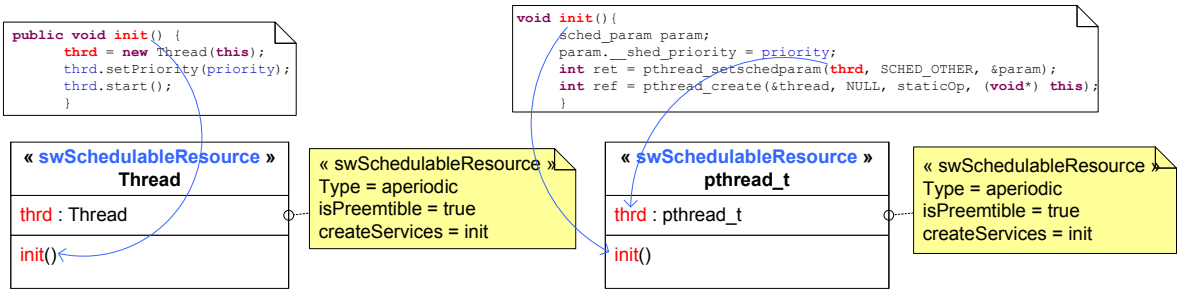

FIGURE 4.20 – création des ressources

L'initialisation de la ressource créée à des valeurs spécifiées par le concepteur d'une application, est réalisée à l'aide des propriétés typées. Par exemple, la valeur de la priorité des threads est précisée par le concepteur de l'application. Cette valeur doit être passée ensuite comme paramètre pour des implémentations spécifiques à la plate-forme afin de fixer la priorité du thread. Or on ignore cette valeur au moment de la modélisation, cela permet alors de remplacer la valeur concrète de la priorité par une propriété typée par le

même type de cette valeur concrète imposée par l'implantation sur la plate-forme spécifique. Ensuite, la valeur de la propriété est fixée à la valeur spécifiée par le concepteur au moment du démarrage de la transformation. Dans la figure 4.21, la propriété *priority* de type *int*, dans les opérations *init*s (ligne 3), est utilisée pour fixer la priorité des threads Java et C++/POSIX. Cette propriété est ajoutée dans les modèle des ressources *Thread* et *pthread_t*. La figure 4.21 illustre une telle modélisation.

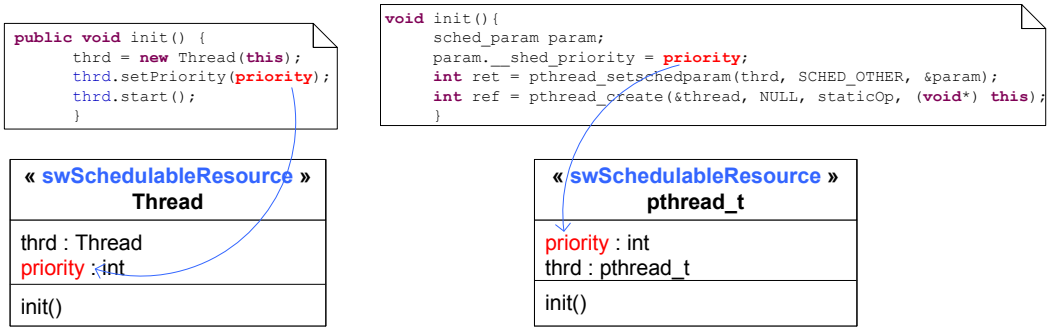

FIGURE 4.21 – Initialisation des ressources

A noter que cette technique d'initialisation ne peut être utilisée que si la propriété à initialiser peut être référencée par une propriété de type *TypedElement* d'un stéréotype SRM. En effet, si la propriété ajoutée est référencée par une propriété d'un stéréotype SRM, une transformation de modèle générique à base de SRM peut l'identifier et la reconnaître. Sinon, la transformation ne peut pas l'atteindre et dans ce cas, une solution consiste à initialiser la propriété par une valeur par défaut.

Identification des patrons d'utilisation

Pratiquement, lors de l'élaboration d'une application temps réel, l'utilisation de certaines ressources exige l'application d'un patron d'utilisation. Cette approche propose de capturer les patrons d'utilisation dans le modèle de la ressource. Il pourra ensuite être utilisé dans les différentes situations où une telle ressource est réclamée.

Par exemple, pour utiliser un thread dans le contexte d'une plate-forme C++/POSIX, cette dernière impose, comme présenté dans la section 2.1.1, que lors de la création et l'activation de ce thread, une fonction ou une méthode statique soit activée. Par conséquence, dans la figure 4.22.a, l'opération statique *staticOp* est modélisée dans le modèle de la ressource *pthread_t*. Pour spécifier que l'opération *staticOp* est la routine qui doit être exécutée dans le contexte de ce thread POSIX et permettre donc à l'outil de transformation de l'identifier, elle est référencée par la propriété *entryPoints* du stéréotype *SwSchedulableResource* annotant la ressource *pthread_t*. Ainsi, lors du démarrage de la transformation, le comportement de l'opération *staticOp* sera fixé au flot d'exécution décrit au niveau du modèle de l'application.

Concernant la ressource *Thread*, la plate-forme Java impose qu'elle implémente l'interface *Runnable*. Par conséquence, la ressource *Thread* doit toujours encapsuler la méthode *run*. Cette méthode est exécutée directement lors de la création et de l'activation du thread Java.

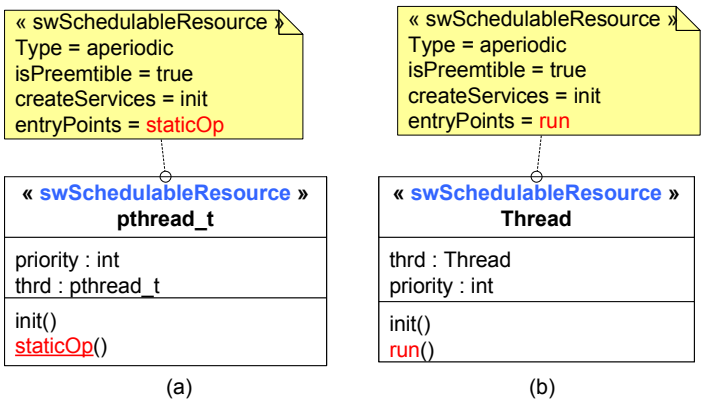

FIGURE 4.22 – Mise en œuvre des patrons d'utilisation

Elle est donc référencée par la propriété *entryPoints* du stéréotype *SwSchedulableResource*. La figure 4.22.b montre la méthode *run* ajouté au modèle de la ressource Java avec l'interface *Runnable* implémentée par cette ressource.

Identification des comportements des ressources ajoutées

Pour identifier les comportements des ressources ajoutées, les deux sous-étapes de modélisation précédentes sont appliquées : 1) création et initialisation de la ressource et 2) Identification des comportements de la ressource. Cependant, la première sous-étape peut être ignorée si tous les cas suivants sont vérifiés :

- La ressource ajoutée étend une ressource déjà modélisée.
- La ressource déjà modélisée contient une opération référencée par la propriété *createServices* du stéréotype annotant la ressource.
- L'opération existante peut être utilisée pour créer la ressource ajoutée.

Si ces trois cas ne sont pas vérifiés une nouvelle opération doit être créée et référencée à la propriété *createServices* du stéréotype annotant la nouvelle ressource.

Par exemple, pour les threads périodiques modélisés dans la figure 4.18, les mêmes opérations ajoutées pour créer et initialiser les classes parents peuvent être utilisées pour créer ces threads périodiques et le concepteur n'a pas besoin de créer une autre opération. Par conséquent, dans la figure 4.23, la propriété *createServices* du stéréotype *SwSchedulableResource* annotant le la ressource périodique *PeriodicThread* de Java référence l'opérations *init* dans la classe *Thread* parent.

Le patron d'utilisation des ressources ajoutées est mis en œuvre comme indiqué dans la sous-étape précédente.

Par exemple, une mise en œuvre des threads périodiques Java ou C++/POSIX consiste à définir un comportement pour les opérations *run* et *staticOp* des classes *PeriodicThread* et *PeriodicPthread_t* qui appelle périodiquement une opération *launch*. La figure 4.24 illustre la mise en œuvre d'un tel comportement dans la classe *PeriodicThread* de Java. Le comportement périodique associé à l'opération *run* correspond à la routine exécutée une fois que le thread est activé. L'opération *launch*, appelée périodiquement, est modélisée

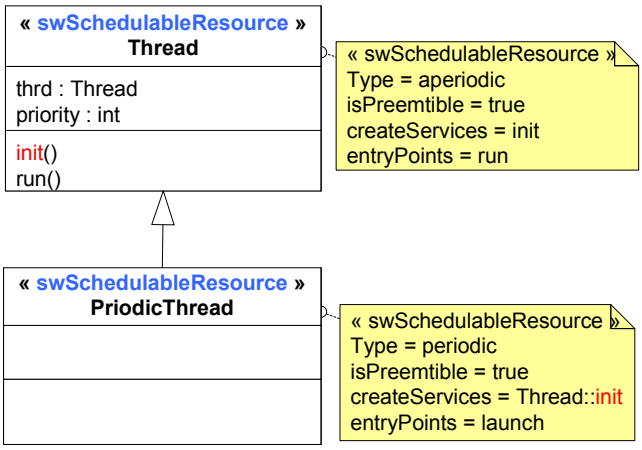

FIGURE 4.23 – Création d'une ressource ajoutée

dans le modèle des ressources et elle est référencée par la propriété *entryPoints* du stéréotype *SwSchedulableResource*. La valeur de la période est spécifiée par le concepteur. Par conséquent, une propriété *period* est utilisée pour définir la valeur de la période. Elle est ajoutée au modèle de la ressource *Thread* et P*thread_t*.

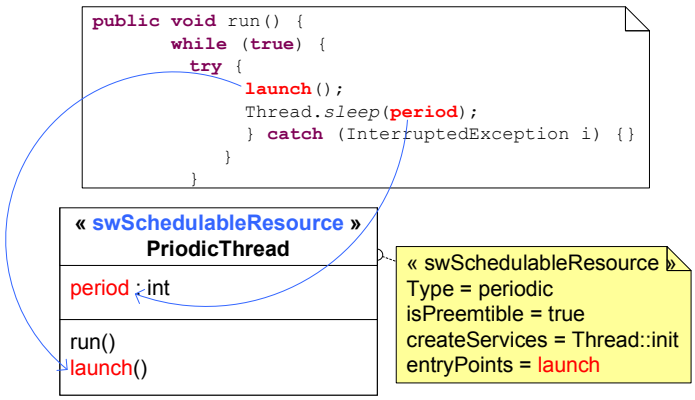

FIGURE 4.24 – Mise en œuvre d'un comportement périodique

3.1.5 Modélisation des services fournis

FIGURE 4.25 – Cinquième action : modélisation des services fournis par la ressource

La modélisation d'un service de la ressource se concrétise par le référencement de ce service par une propriété typé par un *BehavioralFeature* d'un stéréotype SRM. Dans ce cas, la sémantique dénotant la propriété du stéréotype décrit la sémantique du service référencé. Cependant, lors de la modélisation de la plate-forme, on peut être confronté à différents

cas de modélisation. Ainsi, un ou plusieurs services de la plate-forme peuvent être décrits par la même propriété du stéréotype. Réciproquement, une ou plusieurs propriétés du stéréotype peuvent être utilisées pour décrire un même service. Cette diversité des choix des services empêche l'automatisation des appels des services de la ressource dans des transformations génériques à base de SRM. Par conséquence, la définition supplémentaire d'une correspondance bijective entre propriétés du stéréotype et services de la ressource est nécessaire.

On distingue alors deux cas selon si on doit déclarer une propriété référençant plusieurs services ou plusieurs propriétés référence un seul service.

Une propriété du stéréotype référence plusieurs services

Lorsque deux ou plusieurs services peuvent être référencés par la même propriété du stéréotype un seul service doit être choisi par le modélisateur de la plate-forme. Le choix du service sera guidé par la sémantique et le rôle que doit accomplir la ressource qui fournit ces services. Ce rôle est identifié par le stéréotype appliqué et les valeurs de ses propriétés fixées.

Par exemple, dans la figure 4.26, la ressource *linkedList* de la plate-forme Java est annotée par le stéréotype *MessageComResource*. Les propriétés *sendServices* et *receiveServices* de ce stéréotype doivent référencer chacun un seul service de la ressource. Cependant, la ressource *LinkedList* fournie plusieurs opérations qui peuvent être référencées par *sendServices* comme par exemple les opérations *get*(int) et *poll*. Ces deux opérations permettent de retrouver et d'envoyer un élément de la liste. De même, la ressource *LinkedList* fournit plusieurs opérations qui peuvent être référencées par *recieveServices* tels que les opérations *add* et *addFirst*. Ces deux opérations permettent d'ajouter des éléments à la même liste. Or, pour que la transformation soit déterministe, une seule opération doit être référencée à la propriété *sendServices* ainsi qu'à la propriété *receiveServices*, le choix de l'opération appropriée dépend alors du politique du traitement des données adoptée pour la *LinkedList*. Dans la figure 4.26, la politique de traitement des données sélectionné pour la *LinkendList*

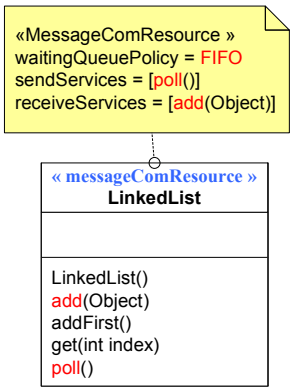

FIGURE 4.26 – Modélisation des services des ressources

est le FIFO (premier arrivé, premier servi) (la valeur de la propriété *waitingQueuePolicy* est fixé à FIFO). Par conséquent, l'opération qui doit être référencée par la propriété *sendServices* est l'opération qui récupère et renvoie le premier élément de la liste, ce qui correspond

à l'opération *poll*. Et l'opération qui doit être référencé à la propriété *receiveServices* est l'opération *add*(Object). De cette manière, les services de la ressource sont référencés aux propriétés du stéréotype.

Plusieurs propriétés du stéréotype référencent un service de la ressource

Ce cas se manifeste lorsque la sémantique d'un service offert nativement par la plate-forme est composée de la sémantique de deux ou plusieurs services réunis. Dans ce cas, cette opération est référencée par les deux propriétés du stéréotype.

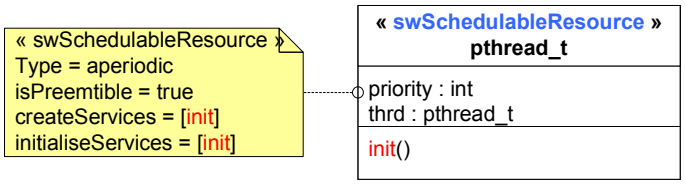

FIGURE 4.27 – Modélisation d'un service référencé par deux propriétés

Par exemple, dans la figure 4.20, dans l'étape de la modélisation 4.19, l'opération *init* ajoutée dans le modèle de la ressource pthread_t permet de créer et d'initialiser la ressource. Cette opération est référencée par la propriété *createServices* car elle permet de créer la ressource. Cette opération permet aussi d'initialiser la ressource, comme présenté dans l'étape 4.19, elle peut donc être référencée par la propriété *initializeServices* du stéréotype *SwSchedulableResource*(figure 4.27).

3.1.6 Ajout des services manquants

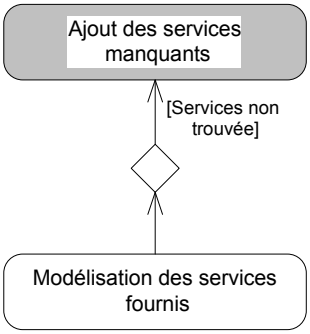

FIGURE 4.28 – Sixième action : modélisation des services manquantes

Si les propriétés typées par un *BehavioralFeature* du stéréotype référencent tous un service de la plate-forme, le concepteur est invité à passer à l'étape de modélisation suivante. Si, par contre, une ressource ne fournit pas un service qui correspond à une propriété d'un stéréotype SRM, un service correspondant doit être ajouté au modèle de ressources. Celui-ci est est mis en œuvre dans le modèle de la ressource.

Par exemple, les ressources Thread de Java et pthread_t de POSIX ne fournissent pas

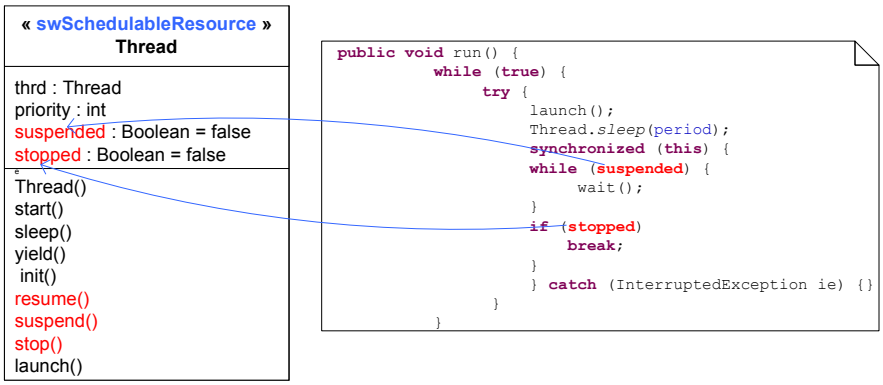

FIGURE 4.29 – Modélisation des services manquantes

```
void resume() {
      suspended = false;
      notify();
      }
```

```
void suspend() {
      suspended = true;
      }
```

FIGURE 4.30 – Opérations de suspension et de reprise de l'exécution du thread

nativement des services de suspension, de reprise et d'arrêt d'un thread. Afin d'ajouter des tels services, la mise en œuvre peut être conçu de sorte que la méthode *run* vérifie périodiquement si ce thread devrait suspendre, reprendre ou arrêter sa propre exécution. Typiquement, cela se réalise par la création de deux variables booléennes : une pour suspendre et reprendre, et l'autre pour arrêter. Pour suspendre et reprendre, tant que la variable correspondante est mise à vrai, la méthode *run* doit continuer à laisser le thread s'exécuter. Si cette variable est définie à faux, le thread doit se mettre en pause. Pour la variable d'arrêt, si elle est fixée sur vrai, le thread doit se terminer.

Dans la classe Thread de la figure 4.29, les opérations *resume*, *suspend* et *stop* sont ajoutées. En plus, deux propriétés booléennes *suspended* et *stopped* sont définies dans la classe Thread. Elles sont de type booléen initialisées à faux. La valeur faux représentent le fait que lorsque la ressource est créée, elle n'est pas arrêtée ni suspendue. Le test des valeurs des propriétés se fait dans un bloc synchronisé (figure 4.29, opération *run*) encapsulé dans l'opération *run*. Si la variable *suspended* est vrai, la méthode *wait* est invoquée pour suspendre l'exécution du thread. Pour suspendre l'exécution du thread, l'opération suspend doit être appelée pour fixer la propriété *suspended* à vrai. Pour reprendre l'exécution, l'opération *resume* est appelée pour fixer la propriété *suspended* à faux et invoquer *notify* qui redémarre le thread (figure 4.30). Enfin, pour arrêter le thread, l'opération *stop* fixe la propriété *stopped* à vrai.

3.1.7 Modélisation des éléments typés

Les stéréotypes du profil SRM possèdent des propriétés de type *TypedElement*. Ces propriétés sont utilisées pour référencer les éléments typés des ressources de la plate-forme, comme par exemple la propriété *priorityElements* du stéréotype *SwSchedulableResource* typée par *TypedElement[*]*. Dans cette approche, les éléments typés des ressources sont les propriétés qui étaient ajoutées aux modèles des ressources durant les étapes de modélisation.

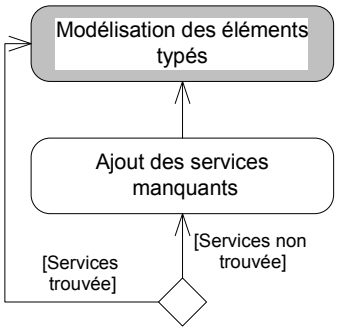

FIGURE 4.31 – Septième action : modélisation des éléments typés

En d'autres termes, ils correspondent aux propriétés utilisées lors du mise en œuvre des aspects comportementaux spécifiques aux ressources. Par exemples, les propriétés *priority* et *period* utilisées pour initialiser les ressources threads.

Parmi ces éléments typés, si un élément correspond à une propriété de type *TypedElement* du stéréotype annotant sa ressource, il est référencé par cette propriété. Cela veut dire que des éléments typés peuvent rester sans référencement comme par exemple les propriétés ajoutées *suspended* et *stopped*. La figure 4.32 montre le référencement des éléments typés des ressources concurrentes Java et C++/POSIX par les propriétés correspondantes du stéréotype *SwSchedulableResource*. Les propriétés *period* et *priority* sont référencées respectivement par les propriétés *periodElements* et *priorityElements* du stéréotype *SwSchedulableResource* et les propriétés *thrd* sont référencées par la propriété *identifierElements* du stéréotype *SwSchedulableResource*.

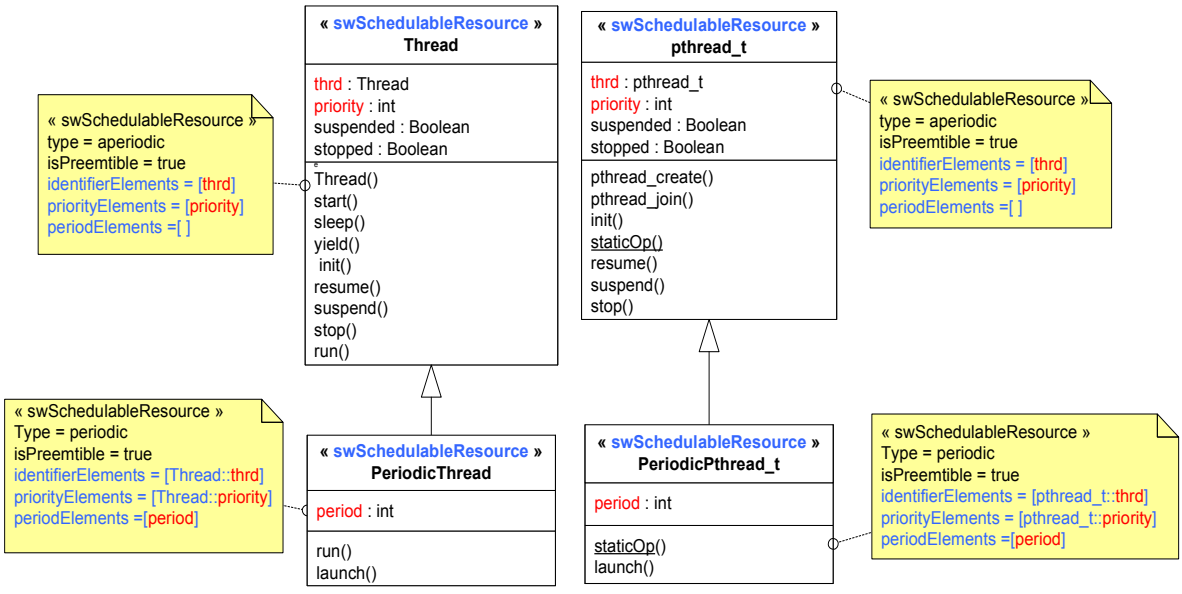

FIGURE 4.32 – Modélisation des éléments typés des ressources

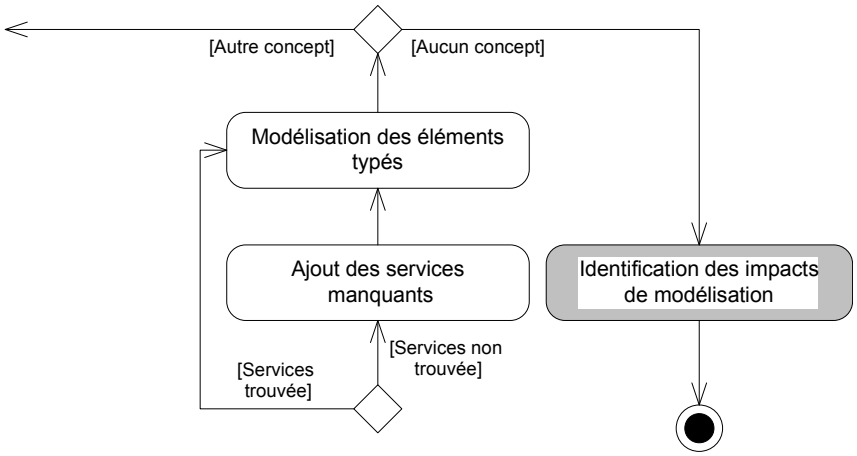

FIGURE 4.33 – Huitième action : impacts des règles de modélisation

3.1.8 Identification des impacts des règles de modélisation

L'approche proposée pour modéliser la plate-forme logicielle a entraîné l'ajout d'opérations et de propriétés dans le modèle des ressources de la plate-forme. Il faut distinguer les opérations et les propriétés qui étaient ajoutées, de celles fournies nativement par la plate-forme. En effet, tout élément ajouté dans le modèle de la ressource est une implémentation spécifique à la plate-forme nécessaire à l'exécution de l'application. Il doit être donc généré dans le modèle de l'application spécifique à la plate-forme produit par la transformation générique. Ainsi, lorsque tous les éléments ajoutés sont distingués de ceux fournis nativement, la transformation exécute un simple copie de ces éléments dans le modèle de l'application généré spécifique à la plate-forme. Le résultat est donc un modèle spécifique à la plate-forme qui encapsule toutes les implémentations spécifiques à la plate-forme qui avant étaient générées par des transformations spécifiques.

