Approche fondée sur les modèles pour java temps-réel

Chaker Nakhli

No ENSC-XXXX

THESE DE DOCTORAT DE
L'ECOLE NORMALE SUPERIEURE DE CACHAN

Présentée par
Monsieur CHAKER NAKHLI

pour obtenir le grade de
DOCTEUR DE L'ECOLE NORMALE SUPERIEURE DE CACHAN

Domaine :
INFORMATIQUE

Sujet de la thèse :

APPROCHE FONDEE SUR LES MODELES POUR JAVA TEMPS-REEL

Thèse présentée et soutenue le 2 septembre 2005 devant le jury composé de :

Laurent Fribourg	Directeur de Recherche CNRS	Président
Philippe Clauss	Professeur	Rapporteur
Klaus Havelund	Directeur de Recherche NASA Ames	Rapporteur
Jean-Pierre Talpin	Professeur	Examinateur
Antoine Petit	Professeur	Directeur de Thèse
Joseph Sifakis	Directeur de Recherche CNRS	Co-directeur de Thèse

Laboratoire VERIMAG - UMR 5104
2, avenue de Vignates, 38610 GIERES (FRANCE)

Remerciements

Je tiens à remercier Joseph Sifakis pour m'avoir accueilli dans son laboratoire et pour m'avoir encadré durant ma thèse. Je lui sais gré d'avoir su me consacrer une partie de temps, en particulier aux moments opportuns, malgré ses nombreuses activités.

Je voudrais remercier les membres de mon jury, Philippe Clauss et Klaus Havelund qui ont accepté d'être rapporteurs, Antoine Petit, Laurent Fribourg et Jean-Pierre Talpin.

Je tiens également à remercier Sergio Yovine pour ses encouragements et pour les innombrables discussions scientifiques et techniques que nous avons eues. Son aide et sa contribution ont été précieuses dans ce travail. Je voudrai aussi le remercier chaleureusement de m'avoir soutenu sans réserve dans les moments difficiles. J'ai eu beaucoup de plaisir à le connaître et à travailler avec lui.

Je remercie Ramzi Ben Salah pour toutes les discussions très enrichissantes, scientifiques ou non, que nous avons eues, d'avoir relu mon manuscrit et organisé le pot de ma soutenance. Je suis extrêmement fier de le compter parmi mes amis.

Un grand merci à Sahbi Bahroun, Kaïs Tmar, Chiheb Kossentini et Amine Dhia pour leur amitié, leur soutien et d'avoir toujours été patients face à mon indisponibilité et mes sauts d'humeur.

Finalement, je remercie ma petite famille : mon père Mohammed, ma sœur Leïla et ma mère Sondes. Sans vous, votre amour et votre soutien, je ne serais pas là à écrire ces lignes. Merci papa d'avoir toujours cru en moi. Merci Leïla pour ta tendresse et tes encouragements. Merci Maman de m'avoir soutenu corps et âme pendant ces années, cette thèse est aussi la tienne.

Table des matières

Chapitre 1

Introduction

1.1 Systèmes temps-réel

1.1.1 Définition, besoins et défis

Les progrès technologiques en informatique aujourd'hui font que non seulement la plupart des objets qui nous entourent sont de plus en plus dotés d' "intelligence", mais voient aussi leurs fonctionnalités constamment améliorées et surchargées par de nouveaux emplois et capacités. Les exemples ne manquent pas : les téléphones mobiles enregistrent des filmes vidéo, les machines à laver se connectent sur Internet et les voitures proposent le meilleur itinéraire à suivre tout en corrigeant automatiquement le freinage et la trajectoire.

Cette avancée technologique est le fruit de l'intégration de *composants embarqués* logiciels et matériels. Ces composants sont souvent réactifs, amenés à interagir et communiquer avec d'autres systèmes et leurs environnements. L'exécution d'un tel composant est liée à la dynamique des stimuli auxquels ce dernier doit réagir et est souvent contrainte par des échéances et des exigences temporelles nécessaires au bon fonctionnement du système mais aussi à la qualité de service. Il s'agit alors de *systèmes temps-réel* où faillir à ces contraintes donne lieu à une dégradation importante des performances, voire la mise en péril du système et son environnement. Lorsque l'échec à une contrainte temporelle est considéré comme défaillance critique du système portant atteinte à son intégrité, le système est dit *temps-réel dur*. Dans le cas où l'effet d'une telle défaillance se réduit à une dégradation de la qualité de service sans forcément empêcher le bon fonctionnement global, le système est dit *temps-réel mou*.

Le développement des systèmes temps-réel est non seulement un enjeu économique important, mais aussi un domaine de recherche particulièrement actif. L'élaboration des systèmes temps-réel soulève des défis importants et amène à répondre à de nombreux besoins :

- Diminuer les coûts de développement et assurer un temps d'accès au marché[1] minimal.
- S'adapter et répondre à des environnements de plus en plus rapides et complexes.

[1] En anglais : *time-to-market*

- Combiner des activités temps-réel dur et mou. Ceci conduit à concevoir des politiques d'ordonnancement plus complexes afin de satisfaire des exigences plus subtiles que le respect strict des échéances. Par exemple l'optimisation des temps de réponses globaux et la qualité de service.
- Concevoir des systèmes complexes dans leurs fonctionnalités mais aussi capables de se mettre à jour, se configurer, se réparer et s'adapter automatiquement ou avec peu d'intervention, et à distance.
- Obtenir des systèmes fiables : assurer la sécurité, sûreté, disponibilité *etc.* La nature dynamique et l'hétérogénéité des systèmes rend les techniques classiques d'évaluation de fiabilité obsolètes.

Le développeur des systèmes temps-réel ne dispose pas d'outils, de méthodologies et techniques unifiées afin d'accompagner toutes les étapes de développement. Une modélisation correcte des systèmes temps-réel permettrait l'analyse formelle et l'étude et la conception de contrôleurs adéquats.

1.1.2 Distinctions entre logiciel et implémentation

Il existe des différences fondamentales entre le logiciel et le système temps-réel qui l'implémente. Le logiciel est écrit dans des langages tel que Ada 95, C, Java où le développeur fait différentes sortes d'abstractions sur le comportement des composants et leurs interactions. Le logiciel est une description *haut niveau* du système. Une multitude de détails sont omis, comme la concurrence, la gestion des ressources, l'ordonnancement, le mode d'exécution, ainsi que les caractéristiques propres à la plate-forme de déploiement. Le développement du logiciel s'accompagne donc d'un certain nombre d'hypothèses et d'abstractions nécessaires à la maîtrise de la complexité de l'ensemble et qui contribuent à simplifier la description. Dans ce sens, le logiciel est généralement conçu pour être très peu dépendant de l'architecture de la plate-forme cible. Le développeur admet, de plus, certaines propriétés comme l'atomicité, la concurrence, une communication instantanée et l'absence de délais. Le logiciel forme par le biais des règles de la sémantique opérationnelle du langage une machine abstraite réactive dont les actions sont déclenchées par les événements externes. Même si le code du logiciel contient des références au temps à travers des garde-fous ou des temps d'expiration, le temps reste totalement externe à cette machine. Les évènements temporels sont traités de même que tout autre évènement externe.

La différence fondamentale entre le logiciel est l'implémentation réside dans le fait que le logiciel est immatériel et donc dépourvu de toutes informations temporelles. L'implémentation, quant à elle, correspond au comportement décrit par le logiciel mais tournant à une vitesse précise dictée par la plate-forme et est couplée avec un environnement ayant une dynamique particulière. De plus, l'implémentation ne respecte pas forcément les abstractions du logiciel. Par exemple, les calculs ne sont pas instantanés, le nombre de processeurs est fixé limitant

ainsi le parallélisme, l'indéterminisme inhérent à certains calculs indépendants est éliminé *etc.*

Il est essentiel de disposer d'une méthodologie rigoureuse, c'est à dire fondée sur l'utilisation de modèles formels, pour que le passage du logiciel à son implémentation se fasse en respectant certaines conditions importantes. Il faut préserver les propriétés vérifiées lors de la conception fonctionnelle. D'autre part, il faut prendre en compte la dynamique de l'implémentation pour la simulation et la vérification de celle-ci. L'analyse d'un système temps-réel nécessite l'utilisation de modèles temporisés, qui puissent décrire à la fois l'effet des actions du logiciel et la progression du temps. C'est dans cette optique que ce travail présente une approche fondée sur les modèles dans le cadre Java temps-réel.