Pour cela, nous proposons d'identifier l'impact de la modélisation détaillée de la plate-forme à l'aide d'un ensemble d'annotation (stéréotypes) regroupés dans un profil minimal appelé DPD (Detailed Platform Description). Il est composé de quatre stéréotypes et pourra être étendu ultérieurement si besoin :

- *Added_Property_Source* est un stéréotype appliqué à des propriétés qui prennent leurs valeurs à partir du modèle d'application. Par exemple, les propriétés *period* et *priority* dans le modèle de la ressource *Pthread_t* (figure 4.21) et utilisées pour initialiser un thread dans l'étape de modélisation 4.19.
- *Added_Property_Target* est un stéréotype appliqué aux propriétés qui sont nécessaires à la réalisation des services non fournis par une ressource. Ces propriétés peuvent avoir des valeurs par défauts dans le modèle de la ressource qui seront modifiées par les services ajoutés pour réaliser le comportement désiré. Par exemple, les propriétés *stopped* et *suspended* qui sont utilisées pour réaliser la suspension, la reprise ou l'arrêt d'un thread dans la figure 4.29 de l'étape de modélisation 4.28, et les propriétés représentant les instances des ressources créées comme les propriétés *thrd*s des ressources *Thread* et pthread_t dans la figure 4.20.
- *Added_Operation* est un stéréotype appliqué aux opérations qui sont ajoutées aux

modèles des ressources de la plate-forme. Le comportement de ces opérations peut encapsuler des implémentations spécifiques à la plate-forme. Par exemple, toutes les opérations ajoutées pour réaliser la mise en œuvre des aspects comportementaux des ressources, tels que les opérations *staticOp* et *launch* proposées dans l'étape de modélisation 4.17 pour réaliser un comportement périodique. Les opérations ajoutées pour réaliser les services des ressources manquantes dans l'étape de modélisatoin 4.28, sont annotées aussi avec ce stéréotype.

– Added_Inner_Class est un stéréotype utilisé pour capturer le cas où des classes internes sont utilisées pour réaliser un certain patron d'utilisation. Ce stéréotype a été identifié lors de la modélisation détaillée de la plate-forme Java temps réel. Par exemple, dans Java temps réel [52], les classes internes qui implémentent l'interface *Runnable* sont souvent utilisées pour réaliser un patron d'utilisation d'une ressource.

La figure 4.34 présente les modèles détaillés des deux plates-formes Java et C++/POSIX annotés avec SRM et DPD. Dans cette figure, des ressources de communication et de synchronisation sont modélisées en plus en suivant la même approche de modélisation.

3.2 Bilan

Dans cette section, le profil SRM a été utilisé pour décrire les modèles de plates-formes logicielles d'exécution. En plus de la modélisation structurelle que la profil permet, une modélisation des comportements observables des ressources et des services de la plate-forme a été proposé. Le résultats est donc des modèles des plates-formes décrits d'une manière détaillé. Ces modèles des plates-formes peuvent être réalisés avec des heuristiques de modélisation qui été définis.

Les heuristiques proposées dans cette section supposent que pour chaque concept SRM une ressource correspondante de la plate-forme doit être trouvée. Cette hypothèse n'est pas toujours vraie. En effet, on peut être confronté à des situations où une plate-forme ne fournit pas une ressource qui correspond à un concept SRM, et ne permet pas de mettre en œuvre un patron de conception qui correspond à ce concept, ou au contraire, elle offre plusieurs choix de modélisation. Cette étude ne traite pas ces situations. Ils peuvent être considérés comme perspective de ce travail.

4 Conclusion

L'objectif de ce chapitre était de proposer des heuristiques de modélisation permettant de décrire la structure des ressources et des services offerts par les plates-formes ainsi que les comportements associés à ces ressources et ces services. L'application de ces heuristiques de modélisation sur une plate-forme logicielle d'exécution a fourni un modèle détaillé de cette plate-forme. Ces modèles encapsulent les implémentations spécifiques à la plate-forme qui, jusqu'à maintenant, été exprimées dans les transformations de modèles.

Pour atteindre cette objectif, le profil SRM a été présenté au début. Ensuite, une comparaison a été réalisée entre une transformation de modèles générique basée sur SRM et une transformation de modèles spécifique. Le résultat de cette comparaison a permis d'identifier eux problématiques pour lesquelles il faut trouver une solution afin de pouvoir

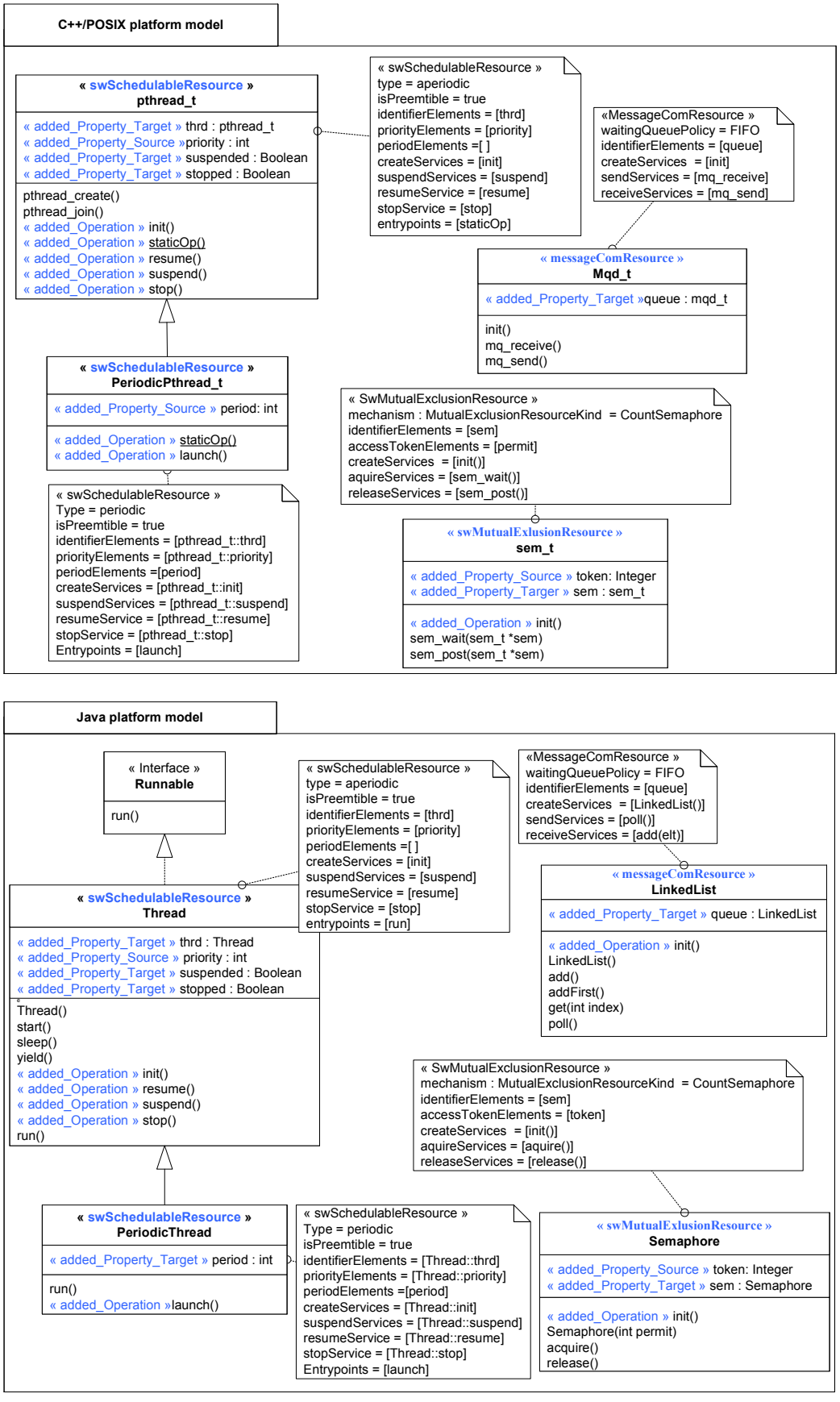

FIGURE 4.34 – Extrait des modèles détaillés des plates-formes Java et C++/POXIX

utiliser les transformations génériques pour le déploiement des systèmes multitâches. L'une des problématiques est la gestion des implémentations spécifiques à la plate-forme. Pour cela, les heuristiques des modélisations de la plate-forme ont été proposées.

Ces heuristiques de modélisation et les travaux d'analyse des besoins qui ont abouti à la proposition de ces heuristiques ont été publié dans [48, 49].

Avec cette séparation de préoccupation réalisée, les modèles détaillés doivent permettre la réalisation des transformations de modèles génériques, indépendant de la plate-forme sous-jacent et produire des applications à partir desquelles un code exécutable peut être généré. Afin de confirmer cette hypothèse, le chapitre suivant expérimente l'intégration des modèles détaillés de plates-formes dans une ingénierie générative dirigée par les modèles.

CHAPITRE 5

Intégration des modèles de plates-formes détaillés dans une ingénierie générative

Ce chapitre décrit un cadre de conception et de déploiement des systèmes multitâches à travers différentes plates-formes d'exécution. Le fondement principal dans ce cadre expérimental est le modèle détaillé des plates-formes, présenté dans le chapitre 3, qui permet de décrire les comportements des ressources et des services des plates-formes et, par conséquence, offre la possibilité d'écrire des règles de transformation de modèle génériques qui pourraient être réutilisées, même si la plate-forme cible est modifiée.

Pour cela, ce chapitre est divisé en trois sections. Premièrement, une approche réalisant la correspondance entre modèles applicatifs et plates-formes cibles est proposée. Deuxièmement, une méthode pour modéliser les systèmes multitâches est décrite. Enfin, dans une troisième et dernière section, une infrastructure de transformation générique est définie.

1 Définition du cadre expérimental

Les résultats du chapitre 4 ont montré qu'il existait un grand potentiel envers le développement de transformations de déploiement génériques. Afin de confirmer ce potentiel, ce chapitre propose la mise en place d'un cadre visant à expérimenter l'intégration des modèles détaillés des plates-formes dans un processus génératif outillé.

Le but de ce cadre est de favoriser le principe de séparation de préoccupations tout au long du processus de déploiement de l'application. Cela veut dire qu'à l'issue de ce processus, les concepteurs peuvent réaliser un modèle de leur système multitâche qui peut être utilisé pour des implémentations multi-plateforme. Le développeur des chaînes de transformation peut décrire des règles de transformation génériques, tandis que les fournisseurs de plate-forme, qui développent et vendent des systèmes d'exploitation temps réel, fournissent un modèle détaillé de ces systèmes.

Pour cela, la mise en place de ce cadre nécessite :

a) La définition des hypothèses de modélisation des applications utilisées par les concepteurs des systèmes multitâches.
b) La proposition d'une approche pour établir les correspondances entre les concepts du modèle applicatif (identifiés par les stéréotypes utilisés dans le modèle de l'application) et les concepts de la plate-forme cible.
c) La réalisation d'une infrastructure de transformation générique pour implémenter ces systèmes.

La figure 5.1 montre la structure du cadre expérimental proposé. La suite de cette section présente, tout d'abord, l'approche permettant d'établir les correspondances entre les concepts applicatifs et les ressources de la plate-forme. Cette approche permet de fournir le Modèle de correspondance qui est utilisé à l'entrée de l'infrastructure de transformation générique dans la figure 5.1. Ensuite, une méthode de conception des applications concurrentes, intégrée dans ce cadre, est présentée. Le résultat de cette méthode est le modèle applicatif utilisée aussi à l'entrée de l'infrastructure de transformation (Modèle applicatif de la figure 5.1). Cette dernière est détaillée à la fin de cette section. Elle permet la génération de modèles spécifique à la plate-forme à partir desquels on peut générer du code exécutable.

1.1 Modèle de correspondance

Lors de la modélisation des applications concurrentes multitâches, le concepteur utilise des concepts de haut niveau pour exprimer la concurrence, et par ailleurs les problèmes sous-jacents, à savoir la communication et la synchronisation. Ces concepts correspondent à des ressources fournies par la plate-forme cible. Les modèles détaillés des plates-formes encapsulent toutes les implémentations spécifiques permettant la réalisation de ces concepts de hauts niveaux. Afin d'automatiser la production des applications, il faut trouver un moyen pour réaliser la correspondance entre les concepts de haut niveau utilisés dans le modèle applicatif et les ressources de la plate-forme cible.

Par exemple, dans la figure 5.2, le concept applicatif *RtFeature* est utilisé pour modéliser une entité concurrente. Ce concept applicatif correspond à une *pthread_t* dans le modèle détaillé de la plate-forme C++/POSIX. Pour automatiser le déploiement de l'application,

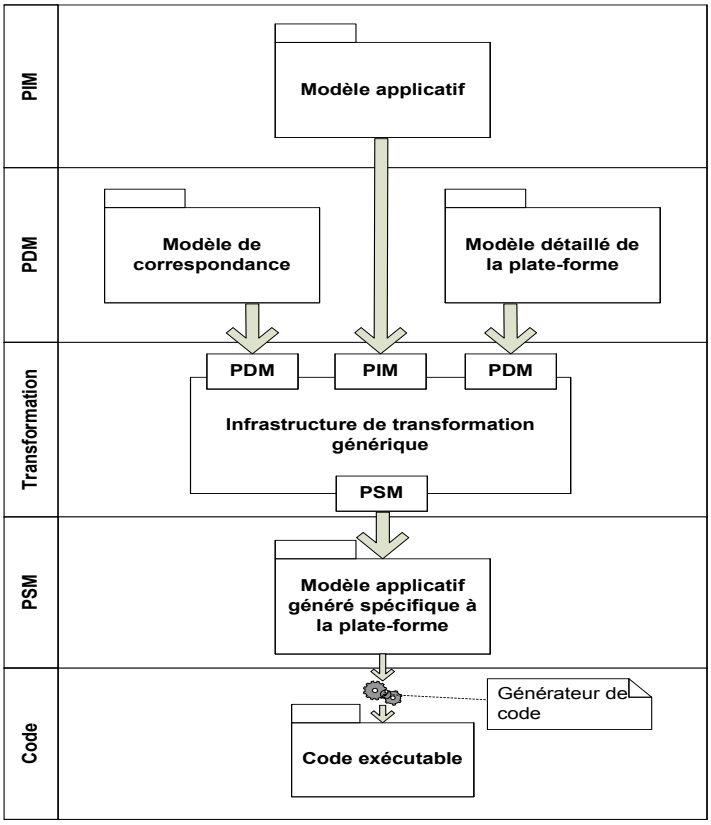

FIGURE 5.1 – Synoptique du cadre générative expérimental

il faut trouver un moyen pour automatiser la déduction des correspondances, comme par exemple entre *RtFeature* et *pthread_t*. Cette automatisation peut se réaliser en se basant sur une identification par stéréotypage.

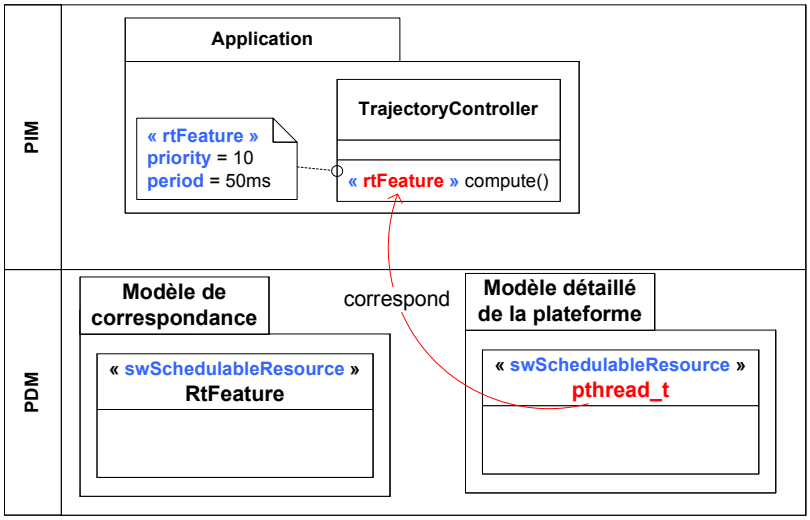

FIGURE 5.2 – Rôle du modèle de correspondance

En effet, puisque les modèles des plates-formes détaillées sont annotés avec SRM,

cette approche propose de décrire les concepts de haut niveau utilisés dans le modèle d'application (les stéréotypes) également avec SRM. Par conséquence, si une correspondance doit être établie entre un concept dans le modèle de l'application et un élément spécifique dans le modèle de la plate-forme, le concept de haut niveau est alors annoté par le même stéréotype SRM sur l'élément du modèle de la plate-forme. Dans ce cas, SRM est utilisé comme un pivot sémantique pour réaliser cette correspondance.

Ainsi, dans l'exemple de la figure 5.2, *pthread_t* est annoté avec *SwSchedulableResource*, pour déduire la correspondance entre *RtFeature* et *pthread_t*, il suffit d'annoter le concept *RtFeature* avec le même stéréotype. Par conséquence, dans la figure 5.2, le concept *RtFeature* est annoté donc par *SwSchedulableResource* dans le modèle de correspondance. Ainsi, une transformation est capable maintenant par identification des stéréotypes appliqués aux concepts applicatifs et aux ressources de la plate-forme de déduire la correspondance.

Par conséquence, afin d'automatiser la réalisation des correspondances, les concepts applicatifs utilisés pour modéliser la concurrence au niveau modèle doivent être annotées par SRM. L'annotation de ces concepts ne peut pas être direct parce que la norme interdis l'application d'un stéréotype sur un autre stéréotype. Pour cela, la section suivante présente une approche permettant la modélisation des concepts applicatifs afin de pouvoir automatiser la déduction des relations de correspondance.

Les principaux points forts de ce modèle de correspondance sont :

- Les concepteurs des applications ne sont pas contraints à un langage de conception de haut niveau pour exprimer la concurrence au niveau modèle. En effet, n'importe quel langage peut être utilisé une fois que la correspondance avec des concepts de SRM peut être établie. Par exemple HLAM, RT-UML [53] sont deux langages de conception de haut niveau basée sur UML, ils sont utilisés pour modéliser des applications concurrentes multitâches. Si les concepts de ces deux langages peuvent être décrits par SRM, le concepteur est alors libre de choisir le langage qui veut. Il peut même proposer son propre langage dont ses concepts seront aussi annotés par SRM.
- Les applications modélisées peuvent être portées vers une ou plusieurs plates-formes à l'aide des transformations génériques. En effet, SRM est utilisé comme un pivot sémantique entre les langages de conception de haut niveau et les différentes plates-formes d'exécution cible, ce qui permet d'écrire des règles de transformation génériques basées sur SRM.

1.1.1 Modélisation des concepts annotant le modèle applicatif

Un stéréotype du profil SRM possède des propriétés. Ces propriétés peuvent être classées en trois catégories selon leur type.

1. La première catégorie représente les propriétés de type *TypedElement* utilisées pour décrire les éléments typés des ressources, comme par exemple la propriété *periodElements* du stéréotype *SwSchedulableResource* qui référence des propriétés des ressources de la plate-forme qui spécifie la période.
2. La deuxième représente les propriétés de type *BehavioralFeature* utilisées pour décrire les services des ressources, comme par exemple, la propriété *createServices* du stéréotype *SwSchedulableResource* qui référence des opérations des ressources de la plate-forme permettant de créer la ressource (L'opération *pthread_create*).

3. La troisième catégorie représente les propriétés des types primitifs (booléen, entier, énumération) qui spécialisent le mode d'exécution de la ressource modélisée, comme par exemple la propriété *type* du stéréotype *SwSchedulableResource* qui permet de spécifier des modes d'exécution périodique, apériodique ou sporadique.

Les propriétés des stéréotypes utilisées pour décrire le modèle applicatif sont des éléments typés. Par conséquence, elles peuvent être référencées par des propriétés de la première catégorie d'un stéréotype SRM. Par exemple, la propriété *period* du stéréotype *RtFeature* utilisée pour annoter l'opération *compute*, dans la figure 5.2, peut être référencée par *periodElements* du stéréotype *SwSchedulableResource*. Cependant, ce référencement ne couvre pas toutes les propriétés du stéréotype. Les propriétés non référencées, peuvent spécifier un certain mode d'exécution des éléments applicatifs modélisés. Le mode d'exécution peut correspondre à un mode d'exécution spécialisé par les propriétés de la troisième catégorie d'un stéréotype SRM. Par exemple, le stéréotype *Ppunit* du HLAM de MARTE possède deux propriétés *concPolicy* et *memorySize*. La première propriété peut être référencée par la propriété *accessTokenElements* du stéréotype *SwMutualExlusionResource* correspondant au concept *PpUnit*, tandis que la deuxième ne correspond à aucune propriété de type *TypedElement* de ce stéréotype. Cependant, la valeur de la propriété *concPolicy* (*concurrent*, *sequentiel* ou *guarded*) peut spécifié un mode d'exécution qui correspond à un concept SRM (le stéréotype SwMutualExlusionResource avec la valeur de la propriété *mechanism* fixée à *CountSemaphore* ou *Mutex*).

Pour pouvoir modéliser toutes ces cas, des heuristiques de modélisation sont définies. La figure 5.3 illustre le séquencement des ces heuristiques dédiées à la modélisation des profils spécifiques aux applications concurrentes multitâches. Elles sont détaillées ci-dessous.

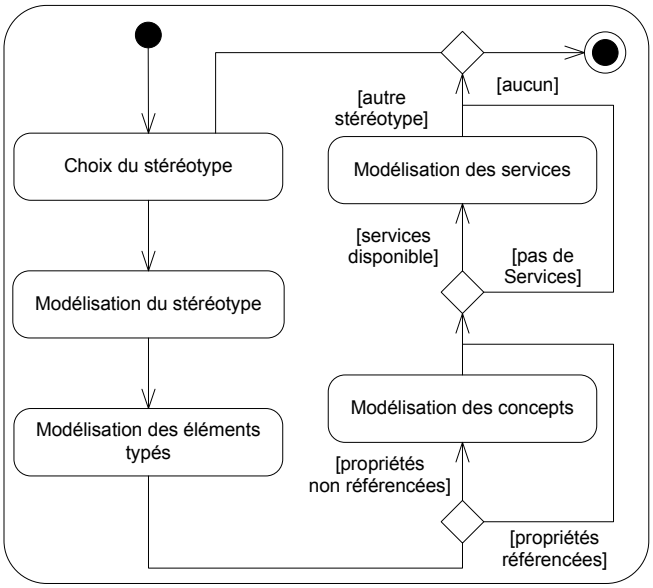

FIGURE 5.3 – Heuristiques de modélisation des stéréotypes

Choix du stéréotype

Lors de la modélisation de l'application, le choix de stéréotypes utilisés pour exprimer la

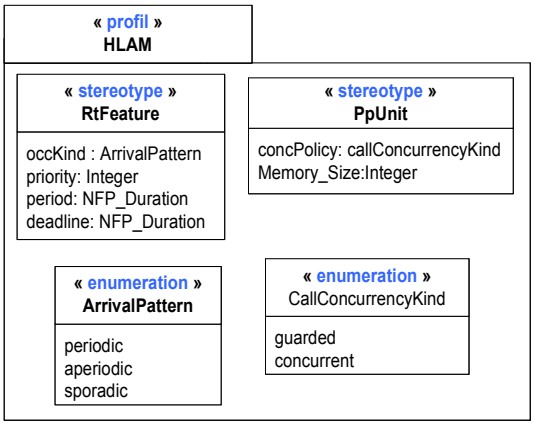

FIGURE 5.4 – Extrait simplifié du profil HLAM

concurrence doit prendre en compte une correspondance éventuelle avec des stéréotypes du profil SRM. La première étape de modélisation consiste donc à bien choisir le stéréotype à modéliser. Par exemple, le profil HLAM de MARTE offre le stéréotype *RtFeature* pour exprimer la concurrence et le stéréotype *PpUnit* pour exprimer la synchronisation. La figure 5.4 présente le modèle simplifié de ces deux stéréotypes. Ils ont une correspondance éventuelle avec les stéréotypes *SwSchedulableResource* et *SwMutualExlusionResouce*. Par conséquence, ces deux stéréotypes peuvent être utilisés lors de la modélisation de l'application.

Modélisation du stéréotype

Dans UML, le fait d'appliquer un stéréotype sur un autre stéréotype est interdit par la norme. Par conséquence, l'annotation des stéréotypes utilisés dans le modèle applicatif par les stéréotypes du profil SRM est interdite. Pour cela, dans cette approche, un stéréotype d'un profil est représenté dans le modèle de correspondance par une classe UML portant le même nom. Ensuite, la classe est annotée par le stéréotype du profil SRM correspondant. La figure 5.5 illustre le modèle des concepts *RtFeature* et *PpUnit*. Dans ce modèle, la classe *RtFeature* est annotée par *SwSchedulableResource* et la classe *PpUnit* Par *SwMutualExlusionResource*.

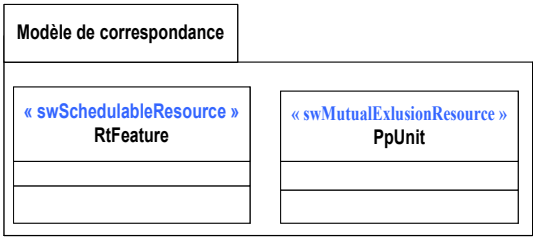

FIGURE 5.5 – Annotation des concepts du stéréotype par SRM

Modélisation des éléments typés du stéréotype

Les propriétés du stéréotype de haut niveau qui peuvent être référencées par des propriétés du stéréotype SRM, sont modélisées comme des propriétés dans la classe UML créée. Ces propriétés sont ensuite référencées par les propriétés des stéréotypes SRM correspondant. Par exemple, dans la figure 5.6, les propriétés *period* et *priority* du stéréotype *RtFeature* correspond aux propriétés *periodElements* et *priorityElements* du stéréotype *SwSchedulableResource*. Elles sont modélisées par des propriétés portant le même nom dans la classe *RtFeature* créée, et référencées par les propriétés *periodElements* et *priorityElements* du stéréotype *SwSchedulableResource*.

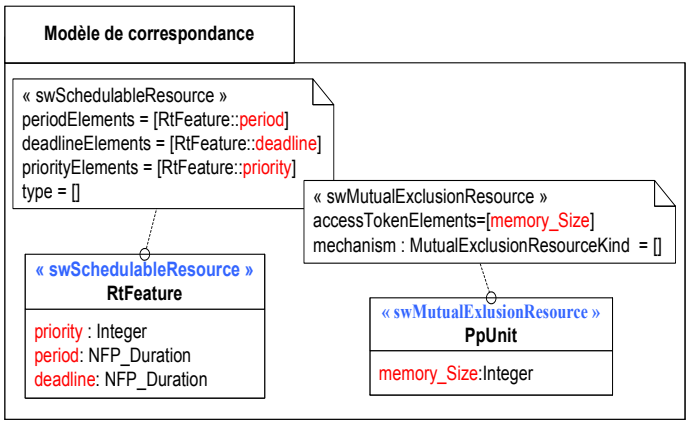

FIGURE 5.6 – Modélisation des propriétés référencées du stéréotype

Modélisation des concepts

Ce cas se présente lorsqu'un stéréotype d'un profil appliqué au niveau applicatif fournit des propriétés non référencées par des propriétés d'un stéréotype SRM. Si la valeur de ces propriétés spécifie un certain mode d'exécution qui correspond à un concept de SRM, alors une nouvelle classe UML est créée pour modéliser ce mode d'exécution. La nouvelle classe UML contient alors les propriétés non référencées avec une valeur par défaut qui correspond à ce mode d'exécution. Une relation d'héritage est définie entre cette nouvelle classe et la classe qui correspond au stéréotype déjà modélisé.

Par exemple, le stéréotype *RtFeature*, dans la figure 5.4, possède la propriété *occKind* de type *ArrivalPattern* qui est une *Enumeration*. Cette *Enumeration* spécifie si le mode d'exécution est périodique, apériodique etc. Cette propriété n'est pas référencée par une propriété du stéréotype *SwSchedulableResource*, mais sa valeur spécialise un mode d'exécution qui peut être référencé par le stéréotype *SwSchedulableResource* spécialisé par sa propriété type. La figure 5.7 illustre ce cas de modélisation. Dans cette figure deux nouvelles classes UML sont créées (*PeriodicRtFeature*, *AperiodicRtFeature*), elles contiennent la propriété *occKind* avec une valeur par défaut fixé à *periodic* dans la classe *PeriodicRtFeature* et à *aperiodic* dans la classe *AperiodicRtFeature*. Ces deux classes sont annotées par *SwSchedulableResource* en fixant la propriété *type* de ce stéréotype à *periodic* et *aperiodic* respectivement pour chaque classe. Enfin, une relation d'héritage est définie entre ces deux classes et la classe *RtFeature*

déjà modélisée.

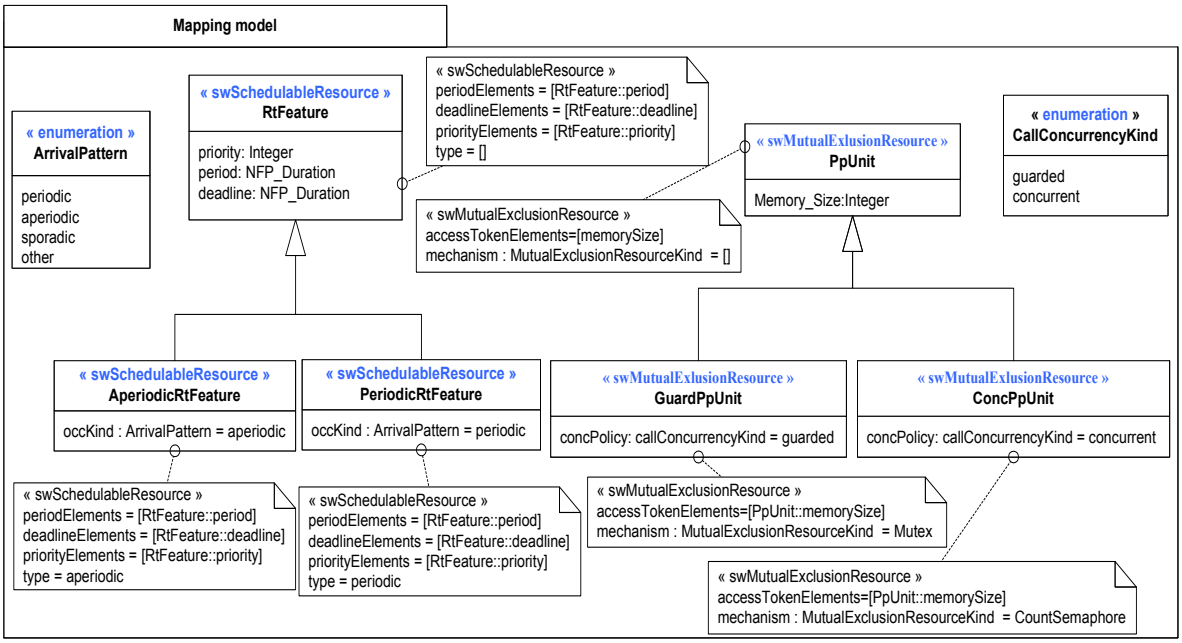

FIGURE 5.7 – Modèle de correspondance

Modélisation des services associés à un stéréotype

Un stéréotype d'un profil appliqué au modèle applicatif qui modélise un certain mécanisme de communication fournit des services qui sont utilisés par le concepteur pour réaliser la communication. Ces services sont des opérations dont leur rôle est de proposer des services d'envoi et de réception d'une structure de donnée ou encore des services de lecture et d'écriture des données. Ces services sont référencés par les propriétés des stéréotypes SRM correspondant.

Par exemple, dans [56] le stéréotype Connector du profil FCM[1] est utilisé pour la modélisation des mécanismes de communication. Avec ce stéréotype, le concepteur peut spécifier le type du mécanisme de communication qu'il souhaite à travers la propriété *connType* de ce stéréotype (*PushPull*, *PushPush*, etc.). L'implémentation souhaitée (FIFO, LIFO, etc.) du mécanisme communication peut être spécifiée par la propriété *connImpl* du stéréotype. Ces implémentations sont fournies sous la forme des composants d'interaction regroupées dans une bibliothèque. Pour réaliser les services d'envoi et de réception des données, le concepteur utilise les opérations des interfaces qui type les ports des composants d'interaction.

La figure 5.8 illustre la modélisation du stéréotype *Connector*. Dans cette figure, une classe *Connector* est créée. Elle est annotée par le stéréotype *MessageComResource*. Les propriétés *connType* et *connImpl* ne peuvent pas être référencées par des propriétés du stéréotype *MessageComResource*. Cependant, les valeurs de la propriété *connImpl* spécifient des modes d'exécution qui peuvent être référencés par le stéréotype *MessageComResource* spécialisé par

1. http ://www.flex-eware.org/

sa propriété *messageQueuePolicy*. Pour cela, en suivant l'étape de modélisation dans 1.1.1, deux classes UML sont créées qui héritent de la classe *Connector* (*FifoConnector*, *LifoConnector*). Elles sont annotées par *MessageComResource* avec la propriété *messageComPolicy* est renseignées par FIFO dans la première classe et par LIFO dans la deuxième. En plus, les propriétés *SendServices* et *ReceiveServices* de ce stéréotype référencent respectivement les opérations *send* et *receive*, de l'interface *PushPull*, offertes pour réaliser les services d'envoie et de réception des données.

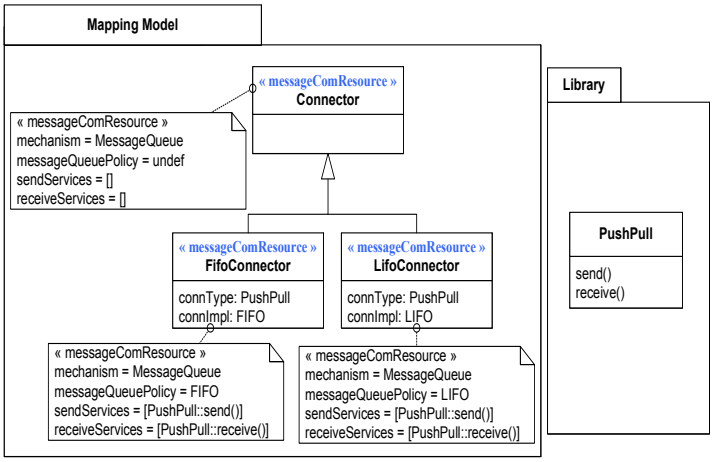

FIGURE 5.8 – Modélisation des services associés à un stéréotype

1.2 Méthode de modélisation des systèmes multi-tâches

La mise en œuvre d'un cadre expérimental pour le développement des systèmes multitâches nécessite la proposition d'une méthode permettant la conception de tels systèmes en termes de modélisation de leurs structures, leurs comportements et la prise en compte des contraintes qualitatives et quantitatives spécifiques à leur domaine d'application. La prise en compte de ces contraintes se concrétise par l'utilisation des concepts de haut niveau qui peuvent être modélisé dans le modèle de correspondance comme présenté dans la section précédente.

Dans le cadre de ce travail, nous avons adopté une méthode, simplifiée, issues des pratiques du domaines et notamment inspirées des méthodes proposées dans [54, 55, 56]. Elle se décompose en deux volets : 1) structure et 2) comportement.

1.2.1 Spécification de la structure de l'application

Pour spécifier la structure de l'application multitâche, la méthode proposée se base sur les diagrammes de classes et de structure composite. En plus, pour modéliser les contraintes qualitatives en relation avec la concurrence, la synchronisation et la communication, nous allons utiliser les stéréotypes *RtFeature*, *PpUnit* et *Connector* présentés dans la section précédente.

Diagramme des classes

Le principal diagramme UML pour décrire la structure des systèmes est le diagramme de classes. Il permet d'identifier la structure statique des systèmes multitâches en cours de conception. La structure est composée essentiellement des classes, des attributs, des opérations, des associations et d'héritages.

Pour réaliser un diagramme de classes d'une application, le concepteur doit premièrement, comme proposé dans la méthodologie ACCORD [55] dont les principaux concepts sont intégrés au profil HLAM MARTE, extraire des mots-clés qui définiront les classes de l'application. Ensuite, il doit définir les dépendances entre les classes (relations d'héritage et d'association) avec précisions de la cardinalité et de l'orientation des relations d'association. Après, le concepteur décrit les opérations offertes en précisant le nom, le type de la valeur de retour et les listes des paramètres avec leurs modes de passage respectifs. Enfin, il décrit les attributs de chaque classe.

A ce stade de modélisation, cette méthode permet au concepteur de spécifier les entités concurrentes de l'application. Une entité concurrente s'exécute en parallèle avec d'autres entités concurrentes. Dans cette approche, une entité concurrente représente une opération qui doit s'exécuter sur son propre thread d'exécution. On se limite ici au cas où une classe modélisée ne peut avoir qu'une seule entité concurrente. Les opérations qui représentent des entités concurrentes sont annotées par le stéréotype *RtFeature* de HLAM. Les propriétés quantitatives telles que les périodes et les priorités des threads peuvent être fixées par le concepteur à travers les propriétés des stéréotypes annotant les opérations spécifiées comme entités concurrentes (cf. figure 5.9).

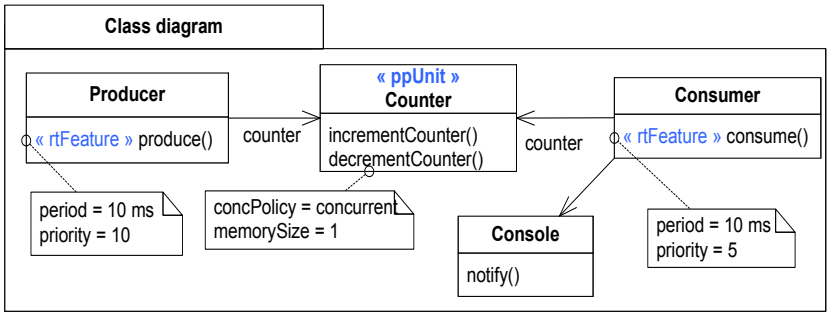

FIGURE 5.9 – Diagramme de classes de l'application Producer/Consumer

Outre les entités concurrentes, le concepteur doit s'intéresser aux objets qui risquent d'être utilisés en parallèle par plusieurs ressources concurrentes dans l'application. Dans ce cas, il faut adjoindre à ces objets un mécanisme de protection interne afin d'éviter que deux ou plusieurs threads ne s'entremêlent et créent des incohérences dans les valeurs des attributs de cet objet partagé. Afin de permettre un accès protégé à ces objets, il est permis au concepteur de décrire ce type des objets avec le stéréotype *PpUnit*. Dans ce cas, au moment de la réalisation de l'implémentation, un mécanisme interne de protection est associé à ce type d'objet.

La figure 5.9 montre le diagramme de classes de l'exemple Producteur/Consommateur. Les Classes *Producer* et *Consumer* contiennent deux entités concurrentes spécifiées par l'application du stéréotype *RtFeature* sur les opérations *produce* et *consume* respectivement.

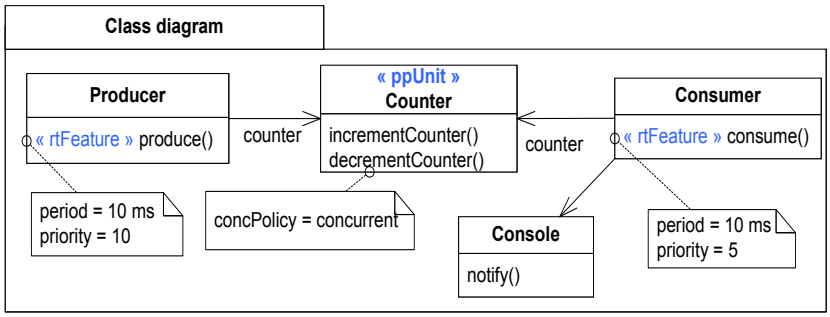

FIGURE 5.10 – Diagramme de structure composite de l'application Producer/Consumer

Par conséquence, les objets typés par Producer ou Consumer sont des objets concurrents. La classe *Counter* est partagée par les deux classes *Producer* et *Consumer*, elle est une ressource partagée. Par conséquence, elle est annotée par le stéréotype *PpUnit*.

Diagramme de structure composite

Après la description de la structure statique en utilisant le diagramme de classes, le concepteur doit spécifier les instances concrètes qui composent le système. Ainsi, tous les objets nécessaires à la phase d'exécution du système sont spécifiés à l'aide des propriétés d'une classe représentant le système dans un diagramme de structure composite [56].

A ce niveau de modélisation, le concepteur peut spécifier aussi les mécanismes de communication entre les objets en utilisant l'élément *Connector* dans UML. Cependant, les connecteurs dans la norme UML ne portent aucune information quant à la réalisation d'une interaction. Pour cela, cette approche propose l'utilisation d'une version simplifiée de la sémantique d'interaction présentée dans la section 1.1.1. En effet, une extension du connecteur UML est réalisée pour spécifier cette sémantique d'interaction. Ainsi, les connecteurs peuvent être annotés avec le stéréotype *Connector* afin de modéliser explicitement le type et l'implémentation de l'interaction. Le type de l'interaction est spécifié par la propriété *connType* du stéréotype *Connector*, il représente le mécanisme de communication entre objets. Le concepteur peut trouver l'implémentation des types de communication dans la bibliothèque offerte avec cette approche.

Le diagramme de structure composite de l'exemple Producteur/Consommateur est illustré dans la figure 5.10. L'ensemble du système en cours de conception est représenté par la classe *System*. Cette classe encapsule des propriétés représentant l'ensemble des objets à créer du système. Les propriétés de la classe *System* sont typées par les classes correspondantes du diagramme de classes. La multiplicité de ces propriétés peut être fixée au nombre des objets que le concepteur souhaite instancier de chaque classe (par exemple, la multiplicité de la propriété *producer* est fixée à 2). La communication entre les instances de la classe *Producer* et celle de la classe *Consumer* est modélisée par un connecteur UML. Ce connecteur est annoté avec le stéréotype *Connector* pour spécialiser le type de communication. Une communication asynchrone de type FIFO doit être établie entre les producteurs et le consommateur, par conséquence, la propriété *connType* est fixée à *PushPull*. L'implémentation du mécanisme est réalisée par une FIFO, par conséquence la valeur de

la propriété *connImpl* est renseignée par FIFO. Pour envoyer les mesures effectuées par les producteurs et les recevoir par le consommateur, le concepteur utilise les opérations fournies par l'interface *PushPull*. Cette interface offre une opération qui joue le rôle de *sendServices* (opération *send*) et une autre qui joue le rôle de *receiveServices* (opération *receive*).

1.2.2 Spécification du comportement de l'application

Pour être indépendant du langage de programmation de la plate-forme cible, UML offre plusieurs diagrammes pour modéliser le comportement des applications. Selon les objectifs de la modélisation, seuls certain d'entre eux sont utiles. Dans cette approche, on s'intéresse à la modélisation du comportement associé aux opérations des classes UML. Le diagramme d'activité, le diagramme de séquence ou encore le diagramme de machine à états peuvent répondre à notre besoin. Pour cette étude, on s'est limité à l'usage du diagramme d'activité pour spécifier les aspects comportementaux des applications multitâches.

Le diagramme d'activité décrit le comportement, associés aux opérations du système, en termes des activités. Les activités sont une spécification d'un comportement construit à partir de l'enchaînement des sous unités dont les éléments de base sont les actions. Le passage d'une action vers une autre est matérialisé par une transition. Les transitions sont déclenchées à la fin de chaque action et provoquent le début immédiat d'une autre. La sémantique d'exécution de ces diagrammes est basée sur une logique de jetons comme dans les réseaux de Petri.

Par exemple, la figure 5.11 illustre l'activité qui spécifie le comportement de l'opération *consume* dans la classe Consumer (figure 5.9). Cette activité est composée de trois actions. La première action appelle le service de réception des données à travers l'opération `send` de l'interface *PushPull* référencée par la propriété *sendServices*. La deuxième action appel l'opération de décrémentation du compteur dans la classe Counter. La dernière action est un appel d'une opération qui vérifie les données.

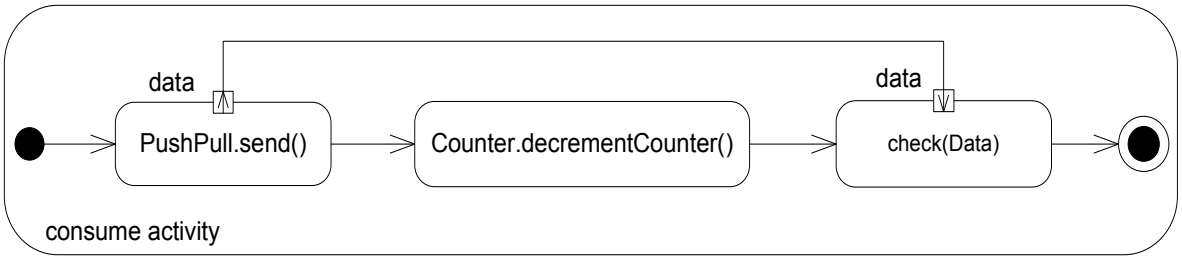

FIGURE 5.11 – Modélisation du comportement indépendamment de la plate-forme

Outre le diagramme d'activité, le concepteur peut utiliser les *OpaqueBehavior*(s) d'UML pour décrire le comportement des opérations. Dans ce cas, les comportements associés aux opérations ne sont pas portables entre les plates-formes d'exécution parce qu'ils sont codés par le langage de programmation supporté par la plate-forme d'exécution cible. Pour assurer la portabilité le concepteur est obligé de recoder l'implémentation des opérations à chaque fois que la plate-forme cible et, par conséquence, le langage de programmation, sont modifiés. Une autre alternative est l'utilisation du nouveau langage d'action standardisé par l'OMG, ALF [57]. A partir de ce langage d'action, et en utilisant des générateurs de code

ALF/Langage de programmation, la portabilité du code peut être assurée.

1.3 Description de l'infrastructure de transformation générique

L'objectif de cette infrastructure de transformation est d'automatiser le déploiement des applications multitâches indépendantes de la plate-forme sur une ou plusieurs plates-formes cibles. Pour cela, le processus de transformation mis en œuvre doit, premièrement, réifier les concepts abstraits utilisés pour modéliser les contraintes de concurrence, et, deuxièmement, il doit assurer la production d'un modèle spécifique à la plate-forme complet et, par conséquence, la génération d'un code exécutable.

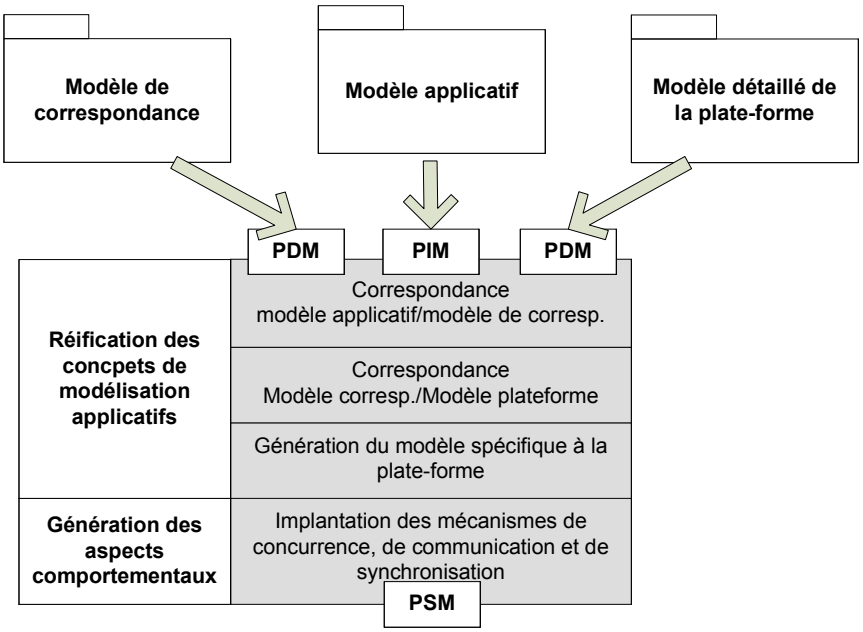

FIGURE 5.12 – Transformation générique

1.3.1 Réification des concepts de modélisation de l'application

Les stéréotypes utilisés pour annoter le modèle applicatif doivent être réifiés. Concrètement, la transformation générique doit trouver pour chaque stéréotype la ressource correspondante dans la plate-forme cible. Pour cela, la transformation générique prend trois modèles à son entrée (figure 5.12) : le modèle applicatif décrit à l'aide d'un profil dédié à la conception des systèmes concurrents, le modèle détaillé de la plate-forme cible présenté dans le chapitre 4 et le modèle de correspondance présenté dans la section 1.1 de ce chapitre. En se basant sur ces trois modèles, le processus de réification des concepts se réalise avec une seule transformation opérant en trois étapes :

1. établissement des correspondances entre les concepts du modèle applicatif et ceux du modèle de correspondance,
2. établissement des correspondances entre les ressources du modèle de correspondance et ceux du modèle détaillé de la plate-forme,

3. création du modèle spécifique à la plate-forme cible.

Correspondance modèle applicatif/modèle de correspondance

Dans cette phase, la transformation se base sur un algorithme d'équivalence qui consiste à trouver pour chaque élément stéréotypé dans le modèle applicatif son équivalent dans le modèle de correspondance. Pour cela, l'algorithme 1 prend en entrée l'ensemble des éléments du modèle applicatif. De cet ensemble, il choisit les éléments qui sont stéréotypés. À partir des éléments stéréotypés, l'algorithme recherche pour chaque stéréotype annotant l'élément, la classe correspondante dans le modèle de correspondance (*getCorrespondantClass*, ligne 7). Si celle-ci existe, il créé un couple rassemblant l'élément stéréotypé et la classe correspondante. L'identification de cette classe s'appuie sur l'algorithme de correspondance définie dans Algorithme 2.

L'algorithme de correspondance compare le stéréotype appliqué à un élément du modèle applicatif avec les classes dans le modèle de correspondance. La comparaison consiste, d'une part, à identifier la classe qui porte le même nom que le stéréotype ainsi que toutes les classes descendantes de cette classe identifiée. D'autre part, elle consiste à comparer les valeurs par défaut affectées aux propriétés de classes identifiées avec les valeurs affectées aux propriétés du stéréotype. Un stéréotype est ainsi équivalent à une classe dans le modèle de correspondance si les valeurs de leurs propriétés sont renseignées de la même façon.