1.1.3 Temps-réel synchrone et asynchrone

Les pratiques actuelles pour la conception des systèmes temps-réel s'inscrivent dans deux paradigmes bien définis : synchrone et asynchrone.

Dans le paradigme synchrone [Hal98, BB91, Hal93], le système propage les effets des stimuli de l'environnement à travers ses différents composants suivant un ordre de causalité. Le système réagit à l'environnement en effectuant un pas global de calcul où différents composants prennent part. L'*hypothèse synchrone* stipule que la réaction du système est assez rapide pour répondre *à temps* à *tout* stimulus de l'environnement. En d'autres termes, tout les calculs sont finis avant qu'un nouveau flot de données arrive et enclenche un nouveau pas de calcul. La réactivité et la précision du système sont donc bornées par la durée du pas.

Les programmes synchrones sont implémentés en utilisant un ordonnancement simple respectant l'ordre de causalité entre les composants. Ceci permet des implantations efficaces qui ne nécessitent pas de systèmes d'exploitation. Pendant un pas de calcul, chaque tâche reçoit un fragment de temps de calcul. L'implémentation résulte typiquement en une seule tâche constituée d'une boucle lecture/traitement/écriture.

Le paradigme synchrone est adopté par différents langages, dits langages synchrones, ayant acquis un large succès en industrie [BCE+03] comme le langage impératif Esterel [BG92] ou les langages de flot de données Lustre [HCRP91] et Signal [LGLL91]. Ils sont utilisés, entre autres, pour les applications multimédia, de traitement de signal et de contrôle automatique.

Dans le paradigme asynchrone, il n'y pas de pas global d'exécution. Hormis les contraintes imposées par l'interaction éventuelle des composants, ces derniers sont indépendants. C'est la politique d'ordonnancement qui détermine, à certains états du système, le composant à exécuter. Le composant élu s'accapare alors des ressources de calcul jusqu'à ce qu'il s'arrête ou perde la main, selon les critères d'ordonnancement. L'ordonnanceur remplit une fonction centrale dans un système asynchrone. Son rôle est de restreindre l'accès aux ressources, en engageant des actions du système, dans le but de satisfaire les contraintes fonctionnelles et temporelles.

La plupart des langages généralistes comme C, Ada 95 et Java sont asynchrones. La notion de concurrence n'est pas toujours apparente au niveau du langage. Pour exprimer le parallélisme et la synchronisation, il faut souvent se baser sur des librairies du système d'exploitation comme *Win32 Threads* [Har97] ou *POSIX Threads* [Ins95]. D'autres langages comme, Ada et Java, permettent d'exprimer la concurrence au niveau du programme et offrent une certaine indépendance de la plate-forme d'exécution.

Deux approches majeures abordent l'ordonnancement des systèmes temps-réel asynchrones. La première approche est la théorie d'ordonnancement temps-réel. Elle propose des algorithmes basés sur des modèles analytiques simples qui proposent des critères suffisants pour l'ordonnançabilité. La deuxième approche est l'approche dirigée par les modèles.

La théorie d'ordonnancement temps-réel consiste à vérifier une condition suffisante appelée *critère d'ordonnancement* sous laquelle l'ordonnancement est dit *faisable*, c'est à dire les exigences temporelles sont satisfaites. Ces méthodes ont été appliquées avec succès au développement des systèmes temps-réel et certaines de ces politiques sont aujourd'hui supportées sur un grand nombre de systèmes d'exploitation temps-réel. Une synthèse et vue d'ensemble sont présentées en [KRP$^+$93, ABD$^+$95, But97]. Dans le cadre de ces méthodes, les critères d'ordonnancement exigent des conditions strictes sur les propriétés temporelles des tâches et leurs interactions, sous lesquelles s'applique la faisabilité de l'ordonnancement. Ces conditions concernent les périodes, temps d'exécution, échéances, taux d'utilisation du processeur *etc.* Les politiques RMA (*Rate Monotonic Analysis*) [LL73, SKG91] et EDF (*Earliest Deadline First*) [LL73, SRS98] sont des exemples de cette approche d'analyse.

La théorie d'ordonnancement temps-réel bénéficie de certains avantages. En effet, les critères d'ordonnancements reposent sur des formules analytiques simples, faciles à mettre en oeuvre et à vérifier. Dans la majorité des cas, l'ordonnanceur est basé sur des critères statiques et peu dépendants de l'état du système. Il est par conséquent simple à implanter et son exécution n'introduit pas de délais considérables à l'exécution du système.

Cependant, ces méthodes et algorithmes d'ordonnancement restent fortement limités et restrictifs. Outre le fait que ces méthodes ne sont applicables que pour une structure spécifique de l'application, elles sont strictement réduites à garantir les contraintes temporelles d'échéance pour chaque tâche. Elles ne permettent pas d'exprimer d'autres types de contraintes. Par exemple, la plupart de ces méthodes ne prennent pas en compte les mécanismes de communication entre les tâches ou le partage de ressources. La gestion des ressources partagées, si le partage des ressources est autorisé, se fait par des politiques simples comme PIP (*Priority Inheritance Protocol*) [SRL90] ou PCP (*Priority Ceiling Protocol*) [RSLR95]. Il n'est pas possible d'avoir des politiques plus complexes ou adaptée à chaque type de ressources en fonction des besoins de l'application. A cause du manque d'expressivité de ces méthodes, il n'est pas possible d'exprimer des contraintes intimement liées à la nature et comportement de l'application, tel que les contraintes de Qualité de Service (*QoS*). Ces dernières sont souvent

importantes car elles permettent d'optimiser et/ou garantir des propriétés substantielles des systèmes embarqués, comme la consommation d'énergie ou l'espace mémoire utilisé.

L'approche dirigée par les modèles est une approche alternative qui a pour but de pallier les déficiences des méthodes d'ordonnancement temps-réel classiques. L'analyse d'ordonnancement du système est basé sur un modèle temporisé qui décrit le comportement, les interactions, contraintes temporelles *etc.* de l'application. L'approche consiste à : soit *vérifier l'ordonnançabilité du système*, étant donné une politique d'ordonnancement particulière ; soit *synthétiser un ordonnanceur*, en se basant sur modèle, qui vérifie par construction les contraintes d'ordonnancement exigées.

La vérification du modèle du système associé à une politique d'ordonnancement représente des avantages intéressants. Le modèle du système ordonnancé décrit à la fois le comportement fonctionnel et non-fonctionnel du système et permet ainsi de mener l'analyse conjointe des deux types de propriétés. Le déterminisme introduit par les choix de la politique d'ordonnancement réduit l'espace d'état du modèle initial du système permettant ainsi de contenir l'explosion combinatoire. Cela permet aussi d'éviter les faux contre-exemples qui émanerait de la vérification du modèle libre de l'application sans prendre en compte l'ordonnancement.

La synthèse d'ordonnanceur [MTdR96, KLP+98, AGS00, AGS02, AFM+04] consiste à extraire, à partir du modèle de l'application, un ordonnanceur qui répond aux exigences et contraintes imposées au système. Cette démarche ne considère pas de politiques d'ordonnancement particulières, elle produit un ordonnanceur spécifique. La construction d'ordonnanceur à partir du modèle de l'application constitue le cadre le plus libre et le plus expressif pour ordonnancer l'application. Cette méthode souffre malheureusement de l'explosion combinatoire due à l'énumération exhaustive de l'espace d'état. Malgré ce problème, la synthèse d'ordonnanceur a pu être menée sur des exemples de taille industrielle [NY01, GV03]. Les outils de vérification comme SMV [McM93], Kronos [Yov97] et Uppaal [BLL+95] offrent un support indispensable et efficace pour la synthèse.

L'approche dirigée par les modèles forme un cadre unifié pour aborder l'analyse des propriétés fonctionnelle et non-fonctionnelle des systèmes. Elle offre le moyen d'exprimer des contraintes plus riches que le strict respect des échéances temps-réel, auxquelles sont réduites les théories classiques d'ordonnancement. Dans cette approche, la modélisation du système joue, par définition, un rôle central. Il est essentiel de construire des modèles fidèles et cohérents. C'est dans cette direction que le présent travail se place en proposant un cadre, une méthodologie et un outil pour la modélisation des systèmes temps-réel écrits en Java.

1.2 Etat de l'art

Le travail présenté dans cette thèse s'inscrit au carrefour de trois domaines et approches importants dans l'élaboration des systèmes temps-réel :

- Approches fondées sur les modèles
- Analyse et vérification des logiciels
- Langages et profils de programmation temps-réel

Les sections qui suivent définissent ces domaines et présentent quelques travaux et méthodologies qui s'y rapportent.