Par exemple, la classe Counter dans la figure 5.9 est annotée par le stéréotype *PpUnit*. La politique d'accès à cette ressource est concurrente (la valeur de la valeur de la propriété *concPolicy* est fixée à *concurrent*). Alors d'après l'algorithme 2, la classe qui porte le même nom que le stéréotype ainsi que les classes descendant de ce stéréotype sont sélectionnées depuis le modèle de correspondance. Cela correspond donc à l'ensemble des classes *PpUnit*, *ConcPpUnit* et *GuardPpUnit* de la figure 5.7. Ensuite, les valeurs par défaut affectées aux propriétés de ces classes sont comparées avec les valeurs des mêmes propriétés du stéréotype *PpUnit*. La comparaison montre que la valeur de la propriété *concPolicy* de la classe *ConcPpUnit* est égale à la valeur de la propriété *concPolicy* du stéréotype *PpUnit*, comme illustrer dans la figure 5.13. Par conséquence, le résultat est un couple regroupant la classe *Counter* et la classe *ConcPpUnit*. De la même manière, les opérations *produce* et *consume* correspondent à la classe *PeriodicRtFeature* car la valeur de la propriété *type* du stéréotype *RtFeature* les annotant est fixée à *periodic*.

Correspondance modèle de correspondance/modèle de plate-forme

La première phase de la transformation a fourni un ensemble de couples combinant chacun un élément stéréotypé du modèle applicatif et la classe correspondante dans le modèle de correspondance. La deuxième phase consiste maintenant à établir la correspondance entre les éléments stéréotypés et les ressources équivalentes dans le modèle détaillé de la plate-forme cible. Cela est possible grâce à la description commune des ressources dans le modèle de correspondance et celles dans le modèle détaillé de la plate-forme. Par conséquence, un algorithme d'équivalence peut être écrit entre ressources applicatives et ressources cibles. Cet algorithme est spécifié en Algorithme 3. Il s'appuie sur l'opérateur

Algorithme 1: Algorithme de correspondance entre modèle applicatif et modèle de correspondance

Entrées :
A : L'ensemble des éléments applicatifs
C : L'ensemble des classes du modèle de correspondance
Résultat :
L'ensemble des couples (Elément applicatif stéréotypé, Classe correspondante dans C)
$T_{\text{ac}} = \{t_{\text{ac}} | \exists\, a \in A, c \in C : t_{\text{ac}} = (a, c)\}$
Données :
S(e) : Le stéréotype décrivant l'élément e

```
1  Initialisations :
2  T_ac ← ⊘
3  début
4      Trouver les correspondances entre A et C
5      pour chaque a ∈ A faire
6          si S(a) != ⊘ alors
7              c ← getCorrespondantClass(S(a), C);
8              t_ac ← t_ac ∪ (a, c);
9          fin
10         retourner T_ac;
11     fin
12 fin
```

Algorithme 2: Algorithme de sélection de la classe du modèle de correspondance qui correspond au stéréotype

Entrées :
s : Le stéréotype appliqué à l'élément applicatif
C : L'ensemble des classes du modèle de correspondance
Résultat :
Retourne la classe dans l'ensemble C qui correspond au stéréotype s : class
Données :
C_{select} : Ensemble des classes sélectionnées de C
D_{shild} : Ensemble des classes qui spécialise une classe
P(s) : L'ensemble des propriétés du stéréotypes s

```
1  Initialisations :
2  C_select ← ∅
3  D_shild ← ∅
4  début
5      Trouver la classe qui correspond à s
6      pour chaque c ∈ C faire
7          si c.name = s.name alors
8              C_select ← C_select ∪ c;
9              D_shild ← D_shild ∪ getShilds(c);
10             C_select ← C_select ∪ D_shild;
11         fin
12     fin
13     pour chaque c ∈ C_select faire
14         L'ensemble des propriétés de la classe qui ont une valeur par défaut
           P_c(c) = {p_c|defaultValue(p_c)notnull};
15         le nombre des propriétés n = P_c(c).size;
16         définition d'un entier i = 0;
17         pour chaque p_c ∈ P(c) faire
18             pour chaque p_s ∈ P(s) faire
19                 si defaultValue(p_c) = defaultValue(p_s) alors
20                     i++;
21                 fin
22             fin
23         fin
24         si n = i alors
25             class = c;
26         fin
27     fin
28     retourner class;
29 fin
```

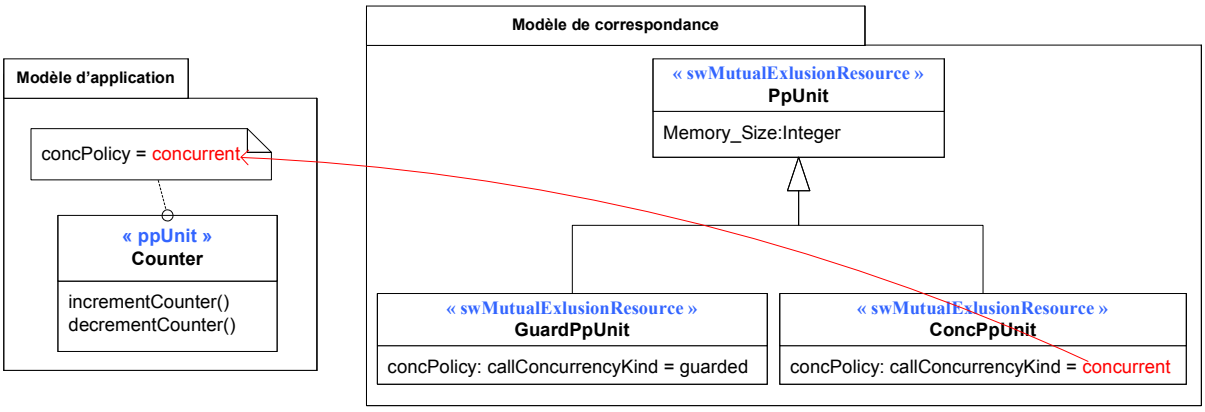

FIGURE 5.13 – Correspondance entre la classe *Counter* et la classe *ConcPpUnit*

de conformité $\approx$ défini par Frédéric Thomas dans sa thèse [75]. Cet opérateur permet d'identifier si deux ressources sont équivalentes. Pour cela, il compare d'une part, les stéréotypes appliqués sur les ressources passées en paramètre et, d'autre part, les valeurs des métapropriétés primitives, c'est-à-dire des types primitifs (booléen, entier et chaîne de caractère par exemple) et des énumérations. Ainsi, deux ressources sont équivalentes si et seulement si les ensembles des stéréotypes appliqués sur la source et sur la cible sont égaux et si ces stéréotypes sont renseignés de la même façon. En plus de l'opérateur de conformité, l'algorithme proposé permet de spécifier des règles de conformité spécifiques dans le cas ou la relation de conformité ne répond pas seule aux besoins de l'établissement de la correspondance entre les ressources.

L'algorithme 3 se déroule comme suit : pour chaque classe source dans le modèle de correspondance, il recherche la classe correspondante dans le modèle détaillée de plate-forme cible. La classe correspondante doit être conforme à la classe source (résultat de l'opérateur $\approx$ est vrai). En plus de la relation de conformité, la classe de correspondance doit satisfaire les règles de conformité spécifiques, dans le cas où c'est nécessaire. Le résultat de cet algorithme est donc un ensemble de triplets joignant chacun l'élément applicatif stéréotypé, la classe correspondante du modèle de correspondance et la classe du modèle détaillé de la plate-forme qui vient d'être spécifier.

Par exemple, la classe *ConcPpUnit*, dans le modèle de correspondance de la figure 5.7, est annotée par le stéréotype *SwMutualExlusionResource* de SRM. La propriété *mechanism* de ce stéréotype est fixée à *countSemaphore*. La figure 5.14 montre que la ressource dans le modèle détaillé de la plate-forme Java , annotée par le même stéréotype et renseignée de la même façon, correspond à la classe *Semaphore*.

Génération du modèle spécifique à la plate-forme

Les deux phases précédentes de la transformation ont permis l'identification des ressources de la plate-forme correspondants aux éléments applicatifs stéréotypés et dont leur réalisation nécessite des implémentations spécifiques à la plate-forme. Grâce à la modélisation détaillée des plates-formes d'exécutions proposée dans le chapitre 4, toutes ces implémentations spécifiques sont déjà encapsulées dans les modèles des ressources et

Algorithme 3: Algorithme de correspondance entre les éléments applicatifs et les ressources de la plate-forme

Entrées :
L'ensemble des couples (Elément applicatif stéréotypé, Classe correspondante dans C) $T_{\text{ac}} = \{t_{\text{ac}} | \exists\ a \in A, c \in C : t_{\text{ac}} = (a, c)\}$
R : L'ensemble des classes du modèle détaillé de la plate-forme
Résultat :
L'ensemble des triplets (Elément applicatif stéréotypé, Classe correspondante dans C, Ressource de la plate-forme) $T_{\text{acr}} = \{t_{\text{acr}} | \exists\ a \in A, c \in C, r \in R : t_{\text{acr}} = (a, c, r)\}$

Initialisations :
$T_{acr} \leftarrow \oslash$
début
  Trouver les correspondances entre A et C
  pour chaque $t_{ac} \in T_{ac}$ **faire**
    pour chaque $r \in R$ **faire**
      si $t_{ac}.c \approx r$ **alors**
        si $specificCorrespondance(t_{ac}.c, r)$ **alors**
          $t_{\text{acr}} \leftarrow t_{\text{acr}} \cup (t_{\text{ac}}.a, t_{\text{ac}}.c, r)$;
        fin
      fin
    fin
  fin
  retourner T_{acr};
fin

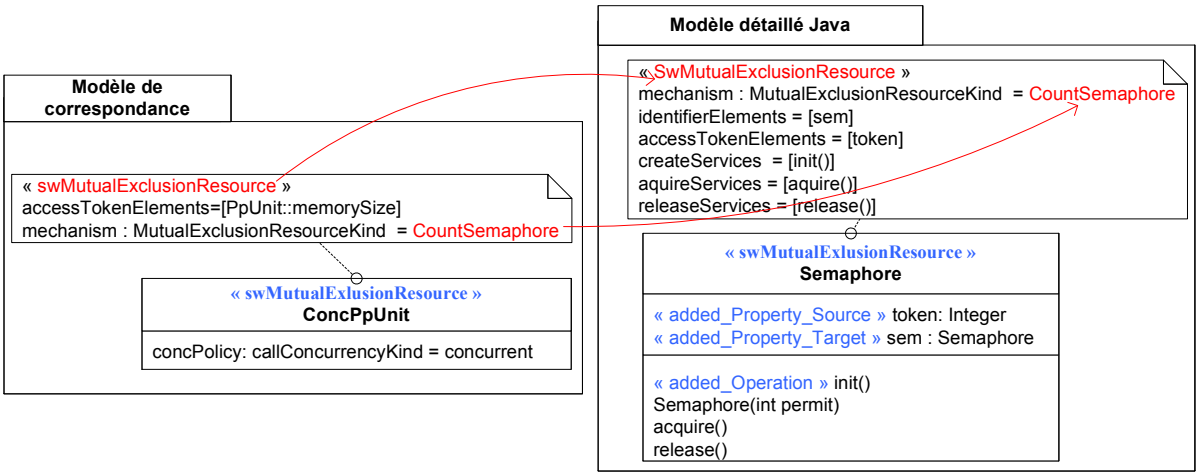

FIGURE 5.14 – Correspondance entre la classe *ConcPpunit* et la classe *Semaphore*

elles sont identifiées par les stéréotypes du profil DPD.

Par conséquence, pour réaliser des modèles applicatifs spécifiques à une plate-forme cible, la transformation crée, pour chaque élément applicatif stéréotypé, une classe UML équivalente dans le modèle de sortie portant le même nom. Ensuite, elle ajoute à chaque

classe créée toutes les éléments de la ressource de la plate-forme cible annotés par un stéréotype du profil DPD. La ressource de la plate-forme est celle qui est identifiée comme équivalente à l'élément applicatif dans l'étape de transformation précédente. La figure 5.15 présente le modèle spécifique à la plate-forme Java qui peut être généré à ce stade de la transformation pour l'application Producteur/Consommateur. Dans cette figure, les classes *Producer* et *Consumer* générées encapsulent les implémentations spécifiques de la classe *PeriodicThread* du modèle détaillé de la plate-forme Java. La classe *Counter* générée encapsule les implémentations spécifiques de la classe *Semaphore*. Et enfin, la classe *ProducerConsumerConnector* encapsule les implémentations spécifiques de la classe *LinkedList*.

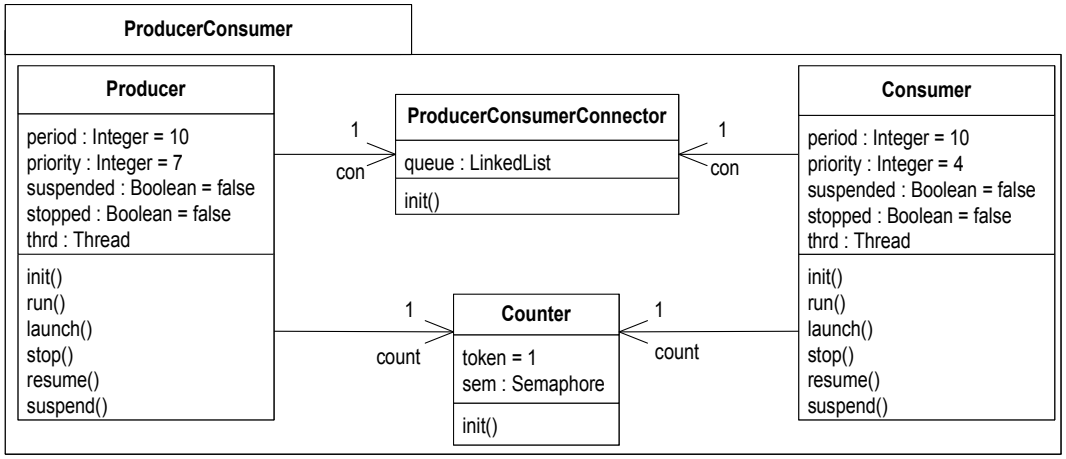

FIGURE 5.15 – Modèle applicatif spécifique à la plate-forme Java

Parmi les éléments ajoutés aux classes générées, les propriétés qui sont annotées par le stéréotype *Added_Property_Source* dans les ressources du modèle détaillé de la plate-forme. Ces propriétés prennent leurs valeurs depuis le modèle applicatif, comme précisé lors de la présentation du profil DPD 3.1.8. Pour cela, une valeur est affectée à chacune de ces propriétés dans le modèle de sortie. Elle correspond à la valeur affectée à la propriété équivalente du stéréotype utilisé au niveau applicatif. Cette propriété équivalente est référencée, dans le modèle de correspondance, par la même propriété du stéréotype SRM qui référence la propriété générée dans le modèle de sortie. Ainsi, dans la figure 5.16, les propriétés *token* et *period* dans les classes *Counter* et *Producer* générées proviennent respectivement des classes *PeriodicThread* et *Semaphore* (relation 1). Ces deux propriétés sont référencées par la propriété *periodElements* du stéréotype *SwSechdulableResource* annotant la ressource *PeriodicThread* et la propriété *AccessTokenElements* du stéréotype *SwMutualExlusionResource* annotant la ressource *Semaphore*. Dans le modèle de correspondance, la propriété *period* de la classe *PeriodicRtFeature* et la propriété *memorySize* de la classe *ConcPpUnit* sont référencées de la même façon que les propriétés *period* et *token* des ressources *PeriodicThread* et *Semaphore* (relation 2). Dans le modèle applicatif, ces deux propriétés correspondent à la propriété *period* du stéréotype *RtFeature* annotant l'opération *produce* et à la propriété *memorySize* du stéréotype *PpUnit* annotant la classe *Counter* (relation 3). Ces deux propriétés possèdent des valeurs affectées par le concepteur. Alors, la transformation affecte ces valeurs aux valeurs des propriétés *period* et *token* dans le modèle de sortie. Par conséquence, dans la figure 5.15,

la valeur par défaut de la propriété *period* est fixée à 10 et celle de *token* à 1.

Pour les éléments applicatifs qui ne sont pas stéréotypés, la transformation les reproduit dans le modèle de sortie sans aucune modification. C'est la transformation identique.

1.3.2 Génération des aspects comportementaux

Le processus de transformation présenté dans la section précédente a permis la génération de la structure du modèle applicatif enrichie des implémentations spécifiques nécessaires à la mise en œuvre des concepts de modélisation des applications concurrentes. Cependant, en plus de ces implémentations spécifiques générées, la réalisation des nombreux concepts applicatifs repose sur des appels vers les services des ressources de la plate-forme ou la résolution d'une dépendance entre l'application et la plate-forme, comme par exemple, la concurrence et les mécanismes de communication et de synchronisation.

Notre approche offre au concepteur la possibilité de spécifier de tels concepts. Pour réaliser l'implantation de ces concepts indépendamment de la plate-forme, il faut gérer l'appel aux services des plates-formes d'une manière générique. Pour cela, il faut définir des patrons de conception qui permettent la mise en œuvre de ces mécanismes, tout en conservant la généricité de la transformation. Une fois définis, ces patrons seront alors intégrés dans la transformation de modèle.

Implantation de la concurrence

Lors de la modélisation détaillée de la plate-forme, la séquence d'action à exécuter par la ressource concurrente lorsqu'elle activée, est modélisée par une opération référencée par la propriété *entryPoints* du stéréotype *SwSchedulableResource*. Dans le modèle de sortie généré, cette opération est aussi générée. Le comportement associé à cette opération de point d'entrée doit être spécifié par la séquence d'action de l'entité concurrente définie par le concepteur dans le modèle applicatif. Selon notre approche, cette séquence d'action est l'opération annotée par le stéréotype *RtFeature*. Par conséquence, pour spécifier le comportement de l'opération de point d'entrée dans le modèle de sortie, on doit définir le patron de conception comportemental *entryPoint*. C'est une activité composée d'une seule action. La figure 5.17 illustre une telle activité. Elle consiste à appeler l'opération générée dans le modèle de sortie dont l'exécution est spécifiée comme concurrente dans le modèle applicatif.

Ainsi, pour chaque classe générée dans le modèle de sortie et qui correspond à une ressource concurrente, la transformation de modèle identifie tout d'abord l'opération jouant le rôle du point d'entrée, ensuite, elle génère une activité qui correspond au patron de conception *entryPoint* avec l'opération appelée bien identifiée. Enfin, la transformation spécifie la nouvelle activité comme étant la mise en œuvre de l'opération modélisée comme point d'entrée.

Par exemple, dans le modèle Producteur/Consommateur spécifique à la plate-forme Java de la figure 5.18, les deux opérations **launch**, générées dans les classes *Producer* et *Consumer*, sont référencées par la propriété *entryPoints* dans le modèle de la plate-forme. Ainsi, la transformation générique spécifie leurs comportements respectivement par une activité qui appelle l'opération *produce* générée dans la classe *Producer* et une

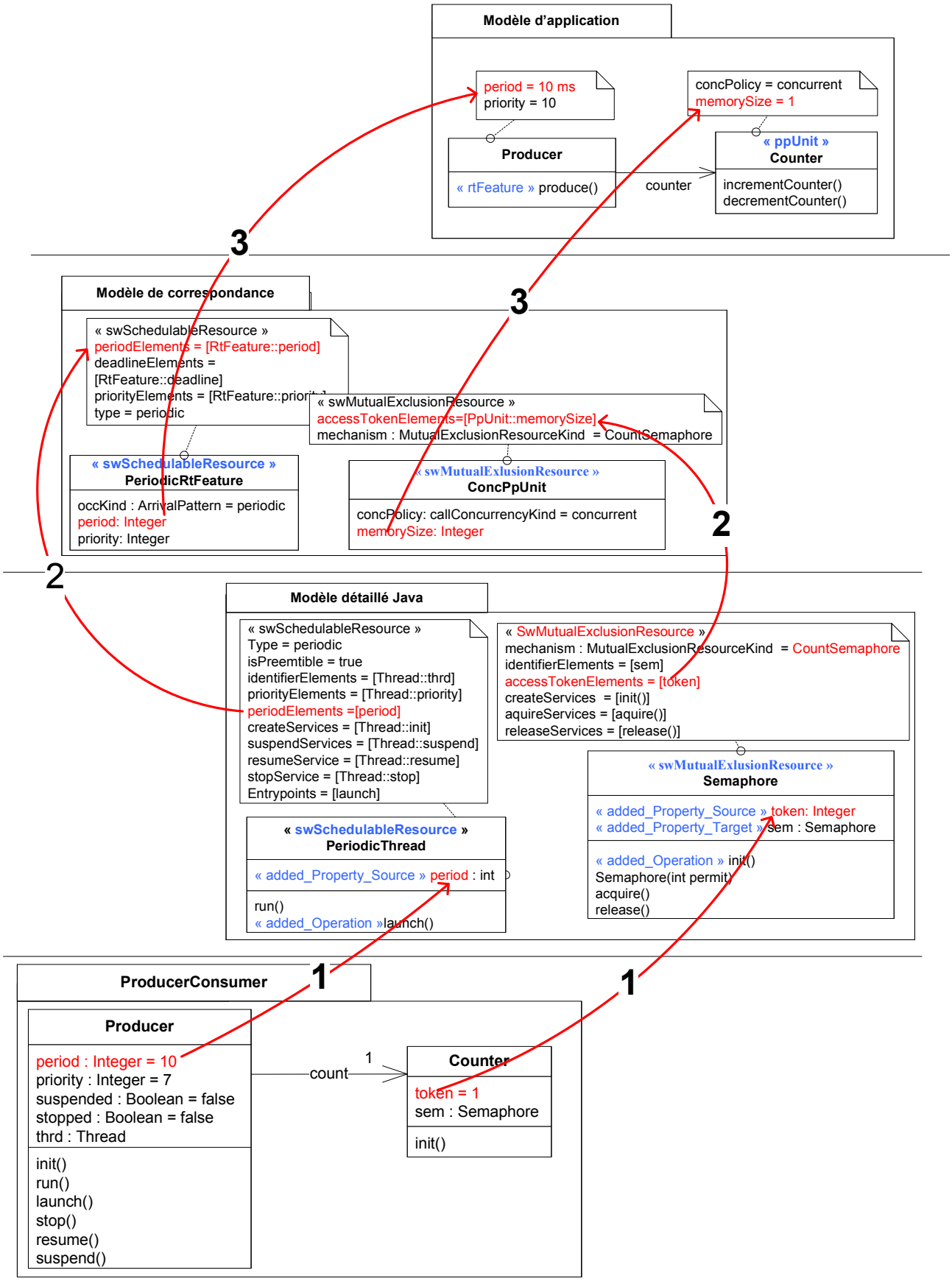

FIGURE 5.16 – Illustration de l'affectation des valeurs des propriétés

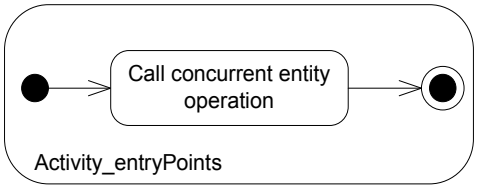

FIGURE 5.17 – Comportement associé à l'opération de point d'entrée

autre qui appelle l'opération *consume* générée dans la classe *Consumer*. Le comportement associé aux deux opérations *produce* et *consume* est spécifié par le concepteur de l'application.

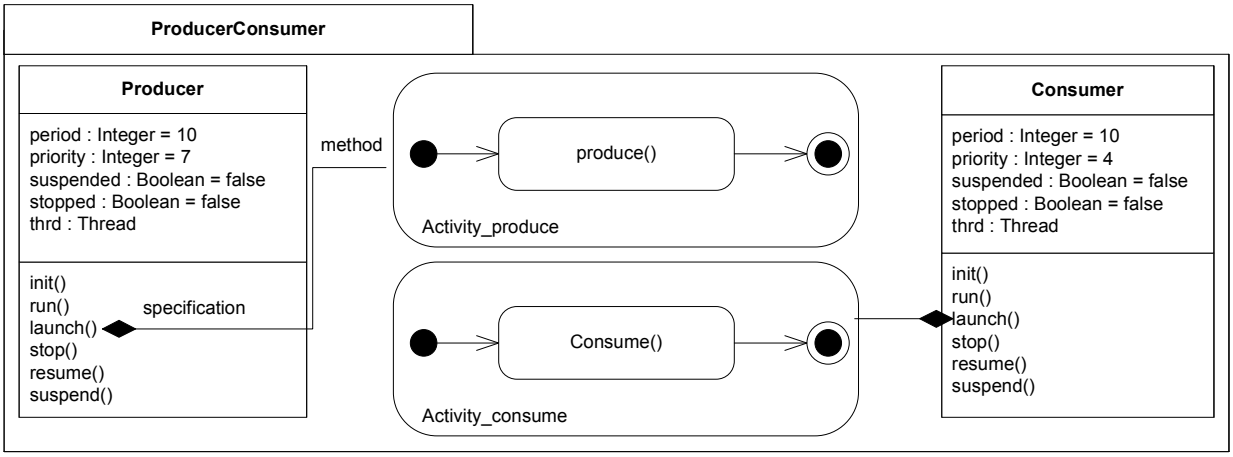

FIGURE 5.18 – Comportement associé à l'opération *launch*

Implantation des mécanismes de communication

Comme présenté lors de la présentation du profil SRM, les mécanismes de communication sont classés en deux grandes familles :

1. Échange par message : se concrétise par des appels des services d'envoi et de réception des données de la ressource correspondante de la plate-forme.
2. Échange par variable partagée : se concrétise par des appels des services de lecture ou d'écriture des donnés.