1.2.1 Approches fondées sur les modèles

Dans ces approches, le modèle constitue l'élément central du développement et de la conception d'un système. Utilisé à différents niveaux d'abstraction et en cours de développement, il est essentiel à la vérification des choix de conception, surtout non-fonctionnels. Ces choix expriment des décisions d'implémentation importantes, telles que la décomposition en tâches, l'ordonnancement et la gestion de la mémoire.

On distingue deux types d'approches fondées sur les modèles :
- les approches utilisant des langages de modélisation systèmes
- les approches étendant des langages de programmation existants pour les doter avec des concepts et des constructions adéquates pour la modélisation des systèmes.

Les formalismes suivants représentent la première approche :

UML [2] est un standard de modélisation [RJB98] largement utilisé dans la modélisation des logiciels. Avec les profils destinés à l'ordonnancement et à la qualité de service [Obj05a, Obj05b], le standard s'oriente vers la spécification des systèmes temps-réel. Cependant, cette méthodologie souffre de l'imprécision de sa sémantique, notamment pour exprimer les aspects temporels. Le projet *Omega* [GOO03, OGO05] définit un sous-ensemble de UML, étendu par des constructions temporelles, et lui donne une fondation formelle pour la spécification des systèmes temps-réel. Plusieurs outils commerciaux supportent la modélisation UML pour le temps-réel, tels que *Rational Rose* et *I-Logix Rhapsody*.

Matlab / Simulink est un ensemble d'outils pour la modélisation et la simulation des systèmes synchrones (notamment pour le contrôle commande) continus, échantillonnés et hybrides. Simulink est basé sur l'hypothèse synchrone : les temps de calculs sont considérés nuls. Il fournit un environnement d'édition graphique permettant une modélisation intuitive des systèmes.

[2] *Unified Modeling Langage*

Giotto fournit un langage de description d'architecture (indépendant de la plate-forme), un compilateur et un exécutif (dépendants de la plate-forme) pour la spécification et implémentation des systèmes temps-réel. Le langage permet de décrire l'architecture du système par le biais de tâches périodiques et ports de communication. La spécification est annotée par des informations sur la plate-forme cible afin de guider/restreindre le compilateur. Le code associé aux composants déclarés dans Giotto est fourni par l'utilisateur (essentiellement du C).

Ptolemy est un outil de modélisation et simulation des systèmes embarqués. Il utilise une conception basée sur les *acteurs* : ce sont des entités auxquelles est associé un comportement et qui sont liées par des interactions. Le tout forme une description de l'architecture du système étudié. Cette architecture est liée à un *modèle de calcul* qui renseigne sur l'ordonnancement et la nature des communications. Le modèle de calcul est choisi parmi ceux prédéfinis ou fourni par l'utilisateur par extension de l'outil (à code ouvert). Ptolemy dispose d'un éditeur graphique et de différents modules de simulation.

Parmis les principaux formalismes de modélisation de systèmes par extension de langage de programmation existants, on a :

SystemC [Ope03] est un formalisme pour décrire les architectures matérielles basées sur le langage C++. SystemC enrichit C++ avec des concepts comme les portes, les modules, les processus et les signaux. Un noyau de simulation synchrone dirigé par les évènements est fourni afin d'exécuter les modèles.

Le projet *Taxys* [BPPS00, BCP+01] présente une chaîne complète mettant en oeuvre une approche fondée sur les modèles pour la modélisation, l'analyse et l'implémentation des systèmes temps-réel embarqués.

Taxys étend le langage de programation synchrone Esterel [BG92] par des annotations temporelles qui définissent les temps d'exécution et les échéances. L'objectif de l'analyse est de vérifier, pour une application donnée, le respect des échéances et la validité de l'hypothèse synchrone.

La chaîne expérimentale du projet, composée par le compilateur SAXO-RT [WBC+00] et l'outil de vérification Kronos, a été appliquée avec succès sur des systèmes de télécommunication. Cette méthodologie a été aussi réutilisée pour un système de contrôle de véhicules [TY01].

La méthodologie et l'outil *JediTool* que nous présentons dans cette thèse sont dans la ligne directe et dans l'esprit du projet Taxys. Notre contribution concerne les systèmes temps-réel dans le cadre de la technologie Java. Il est à noter que dans le cadre de Taxys on a une notion d'état claire (*built-in*) dans le langage, des pas de calculs sans effets de bord et une exécution déterministe du système. Dans le cas de Java, il est nécessaire de gérer le dynamisme et le

comportement asynchrone, et établir une notion d'observabilité afin de définir les états et pas d'exécution du système modélisé.

1.2.2 Analyse et vérification des logiciels

La vérification formelle consiste à comparer deux types de description modulo une relation de satisfaction. Elle est largement utilisée aujourd'hui pour la validation des circuits électroniques et des protocoles de communication pour lesquels il est facile de construire des modèles analysables.

Pour les logiciels écrits dans des langages de programmation généraux, il est difficile de construire des modèles analysables qui prennent en compte le comportement lors de l'exécution.

Les outils principaux de vérification dédiés au langage Java sont :

Java Path Finder (JPF) [VHBP00, Rob05] est une machine virtuelle spécialement conçue pour la vérification par exploration de l'espace d'état des programmes Java. La vérification s'opère directement sur le *bytecode*[3] et permet de détecter les blocages et la violation d'invariants. JPF peut être paramétré par des heuristiques afin d'orienter l'exploration. Les invariants et heuristiques sont fournis par l'utilisateur sous forme de classes Java.

Bandera [CDH+00] se base sur des assertions (exprimées en *Bandera Specification Langage* [CDHR00]) insérées dans le code source Java afin d'extraire un modèle du programme. Ce modèle peut être traduit vers différents formats d'outils d'analyse, principalement Spin [Hol97], dSpin [DIS99] et NuSmv[CCGR00]. Les assertions sont traduites en LTL [Pnu77] ou CTL[CES86], selon l'outil d'analyse choisi. Bandera est limité par certains aspects dynamiques du langage, tel que l'analyse inter-méthode des pointeurs.

ESC/Java 2 [FLL+02] est basé sur un prouveur de théorèmes afin de vérifier des annotations de pré-conditions, post-conditions et invariants de boucles spécifiés dans le code source par des annotations exprimées en JML[CK04]. L'outil permet d'analyser certaines propriétés fonctionnelles, telles que les blocages et la corruption de données.

Ces outils ne prennent pas en charge les aspects temps-réel et les propriétés non fonctionnelles. Une contribution essentielle de notre approche, qui se distingue des travaux sus-mentionnés, est l'incorporation des choix d'implémentation dans les modèles. Les aspects non-fonctionnels, tels que les temps d'exécution, les contraintes d'échéance et les politiques d'ordonnancement sont associés au modèle du logiciel. Notre méthodologie est supportée par l'outil de modélisation et d'analyse *JediTool*.

[3] Format binaire produit par le compilateur Java (*cf.* paragraphe 2.1).

1.2.3 Langages et profils de programmation temps-réel

Plusieurs profils de programmation ont été définis pour le développement des systèmes temps-réel. Ils consistent en un ensemble de restrictions qui définissent un sous-ensemble d'un langage et/ou librairie donnés. Les profils temps-réel sont largement utilisés en industrie et sont souvent supportés par des outils de renforcement et de vérification. Le but des profils temps-réel est d'assurer des propriétés importantes lors du développement, telles que la prédictibilité en temps de calcul et mémoire, l'élimination des schémas d'erreurs classiques et permettre la vérification formelle.

Nous présentons quelques profils, utilisés en industrie, qui nous ont inspiré dans la définition du sous-ensemble Java que nous considérons :

Misra C est un guide composé de règles de développement pour le langage C publié en [MIS98]. Ces règles restreignent essentiellement : l'utilisation des pointeurs et tableaux, le flot de contrôle, les opérateurs arithmétiques, les structures et unions ainsi que les directives de pré-*prossessing* et l'utilisation des librairies standards du langage C. Plusieurs outils d'enforcement existent afin de détecter la violation des règles ; ces outils restent imprécis [Tec01, Cha02]. Il est à noter que seuls les programmes séquentiels sont concernés par ce porfil, la programmation concurrente n'est pas traitée.

Spark Ada [BB03] définit un sous-ensemble du langage Ada étendu par des annotations sous forme de commentaires spéciaux. Ces annotations servent principalement à la définition de pré-conditions, post-conditions et assertions sur les sous programmes. Le code est analysé afin d'extraire des conjectures vérifiées par des prouveurs de théorèmes. Parmi les restrictions adoptées par le profil Spark nous pouvons citer l'exclusion des constructions suivantes : les exceptions, les types génériques, les types pointeur ainsi que les instructions de saut (telles que `goto`). L'utilisation de Spark dans le processus de développement de trois applications critiques de taille réelle a été présentée en [Cha00]. Comme dans Misra C, ce profil n'adresse pas la programmation concurrente. De plus, le lien tardif et l'héritage multiple sont totalement interdits.

Embedded C++ [The02] définit d'un sous-ensemble du langage C++ et restreint ses librairies standards ; les restrictions sont inspirées de [PS91]. Le comité technique de Embedded C++ regroupe principalement des industriels japonais de semi-conducteurs, le profil est destiné à la programmation des nouvelles puces et micro-contrôleurs. Une première version de la grammaire du sous-ensemble du langage a été publiée mais le guide de programmation reste incomplet. Les restrictions adoptées comprennent les interdictions de : l'héritage multiple, les exceptions, les types génériques, la surcharge d'opérateurs, ainsi que toutes utilisations des librairies non conformes au profil. Le compilateur C++ de *Green Hills* [Gre04] supporte ce sous-ensemble. Embedded C++ ne fournit aucune information concernant la programmation concurrente.

Ravenscar Ada [DB98, BDV04] définit un patron d'architecture pour les applications temps-réel concurrentes écrites en Ada. Le but du profil est de faire correspondre les applications qui s'y conforment au modèle analytique de la politique d'ordonnancement RMA et gestion de ressources PCP. Ravenscar restreint l'usage de la librairie temps-réel Ada et réglemente la communication et la création d'objets partagés et des tâches. Essentiellement, les restrictions imposent un nombre fixe de tâches communicant à travers un nombre fixe d'objets partagés. Ce profil a été adapté au langage Java par le profil Ravenscar Java [KWK02, CW03] sur lequel est basé le projet *Expresso* (*cf.* paragraphe 1.3.1).

1.3 But et approche de la thèse

1.3.1 Contexte

Ce travail a été démarré dans le cadre du projet *RNTL-Expresso*[4] regroupant des partenaires du monde industriel et académique. L'objectif du projet Expresso était de définir une extension Java temps-réel et fournir un ensemble d'outils nécessaires à l'environnement de développement des systèmes temps-réel critiques susceptibles d'être certifiés au sens des standards tel que le DO178-B [Eur92]. Afin de pallier le manque de performance introduit par l'interprétation du code compilé, le code Java est compilé en code natif de la plate-forme de déploiement choisie. La machine virtuelle est remplacée par un exécutif installé sur le système d'exploitation hôte. Les membres partenaires de ce projet comprennent les industriels *Thales Systèmes Aéroportés* et *EDF* qui définissent les besoins et évaluent les résultats du produit, les sociétés de produits et services informatiques *Aonix* et *Silicomp RI* qui s'associent pour fabriquer la chaîne de compilation et l'exécutif Java, les laboratoires *INRIA/IRISA* et *Verimag* comme experts en techniques formelles qui complètent l'atelier logiciel avec les outils d'analyse temps-réel.

Nous avons livré au terme de ce projet une première version de notre outil *JediTool* pour l'extraction automatique de modèles temporisés à partir du logiciel applicatif accompagné d'informations complémentaires qui caractérisent les choix d'implantation. Cet outil met en œuvre et supporte la méthodologie présentée dans ce travail. *JediTool* est présenté au chapitre 5. Nous avons utilisé *JediTool* sur une application concrète afin de générer un modèle temporisé qui a été utilisé pour la synthèse et la génération du code de l'ordonnanceur. Grâce à la chaîne de compilation Expresso, nous avons porté avec succès l'application et l'ordonnanceur généré sur un système d'exploitation temps-réel. Cette application [KNY03] est présentée au chapitre 6. De plus, nous avons contribué par un outil d'analyse et transformation des programmes Java basés sur une allocation automatique (gérée par le ramasse-miettes) vers une allocation

[4] Réseau National des Technologies Logicielles. Projet financé par le Ministère de l'Economie des Finances et de l'Industrie.

par régions. Notre approche [GNYZ04] est également présentée au chapitre 6.

1.3.2 Objectif et démarche

Nous proposons une approche fondée sur les modèles pour l'analyse des systèmes temps-réel dans le cadre de la technologie Java. La figure 1.1 illustre le principe adopté pour l'extraction et l'analyse des modèles.

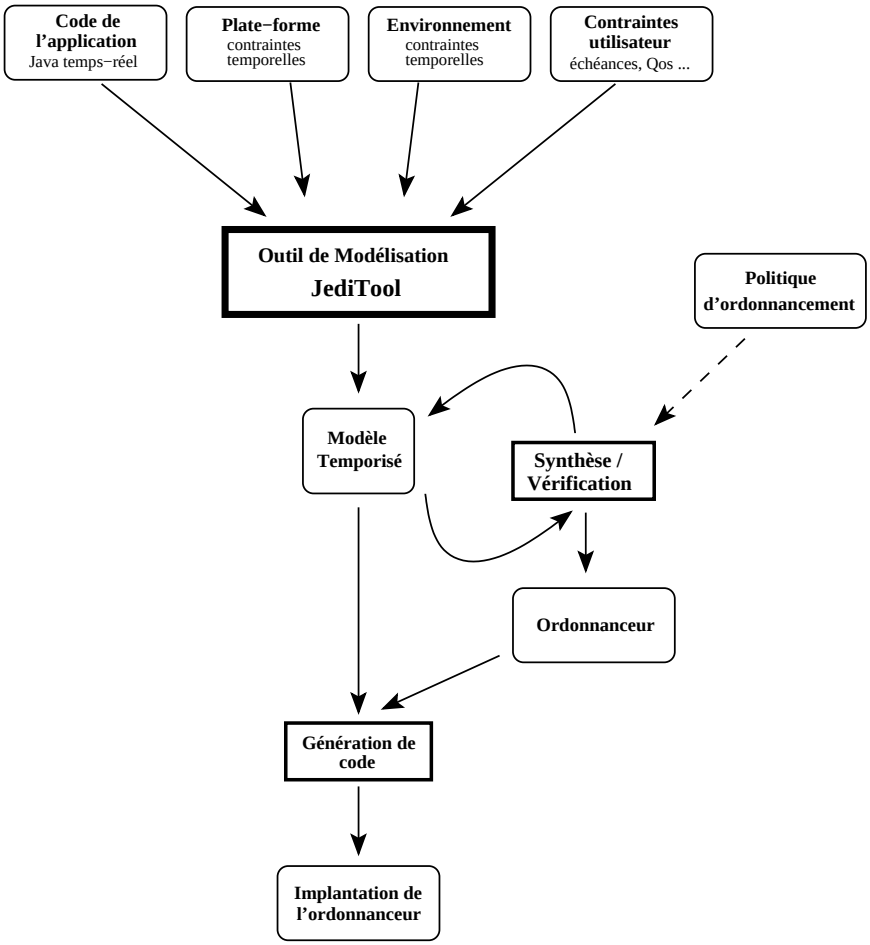


FIG. 1.1 – Schéma de principe de l'approche de modélisation.

Notre objectif est de construire un modèle temporisé qui représente fidèlement le comportement d'un système temps-réel. Le modèle est construit en augmentant la description fonctionnelle déterminée par le logiciel avec les contraintes non-fonctionnelles introduites par l'implantation. Le modèle obtenu est utilisé, soit pour la synthèse d'ordonnanceur, soit pour la vérification de l'ordonnançabilité du système pour une politique d'ordonnancement donnée. L'ordonnanceur qui en résulte pourrait alors être implanté par génération automatique de code.

Pour réaliser cette approche, nous adoptons la démarche suivante :

- Définition d'un sous-ensemble du langage Java pour la programmation des systèmes temps-réel. Ce sous-ensemble correspond à un compromis entre, d'une part l'expressivité et le confort de programmation, et d'autre part la possibilité de construire un modèle calculé statiquement. Pour sa définition, il a été pris en compte les normes adoptées pour d'autres langages, telles que celles mentionnées plus haut, notamment le profil Ravenscar Ada.