Dans un modèle applicatif, pour être indépendant de la plate-forme, ces services ne peuvent pas être utilisés explicitement. En revanche, le concepteur peut utiliser des opérations qui jouent le rôle de ces services au niveau modèle. Par exemple, dans cette approche le concepteur utilise le stéréotype *Connector* pour spécifier le mécanisme de communication. Ensuite, il réalise l'envoi et la réception, la lecture et l'écriture des données en utilisant les opérations des interfaces qui sont définies dans la librairie fournie avec la méthode de conception.

Lors de l'implantation de ce mécanisme, la transformation de modèle doit pouvoir définir la mise en œuvre des ces opérations. Pour cela, la transformation :

1. identifie le rôle joué par l'opération dans le modèle de correspondance. Le rôle

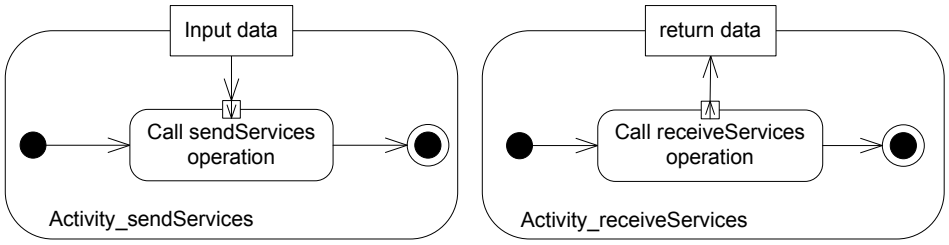

FIGURE 5.19 – Comportement associé à l'opération d'envoie et de réception de données

est déterminé par la propriété du stéréotype SRM qui référence cette opération (*readServices* ou *writeServices* du stéréotype *SharedDataComResource* et *sendServices* ou *receiveServices* du *MessageComResource*).

2. cherche l'opération équivalente dans le modèle détaillé de la plate-forme. L'opération équivalente est l'opération de la ressource de la plate-forme qui est référencée par la même propriété du même stéréotype SRM annotant cette ressource. La ressource est celle du modèle détaillé de la plate-forme qui correspond à l'élément applicatif stéréotypés. Elle était déjà identifiée pendant la deuxième phase de la transformation.
3. génère dans la classe générée, dans le modèle de sortie, qui correspond à l'élément applicatif, les opérations de communication utilisées au niveau modèle et spécifie le comportement associé à chaque opération par une activité UML composée d'une action qui appelle l'opération équivalente dans la ressource de la plate-forme.

La figure 5.19 illustre les patrons de conception de communication, utiliser pour envoyer et recevoir des données.

Par exemple, pour mettre en œuvre le mécanisme de communication FIFO entre le producteur et le consommateur (figure 5.10), un appel des services *sendServices* et *receiveServices* de la ressource de la plate-forme cible doit être effectué. Par conséquence, les comportements associés aux opérations receive et send de l'interface PushPull, utilisées pour réaliser ce mécanisme au niveau modèle, seront spécifiés respectivement par les activités illustrées dans la figure 5.20, dans la classe *ProducerConsumerConnector* du modèle de sortie. Dans la première activité, un appel à l'opération *add* de la ressource *LinkedList* de la plate-forme Java est effectué (opération dans le modèle détaillé de plate-forme qui est référencée par la propriété *receiveServices* du stéréotype *MessageComResource*). Dans la deuxième activité, un appel à l'opération *poll* de la même ressource est effectué (opération dans le modèle de plate-forme détaillée qui est référencée par la propriété *sendServices* du stéréotype *MessageComResource*).

Implantation des mécanismes de synchronisation

SRM identifie deux mécanismes de synchronisation : l'exclusion mutuelle et la notification. L'implantation de ces mécanisme nécessite des appels à des services spécifiques des ressources de la plate-forme. Dans cette approche, la transformation proposée gère ces implantations d'une manière générique indépendamment de la plate-forme cible.

Au niveau modèle, le mécanisme de synchronisation à exclusion mutuelle est modélisé

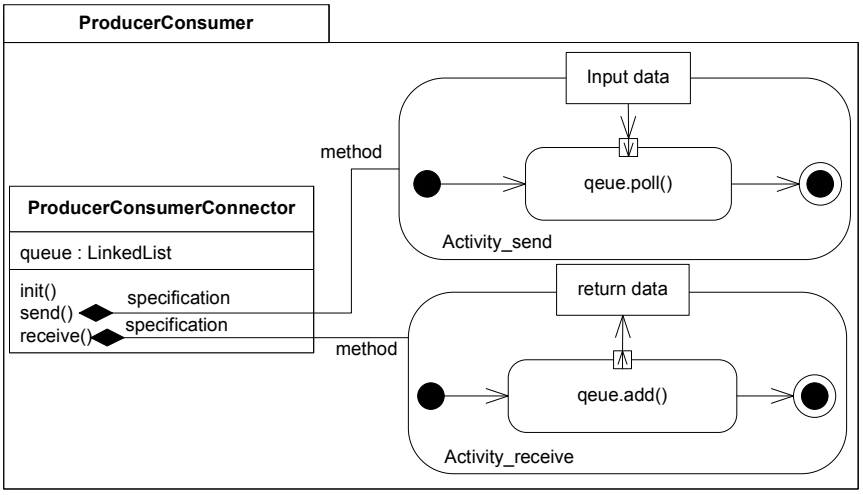

FIGURE 5.20 – Comportement associé aux opérations *send* et *receive*

par un stéréotype qui annote une classe. Ainsi, il faut protéger tous les appels des opérations de la classe modélisée comme étant à exclusion mutuelle. La concrétisation de ce mécanisme implique systématiquement des appels des services de prise et de libération de la ressource d'exclusion mutuelle de la plate-forme.

La transformation réalise ce mécanisme de synchronisation comme suit :

1. Pour chaque opération de la classe protégée, elle génère la même opération avec le même comportement dans la classe correspondante dans le modèle de sortie, mais en ajoutant la suffixe " _Protected " au nom de l'opération.
2. La transformation crée une nouvelle opération identique à l'opération de la classe protégée du modèle applicatif.
3. La transformation spécifie le comportement associé à l'opération créée par une activité. Cette activité est composée de trois actions. La première action est un appel de l'opération de la ressource de la plate-forme qui est référencée par la propriété *acquireServices* du Stéréotype *SwMutualExlusionResource*. La deuxième action est un appel de l'opération marquée par le suffise " _Protected " et qui correspond à la reproduction de l'opération de la classe protégée du modèle applicatif. Enfin, la troisième action est un appel de l'opération de la ressource de la plate-forme qui est référencée par la propriété *releaseServices* du stéréotype *SwMutualExlusionResource*.

La figure 5.21 illustre l'activité de protection générée. Ce mécanisme de protection reprend les mécanismes connus dans le monde objet, comme par exemple les patrons de conception proposés par D. Schmidt dans ACE [71] ou encore les mécanismes de protection dans ACCORD [54].

Par exemple, la classe *Counter* dans l'application Producer/Consumer est annotée par le stéréotype *PpUnit*. Les appels de ses opérations doit être réalisés en exclusion mutuelle. Ainsi, dans la figure 5.22, pour une implantation sous C++/POSIX, le comportement de l'opération *incrementCounter* dans le modèle de sortie sera une activité qui appelle tout d'abord l'opération *sem_wait* (opération référencée par *acquireServices* du stéréotype *SwMutualExlusionResource* annotant la ressource sem_t) ensuite l'opération

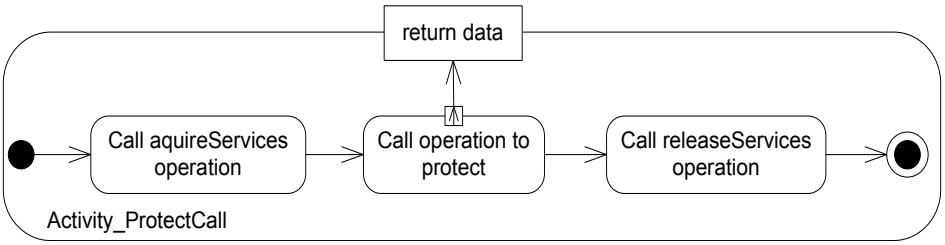

FIGURE 5.21 – Comportement associé aux opérations d'une ressource protégée

incrementCounter_Protected générée et, enfin, l'opération *sem_post* (opération référencée par *releaseServices* du stéréotype *SwMutualExlusionResource* annotant la ressource *sem_t*).

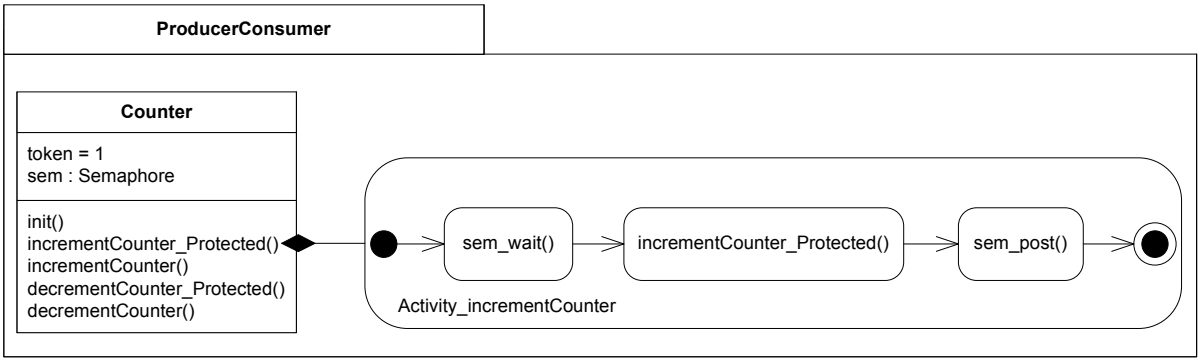

FIGURE 5.22 – Comportement associé à l'opération *incrementCounter*

Démarrage de l'application

A ce stade de la transformation, le modèle spécifique à la plate-forme qui est généré contient tous les aspects fonctionnels et non-fonctionnels nécessaires pour la génération d'un code exécutable. Cependant les aspects concernant le démarrage du système ne sont pas encore générés. Ces aspects traitent les actions de création et d'initialisation les objets du système. Au niveau modèle, les informations concernant les objets du système sont spécifiées dans la classe système du diagramme de structure composite. En effet, les propriétés typées par les classes de l'application représentent les objets à créer.

Pour initialiser et démarrer le système, la transformation premièrement génère dans le modèle de sortie une classe qui correspond à la classe système dans le diagramme composite. Elle ajoute à cette classe générée une opération. Le comportement associé à cette opération est une activité composée des actions qui sont définies comme suit :

– pour chaque propriété typée dans la classe de départ, une instance de la classe générée dans le modèle de sortie qui correspond à son type est créée. Cela est réalisé par l'action *CreateObjectAction*. La multiplicité de la propriété au niveau modèle est concrétisée par le nombre des instances qui sont créées de la classe qui correspond à son type. Par exemple, dans la figure 4.6, la multiplicité de la propriété *producer* est fixée à deux, par conséquence, deux actions de création d'objet de type *Producer* doivent être générées dans l'activité.

– Une fois que les objets de l'application sont créés, la transformation génère des actions d'appel d'opération (CallOperationAction) de création de service dans le cas où le type d'objet créé encapsule des éléments qui proviennent d'une ressource de la plate-forme. Par exemple, l'objet *counter* est une instance typée par la classe *Counter*. Cette classe encapsule des implémentations spécifiques à un sémaphore de la plate-forme. Ainsi, pour créer et initialiser cette sémaphore une action d'appel de l'opération *init* doit être créé. Cette opération encapsule un comportement spécifique permettant de créer et d'initialiser le sémaphore. Elle est référencée par la propriété *createServices* du stéréotype *SwMutualExlusionResource* dans le modèle détaillé de la plate-forme.
– Dans le cas où l'objet partage une ressource avec des autres objets, une même instance de cette ressource doit être transmise à eux. Pour cela, dans chaque classe générée qui correspond au type de l'objet qui partage une ressource, la transformation de modèle crée une opération d'initialisation qui initialise la référence de l'association vers la classe partagée. Ensuite, elle crée un appel d'opération de création de service. Ce comportement est spécifié dans une activité. Par exemple, dans la figure 5.10, l'objet consumer partage l'objet counter avec deux autres objets. En plus, cet objet correspond à une ressource concurrente qui doit être créée. Ainsi, la transformation ajoute dans la classe Consumer générée, l'opération createObject() associée avec un comportement qui affecte l'objet partagé counter à la propriété count de type Counter dans la classe Consumer. Ensuite, elle effectue un appel de l'opération init dans la classe Consumer. Cette activité est illustrée dans la figure 5.23.

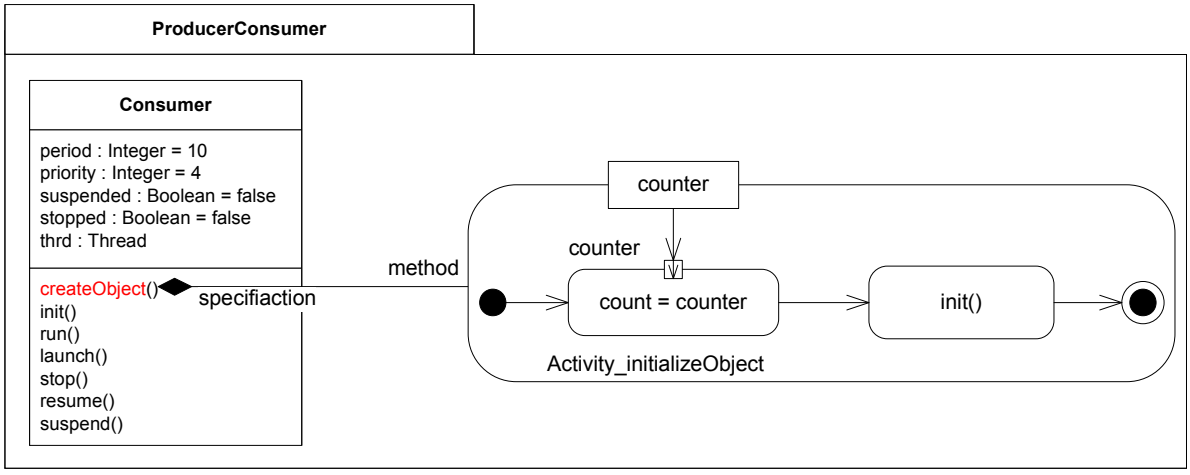

FIGURE 5.23 – Comportement associé à l'opération d'initialisation des propriétés typées par des ressources partagées

– Après l'ajout des opérations d'initialisation dans les classes concernées, la transformation crée alors des actions d'appel des ces opérations dans l'activité globale générée pour créer et initialiser le système.

La figure 5.24 illustre l'activité complète de démarrage de l'application générée pour l'application Producteur/Consommateur.

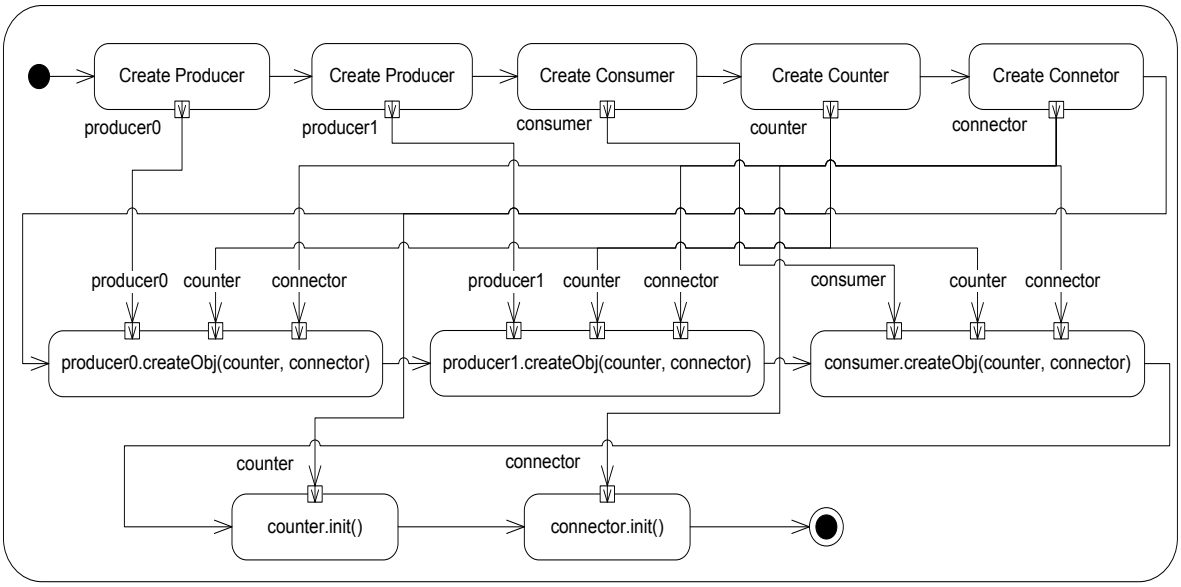

FIGURE 5.24 – Comportement associé à l'opération de démarrage du système

1.4 Génération du code exécutable

Le modèle spécifique à la plate-forme généré à la fin du processus de transformation générique est un modèle UML qui encapsule les aspects fonctionnels et les aspects spécifiques à la plate-forme cible :

- Les aspects fonctionnels sont ceux qui sont spécifiés par le concepteur au niveau du modèle de l'application indépendant de la plate-forme.
- Les aspects spécifiques à la plate-forme correspondent aux opérations et propriétés qui sont annotées par les stéréotypes du profils DPD dans le modèle détaillé de plate-forme.
- Les comportements spécifiés par des activités nécessaires à l'implantation des mécanismes de concurrence, de synchronisation et de communication.

Les figures 5.25 et 5.26 représentent les modèles de l'application Producer/Consumer spécifiques aux plates-formes Java et C++/POSIX. A partir de ces modèles, un code exécutable peut être généré en utilisant des générateurs de code standard UML/Java et UML/C++. Ces générateurs doivent supporter la génération de codes depuis les éléments des packages UML qui supporte la modélisation structurelle et comportementale.

Pour notre travail, la génération de code est réalisée à travers les générateurs de codes Java et C++ qui sont intégrés dans Papyrus. Papyrus est développé au sein du laboratoire CEA LIST, c'est un outil d'édition graphique de modèles UML2.3, rigoureusement conforme à la norme UML.

2 Discussion

L'infrastructure de transformation proposée dans cette étude a été décrite dans le langage de transformation ATL [21]. Les premiers résultats de l'utilisation de cette infrastructure sont

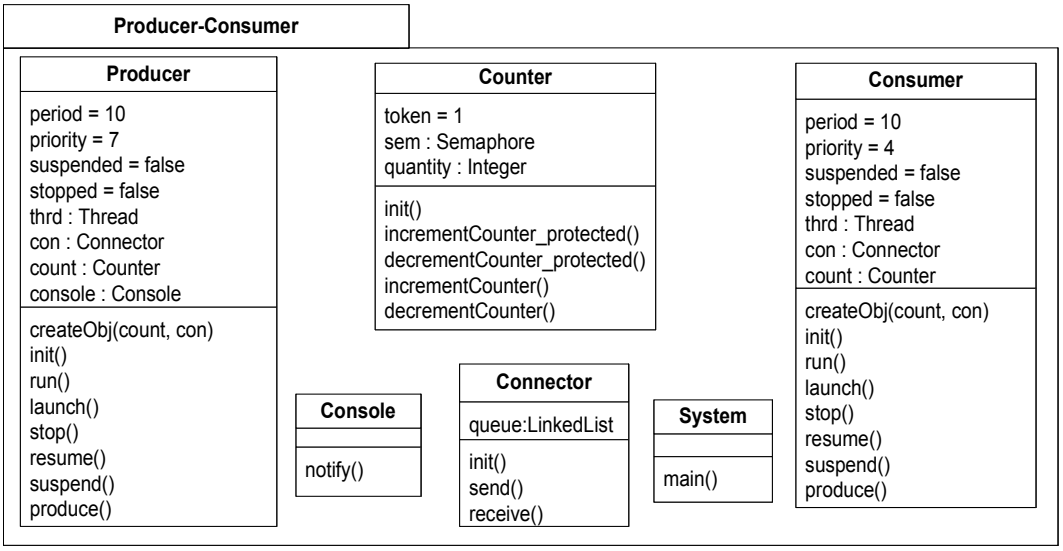

FIGURE 5.25 – Modèle généré spécifique à la plate-forme Java

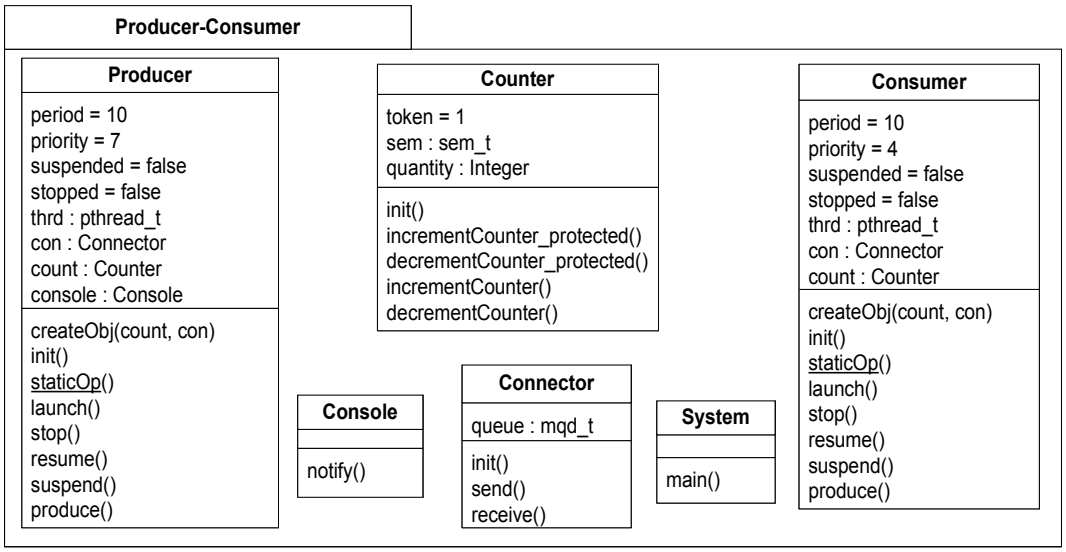

FIGURE 5.26 – Modèle généré spécifique à la plate-forme C++/POSIX

encourageants puisque, d'une part, ils montrent que l'implantation d'une infrastructure de transformation basée sur des plates-formes existe et que, d'autre part, une infrastructure générique est envisageable pour aider à la construction des transformations ciblant plusieurs plates-formes tout en préservant les critères de qualité. Ces critères ont été définis dans le chapitre 3 pour juger la qualité de la transformation de modèle.

En effet, les transformations construites dans cette étude satisfont les critères suivants :

- La réutilisabilité : les transformations sont indépendantes des concepts applicatifs et de la plate-forme cible
- La séparation de préoccupation : Les transformations sont réalisées par un expert IDM sont se préoccuper des informations spécifiques à la plate-forme.
- La préservation des propriétés et du comportement de l'application : Les propriétés

de l'application multitâche ainsi que ses comportements associées sont réifiés lors de déploiement par rapport à la plat-forme cible.
– la complétude : le modèle produit en sortie de la transformation est complet. A partir de ce modèle, on peut générer du code exécutable.

L'explicitation des plates-formes a permis de capitaliser une grande partie des règles de transformation. Cependant, ces règles sont dépendantes :

1. de la méthode de modélisation proposée pour la conception des systèmes multitâches.
2. du profil SRM utilisé comme langage pivot pour la modélisation des plates-formes d'exécution et des concepts applicatifs de haut niveau.
3. des hypothèses de correspondance entre l'application et les ressources de la plate-forme. Ces hypothèses peuvent être spécialisés pour établir une correspondance spécifique entre un concept applicatif et une ressource de la plate-forme.

La transformation proposée est donc générique pour des modèles applicatifs et des modèles de plates-formes respectant les hypothèses de modélisations proposés dans cette étude.

3 Conclusion

L'objectif de ce chapitre était d'expérimenter l'intégration des modèles détaillés de plates-formes dans une ingénierie générative dirigée par les modèles. Pour cela, dans un premier temps, un cadre de conception et de déploiement été proposé. Ce cadre vise à déployer une application multitâche indépendante de la plate-forme sur un ensemble de plates-formes cibles. Le cadre proposé offre une méthode de modélisation des applications multitâches. Il supporte aussi une infrastructure de transformation générique permettant de déployer l'application multitâches sur une plate-forme cible spécifiée par son modèle détaillé en entrée de cette infrastructure. Cette infrastructure gère les mécanismes de communication et de synchronisation d'une manière indépendante de la plate-forme.