- Pour construire le modèle temporisé d'un système, il a fallu d'abord extraire un modèle non temporisé représentant une abstraction du comportement du logiciel Java. Nous avons défini la sémantique des instructions du langage, puis nous donnons une méthodologie basée sur des critères d'observabilité et permettant de construire les modèles fonctionnels des instructions d'un programme. Ces modèles sont les briques de base pour la construction du modèle d'un système temps-réel.

- Le modèle d'un système temps-réel est construit d'une manière modulaire et par couches. Trois couches principales forment le modèle : la première est constitué par la composition des modèles des threads, la deuxième modélise la synchronisation et la communication, et la troisième est constituée par l'ordonnanceur. Les modèles des threads sont obtenus en ajoutant des contraintes temporelles aux modèles fonctionnels construits à partir des instructions de leurs corps. Ces contraintes temporelles émanent des temps d'exécution, de la dynamique de l'environnement et les contraintes de QoS. La composition des threads prend en compte l'ensemble des actions synchronisées calculé en fonction de leurs interactions. Les deux dernières couches du modèle sont basées sur les priorités dynamiques [AGS00, AGS02]. Ceci nous permet d'exprimer d'une manière unifiée des propriétés fonctionnelles, telles que l'exclusion mutuelle, et non-fonctionnelles, telles que celles dédiées à l'ordonnancement.

Nous avons implanté l'outil *JediTool* afin de valider les idées et la démarche que nous avons établies. *JediTool* est connecté à la boîte à outils d'analyse et vérification IF-2 [BGM02] développée à *Verimag*. Il est intégré comme composant dans l'environnement de développement à code ouvert *Eclipse* [OTI03].

1.4 Plan du document

De la décomposition du travail, présentée au paragraphe précédent, découle l'organisation suivante.

Le chapitre 2 définit un sous-ensemble du langage Java pour la programmation des systèmes temps-réel, adapté à la construction statique de modèles. Le chapitre 3 traite la modélisation des instructions du programme. Une méthodologie est proposée afin de construire

des modèles fonctionnels à différents niveaux d'abstraction. Le chapitre 4 présente les étapes de construction du modèle temporisé d'une application. Nous terminons le chapitre par un exemple concret qui illustre notre démarche. Le chapitre 5 présente l'utilisation et le contexte de l'outil *JediTool*. Le chapitre 6 présente des applications effectuées avec *JediTool*. Finalement, nous concluons et présentons les perspectives de ce travail au chapitre 7.

Chapitre 2

Un Sous-ensemble du Langage Java

La technologie Java connaît un engouement et succès rarement acquis par un langage de programmation en si peu de temps. Au départ destiné pour programmer les appareils électroménagers, le langage Java devient rapidement le langage de référence pour la programmation Web et des systèmes embarqués. Le succès du langage est dû à ses nombreuses qualités, notamment le paradigme orienté objet, les mécanismes de sécurité, la simplicité de sa syntaxe et la portabilité.

Les avantages offerts par le langage Java et son succès pour les systèmes embarqués suscitent beaucoup d'intérêt auprès des communautés scientifique et industrielle afin de l'adopter pour le développement des systèmes temps-réel. L'adaptation de Java pour le temps-réel, comme cela a été le cas pour d'autres langages généralistes, nécessite la définition d'extensions et librairies temps-réel et doit s'accompagner d'un ensemble de restrictions nécessaires à l'analyse formelle des programmes. Notre objectif est de définir un sous-ensemble du langage Java afin de permettre la construction statique de modèles temporisés des systèmes temps-réel. Ce sous-ensemble est défini par des règles de programmation indépendantes des librairies temps-réel envisagées.

La section 2.1 présente le langage Java. Ensuite, la section 2.2 détaille les extensions temps-réel apportées à Java dans le cadre du projet Expresso, basées sur la spécification RTSJ[BG00] et le profil Ravenscar-Java[KWK02, CW03]. La section 2.3 établit la syntaxe abstraite du sous-ensemble du langage Java que nous considérons et met en oeuvre les restrictions adoptées pour la programmation objet et concurrente. Nous concluons au paragraphe 2.4.

2.1 Le langage Java

Java est un langage orienté-objet avec un typage fort et statique. Le langage Java se compile vers un format binaire, appelée *bytecode*, destiné à être interprété par une *machine virtuelle*.

Depuis son lancement en 1995 par Sun Microsystems, le langage connaît une forte croissance en terme d'utilisateurs et outils de développement. Ce succès est dû aux avantages du langage Java, comme la gestion automatique de la mémoire, les mécanismes de sécurité contre l'introduction de code malicieux, la portabilité du bytecode *etc.* Dans les paragraphes suivants est donnée une vue d'ensemble des propriétés du langage et une discussion sur l'adéquation de Java pour le développement des systèmes temps-réel.

Objets et threads

Contrairement aux langages Ada 95 et C++ qui ont été écris comme extensions orientés objet de langages procéduraux (respectivement Ada 83 et C) le langage Java est orienté objet par construction. Les concepts tels que l'abstraction, l'encapsulation et l'héritage permettent une meilleure maîtrise de la complexité du logiciel et facilitent la programmation des systèmes complexes et de grande taille. De plus, Java hérite ainsi du savoir-faire (par exemple les patrons de conception objet [GHJV93]) et des méthodologies de conception orientées objet (telles que UML [RJB98]).

Un programme Java est un ensemble de déclaration de classes. Toute classe Java est implicitement sous-classe de `java.lang.Object`, racine de la hiérarchie des classes du langage. Les classes définissent les attributs et les méthodes des objets de l'application. Les threads sont des objets *actifs* qui spécialisent la classe prédéfinie `java.lang.Thread`. A chaque thread est associée, programmatiquement, une méthode appelée *logique du thread*. Cette méthode est exécutée d'une manière asynchrone lors du lancement du thread associé. Le lancement d'un thread s'effectue par l'invocation de la méthode `start()` définie dans le type `Thread`.

Le langage offre la possibilité de définir des sections critiques afin d'assurer l'exclusion mutuelle entre les threads. A chaque objet de l'application est implicitement associé un moniteur. Ces moniteurs peuvent être utilisés par les threads afin de protéger l'exécution des sections critiques. Le mot-clé `synchronized` permet de définir des blocs de code dont l'exécution est protégée par le moniteur d'un objet donné. Par exemple, dans l'instruction de synchronisation suivante :

```
synchronized( sharedObj ) {
     //début de la section critique
...
}    // fin de la section critique
```

le moniteur de l'objet référencé par la variable `sharedObj` protège de tout accès concurrent à une section critique protégé par une synchronisation sur le même objet.

En plus de l'exclusion mutuelle, les threads peuvent coordonner leurs exécutions par un mécanisme d'attente/notification. Ce mécanisme est utile lorsque pendant l'accès en exclusion mutuelle à un objet par un thread, l'état de l'objet (*ie.* valeurs de ses attributs) ne convient pas

aux traitements envisagés dans la section critique. Le thread relâche alors le verrou de l'objet et se met en attente jusqu'à ce qu'un autre thread le notifie, éventuellement après modification de l'état de l'objet, pour reprendre l'exécution. Par exemple, considérons un thread qui accède à une boîte aux lettres, représentée par un objet Java, afin de lire un message alors que celle-ci s'avère vide. Une solution serait de vérifier, à intervalles réguliers, la présence de message dans la boîte aux lettres tout en prenant soin de se synchroniser sur l'objet à chaque fois. Le mécanisme d'attente/notification est une solution alternative. Lors de l'accès à la boîte aux lettres, si elle est vide le thread se met en attente de notification et tout autre thread qui met un message dans la boîte aux lettres est tenu d'émettre une notification afin d'avertir les threads en attente. Ce mécanisme est implémenté par les méthodes `wait()`, `notify()` et `notifyAll()` que tout objet Java hérite de la classe `Object`.

L'invocation de la méthode `wait()` se fait uniquement dans le contexte d'un thread possédant le moniteur de l'objet sur lequel se fait l'appel. Cette invocation a pour effet de relâcher les verrous du moniteur de l'objet et ensuite de bloquer le thread en le mettant en état d'attente de notification. Une fois le signal de notification reçu, le thread notifié est prêt à s'exécuter et il tente de réacquérir les verrous de l'objet relâché. La méthode `wait()` peut prendre en paramètre une valeur de temps d'expiration qui limite le temps d'attente de notification. Sans ce paramètre, l'attente n'est pas bornée et peut durer infiniment longtemps.