Ce cadre d'expérimentation a montré qu'une infrastructure de transformation générique basée sur des modèles détaillés de plates-formes est faisable. Il a mis en avant une véritable séparation de préoccupation entre un concepteur qui modélise son application multitâche suivant la méthode proposé, un fournisseur des chaînes de transformations qui offre une infrastructure de transformation sans se préoccuper de la plate-forme cible et enfin, un expert de plate-forme qui fournit un modèle détaillé de cette plate-forme. Ce cadre d'expérimentation a été publié dans la conférence internationale ICECCS 2011 [50].

Le cadre d'expérimentation, proposé dans ce chapitre, doit être confronté aux critères de qualités définis dans le chapitre 3 en termes de réutilisation des transformations et en termes d'impact sur le coût de portabilité des applications. C'est le sujet du chapitre suivant.

CHAPITRE 6

Validation et expérimentation

L'objectif de ce chapitre est d'évaluer et valider le cadre expérimental proposé dans cette étude. Plus particulièrement il vise deux objectifs : a) étudier la performance du code généré et b) mesurer le coût de portage d'une application vers une nouvelle plate-forme en utilisant le cadre proposée.

Pour cela, ce chapitre est divisé en deux sections principales. La première section évalue la performance de notre approche et confronte les résultats vis-à-vis des implantations orientées objet codées manuellement. La deuxième section présente un cas d'étude où une application est portée vers une nouvelle plate-forme (Java temps réel). Premièrement, elle présente la modélisation de la nouvelle plate-forme avec les heuristiques de modélisation présentées dans le chapitre 4, ensuite, elle propose un ensemble de métriques pour estimer le coût de portage de l'application.

1 Évaluation de la performance

L'objectif principal de cette section est de montrer que le processus de transformation générique proposé n'introduit pas une consommation de mémoire supplémentaire signifiante et donc de prouver que ce processus peut être utilisé pour le développement des systèmes multitâches embarqués. En effet, l'empreinte mémoire est considérée comme une contrainte importante pour les systèmes embarqués. Ceci est particulièrement important dans un sens où la complexité croissante des logiciels embarqués a introduit une nécessité croissante d'utiliser des processus modernes de génie logiciel pour faciliter le développement des tels systèmes. La motivation majeure de ces processus de développement est la réutilisation. Pour garantir la réutilisation, ces processus, comme par exemple les approches orientées aspects [67] ou celles orientées composants [81], adoptent des techniques qui conduisent à une quantité supplémentaire de code et des données utilisés pour réaliser une fonctionnalité du système. Cela introduit une augmentation de l'empreinte mémoire du système et nui aux contraintes des systèmes embarqués qui possèdent des ressources limitées en termes de mémoire. Par conséquence, un processus de développement dédié aux systèmes embarqués ne doit pas introduire une consommation de mémoire supplémentaire significative.

1.1 Description de la procédure d'évaluation

Pour montrer que le processus de développement générique présenté dans le chapitre 5 n'introduit pas une augmentation significative de l'empreinte mémoire, une comparaison est effectuée entre une application développée suivant l'approche générique (DPD) et une version orientée objet codée manuellement du même exemple (OO). L'idée sera de voir combien le code généré par l'approche générique est proche de celui codé manuellement.

L'application considérée pour cette comparaison est l'application Producteur/Consommateur présentée dans la section 2. Cette application est composée des entités concurrentes qui doivent répondre à des contraintes temps réel (producteur et consommateur) et des entités qui ne possèdent pas des critères de concurrence (console). Une telle application représente un bon cas d'étude d'une application multitâche, car elle rassemble les aspects de concurrence, de synchronisation et de communication dans une seule application.

A partir du modèle de l'application Producteur/Consommateur présentée dans la section 1.2.1, une implémentation en C++/POSIX est générée en utilisant le cadre expérimental générique. Ensuite, une implantation de cette application est codée manuellement. Après la compilation des deux programmes, la taille du code généré est mesurée. Ensuite, la mémoire consommée pendant la phase d'exécution des programmes est mesurée.

1.2 Résultats et discussion

Le tableau 6.1 montre la taille en octets de la mémoire consommée par le code généré (texte) et les empreintes mémoire par le processeur lors de l'exécution (*run-time*) des programmes. Nous pouvons remarquer que l'application générée à travers l'approche générique (DPD) consomme 270 octets de texte plus que l'implantation orientée objet. Ceci est dû aux aspects comportementaux et aux opérations et propriétés ajoutées qui sont générées par l'approche générique. Leurs rôles est d'assurer l'indépendance du

TABLE 6.1 – Résultat des empreintes mémoires

Version	text (KB)	run-time (KB)
OO	3	19,4
DPD	3.27	19,4

processus de transformation proposé de la plate-forme cible et donc de garantir la réutilisabilité de ce processus. A l'exécution, nous pouvons remarquer que la mémoire consommée par le processeur est similaire pour les deux approches. Ceci est dû au fait que l'approche générique exécute le même code métier que dans l'approche orientée objet sans modification. Il y a juste des appels indirects des opérations à exécuter.

Bien que le léger surcoût de l'empreinte mémoire observée semble être causé par la taille de l'application utilisée comme cas d'étude, ce surcoût reste faible même avec les grandes applications temps réel embarquées. En effet, le surcoût, dû à la taille de chaque motif comportemental généré avec les opérations et les propriétés ajoutées nécessaires pour réaliser un concept de haut niveau, est constant et indépendant du contexte de l'application. Cette taille constante est additionnée à la taille globale de l'application à chaque fois qu'il s'agit d'une réalisation d'un concept de haut niveau. Bien entendu, les grandes applications appliquent généralement plus de stéréotypes sur des éléments applicatifs et nécessitent donc la génération d'un plus grand nombre des aspects comportementaux et l'ajout d'un plus grand nombre d'opérations et de propriétés ajoutées. Mais, en général, le surcoût reste une petite fraction de la taille de l'application totale, et il est estimé comme un surcoût linéaire. En effet, il peut être calculé exactement à l'aide de la formule suivante :

$$\sum_{i=1}^{N}(Overhead_i * Occurrences_i)$$

Où N est le nombre des stéréotypes distincts appliqués au modèle applicatif.
$Overhead_i$ est la taille des aspects comportementaux et des opérations et propriétés ajoutées qui permettent de mettre en œuvre le concept de haut niveau modélisé par le stéréotype i.
$Occurrences_i$ est le nombre d'occurrences du stéréotype i dans le modèle d'application.

En appliquant cette formule sur notre application, nous constatons que le surcoût total provient de :

$$2 * Overhead_{RtFeature} + 1 * Overhead_{PpUnit} + 1 * Overhead_{connector} = 270$$

avec Overhead$_{RtFeature}$, Overhead$_{PpUnit}$ et Overhead$_{connector}$ sont des constantes.

2 Spécification d'un cas d'étude

Afin de vérifier la validité de notre approche, nous allons présenter dans cette section une étude de cas. Elle consiste à supporter dans le processus de déploiement proposé, la génération de code pour la spécification Java pour le temps réel ou Java temps réel (RTSJ) [52]. Le but de cette étude de cas est de confirmer les résultats qui ont motivé cette thèse

en termes de séparation de préoccupations qui aboutit à une réutilisation des processus de déploiement et à une réduction du coût de la portabilité des applications multitâches. Pour cela, nous allons montrer comment les rôles sont partagés et séparés pour supporter cette nouvelle plate-forme dans le processus de déploiement. Pour cela, le modèle détaillé de la plate-forme RTSJ doit d'abord être fourni, ensuite, il sera utilisé comme entrée du processus de déploiement.

2.1 Présentation de Java temps réel

Le succès du langage de programmation Java a conduit à plusieurs tentatives pour étendre ce langage de sorte qu'il soit plus approprié pour une large gamme de systèmes temps-réel. Le problème majeur qui empêchait l'utilisation de Java avec les systèmes à contraintes temps réel strictes, est le mécanisme du ramasse-miettes (*garbage collector*), qui peut préempter n'importe quel thread pour un temps non spécifié, pénalisant ainsi le déterminisme. La spécification RTSJ résout ce problème en introduisant deux nouveaux threads temps-réel (*RealtimeThread*, *NoHeapRealtimeThread*) et deux nouvelles zones mémoires (*ScopedMemory*, *ImmortalMemory*).

Les *RealtimeThread* et *NoHeapRealtimeThread* (NHRT) sont deux threads temps réel avec comme différence que le dernier est protégé des interruptions provoquées par le ramasse-miettes (Garbage Collector). Il est plus prioritaire que le ramasse-miettes, et donc, il peut le préempter à tout moment. En plus, les NHRTs ne peuvent pas accéder à la mémoire tas (HeapMemory) [84] et ne permettent pas donc l'allocation des objets sur le tas et ne réclament pas des objets alloués dans cette zone mémoire.

RTSJ définit également deux zones mémoire différentes de la mémoire tas (HeapMemory) :

1. La mémoire permanente (*ImmortalMemory*) est une zone mémoire partagée par tous les threads. Les objets alloués dans cette zone mémoire ne sont jamais collectés par le ramasse-miettes et ne sont libérés que lorsque le programme termine.

2. La mémoire à portée (*ScopedMemory*) est une zone mémoire où des objets à durée de vie bien définie peuvent être alloués. Le thread temps réel peuvent alors accéder et utiliser cette mémoire. Lors de l'exécution, il peut créer des objets. Les objets créés restent accessibles aussi longtemps que le thread temps réel occupe la zone mémoire. Une fois qu'il sort de cette zone mémoire, tous les objets alloués ne seront alors plus accessibles.

Étant donné que les mécanismes de gestion de la mémoire sont différents dans les différentes zones mémoires, RTSJ impose certaines restrictions concernant l'affectation entre les différentes zones mémoire. Le but est d'éviter d'avoir une référence à un objet qui est collecté lorsque la mémoire à portée (*ScopedMemory*) est réclamée.

La figure 6.1 résume les restrictions imposées par la spécification java temps réel en relation avec les threads, les zones mémoires et les référencements légaux des objets créés dans ces zones mémoires.

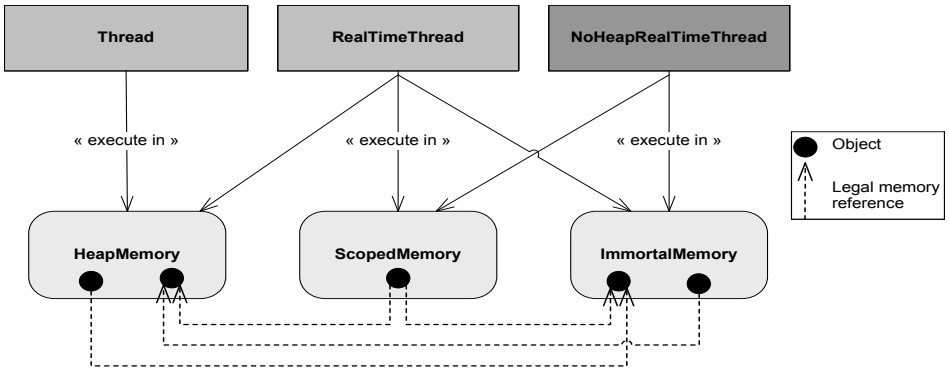

FIGURE 6.1 – Les threads et les zones mémoires de Java temps-réel

2.2 Modélisation de la plate-forme RTSJ

Pour supporter cette plate-forme dans le processus de déploiement, le premier rôle est attribué à un expert en RTSJ pour fournir un modèle détaillé de cette plate-forme, comme présenté dans le chapitre 4. Un des avantages d'une telle modélisation est d'abstraire les développeurs de la complexité du développement en Java temps réel et, par ailleurs, fournir un modèle détaillé de cette spécification qui pourrait être utilisé comme une plate-forme cible dans l'infrastructure de transformation générique proposée. Dans la suite, vu que notre approche ne supporte que les aspects de concurrence, de communications par messages et de synchronisation, alors seuls les modèles détaillés des threads et de communication par messages sont modélisés [1].

2.2.1 Modèle détaillé des threads

La figure 6.2 montre le modèle détaillé des threads de la spécification Java temps-réel. Dans cette figure, on peut distinguer premièrement les trois ressources qui sont fournies nativement par RTSJ. Elles sont annotées par le stéréotype *SwSchedulableResource* et spécialisées par la propriété *isPreemptible* du stéréotype. Cette phase correspond à l'étape de modélisation des ressources existantes de la plate-forme. En effet, le *NHRT* ne peut pas être interrompu par le ramasse-miette, donc la valeur de la propriété *isPreemptible* est fixée à *false*. Le *RealTimethread* peut être préempté donc la valeur de *isPreemptible* est fixée à true.

Ensuite, pour modéliser des ressources concurrentes et périodiques qui correspondent au concept d'une ressource ordonnançable périodique dans SRM, il faut ajouter ces ressources dans le modèle de RTSJ. Dans la figure 6.2, nous pouvons distinguer les ressources *PeriodicImmortalRT*, *PeriodicScopedRt* qui étendent la ressource *RealtimeThread* fournie nativement par la plate-forme et les ressources *PeriodicImmortalNHRT*, *PeriodicScopedNHRT* qui étendent la ressource *NoHeapRealtimeThread*. Ces ressources ajoutées sont annotées par *SwSchedulableResource*.

Cependant, la spécification RTSJ impose des restrictions sur les zones mémoires où un thread temps réel peut s'exécuter. Ainsi, pour distinguer les différentes zones mémoires où

1. les mécanismes de synchronisation sont les même que ceux identifiés lors de la modélisation de la plate-forme Java dans le chapitre 4

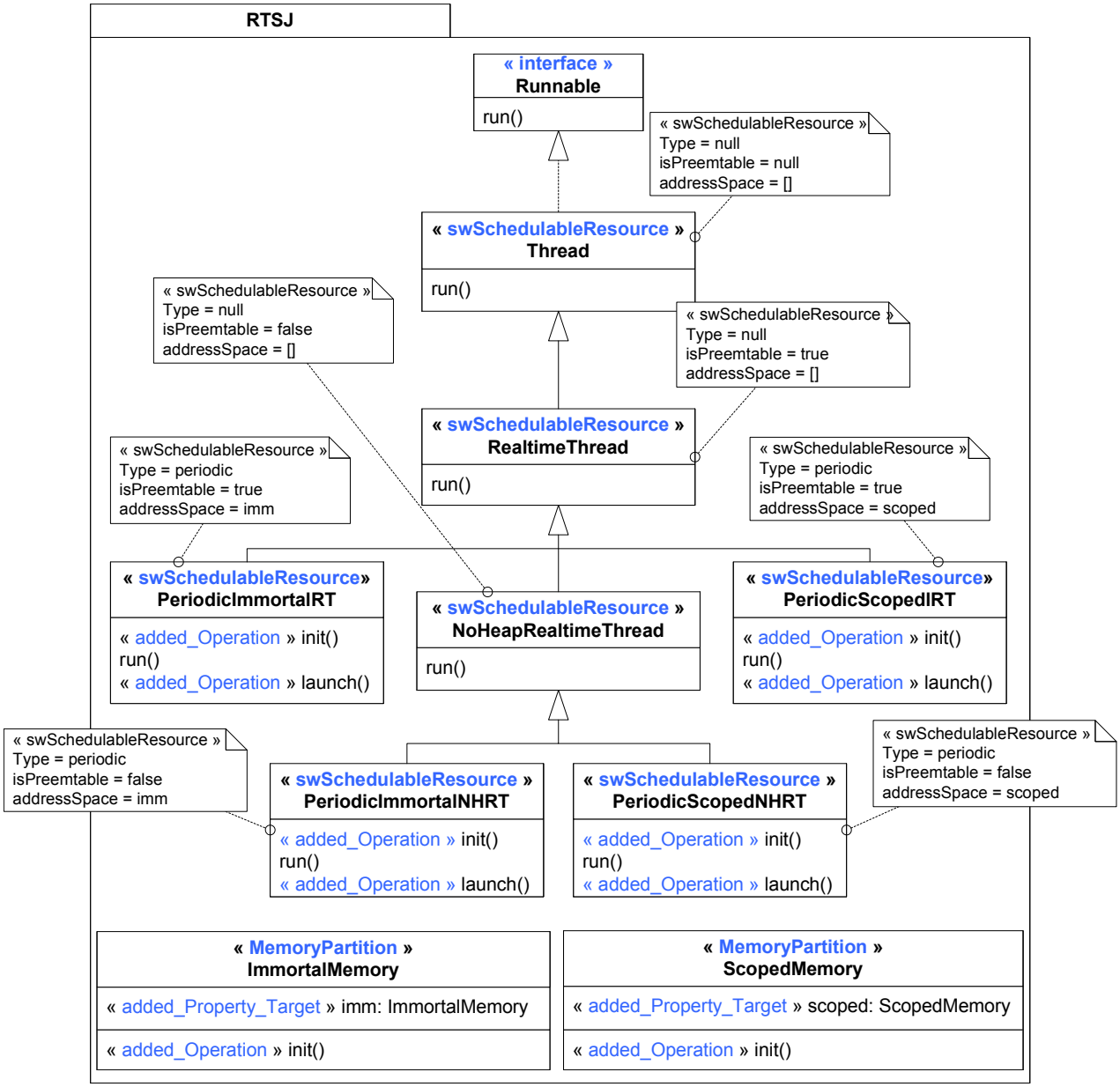

FIGURE 6.2 – Modèle détaillé des threads Java temps réel

```
public void init() {
    PriorityParameters sp = new PriorityParameters(priority);
    RelativeTime start = new RelativeTime(0, 0);
    RelativeTime period_ = new RelativeTime(period, 0);
    RelativeTime cost_ = new RelativeTime(30, 0);
    RelativeTime deadline_ = new RelativeTime(deadline, 0);
    PeriodicParameters rp = new PeriodicParameters(start, period_,
            cost_, deadline_, null, null);
    thrd = new NoHeapRealtimeThread(sp, rp, null,
            ImmortalMemory.instance(), null, this);
    thrd.start();
}
```

FIGURE 6.3 – Opération de création d'un NHRT dans la mémoire immortelle

```
public void run(){
    do{
        launch();
        synchronized (this) {
            while (suspended) {
                wait();
            }
            if (stopped)
                break;
        }
    }while(thrd.waitForNextPeriod());
}
```

FIGURE 6.4 – Comportement périodique de l'opération *run*

les threads peuvent s'exécuter, la propriété *addressSpace* du stéréotype *SwSchedulableResource* est utilisée. Cette propriété est de type *TypedElement*, elle référence donc des éléments typés. Les types de ces éléments doivent être des classes annotées par le stéréotype *MemoryPartition* de SRM[2]. Dans la figure 6.2, les deux zones mémoires *ImmortalMemory* et *ScopedMemory* sont annotées par le stéréotype *MemoryPartition* de SRM. Ces deux ressources possèdent chacune une opération *init* annotée par *Added_Operation* qui permet de créer une instance de chaque ressource. Cette instance créée est capturée par une propriété annotée par *Added_Property_Target* dans le modèle de la ressource (*imm* et *scoped*). Ainsi, dans la figure 6.2, la ressource *PeriodicScopedRt* est une ressource qui modélise un *RealtimeThread* périodique s'exécutant dans une zone mémoire à portée (ScopedMemory). Cela est précisé par la propriété *addressSpace* qui référence la propriété *scoped* typée par la classe *ScopedMemory*.

L'étape suivante consiste à identifier les comportements observables des ressources. Dans la figure 6.2, ces comportements sont mis en œuvre par les opérations qui sont ajoutées aux modèles des ressources et qui sont annotées par *Added_Operation*. Par exemple, les figures 6.3 et 6.4 montrent le comportement associé à l'opération *init* de la ressource *PeriodicImmortalNHRT* pour créer un NHRT dans la mémoire immortelle et le comportement associé à l'opération *run* pour simuler un comportement périodique. Comme nous pouvons constater, les comportements sont des implémentations spécifiques à Java temps réel que le concepteur du modèle détaillé de cette plate-forme spécifie.

2.2.2 Modèle de communication détaillé

La communication entre deux entités concurrentes occupant chacune une zone mémoire différente doit respecter les contraintes imposées par la spécification Java temps réel. Pour

2. Cette restriction est imposée dans la norme

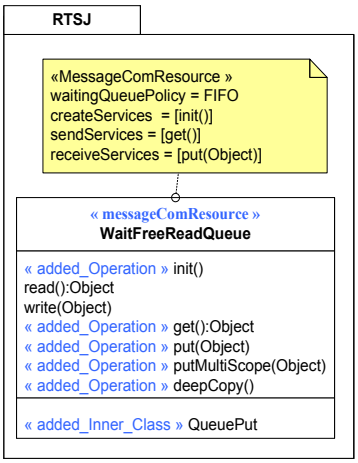

FIGURE 6.5 – le patron Multiscope supporté dans le modèle détaillé de la ressource

cela, plusieurs patrons de conception ont été proposés ([82], [83]) et chacun peut être utilisé selon le besoin du concepteur de l'application.

Pratiquement, les patrons de conception, les plus utilisés et largement connus, sont proposés dans [83]. Et parmi eux, le patron de conception qui permet de répondre au besoin de mise en œuvre d'un mécanisme de communication par message est le patron de conception *Multi-scoped*.

Ce patron représente le cas où des données doivent être envoyées à partir d'une zone mémoire à portée (ScopedMemory) fils vers une zone mémoire parent avec l'intention de conserver ces données et de les tenir à disposition après la réclamation de la zone mémoire fils [3] à portée. Par exemple, supposons qu'un objet concurrent est créé dans la zone mémoire immortelle et le thread associé à cet objet concurrent s'exécute dans une *ScopedMemory*. Si les objets créés dans cette zone mémoire fils doivent être envoyés et stockés dans une ressource de communication créée dans la zone mémoire parent, il n'est pas possible de référencer ces objets créés. La spécification Java temps réel interdit le fait d'avoir une référence de la mémoire parent vers la mémoire fils de type *ScopedMemeory*. Pour cela, il faut appliquer le patron *Multi-Scoped*.

La figure 6.5 montre la prise en compte de ce patron de conception dans le modèle détaillé de la ressource *WaitFreeReadQueue* utilisée pour assurer la communication entre les threads. La ressource *WaitFreeReadQueue* correspond au stéréotype *MessageComResource* de SRM. Les propriétés *sendServices* et *receiveServices* de ce stéréotype référencent respectivement les opérations *get* et *put* annotées par le stéréotype *Added_Operation*. L'opération *put* doit être exécutée dans la zone mémoire où le *WaitFreeReadQueue* est alloué.

La figure 6.6 montre l'extrait de code qui permet de réaliser cette exécution. Dans ce code, l'opération *put* appelle l'opération *getMemoryArea(this)* qui renvoie la zone mémoire où la ressource *WaitfreeReadQueue* est allouée. Ensuite, l'opération *put* est exécutée dans cette zone mémoire. Ceci est réalisé en appelant l'opération *executeInArea(Runnable)* qui prend une classe qui implémente l'interface *Runnable* en paramètre. Pour cela, la classe interne *QueuePut* est ajoutée au modèle de la ressource *WaitFreeReadQueue*. Elle est annotée par le

3. toute zone mémoire créée à partir d'une autre zone mémoire parent est appelé zone mémoire fils.

```
public void put(Object e) {
    MemoryArea mem = MemoryArea.getMemoryArea(this);
    mem.executeInArea(new QueuePut(e));
}

class QueuePut implements Runnable {
    Object i;

    QueuePut(Object e) {
        i = e;
    }

    public void run() {
        putMultiScope(deepCopy(i));
    }
}
```

FIGURE 6.6 – Implémentation du patron de conception Multiscope

stéréotype *Added_Inner_Class*. L'opération *run* de la classe interne ajoutée appelle l'opération *putMultiScope* qui permet d'enregistrer les données dans la zone mémoire où la ressource *WaitFreeReadQueue* est créée. Pour stoker les données, il faut réaliser une copie des données. Ce dernier est réalisé avec l'opération *deepCopy*, ajoutée aussi au modèle de la ressource.

2.3 Intégration du modèle détaillé de RTSJ dans le processus de déploiement

L'intégration de la nouvelle plate-forme modélisée dans le processus de déploiement nécessite l'intervention du fournisseur de la chaîne de transformation sur deux tâches :

1. Il doit offrir au concepteur des systèmes multitâches les outils nécessaires pour spécifier au niveau modèle les différents types des threads temps réel et les zones mémoires dans lesquelles ces threads peuvent s'exécuter. Cela peut être réalisé en ajoutant au stéréotype *RtFeature*, proposé dans la section 1.2.1 du chapitre 5, deux propriétés typées par une énumération qui permet de spécifier respectivement les différents types des threads et les différentes zones mémoire. La figure 6.7 illustre ce stéréotype.