L'appel d'une méthode de notification `notify()` ou `notifyAll()` sur un objet donné a pour effet d'envoyer un message de notification aux threads en attente de notification sur le même objet. La différence entre `notify()` et `notifyAll()` est que `notify()` envoie le message à un destinataire unique, le choix ce dernier est laissé à l'implémentation de la machine virtuelle ; alors que la méthode `notifyAll()` envoie un message en transmission générale (*broadcast*) : tous les threads en attente reçoivent simultanément la notification.

Exécution des programmes

Afin d'exécuter une application, la machine virtuelle procède au chargement de ses différentes classes. C'est le *chargeur de classe* qui est responsable de cette phase. Lors du chargement d'une classe, le *vérificateur de bytecode* de la machine virtuelle procède à une série de vérifications afin de s'assurer de la validité du format du bytecode de la classe et qu'elle ne comporte pas de code malicieux qui risque de causer des dommages au système : s'assurer que le code ne forge pas de pointeurs, ne viole pas les restrictions d'accès aux membres des objets, ne fait pas de conversions de type dynamiques illicites *etc.* Dès le chargement et tout au long de l'exécution d'une application, le *gestionnaire de sécurité* valide les autorisations d'accès aux ressources, comme l'accès au réseau, informations sur les disques *etc.*

La sécurité et la robustesse sont un point central dans la conception du langage Java. La machine virtuelle procède à différents tests lors de l'exécution du bytecode. Par exemple, chaque accès à un tableau est testé pour éviter les dépassements d'indice. La vérification des

types, effectuée à la compilation, est renforcée à l'exécution. L'arithmétique des pointeurs est interdite et les références aux objets ne peuvent pas être forgées au niveau du programme. Ainsi, les données ne risquent pas d'être corrompues, réécrites par inadvertance ou détournées par un programme hostile.

La mémoire en Java est gérée de manière automatique à l'aide d'un ramasse-miettes. Les objets de l'application sont alloués dans un espace mémoire appelé *tas d'allocation*. L'exécution d'une instruction d'instanciation d'objet du programme, à l'aide de l'opérateur `new`, alloue l'objet dans le tas d'allocation. Cependant, le programme n'est pas responsable de la libération de l'espace mémoire alloué. Le ramasse-miettes détecte automatiquement les objets qui ne sont plus référencés par le programme et les libère. Le déclenchement du ramasse-miette n'est pas contrôlé par le programme, il est mis en route d'une manière automatique par la machine virtuelle. Pendant son activité, tous les threads de l'application sont interrompus, faute de quoi les références des objets du programmes risquent d'être corrompues par les manipulations de la mémoire.

Bien que le langage offre la possibilité de créer et coordonner les threads sans dépendre des plate-formes d'exécution, l'ordonnancement des threads ne peut pas être défini d'une manière précise et reste dépendant de la plate-forme sous-jacente à la machine virtuelle. Notamment, bien qu'il soit possible d'attribuer aux threads des priorités d'exécution, la spécification du langage précise qu'il n'est pas garanti qu'un thread de plus grande priorité s'exécute avant un thread de moindre priorité.

Le choix d'adopter une sémantique faible de l'ordonnancement des threads est motivé par un compromis entre le souci de portabilité et les disparités entre les plate-formes d'exécution. En effet, les threads Java sont souvent associés (liés) aux threads du système d'exploitation hôte afin de profiter, entre autres, du parallélisme en cas de présence de plusieurs processeurs. Cette association cause des problèmes car les niveaux de priorités disponibles, les politiques d'ordonnancement et les propriétés des threads systèmes varient significativement d'un système d'exploitations à un autre.

Adéquation de Java au temps-réel

Dans ce paragraphe sont présentés les avantages et les inconvénient majeurs que présente le langage Java pour le développement des systèmes temps-réel. Parmi les caractéristiques intéressantes pour le développement temps-réel, on a :

- Java est un langage orienté objet. Il facilite le développement des systèmes temps-réel, de plus en plus complexes et intégrés.
- La machine virtuelle est de petite taille. Une machine virtuelle de base tient dans 40 ko plus 175 ko pour le support des threads ; soit 215 ko au total. La machine virtuelle peut donc être embarquée dans des systèmes relativement limités en ressources.

- Le bytecode est portable. Pas besoin de disposer de plusieurs compilateurs et outils de débogage pour chaque plate-forme de déploiement.
- Une sémantique claire et simple qui permet un temps d'apprentissage relativement court (notamment par rapport à C et C++). De plus, le langage bénéficie d'une grande communauté d'utilisateurs.
- Java offre des primitives pour implémenter la concurrence et la coordination des threads au niveau du langage, sans avoir recours aux librairies des plate-formes de déploiement choisies.
- Les mécanismes de sécurité facilitent le développement et augmentent la robustesse des programmes. Des bogues classiques comme le dépassement d'indice pour les tableaux (qui donne lieu à des crashs de système très difficiles à déboguer et souvent utilisés comme faille de sécurité) sont simplement impossibles en Java.

Cependant, certains facteurs contraignent l'utilisation de la technologie Java dans le développement temps-réel. En effet :

- Les librairies de base du langage ne disposent pas de primitives dédiées au temps-réel, tel que les horloges, gestionnaire d'interruptions, accès à la mémoire physique *etc.*
- L'ordonnancement des threads est approximatif et dépendant de la plate-forme d'exécution.
- Le ramasse-miettes rend l'exécution des applications indéterministe. Il est difficile de prévoir les pires temps d'exécution et le comportement des applications à cause des interruptions occasionnées par le ramasse-miettes.
- Les fonctionnalités dynamiques du langage rendent difficile, dans certains cas, l'analyse formelle des programmes. Par exemple, les expressions à effet de bords, le polymorphisme, le chargement dynamique *etc.*
- Java ne dispose pas de norme internationale contrairement à d'autres langages comme Ada et C[1]. Sans un standard reconnu et indépendant de Sun Microsystems, le langage Java pourrait avoir du mal à gagner la confiance des développeurs de solutions pour les systèmes critiques.
- Il n'existe pas, à ce jour, de machines virtuelles ni compilateurs certifiés pour les applications temps-réel critiques, comme c'est le cas pour Ada.

En apportant des solutions adéquates aux problèmes mentionnés ci-dessus, les avantages et qualités de Java en font une technologie attractive pour les développement des systèmes temps-réel. Dans les sections suivantes est présentée l'approche Expresso pour la définition d'un Java temps-réel, solution basée sur la spécification RTSJ et le profil Ravenscar-Java. Ensuite, sont

[1]Standards ISO/IEC 8652 :1995 et ISO/IEC 9899 :1999 respectivement pour Ada et C.

présentées des restrictions que nous imposons sur Java afin de définir un sous-ensemble du langage approprié à l'extraction statique de modèles et l'analyse formelle des systèmes temps-réel écrits en Java.

2.2 Approche temps-réel Expresso

Le but du projet Expresso est de fournir une solution pour le développement et le déploiement des systèmes temps-réel critique pour la technologie Java. L'approche retenue par le projet consiste à compiler le bytecode Java en code natif afin de pouvoir le déployer sur des systèmes d'exploitation temps-réel commerciaux et éliminer ainsi la phase d'interprétation qui pénalise la rapidité d'exécution. Les principaux objectifs du projet sont les suivants :

- Définir une interface programmatique et mettre en oeuvre une librairie qui fournissent les fonctionnalités temps-réel (horloges, threads temps-réel *etc.*).
- Apporter un ensemble d'outils pour la compilation et la mise au point des programmes.
- Fournir un exécutif natif, appelé JRTS (*Java Run-Time System*), qui remplace la machine virtuelle et encapsule les services du système d'exploitation temps-réel hôte.

L'interface programmatique Expresso est basée sur un sous-ensemble de la spécification RTSJ, inspiré du profile Ravenscar-Java.

2.2.1 La spécification temps-réel RTSJ

L'institut national américain des standards et de la technologie (NIST), associé à un ensemble d'industriels, a publié en 1998 les besoins et conditions nécessaires pour une extension temps-réel pour la technologie Java [The99]. Le groupe de réflexion initial, suite à des désaccords sur le contrôle et l'évolution d'une spécification Java temps-réel, se divise alors en deux groupes :

- *RTJWG*[2] dirigé par le *J-Consortium*, un forum d'industriels indépendant de Sun Microsystems. Ce groupe comprend plus d'une centaine de membres et a donné lieu à l'extension *RTCE* [J-C00]. Cette spécification défini un exécutif séparé et indépendant de la machine virtuelle pour l'interprétation des systèmes temps-réel. Ce dernier peut être utilisé d'une manière autonome ou associé à une machine virtuelle classique. Cependant les deux domaines (temps-réel et non temps-réel) restent séparés et avec une communication restreinte. La spécification RTCE n'a toujours pas rencontré de succès et semble être peu à peu abandonnée, surtout après la dissolution du J-consortium. Aucune implémentation de référence de l'exécutif RTCE n'a été livrée à ce jour bien que la spécification ait été publiée depuis quelques années.

[2] *Real-Time Java Working Group*

- *RTJEG*[3] sous l'égide du *Java Community Process* affilié à Sun Microsystems. Cet effort a abouti à l'extension *RTSJ* (*Real-Time Specification for Java*) [BG00]. Malgré une implémentation de référence de qualité contestée [Dib02], plusieurs autres implémentations de la spécification, académiques [BR01, DAR04] et industrielles [Sie02], ont été effectuées. La spécification RTSJ est à ce jour la plus stable et soutenue des extensions temps-réel pour Java.

La spécification RTSJ a pour but d'étendre la machine virtuelle Java et définir une interface programmatique pour le développement des systèmes temps-réel. Contrairement à la RTCE, où le domaine temps-réel est séparé du domaine non-temps réel, la spécification RTSJ a pour but de définir une plate-forme où cohabiteraient les activités des deux domaines. La spécification conserve donc la compatibilité en arrière afin d'exécuter les applications écrites pour un environnement d'exécution Java classique. Cette compatibilité a motivé la mise en place d'un mécanisme de gestion de mémoire indépendant du tas d'allocation classique (géré automatiquement) afin d'éviter toute interférence entre les threads temps-réel dur et le ramasse-miettes. Pour l'ordonnancement, la RTSJ n'impose pas de politique d'ordonnancement spécifique. Il est possible d'implanter différentes politiques d'ordonnancement dans une machine virtuelle RTSJ.

Gestion de la mémoire Le tas d'allocation est conservé dans la machine virtuelle RTSJ, il est géré d'une manière automatique par le ramasse-miettes. En plus du tas, la spécification définit un mécanisme d'allocation par régions géré par le programme. Une région est un espace mémoire alloué à l'extérieur du tas d'allocation. Une fois créée, une région est affectée à une portée (*scope*, d'où le nom de la classe *ScopedMemory*). La portée est définie par un bloc d'instructions Java dont le contexte d'allocation est la région qui lui est associée. En d'autres termes, les allocations, par l'opérateur `new`, dans ce bloc d'instructions sont effectuées dans l'espace mémoire de la région associée. Il est possible d'*imbriquer* les régions : c'est le cas lorsque le code associé à une région en crée une nouvelle ; les régions imbriquées sont gérées par une pile au niveau de la machine virtuelle.

Les régions ne sont pas gérées par le ramasse miettes, les objets créés persistent jusqu'à la fin de l'exécution de sa portée. Dans ce cas, tous les objets alloués sont détruits et l'espace mémoire de la région libéré. La spécification définit une région mémoire spéciale appelée *mémoire immortelle* dont la durée de vie est celle de l'application, cette région est créée automatiquement et n'est jamais libérée, les objets qui y sont alloués persistent durant toute l'exécution de l'application.

Les régions créées par le programme ont des durées de vie différentes. Une région A créée dans le contexte d'une région B a une durée de vie plus courte. Il en est de même des objets allouées, respectivement, dans les régions A et B. Afin d'éliminer le risque de former des

[3] *Real-Time Java Expert Group*

références pendantes, la spécification RTSJ interdit qu'un objet de la région B référence un objet de la région A. D'une manière générale, les objets de plus longue durée de vie n'ont pas le droit de référencer les objets de plus courte durée de vie. Des tests à l'exécutions sont effectués par la machine virtuelle afin de renforcer cette règle.

Threads temps-réel La spécification RTSJ définit deux catégories de threads temps-réel. La différence entre ces deux catégories réside dans l'utilisation de la mémoire et si un thread peut ou non être interrompu par le ramasse-miettes.

Dans la première catégorie, définie par la classe `RealTimeThread`, les threads peuvent utiliser le tas d'allocation et les régions pour la création des objets. Par conséquent, ces threads peuvent être interrompus par le ramasse-miettes, responsable de la gestion automatique des objets alloués.

La seconde catégorie est composée des threads temps-réel dur où les pires temps d'exécutions doivent être déterministes. Ces threads, définis par la classe `NoHeapRealTimeThread`, ont une priorité supérieure au ramasse-miette afin d'empêcher toute interférence avec son exécution. Par conséquent, ils ne sont pas autorisés à accéder au tas d'allocation. Ils sont contraints à utiliser uniquement les régions pour l'allocation des objets.

La spécification définit un mécanisme de gestion asynchrone des événements. Chaque événement du système doit être associé à un objet qui le représente en utilisant la classe `AsyncEvent`. Chaque gestionnaire d'événement est associé aux objets d'événement qu'il traite. Lorsque un événement survient, l'objet associé qui lui ai associé "débloque" les gestionnaires : ils sont ordonnancés pour l'exécution. Les gestionnaires d'évènement sont définis par la classe `AsyncEventHandler`. Un gestionnaire d'évènement est associé à un contexte d'exécution (thread) d'une manière dynamique lors de son exécution. Cette opération dépend de l'implémentation de la machine virtuelle et peut pénaliser par un délai de temps le début du traitement du gestionnaire. Il est possible de créer des gestionnaires d'évènement liés statiquement à des threads en utilisant la classe `BoundAsyncEventHandler`.

Ordonnancement L'ordonnancement est basé sur deux propriétés qui caractérisent les threads en RTSJ : *paramètres d'arrivée* et *paramètres d'ordonnancement.*

- Les paramètres d'arrivée définissent pour un thread donné ses temps d'arrivée, son pire temps d'exécution, et son échéance. Trois types de paramètres d'arrivée sont prédéfinis : périodique, sporadique et apériodique. Ces paramètres permettent aussi d'associer à un thread un gestionnaire de dépassement d'échéance. Ce dernier s'exécute si le thread qui lui est associé rate son échéance. Ces paramètres sont représentés dans la spécification par la classe `ReleaseParameters` et ses classes filles.
- Les paramètres d'ordonnancement définissent l'*éligibilité d'exécution* de chaque thread. L'éligibilité d'exécution est la métrique sur laquelle se base la politique d'ordonnance-

ment pour choisir un thread parmi les threads prêts à être exécuter. Par défaut, l'éligibilité d'exécution est définie par des priorités associées aux threads. Cependant une implémentation RTSJ est libre de définir des paramètres d'ordonnancement alternatifs appropriés aux politiques d'ordonnancement fournies.

La RTSJ associe à l'ordonnanceur trois rôles : affecter le thread de plus haute éligibilité parmi les threads prêts à exécuter au processeur, gérer des moniteurs des objets partagés et fournir un mécanisme d'analyse de faisabilité.

Le choix des threads pour l'exécution se base sur les paramètres d'arrivée et d'ordonnancement des threads selon les critères de la politique d'ordonnancement. La spécification exige la présence d'un ordonnanceur, appelé *ordonnanceur de base*, préemptif basé sur des priorités fixes suivant la politique RMA [SKG91]. La gestion des ressources est dans ce cas gérée par le protocole PIP [SRL90]. Le protocole PCP [RSLR95] est recommandé par la spécification en tant qu'option, une implémentation de la machine virtuelle n'est pas obligée de le fournir. Il est possible d'associer, programmatiquement avec la librairie RTSJ, une des politiques de gestions de moniteurs fournies.

Etant donnée une politique d'ordonnancement, l'analyse de faisabilité consiste à déterminer, pour un ensemble de threads, la satisfaction des contraintes de qualité de services. L'analyse se base sur les paramètres d'arrivée et d'ordonnancement des threads. L'analyse de faisabilité est utile dans le cas des politiques d'ordonnancement temps-réel classiques basées sur des modèles analytiques simples à vérifier. Cette fonctionnalité est optionnelle dans la spécification RTSJ. Elle est utile afin qu'une application vérifie la faisabilité de l'ordonnancement avant d'ajouter, à l'exécution, de nouveaux threads au système.