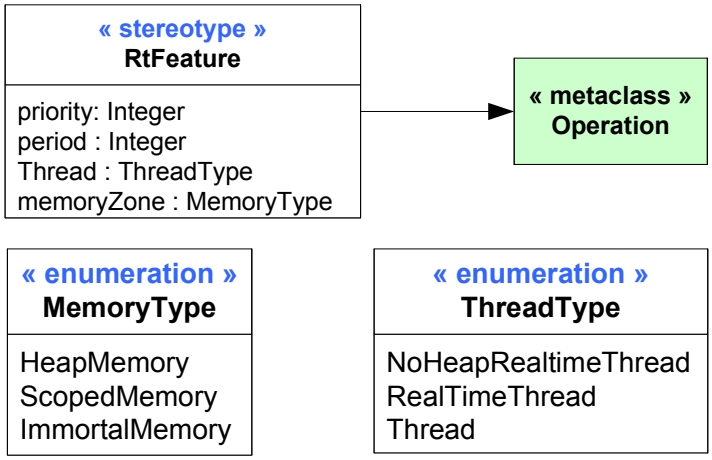

FIGURE 6.7 – Prise en compte des concepts de RTSJ au niveau modèle

2. Le fournisseur de la chaîne de transformation doit spécialiser la relation de correspondance qui permet d'établir les liens entre les classes du modèle de correspondance et les ressources du modèle détaillé de RTSJ. En effet, l'opérateur de conformité $\approx$, utilisé dans l'algorithme 3, ne suffit pas pour distinguer les différentes ressources annotées par *SwSchedulableResource* dans le modèle détaillé de RTSJ (figure 6.2. Il faut spécialiser des règles de conformité spécifiques qui permettent de comparer la zone mémoire de chaque ressource concurrente de la plate-forme avec celle spécifiée au niveau modèle. Cette règle de conformité spécifique sera alors vérifiée comme expliqué dans l'algorithme 3 (Elle est présenté dans l'annexe A).

2.4 Évaluation du coût de la portabilité

Notre approche vise à définir des processus de déploiement réutilisables pour favoriser la portabilité des applications multitâches à travers différentes plates-formes d'exécutions. Le but de cette section est de montrer que l'approche proposée permet de réduire le coût de la portabilité des applications.

Selon Mooney [58], une unité logicielle est portable à travers différents environnements si le coût de portage au nouvel environnement est inférieur au coût de redéveloppement. Dans le contexte de cette étude, l'environnement représente la plate-forme logicielle d'exécution sur laquelle une unité logicielle qui représente l'application multitâche est déployée.

2.4.1 Définitions des métriques pour la mesure de la portabilité

Dans la littérature, plusieurs métriques sont proposées pour mesurer la portabilité. Cependant, toutes sont dédiés à la mesure de la portabilité du code dans un système. Cette section présente des métriques pour mesurer la portabilité d'une application développée suivant des approches dirigée par les modèles. L'intention derrière ces métriques est de refléter l'effort avec lequel une application peut être redéployée à travers différentes plates-formes d'exécutions en suivant une approche d'ingénierie dirigée par les modèles. Afin d'atteindre cet objectif, la notion de degré de portabilité (DoP) proposée par Mooney dans [58] est utilisée. Pour calculer le degré de portabilité, l'équation suivante est proposée :

$$DoP = 1 - \frac{cout\ de\ portage}{cout\ de\ redeveloppement}$$

Dans cette équation, le *coût de portage* est le coût nécessaire pour adapter et porter une unité logicielle (développée dès l'origine pour être portable). Le *coût de redéveloppement* est le coût nécessaire pour redévelopper entièrement une unité logicielle pour s'exécuter sur une nouvelle plate-forme.

Suite à cette équation, si le *DoP* est supérieur à zéro, le portage est plus rentable que le redéveloppement. Dans le cas où la valeur de DoP est égale à 1, cela représente une portabilité maximale.

Afin d'appliquer cette équation aux applications multitâches développées suivant des approches dirigées par les modèles, le processus de développement suivi par ces approches doit être explicité afin de calculer le coût de chaque phase de développement.

Plus particulièrement, nous allons nous intéresser aux approches MDA qui utilisent des transformations spécifiques et à notre approche à base des transformations génériques.

En effet, dans une approche dirigée par les modèles, le processus de développement consiste en une phase de modélisation suivie d'une phase de transformation de modèle. Le coût de développement d'une application pour une plate-forme cible peut être représenté alors par l'équation suivante :

$$C_{globale} = C_{modelisation} + C_{transformation}$$

La modélisation de l'application est une étape commune à toutes les approches IDM. Elle est réalisée par un concepteur qui s'intéresse juste à l'aspect fonctionnel de l'application. Le modèle obtenu lors de cette modélisation est indépendant de la plate-forme. Donc, d'un point de vue de conception, le modèle est portable et le coût de la portabilité pour le concepteur est nul. Par conséquence le coût de la portabilité globale de l'application $C_{globale}$ peut être restreint au coût du développement de la transformation de modèle. Cependant, les transformations de modèles peuvent être basées sur des transformations spécifiques, à la base des approches traditionnelles, ou des transformations génériques qui sont à la base de notre approche.

Pour cela, Nous allons supposer que C_{Tdev} est le coût de développement d'une transformation de modèles suivant des règles de transformation spécifiques et C_{Gdev} est le coût de développement d'une transformation de modèles en suivant des règles de transformation génériques. Ces coûts peuvent être divisés en plusieurs parties comme le montre les équations suivantes :

$$C_{Tdev} = C_{etudePlatforme} + C_{transfSpecifique}$$

$$C_{Gdev} = C_{transfGenerique} + C_{modelisationPlateforme}$$

En effet, ces coûts sont partagés entre deux contributeurs potentiels impliqués dans le processus de transformation : le fournisseur de la chaîne de transformation et le fournisseur de plate-forme. Dans les approches traditionnelles, les fournisseurs des chaînes de transformation étudient premièrement la plate-forme cible, puis ils écrivent des règles de transformation spécifiques à cette plate-forme. Selon l'approche générique proposée, les fournisseurs de plate-forme offrent des modèles détaillés de leurs plates-formes en suivant l'approche de modélisation des plates-formes proposée dans le chapitre 4, alors que les fournisseurs des chaînes de transformation écrivent des règles de transformation génériques.

En réalité, l'étude d'une nouvelle plate-forme exécution est coûteuse en temps et la modélisation détaillée de la plate-forme logicielle d'exécution nécessite la maîtrise des techniques de modélisation de la part des fournisseurs des plates-formes. Cependant, lorsque les fournisseurs de plate-forme offrent un modèle détaillé de leurs plates-formes, la qualité du modèle de plate-forme sera garantie car elle est réalisée par des experts qui connaissent bien leur plate-forme. Dans cette partie, on n'est pas en mesure d'estimer si l'étude d'une nouvelle plate-forme est plus coûteuse ou non qu'un modèle détaillé de cette

même plate-forme, mais il faut juste signaler que la séparation de préoccupation est toujours recommandée dans le développement logiciel [60].

Afin de calculer le degré de portabilité, il faut maintenant estimer le coût de développement des transformations de modèle. Une transformation se compose principalement de règles qui sont utilisées pour concrétiser les concepts utilisés au niveau du modèle applicatif. Dans le calcul du coût de développement des règles de transformations, il est difficile d'estimer avec précision le gain financier lorsqu'une règle doit être spécifique ou générique et portée ou redéveloppée pour une autre plate-forme. Alternativement, une certaine mesure de la taille du logiciel pourrait être considérée comme une mesure raisonnable indirecte des coûts de développement [59]. Dans la pratique, le coût de portage ou de redéveloppement peut être représenté par le nombre de lignes de code qui doit être modifié dans les règles de transformation lors de changement de la plate-forme d'exécution cible. Ainsi, le coût de développement global d'une application en utilisant des transformations spécifiques et génériques est représenté par :

$$C_{Tdev} = N_{GT}$$

$$C_{Gdev} = N_{SP}$$

N_{SP} et N_{GT} représente le nombre des lignes de code qui sont écrits afin de réaliser les transformations spécifiques et génériques respectivement pour une plate-forme cible. En se basant sur ces coûts de développement, notre équation de degré de portabilité sera comme suit :

$$DoP = 1 - \frac{N_{GT}}{N_{SP}}$$

2.4.2 Evaluation du coût des transformations de modèles

Le degré de portabilité de l'approche basée sur une transformation générique par rapport aux approches traditionnelles peut être estimé en utilisant l'équation définie dans la section précédente. Pour cela, il faut calculer premièrement le coût de développement des transformations spécifiques et génériques.

Pour calculer le degré de la portabilité, une transformation de modèle spécifique ciblant la plate-forme Java standart est tout d'abord développée au début. C'est une transformation spécifique de type modèle/texte. Ensuite, cette même transformation spécifique est réutilisée pour cibler deux autres plates-formes : Java temps réel (RTSJ) et C++/POSIX. La transformation spécifique développée encapsule des implémentations spécifiques relatives à la concurrence, la synchronisation et les mécanismes de communication. Enfin, une comparaison est réalisée entre la transformation spécifique et les efforts requis pour sa réutilisation avec des nouvelles plates-formes cibles et la transformation générique proposée.

La comparaison est basée, comme proposé dans la section précédente, sur le calcul du nombre de lignes de code qui ont été écrites pour réaliser la transformation vers une plate-forme cible. Par exemple, pour une transformation spécifique ou générique développée à

l'origine pour une plate-forme donnée, ce nombre représente le nombre total de lignes de code nécessaires pour implémenter cette transformation. Lors de modifications de la plate-forme cible, ce nombre représente le nombre de lignes de code qu'il faut modifier dans l'implémentation de la transformation originale pour qu'elle soit réutilisable avec la nouvelle plate-forme.

Le tableau 6.2 indique le nombre de lignes de code qui ont été écrites pour la mise en œuvre d'un ensemble de concepts, choisis à titre indicatif, liés aux stéréotypes *SwSchedulabeResource*, *MessageComResource* et *SwMutualExclusionResource* dans la transformation de modèles spécifique (N_{SP}) et la transformation de modèle générique (N_{GT}). A partir de ces chiffres, le degré de portabilité de l'approche basée sur une transformation générique est calculé.

TABLE 6.2 – Estimation du coût des transformations de modèles

Platform	N_{SP}	N_{GT}	DoP
Java	350	454	-0.3
RTSJ	250	20	0.92
C++/POSIX	320	0	1
Total	920	474	0.48

Il est clair que le développement d'une transformation générique pour une seule plate-forme est plus coûteux que le développement d'une transformation spécifique (DoP Java <0). Cependant, lors du portage des transformations vers la plate-forme RTSJ, le degré de portabilité est devenue grand ce qui signifie que nous parvenons à une réduction du coût de la portabilité. Lors du portage de la transformation sur le C++/POSIX, nous constatons que le degré de portabilité est égal à 1, ce qui représente une portabilité maximale. La raison de cette portabilité maximale est principalement due à l'ensemble restreint des caractéristiques des plates-formes supportées par notre transformation générique à ce jour. La dernière ligne du tableau indique le nombre total de lignes de code qui été écrites pour réaliser la portabilité au travers des trois plates-formes cibles. Il montre également une réduction de l'effort pour réaliser la portabilité lors de l'utilisation de l'approche générique. En effet, le degré de portabilité indique une réduction d'environ 48% du coût de portabilité. Cette valeur augmente encore lors de l'augmentation du nombre de plates-formes.

3 Conclusion

Dans ce chapitre, une évaluation du cadre d'expérimentation proposé dans le chapitre 5, a été présentée. L'évaluation s'est portée sur l'étude de la performance du code généré pour les applications multitâches implantées en utilisant le cadre proposé. Et deuxièmement, l'expérimentation de l'intégration d'une nouvelle plate-forme cible dans le processus de déploiement.

Les expérimentations que nous avons menées révèlent que le code généré est très proche de celui codé manuellement. Donc, le processus de déploiement proposé n'introduit pas un surcoût remarquable de consommation de mémoire. En plus, l'étude de l'intégration de la spécification Java temps réel comme plate-forme cible dans le processus de déploiement,

nous a permis de confirmer les critères de qualités qui ont été définis dans le chapitre de l'état de l'art. En effet, une véritable séparation de préoccupation a été réalisée, l'infrastructure de transformation a été réutilisée et, enfin, une réduction du coût de la portabilité des applications a été constatée lorsque plusieurs plates-formes sont envisagées comme plate-forme cible.

CHAPITRE 7

Conclusion

1 Bilan

Ce travail de thèse porte sur la prise en compte des plates-formes logicielles d'exécution et de leurs aspects comportementaux dans une ingénierie générative dirigée par les modèles. L'objectif est de favoriser la séparation de préoccupation durant toutes les phases de développement des systèmes multitâches. L'ingénierie dirigée, et plus particulièrement, l'approche MDA par les modèles offre les moyens pour une telle séparation. En effet, en exprimant les fonctionnalités du système d'une manière indépendante de technologies cibles, une première étape de séparation de préoccupations est atteinte. Elle a permis de libérer le concepteur de ce préoccuper des implémentations spécifiques à la plate-forme, mais pas les experts IDM qui doivent maintenant assurer la prise en compte de ces implémentations spécifiques. Cette pratique rend les transformations non réutilisable.

Pour cela, nous avons proposer d'appliquer le principe de séparation de préoccupation lors du développement des transformations. Cela se concrétise par une approche qui permet à l'expert de l'IDM de fournir une infrastructure de transformation générique qui sera utilisée pour déployer des modèles des applications multitâches sur des plates-formes logicielles d'exécution dont leurs modèles détaillés sont fournis par des experts des plates-formes.

Pour cela, le contexte applicatif de cette étude a été centré sur les systèmes logiciels multitâches, c'est-à-dire sur les systèmes applicatifs s'exécutant sur des plates-formes logicielles d'exécution multitâches. Concernant, le contexte technologique, l'ingénierie dirigée par les modèles a été adoptée pour modéliser l'application et les plates-formes, et réaliser de déploiement des applications. Ensuite, la démarche d'étude a été agencée en trois parties.

Premièrement, un état de l'art a été mené sur la modélisation des plates-formes logicielles d'exécution et le déploiement des systèmes multitâches. Concernant la modélisation des plates-formes, il est apparu que toutes les approches s'intéressent d'abord à la modélisation structurelle des plates-formes en laissant les aspects comportementaux de coté. Pour le déploiement, deux types d'approches ont été distingué : celle qui se basent sur

des transformations de modèles spécifiques pour produire l'implantation du système et celles qui se base sur des transformations génériques pour produire des implantations spécifiques non complètes où les caractéristiques comportementales ne sont pas abordées. Par conséquent, la deuxième partie de cette thèse a été orientée sur la modélisation structurelle et comportementale des plates-formes d'exécution.

La seconde partie de cette thèse s'est intéressée à la définition d'une approche de modélisation des plates-formes qui permette de capturer toutes les caractéristiques structurelles et comportementales des ces plates-formes dans les modèles. Pour cela, cette approche propose des heuristiques de modélisation permettant d'offrir des modèles détaillés des plates-formes. Ces modèles détaillés sont décrits d'une manière commune par le profil SRM du standard MARTE. Une telle description doit donc permettre la réalisation d'un cadre de développement des applications multitâches, basé sur des transformations de modèles génériques indépendantes de concepts applicatifs et de la plate-forme cible. Ce cadre a été expérimenté dans la troisième partie.

La troisième, et dernière, partie s'est focalisée sur la mise en œuvre d'un processus de déploiement des applications multitâches, basé sur les modèles détaillés des plates-formes. Pour cela, vu que toutes les informations relatives aux plates-formes sont capturées dans des modèles de plates-formes décrits d'une manière commune, une infrastructure de transformation de modèles est proposée. Cette transformation a permis le déploiement des modèles applicatifs multitâches, réalisé, par un concepteur, sur des plates-formes cibles identifiées par leurs modèles détaillés à l'entrée de la transformation. La transformation a exploité la notion par des patrons de conceptions comportementaux permettant la gestion des mécanismes de communication et de synchronisation d'une manière indépendante de la plate-forme. Enfin, le processus de déploiement proposé a été évalué. Il a montré une réduction dans le coût de déploiement des applications sur plusieurs plates-formes sans impliquer un surcoût de performance.

2 Perspectives

2.1 Perspectives à court terme

Les travaux à court terme concernent essentiellement l'amélioration du cadre d'expérimentation proposé. Cette amélioration doit se porter sur les points suivants :

1. Proposer des techniques qui permettent au concepteur d'effectuer des appels aux services de la plate-forme d'une manière générique. Cela permet de répondre au besoin identifié dans la section 2.2.1 du chapitre 4.
2. Analyser la compatibilité des informations spécifiées au niveau modèle par rapport à ce que fournit la plate-forme. En effet, le cadre d'expérimentation supporte une transformation de modèles générique bijective, c'est-à-dire, pour chaque concept identifié au niveau applicatif, la transformation suppose qu'il y a une ressource de la plate-forme qui correspond à ce concept. En réalité, cette supposition n'est pas toujours valable. Ainsi, un concepteur peut spécifier une certaine politique de communication au niveau modèle qui est supportée par une plate-forme d'exécution mais pas par une autre. En plus, ce concepteur peut spécifier des contraintes temporelles pour une

application multitâche qui ne seront pas valables une fois le code généré, comme par exemple lorsqu'il spécifie une priorité pour une ressource concurrente qui est plus grande de la priorité maximale que peut avoir une tâche dans une plate-forme. Tout cela nécessite une analyse préliminaire du modèle applicatif par rapport à la plate-forme cible. Ce point fait actuellement une partie d'une étude à part entière.

3. Effectuer une étude sur des optimisations éventuelles dans le code généré qui peuvent être réalisées avec notre approche de développement dirigée par les modèles. Une étude sur l'intérêt d'une telle approche pour réaliser des optimisations dans le code généré, a été présentée dans [85]. La prise en compte des ces optimisations est possible en enrichissant notre transformation générique par des règles de transformation qui réalisent ces optimisations (par exemple, la détection des états inatteignables dans une machine à état [85]).

2.2 Perspectives à long terme

Les études à long terme doivent :

1. Améliorer la méthode de modélisation des systèmes multitâches proposée et définir une méthodologie outillée complète basée sur des modèles détaillés des plates-formes. La méthodologie $ACCORD|UML$ [55] peut être étendue et utilisée pour cibler, plusieurs plates-formes logicielles d'exécution.
2. Étendre le cadre de développement proposé pour supporter le déploiement des systèmes multitâches distribués. Pour cela, il est possible, par exemple, d'intégrer cette approche dans la plate-forme de modélisation et de déploiement eC3M [56] qui définit à travers les connecteurs des mécanismes de communication spécifiques à ce domaine d'application et fournit un support de déploiement permettant de spécifier le nœud sur lequel chaque composant du système doit être déployé.

ANNEXE A

Implantation des algorithmes d'équivalence

1 Implantation de l'équivalence entre les stéréotypes et classe du modèle de correspondance

Le *helper correspondToStereotype* A.1 est utilisé pour retrouver la classe dans le modèle de correspondance équivalente au stéréotype appliqué dans le modèle applicatif. Ce *helper* compare les valeurs des propriétés de la classe avec les valeurs des propriétés du stéréotype. L'équivalence est établie si toutes les valeurs des propriétés sont renseignées de la même façon.

Listing A.1 – Détéction de la classe dans le modèle de correspondance qui correspond au concept applicatif

```
helper context UML2!Class
def: correspondToStereotype(stereo: UML2!Stereotype, elt:UML2!Element):
        Boolean = let size : Integer = self.getAttributes()
        ->select(att|not att.defaultValue.oclIsUndefined())->size() in
        if self.getAttributes()->select(att|not att.defaultValue.oclIsUndefined())
        ->iterate(att; seq:Sequence(UML2!Property)= Sequence{}|
                if stereo.getAttributes()->select(stereoAtt|
                        if stereoAtt.name = att.name then
                                if elt.getValue(stereo, stereoAtt.name) =
                                att.defaultValue.value then
                                true
                                else false endif
                        else false endif)->isEmpty()
                then seq->flatten() else seq->append(att) endif)->size()=size
        then true else false endif;
```

2 Implantation de l'équivalence entre la classe et ressource

Le helper présenté dans le listing A.2 permet de trouver la ressource dans le modèle détaillé de la plate-forme qui est annotée par le même stéréotype et dont les valeurs des ces propriétés de type primitifs sont renseignées de la même façon. Dans le cas ou cette relation de conformité ne suffit pas pour déduire les correspondances, nous pouvons remarquer dans *helper* un appel d'un autre *helper specificCorrespondance*. Ce dernier permet de spécialiser la déduction de correspondances entre les classes du modèle de correspondance et les ressources de la plate-forme. Pour une plate-forme donnée, si il n'y a pas un besoin pour la spécialisation, le helper renvoie toujours *true*. Lorsqu'il y a besoin, une spécialisation peut être définie.

Listing A.2 – Détéction de la classe dans le modèle de correspondance qui correspond au concept applicatif

```
helper context UML2!Element
def: getElementFromTargetWithSameStereotype(sourceElt: UML2!Class,
stereo:UML2!Stereotype):
        UML2!Element=    thisModule.stereotypedTargetElements
        ->select(e|e.isStereotypedWith(stereo))
        ->select(targetElt|let size : Integer =
        targetElt.getAppliedStereotype().getStereotypeAllInheritedTags()
        ->select(att|not targetElt.getValue(targetElt.getAppliedStereotype(),
        att.name).oclIsUndefined()and not targetElt.getValue(targetElt.
        getAppliedStereotype(), att.name).
        oclIsKindOf(Sequence(OclAny)))->size() in
        if targetElt.getAppliedStereotype().
        getStereotypeAllInheritedTags()
         ->select(att|not targetElt.getValue(targetElt.
         getAppliedStereotype(),
          att.name).oclIsUndefined()and not targetElt.getValue(
          targetElt.getAppliedStereotype(),
           att.name).oclIsKindOf(Sequence(OclAny)))->select(att|
                        if targetElt.getValue(
                        targetElt.getAppliedStereotype(), att.name) =
                                sourceElt.getValue(stereo, att.name) then
                                true
                        else false endif)->size()=size-1
                        and thisModule.specificCorrespondance(targetElt,
                         sourceElt)then
                        true
                else false endif)->first();
```

Par exemple, avec la plate-forme Java temps réel, la propriété addressSpace du stéréotype SwSchedulableResource est utilisée pour spécifier la zone mémoire dans laquelle un thread temps réel doit s'exécuter. Comme déjà présenté, il existe trois types des zones mémoires :1) mémoire immortelle, 2) mémoire à porter, et 3) mémoire tas. Pour définir la zone mémoire de chaque threah temps réel, la propriétés addressSpace référence un élément typé par la zone mémoire choisie. De même, dans le modèle de correspondance, la propriété addressSpace du stéréotype SwSchedulableResource est utilisée pour spécifier la zone mémoire du concept RtFeature. Avec la relation de conformité initial, la transformation ne peut pas déduire les correspondances entre les concepts de concurrence de haut niveau et les threads temps réel. Une spécialisation de cette relation est nécessaire pour permettre à

la transforamtion de vérifier aussi la zone mémoire définie pour l'exécution.

Ainsi, dans le listing A.3, une spécialisation de la relation de conformité est proposée. Elle vérifie si les valeurs de la propriété addressSpace dans deux stéréotypes SwSchedulableResource annotant respectivement une ressource dans le modèle de correspondance et une dans le modèle de la plate-forme sont conforme.

Listing A.3 – Spécialisation de la relation de conformité

```
helper def: specificCorrespondance(targetElt:UML2!Element,
sourceElt:UML2!Elt):Boolean =
 if sourceElt.getAppliedStereotype().name = 'SwSchedulableResource' then
        if targetElt.getValue(targetElt.getAppliedStereotype(),
                'addressSpace')->first().type.toString() =
                 sourceElt.getValue(sourceElt.getAppliedStereotype(),
                        'addressSpace')->first().name then
                        true
        else false endif
 else true endif;
```

ANNEXE B

Librairie ATL dédiée à l'implantation des aspects comportementaux

Ce chapitre est une illustration des implantations des patrons de conception comportementaux définis dans le chapitre 5 pour la mise en œuvre de la concurrence et des mécanismes de communication et de ressources partagées. Dans la suite nous présenterons l'ensemble des règles de transformations ATL décrites pour la mise en œuvre des ces patrons.

1 Implantation du patron de conception *EntyPoint*

L'implantation de ce patron est réalisé dans le listing B.1. Elle est constitué d'un ensemble des règles et des helpers permettant la création d'une activité (*createEntryPointsBehavior*) et les nœuds (*createEntryPointsNodes*) et les connections entre ces nœuds (*createObjectEdges*, *createFlowEdges*). Le helper *findEntryPointsSpecification* trouve l'opération dont l'activité créée est son spécification. C'est l'opération générée qui correspond à l'opération référencée par la propriété *entryPoints* du stéréotype *SwSchedulableResource*.

Listing B.1 – implantation du patron *entryPoint*

```
rule createEntryPointsBehavior(elt:UML2!Operation){
        to d:UML2!Activity(
          name<-'entry_Points_behavior_'+elt.name,
          specification<-thisModule.findEntryPointsSpecification(elt),
          node<-thisModule.createEntryPointsNodes(elt),
          edge<-thisModule.createObjectEdges()
                ->union(thisModule.createFlowEdges())->flatten()
        )
        do{
            d;
           }
}
helper def: findEntryPointsSpecification(elt: UML2!Element):
```

```
        UML2!Operation= thisModule.operationAndTransformedOperation
        ->select(tuple|tuple.sourceElement.name = elt.name)
        ->select(tuple|tuple.platformOperation.isEntryPoints())
        ->collect(gen|gen.generatedOperation)->first();

helper context UML2!Operation
def:isEntryPoints():Boolean =
   if self.class.getValue(self.class.getAppliedStereotype(),'entryPoints')
   ->first().name= self.name then true else false endif;

helper def: createEntryPointsNodes(elt:UML2!Element): Sequence(UML2!Node) =
        thisModule.createInitialFinalNode()->asSequence()
        ->append(thisModule.makeCallOpAction(elt, elt))->flatten();
```

2 Implantation du mécanisme de protection des ressources partagées

ce mécanisme consiste à accéder aux opérations en exclusion mutuelle. Pour cela, dans la listing B.2, la règle *createProtecetedBehavior* crée l'activité. Dans cette règle, on définit toutes les propriétés de cette activité. Premièrement, on trouve l'opération spécifiée par cette activité. C'est une opération dupliquer de l'opération originale à protéger. Deuxièment, on crée les nœuds de cette activité (*createProtectedBehaviorNodes*). Les trois principaux nœuds sont des actions d'appels aux opération acquireServices et à l'opération à protéger. C'est l'opération générer avec un suffixe *_Protected*. Enfin, une action d'appel à l'opération releaseServices.

Notant que cette activité supporte aussi la gestion des arguments et des valeurs de retour des opérations protégées.

Listing B.2 – implantation du patron de protection

```
rule createProtecetedBehavior(op:UML2!Operation, class:UML2!Class){
        to d:UML2!Activity(
        name<-'behavior_create_'+op.name,
        specification<-thisModule.findDuplicateSpecification(op),
        node<-thisModule.createProtectedBehaviorNodes(class,op),
        ownedParameter <-thisModule.parametersForActivityParameterNode,
        edge<-thisModule.createObjectEdges()
                                ->union(thisModule.createFlowEdges())->flatten()
        )
        do{
            thisModule.parametersForActivityParameterNode<-
                                thisModule.parametersForActivityParameterNode->
                                select(p|not d.ownedParameter->includes(p));
            d;
            }
}

helper def: findDuplicateSpecification(op:UML2!Operation): UML2!Operation=
thisModule.duplicatedOperations->select(top|top.name=op.name)->first();

helper def: createProtectedBehaviorNodes(class:UML2!Class, op:UML2!Operation):
        Sequence(UML2!Node)= thisModule.callAcquireServices(class)
```

```
        ->union(thisModule.callBusinessOperation(op))
        ->union(thisModule.callReleaseServices(class))
        ->union(thisModule.createReadStructuralFeature(class, op))
        ->union(thisModule.createActivityParameter(op))
        ->union(thisModule.createInitialFinalNode())->flatten();

helper def: createReadStructuralFeature(class:UML2!Class, op:UML2!Operation):
        Sequence(UML2!ReadStructuralFeatureAction)= class.ownedAttribute
        ->select(att|class.getValue(class.getAppliedStereotype(),
                'identifierElements')->first().name= att.name)
        ->iterate(att;ens:Sequence(UML2!ActivityNode)=Sequence{}|ens
        ->append(thisModule.makeReadStructuralFeature(att)));

rule makeReadStructuralFeature(att:UML2!Property) {
    to t : UML2!ReadStructuralFeatureAction (
        name <- att.name,
        result <- thisModule.makeOutput(att)
       )
           do{
                t;
           }
}

helper def: callAcquireServices(class:UML2!Class):
        Sequence(UML2!CallOperationAction)
        = class.ownedOperation->select(op|op.isAcquireService())
        ->iterate(opt; ens:Sequence(UML2!ActivityNode)=Sequence{}|ens
        ->append(thisModule.makeCallOpAction(opt, opt)));

helper def: callReleaseServices(class:UML2!Class):
        Sequence(UML2!CallOperationAction) = class.ownedOperation
        ->select(op|op.isReleaseService())
        ->iterate(opt; ens:Sequence(UML2!ActivityNode)=Sequence{}|ens
        ->append(thisModule.makeCallOpAction(opt, opt)));

helper def: callBusinessOperation(op:UML2!Operation):
        Sequence(UML2!CallOperationAction) =
        thisModule.applicationOperationAndTransformedOperation
        ->select(tuple|tuple.platformOperation.name = op.name)
        ->iterate(tuple; ens:Sequence(UML2!ActivityNode)=Sequence{}|ens
        ->append(thisModule.makeCallOpAction(op, tuple.generatedOperation)));

rule makeCallOpAction(sourceOp:UML2!Operation,
generatedOperation:UML2!Operation)
to t:UML2!CallOperationAction(
        name<-'call_'+generatedOperation.name+'_'+generatedOperation.class.name,
        operation <- generatedOperation,
        argument <- thisModule.makeArguments(generatedOperation),
        target<-generatedOperation.makeTargetPin(sourceOp),
        result<-thisModule.checkResult(generatedOperation)
)
do{
          t;
          thisModule.NodeElementsCreated<-
            thisModule.NodeElementsCreated->append(t);
           }
```

```
}
```

3 Implantation de patron de communication

Pour chaque opération référencée au niveau modèle par une propriété d'un stéréotype SRM, la transformation , présentée dans la listing B.3, crée une activité qui appelle l'opération de la plate-forme jouant le même rôle de l'opération utilisée au niveau modèle.

Listing B.3 – implantation du patron de protection

```
rule createSendReceiveServiceBehavior(elt:UML2!Operation,
                source:UML2!Element, class:UML2!Class){
        to d:UML2!Activity(
        name<-elt.name,
        specification<-thisModule.findSendReceiveServiceSpecification(elt)
        node<-thisModule.createSendReceiveNodes(elt, source, class)
        edge<-thisModule.createObjectEdges()
                ->union(thisModule.createFlowEdges())->flatten()
        )
        do{
            d;
            }
}

helper def: findSendReceiveServiceSpecification(elt: UML2!Element):
UML2!Operation= thisModule.sourceOperationAndTransformedOperation
->select(tuple| tuple.sourceOperation.name = elt.name)
->collect(gen|gen.generatedOperation)->first();

helper def: createSendReceiveNodes(elt:UML2!Element, source:UML2!Element
, class:UML2!Class): Sequence(UML2!Node) =
thisModule.createInitialFinalNode()->asSequence()
->union(thisModule.createSendReceiveServices(elt, source, class))->flatten();

helper def: createSendReceiveServices(sourceOp:UML2!Operation,
 source:UML2!Element, class:UML2!Class): Sequence(UML2!CallOperationAction)=
  thisModule.sourceOperationAndTransformedOperation
  ->select(tuple|if source.getValue(source.getAppliedStereotype(),
    'sendServices')->first().name=sourceOp.name then
     tuple.sourceOperation.isResourceService('sendServices')and
     tuple.sourceElement.name = class.name else
     tuple.sourceOperation.isResourceService('receiveServices')and
     tuple.sourceElement.name = class.name endif)
     ->iterate(tuple; ens:Sequence(UML2!ActivityNode)=Sequence{}|
ens->append(thisModule.makeCallOpAction(
tuple.sourceOperation, tuple.generatedOperation)));
```

Table des figures

Bibliographie

[1] Mordechai Ben-Ari : *Principles of Concurrent Programming*. Prentice-Hall, 1982.

[2] Andrew Wellings : *Concurrent and Real-Time Programming in Java*, John Wiley and Sons, 2004.

[3] J.C. Laprie, éditeur : *Guide de la Sûreté de Fonctionnement*. Cépaduès, 1995.

[4] Jean-Philippe Babau : *Formalisation et structuration des architectures opérationnelles pour les systèmes embarqués temps réel*. Mémoire d'habilitation à diriger des recherches, Université Institut National des Sciences Appliquées de Lyon et Université Claude Bernard de Lyon 1, Décembre 2005.

[5] Ahmad Badreddin Alkhodre : *Développement formel de systèmes temps réel à l'aide de SDL et IF*. Thèse de doctorat, Institut National des Sciences Appliquées de Lyon, Septembre 2004.

[6] Safouan Taha : *Modélisation conjointe logiciel/matériel des systèmes temps réel*. Thèse de doctorat, Université de Lille, Mai 2008.

[7] Ian Philips : *When less means more ; and more, the-same*. In IEEE Second International Symposium on Industrial Embedded Systems SIES'2007, 2007.

[8] Yvon Trinquet et Jean-Pierre Elloy. *Systèmes d'exploitation temps réel-Principes*. Dans Techniques de l'ingénieur. Maurice POSTELS S.A., Mars 1999.

[9] K. Berlin, J. Huan, M. Jacob, G.Kochhar,J.Prins,W.Pugh, P.Sadayappan,J. Spacco, and C.W.Tseng : *Evaluating the Impact of Programming Language Features on the Performance of Parallel Applications on Cluster Architectures*, Proc. LCPC2003,Springer Verlag,Texas, USA, 2003, pp.194-208.

[10] F.B. Schneider : *On Concurrent Programming*. Springer Verlag, 1997.

[11] J.-P. Elloy. Editorial : Quelle informatique est donc nécessaire pour automatiser en temps réel. *Technique et Sciences Informatique*, 7(5) :395-396, 1988.

[12] J.-P. Elloy : *Systèmes réactifs synchrones et asynchrones*. Dans l'École d'été ETR'99 - Applications, Réseaux et Systèmes, pages 43-52, Poitiers, 1999. ENSMA

[13] Jean Bézivin : *Sur les principes de base de l'ingénierie des modèles*. RSTI-L'Objet, 10(4) :145-157, 2004.

[14] Jean Bézivin : *On the unification power of models*. Software and System Modeling (SoSym), 4(2) :171-188, 2005.

[15] Colin Atkinson et Thomas Kuhne : *Model-driven development :A metamodeling foundation*. IEEE Software, 20(5) :36-41, 2003.

[16] Edwin Seidewitz : *What models mean*. Dans IEEE software, volume 20, pages 26-32, Los Alamitos, CA, USA, Octobre 2003. IEEE Computer Society

[17] Jean-Marie Favre : *Towards a basic theory to model model driven engineering*. Dans Workshop on Software Model Engineering (WISME 2004), Lisbon, Portugal, Octobre 2004.

[18] B. Selic : *The Pragmatics of Model-Driven Development*. IEEE Software, Vol. 20(5), September 2003.

[19] T. Mens, K. Czarnecki, and P. Van Gorp : *A taxonomy of model transformations*. In International Workshop on Graph and Model Transformation. vol. 4101, pp. 2005-02.

[20] Object Management Group : MOF QVT final adopted specification. http ://www.omg.org/docs/ptc/05-11-01.pdf, Novembre 2005.

[21] Frédéric Jouault : *Contribution à l'étude des langages de transformation de modèles*. Thèse de doctorat, Université de Nantes, 2006.

[22] Jonathan Musset, Etienne Juliot et Stéphane Lacrampe : AcceleoTM 2.2 : Guide utilisateur. Obeo, http ://www.acceleo.org/pages/accueil/fr, Avril 2008.

[23] Markus Völter et Thomas Stahl : *Model-Driven Software Development : Technology, Engineering, Management*. Numéro 978-0-470-02570-3. John Wiley & Sons, Juin 2006.

[24] Eclipse Model to Text Project. Java Emission Template. http ://www.eclipse.org/modeling/m2t/ ?project=jet#jet.

[25] IBM rational rose technical developer, 2008. http ://www.ibm.com/- software/awd-tools/developer/rose/.

[26] IBM telelogic rhapsody, 2008. http ://modeling.telelogic.com/- products/rhapsody/-software/developer/index.cfm.

[27] A. W. Brown, D. J. Carney, E. J. Morris, D. B. Smith, and P. F. Zarrella : *Principles of CASE Tool Integration*. New York : Oxford University Press, 1994.

[28] J. Sztipanovits, and G. Karsai : *Model-Integrated Computing*. Computer, Apr. 1997, pp. 110-112.

[29] J. Greenfield, K. Short, S. Cook, and S. Kent : *Software Factories*. Addison-Wesley, 2004.

[30] Object Management Group, Inc. Meta-Object Facility (MOF) 2.0 Specification, 2004-10-15.

[31] Object Management Group : *MDA guide version 1.1*, http ://www.omg.org/mda/, June 2003.

[32] Roger S. Scowen : *Extended BNF-a generic base standard*. In Software Engineering Standards Symposium, 1993.

[33] Fuentes-Fernandez, Lidia and Vallecillo-Moreno, Antonio : *An Introduction to UML Profiles*. The European Journal for the Informatics Professional, 5(2), April 2004.

[34] J. Hugues, B. Zalila, L. Pautet, and F. Kordon : *From the prototype to the final embedded system using the Ocarina AADL tool suite*. ACM Transactions on Embedded Computing Systems (TECS) , ACM Press, New York, USA.

[35] B. Zalila : *Configuration et déploiement d'applications temps-réel réparties embarquées à l'aide d'un langage de description d'architecture*. Thèse de doctorat, École Nationale Supérieure des Télécommunications, nov 2008.

[36] Society of Automotive Engineer : Architecture analysis & design language (aadl) as5506, 2006.

[37] Object Management Group, *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*, November 2007, OMG document number : formal/2007-11-02.

[38] Jack Herrington : *Code Generation in Action*, Manning Publications Co., Greenwich, 2003.

[39] IBM, http ://www.ibm.com/developerworks/rational/library/apr05/brown/index.html, Avril 2005.

[40] Verro D., Pataricza A. : *Generic and Meta-Transformations for Model Transformation Engineering*. In UML 2004, pp. 290-304 (2004).

[41] van Amstel M.F., Lange C.F.J., van den Brand M.G.J. : *Metrics for analyzing the quality of model transformations*. Proceedings of the 12th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Paphos, Cyprus (July 2008) 41-51.

[42] S. Sendall and W. Kozaczynski : *Model Transformation the Heart and Soul of Model-Driven Software Development*. IEEE Software 20(5), 4245, 2003.

[43] Boehm B.W., Brown, J.R. Kaspar, H. Lipow, M. Macleod, G.J. Merrit, M.J. : *Characteristics of Software Quality*. North-Holland (1978)

[44] Mohagheghi P., Dehlen V. : *An overview of quality frameworks in model-driven engineering and observations on transformation quality*. In : Workshop on Quality in Modeling at MODELS 2007. (September 2007)

[45] Matteo Bordin and Tullio Vardanega : *Real-time Java from an Automated Code Generation Perspective*. In JTRES 07 : Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems, pages 63-72, New York, NY, USA, 2007. ACM.

[46] Object Management Group, Inc., editor : UML 2 OCL (*Final Adopted specification*). http ://www.omg.org/cgi-bin/doc ?ptc/2003-10-14, October 2003.

[47] Object Management Group, *A UML profile for MARTE (version Beta 2)*, June 2008, OMG document number : ptc/2008-06-09.

[48] W. El Hajj Chehade, A. Radermacher, A. Cuccuru, S. Gérard and F. Terrier : *Automating the generation of platform specific models*, Fourth IEEE international workshop on UML and AADL. June 2nd, 2009 Potsdam, Germany.

[49] W. El Hajj Chehade, A. Radermacher, S. Gérard and F. Terrier : *Detailed Real-time Software Platform Modeling*, 17th Asia Pacific Software Engineering Conference. Sydney, Australia, December 2010.

[50] W. El Hajj Chehade, A. Radermacher, F. Terrier, B. Selic and S. Gérard : *A Model-Driven Framework for the Development of Portable Real-time Embedded Systems*. 16th IEEE International Conference on Engineering of Complex Computer Systems. Las Vegas, USA, April 2011.

[51] The Open Group Base Specifications : *Portable Operating System Interface(POSIX)*, ANSI/IEEE Std 1003.1, 2004.

[52] G. Bollela, J. Gosling, B. Brosgol, P. Dibble, S. Furr, M. Turnbull : *The Real-Time Specification for Java*, Addison-Wesley, 2000.

[53] Object Management Group : *UML profile for Schedulability, Performance and Time*, Object Management Group, inc., 2005. OMG document :formal/05-01-02.

[54] A. Lanusse, S. Gérard, F. Terrier : *Real-time Modelling with UML : The ACCORD Approach*. In Proceedings of the UML'98, Springer Verlag LNSC 1618.

[55] Sebastien GERARD : *Modélisation UML exécutable pour les systèmes embarqués de l'automobile*. Thèse de doctorat, Université d'Evry, 2000.

[56] A. Radermacher, A. Cuccuru, S Gerard ,F Terrier : *Generating Execution Infrastructures for Component-oriented Specifications With a Model Driven Toolchain*, in proceedings of the 8th International Conference on Generative Programming and Component Engineering (GPCE 2009), pp. 127-136, Denver, Colorado, October 4-5, 2009.

[57] Object Management Group, *Concrete Syntax for a UML Action Language, V0.11*, 23 August 2010, OMG document number : ad/2010-08-11.

[58] J. D. Mooney : *Bringing Portability to the Software Process*, Tech. rept. 1. West Virginia University, 1997.

[59] P. Devanbu, S. Karstu, W. Melo, W. Thomas : *Analytical and Empirical Evaluation of Software Reuse Metrics*, in Proceedings of the 18th IEEE international conference on software engineering, pages 189-199, 1996.

[60] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. : *N degrees of separation : Multidimensional separation of concerns*, in Proceedings of the 21st International Conference on Software Engineering (ICSE 99), 107-119, May 1999.

[61] Raphaël Marvie, Laurence Duchien et Mireille Blay-Fornarino : *Les Plate-formes d'exécution et l'IDM*, chapitre 4, pages 71-86. Numéro ISBN : 2-7462-1213-7. Hermes, January 2006.

[62] Edward A. Lee : *Overview of the Ptolemy Project*. Technical Memorandum UCB/ERL M01/11 March 6, 2001

[63] S.J. Mellor, M.J. Balcer : *Executable UML : A Foundation for Model-Driven Architecture*, Addison Wesley, ISBN 0-201-74804-5, 2002.

[64] Pathfinder solutions : http ://pathfindermda.com/

[65] http ://www.mentor.com/products/sm/model_development/bridgepoint/

[66] T. Schattkowsky, W. Mueller, A. Rettberg : *A Model-Based Approach for Executable Specifications and Reconfigurable Hardware*, in Proc. of the Design, Automation and Test in Europe (DATE), Nov 2005, pp. 692-697.

[67] Wehrmeister, M.A. : *An Aspect-Oriented Model-Driven Engineering Approach for Distributed Embedded Real-Time Systems*, PhD Thesis, Faculty of Electrical Engineering, Computer Science and Mathematics, Department of Computer Science, Paderborn, September 2009.

[68] P. Kukkala, J. Riihimâki, M. Hämäläinen and K. Kronlöf : *UML 2.0 Profile for Embedded System Design*, Automation and Test in Europe Conference (DATE 2005), pp. 710-715, March 2005.

[69] T. Arpinen, M. Setala, E. Salminen, M. Hannikainen and T. D. Hamalainen : *Modeling embedded software platform with a UML profile*. In Engenio Villar, editor : Forum on Specification and Design Languages (FDL'07), Barcelona, Spain, September 2007.

[70] T. Kangas et al. : *UML-Based Multi-Processor SoC Design Framework*. ACM TECS, vol. 5, no. 2, pp. 281-320, May 2006.

[71] D. C. Schmidt : *ACE : an Object-Oriented Framework for Developing Distributed Applications*. In Proceedings of the 6th USENIX C++ Technical Conference, (Cambridge, Massachusetts), USENIX Association, April 1994.

[72] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio : *A SoC design methodology involving a UML 2.0 profile for Systemc*. In Proc. of the conference on Design, Automation and Test in Europe, pages 704-709, Munich, Germany, March 2005. IEEE Computer Society.

[73] E. Riccobene,P. Scandurra, S. Bocchio, A. Rosti, L. Lavazza, L. Mantellini : *SystemC/C-Based Model-Driven Design for Embedded Systems*. ACM Transactions on Embedded Computing Systems (TECS), 8(4) 2009.

[74] Ali Koudri : *Processus de Conception Conjointe Logiciel Matériel Dirigés par les Modèles*. Thèse de doctora, Université des Sciences et Technologies de Lille, Juillet 2010.

[75] Frédéric THOMAS : *Contribution à la prise en compte des plates-formes logicielles d'exécution dans une ingénierie générative dirigée par les modèles*. Thèse de doctorat, Université d'Evry, 2008.

[76] Metropolis Project Team : The metropolis meta model version 0.4. Rapport technique UCB/ERL M04/38, University of California, Berkeley, September 2004.

[77] Rong Chen, Marco Sgroi, Grant Martin, Luciano Lavagno, Alberto Sangiovanni-Vincentelli et Jan Rabaey : *UML and Platform-Based Design*, volume UML for Real : Design of Embedded Real-Time Systems, chapitre 5. Kluwer Academic Publishers, May 2003.

[78] Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Guang Yang, Haibo Zeng et Qi Zhu : :*A next-generation design framework for platform based design*. In Conference on Using Hardware Design and Verification Languages (DVCon), February 2007.

[79] Matthias BRUN : *Contribution à la considération explicite des plates-formes d'exécution logicielles lors d'un processus de déploiement d'application*. Thèse de doctorat, Université De Nantes, 2010.

[80] Tivadar Szemethy : *Domain-Specific Models, Model Analysis, Model Transformations*. Thèse de doctorat, Université de Vanderbilt, Nashville, Tennessee, USA, May 2006.

[81] Frédéric LOIRET : *Tinap : Modèle et infrastructure d'exécution orienté composant pour applications multi-tâches à contraintes temps réel souples et embarquées*. Thèse de doctorat, Université des Sciences et technologies de Lille, Mai 2008.

[82] E. G. Benowitz and A. F. Niessner : *A Patterns Catalog for RTSJ Software Designs*. Lecture notes in Computer Science, 2003.

[83] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek : *Real-Time Java Scoped Memory :Design Patterns and Semantics*. In Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04), pages 101-110, 2004.

[84] Greg Bollella, Tim Canham, Vanessa Carson, Virgil Champlin, Daniel Dvorak, Brian Giovannoni, Mark Indictor, Kenny Meyer, Alex Murray, and Kirk Reinholtz : *Programming with non-heap memory in the real time specification for java*. In OOPSLA'03 : Companion of the 18th annual ACM SIGPLAN conference on Object-oriented

programming, systems, languages, and applications, pages 361-369, New York, NY, USA, 2003. ACM.

[85] A. Charfi, C. Mraidha, P. Boulet, S. Gérard and F. Terrier : *Toward optimized code generation through model-based optimization*. Proc, Design Automation and Test in Europe, DATE 2010., in press.

Résumé

Titre- Contribution au déploiement multiplateforme d'applications multitâches par la modélisation comportementales haut niveau des services d'exécution.

Résumé- Face à la complexité des logiciels multitâches, liée aux contextes économique et concurrentiel très pressants, la portabilité des applications et la réutilisabilité des processus de déploiement sont devenues un enjeu majeur. L'ingénierie dirigée par les modèles est une approche qui aspire répondre à ces besoins en séparant les préoccupations fonctionnelles des systèmes multitâches de leurs préoccupations techniques, tout en maintenant la relation entre eux. En pratique, cela se concrétise par des transformations de modèles capables de spécialiser les modèles pour des plates-formes cibles. Actuellement, les préoccupations spécifiques à ces plates-formes sont décrites implicitement dans les transformations eux même. Par conséquence, ces transformations ne sont pas réutilisables et ne permettent pas de répondre aux besoins hétérogènes et évolutifs qui caractérisent les systèmes multitâches. Notre objectif est alors d'appliquer le principe de séparation de préoccupation au niveau même de la transformation des modèles, une démarche qui garantie la portabilité des modèles et la réutilisabilité des processus de transformation.

Pour cela, cette étude propose premièrement une modélisation comportementale détaillée des plates-formes d'exécutions logicielles. Cette modélisation permet d'extraire les préoccupations spécifiques à une plate-forme de la transformation de modèle et les capturer dans un modèle détaillé indépendant et réutilisable. Dans un second temps, en se basant sur ces modèles, elle présente un processus générique de développement des systèmes concurrents multitâches. L'originalité de cette approche réside dans une véritable séparation des préoccupations entre trois acteurs à savoir le développeur des chaînes de transformation, qui spécifient une transformation de modèle générique, les fournisseurs des plates-formes qui fournissent des modèles détaillés de leurs plates-formes et le concepteur des applications multitâche qui modélise le système. A la fin de cette étude, une évaluation de cette approche permet de montrer une réduction dans le coût de déploiement des applications sur plusieurs plates-formes sans impliquer un surcoût de performance.

Mot clés- Ingénierie Dirigée par les Modèles, Modèle de Plate-forme, Transformation de modèle, Exécution multitâche, UML, MARTE

Abstract

Title- Contrubution to Multiplatform Deployement of Multitasking Applications by High-Level Execution Services Behavioral Modeling

Abstract- Given the complexity of multitasked software, linked to very pressing economic and competitive contexts, application portability and deployment process reusability has become a major issue. The model driven engineering is an approach that aspires to meet these needs by separating functional concerns of multitasking systems from their technical concerns, while maintaining the relationship between them. In practice, this takes the form of model transformations that specializes models for target platforms. Currently, concerns specific to these platforms are described implicitly in the transformations themselves. Consequently, these transformations are not reusable and do not meet the heterogeneous evolutionary needs that characterize multitasking systems. Our objective is then to apply the principle of separation of concern even at the level of transformation models, an approach that guarantees portability and reusability of models transformation process.

To do this, this study provides first a detailed behavioral modeling of software execution platform. This modeling allows to extract specific concerns from model transformation and to capture them in a detailed platform model independent and reusable. In a second step, based on these models, it presents a generic process for developing concurrent systems. The originality of this approach is a true separation of concerns between three actors : the developer of transformation tool, who specifies a generic model transformation, platform providers that provide detailed models of their platforms and the multitasked system designer that models the system. At the end of this study, an evaluation of this approach shows a reduction in the cost of deploying applications on multiple platforms without incurring an additional cost of performance.

Keywords Model Driven Engeneering, Platform model, Model transformation, Multitasking execution, UML, MARTE