2.2.2 Profil Ravenscar-Java

Le profil Ravenscar-Java [KWK02, CW03] définit un sous-ensemble de l'interface programmatique RTSJ pour la programmation des systèmes temps-réel dur. Le profil impose des ensembles de threads et objets partagés statiques et un ordonnancement basé sur la politique RMA et une gestion des moniteurs par PCP. Le profil fournit aussi un ensemble de directives de programmation, composées de 28 règles (directives obligatoires) et 24 conseils (directives optionnelles), inspirées du profil Ravenscar pour Ada [BDV04, DB98]. Les restrictions essentielles apportées à la spécification RTSJ par le profil Ravenscar-Java sont les suivantes :

- **Gestions de la mémoire** — Le tas d'allocation et le ramasse-miettes sont éliminés. La mémoire est allouée uniquement pour une mémoire immortelle et des régions de tailles fixes et à temps d'allocation linéaire. Il est interdit d'imbriquer les régions ou de les partager entre plusieurs threads.

- **Threads temps-réel** — Seul les threads sans accès au tas d'allocation sont autorisés. A chaque thread est associée une unique région de mémoire dans laquelle il alloue les objets locaux. Un objet local créé par un thread ne peut être partagé avec un autre thread. Le mécanisme d'attente/notification est totalement interdit. Les objets partagés sont alloués dans la mémoire immortelle.
- **Ordonnancement** — Ravenscar-Java impose un ordonnancement préemptif basé sur des priorités fixes de type RMA où tout les moniteurs des objets sont synchronisés par la politique PCP. La politique PIP n'est pas permise car elle souffre des problèmes de blocage par acquisition croisée des ressources partagées (*deadlocks*) et des chaînes de blocages (*chains of blocking*) [Aud91]. Le profile n'autorise pas de politiques d'ordonnancement et de gestion de ressource alternatives.

Les programmes Ravenscar-Java sont destinés à tourner sur une machine virtuelle spécialement allégée compte tenu des restrictions imposées. En particulier, une machine virtuelle Ravenscar ne comporte pas de ramasse miette, est dépourvue de tas d'allocation et ne contient pas de mécanismes pour la gestion des attentes/notifications.

L'exécution des applications conformes à Ravenscar-Java est divisée en deux phases distinctes : la première, appelée *phase d'initialisation*, est destinée à la création et initialisation des threads et des objets partagés ; la deuxième, *appelée phase mission*, est réservée à l'exécution des threads. Aucun thread ou objet partagé ne peut être créé dans cette deuxième partie. Ces phases d'exécution, augmentée par une troisième phase de *terminaison*, sont adoptées dans notre sous-ensemble de Java et détaillées au paragraphe 2.3.3.

2.2.3 Implémentation *Expresso*

L'implémentation de la librairie temps-réel Expresso et de l'exécutif JRTS ont motivé certaines modifications de l'interface programmatique de la librairie RTSJ / Ravenscar-Java initialement adoptée. Le profil Expresso se démarque de Ravenscar-Java en quelques points concernant la programmation concurrente, tel que :

- Les gestionnaires d'événement asynchrones sont restreints aux gestionnaires sporadiques avec un temps minimum non nul entre deux arrivées successives, afin de minimiser les situations de surcharge du système.
- La création des moniteurs associés aux objets ne se fait pas dynamiquement. Afin de se synchroniser sur un objet partagé, il faut au préalable créer explicitement son moniteur, faute de quoi une erreur se produit. Ceci permet d'économiser la mémoire et restreindre le mécanisme de verrouillage des objets.
- La politique de gestion des ressources PIP, écartée par Ravenscar-Java, est autorisée par Expresso. Le protocole d'héritage de priorité PIP à l'avantage d'introduire un délai

faible lors de l'exécution et il est supporté par la majorité des systèmes d'exploitations temps-réel commerciaux.

L'exécutif natif Java et la librairie Expresso ont été implémentés pour les systèmes d'exploitation POSIX (notamment *FastOS* de *Thales*) et le noyau temps-réel *Raven* d'*Aonix*. L'interface programmatique de la librairie Expresso, documentation et source, a été publiée en licence logiciel libre [GRF03].

2.2.4 Evaluation

La vision de la RTSJ est de définir une plate-forme pour accueillir des applications non-temps réel, temps-réel mou et temps réel dur. La spécification constitue un cadre riche de fonctionnalités temps-réel afin de répondre aux besoins et apporter confort et flexibilité de développement pour un large domaine d'application. Le projet Expresso, basé sur le profil Ravenscar-Java, définit un cadre plus approprié au développement des systèmes temps-réel dur en considérant un sous-ensemble des fonctionnalités offertes par la spécification RTSJ. Cette démarche n'est pas nouvelle : la librairie temps-réel du langage Ada a été restreinte par le profil Ravenscar, aujourd'hui une norme [Int05] pour le développement en Ada des applications temps-réel dur. Les langages C et C++ disposent aussi de profils pour le développement des systèmes temps-réel dur et embarqués tel que Misra C [MIS98] et les directives pour la conformité au standard IEC 61508 [Gre02].

L'approche Expresso facilite l'analyse et le développement des applications Java/RTSJ temps-réel dur. Cette approche découle de l'expérience et pratiques industrielles utilisées pour le développement des systèmes temps-réel critique. Cependant, il est à noter que :

- L'approche Expresso/Ravenscar-Java ne répond pas au besoin de politiques d'ordonnancement plus flexible vu l'intégration et la complexité de plus en plus grande des applications temps-réel. Expresso se restreint à un ordonnancement basé sur des priorités fixes RMA et une politique de gestion de ressource basée sur PCP et PIP. Ces politiques ne permettent pas d'exprimer des contraintes liées à des exigences de qualité de service plus complexes.
- L'interdiction du mécanisme de communication attente/notification nous semble trop restrictive. En se plaçant dans le cas des attentes avec temps d'expiration borné, les pires temps d'exécution restent déterministes et la communication n'introduit pas de blocage. De plus, la gestion des files d'attente pour chaque objet reste aussi prédictible, en temps d'exécution et espace mémoire, dans le cadre d'un modèle concurrent statique, c'est à dire avec des threads et objets partagés connus à priori.
- Les règles de programmation Ravenscar-Java sont imprécises sur l'utilisation des fonctionnalités orienté objet. La redéfinition (*overriding*) et surcharge (*overloading*) de méthode ne sont pas interdites, bien que le profil décourage ces mécanismes pour des

méthodes (redéfinies ou surchargées) dont "les temps d'exécution ou les logiques diffèrent fortement". Le profil ne précise pas le degré de liberté laissé au programmeur et les contextes favorables ou défavorables à l'utilisation de la redéfinition et la surcharge.

Le projet Expresso adopte une politique d'ordonnancement temps-réel basée sur un modèle analytique simple qui permet de vérifier facilement, sous certaines hypothèses, l'ordonnançabilité des systèmes. Cependant, cette approche manque d'expressivité et ne correspond pas à la réalité et aux besoins des systèmes temps-réel actuels. Afin de permettre une analyse plus fine et adopter des ordonnancements plus flexibles et adaptés, il est nécessaire de disposer de modèles formels plus riches qui prennent en compte la dynamique des systèmes. Dans les paragraphes qui suivent, nous présentons un ensemble de restrictions afin de définir un sous-ensemble du langage Java permettant l'extraction statique de modèles temporisés des systèmes temps-réel.

2.3 Restrictions apportées au langage

Il s'agit d'établir un compromis entre, d'un côté, le confort de programmation et les fonctionnalités dynamiques et, d'autre côté, la construction statique de modèles formels à partir du logiciel applicatif d'un système temps-réel.

Les restrictions sont constituées par un sous-ensemble syntaxique du langage (paragraphe 2.3.1) et de restrictions sur la programmation des classes (paragraphe 2.3.2) et des threads (paragraphe 2.3.3). L'ensemble de ces restrictions forme le sous-ensemble du langage Java utilisé dans ce travail.

2.3.1 Syntaxe abstraite

Le sous-ensemble syntaxique de Java considéré est similaire à celui présenté en [DE97]. La syntaxe abstraite correspondante est donnée à la table 2.1 et est commentée dans les paragraphes qui suivent.

2.3.1.1 Types abstraits de base

$TypeId$ représente l'ensemble des types utilisés dans un programme. Ceux-ci sont composés des types de base du langage Java (types numériques et booléen, essentiellement) et des types définis par l'utilisateur sous forme de classes.

$ClassId$ et $ThreadId$ dénotent respectivement l'ensemble des noms de classe et de thread d'un programme. Nous distinguons volontairement ces deux catégories, bien qu'en Java, cette distinction n'est pas formellement faite, afin de simplifier les considérations de modélisation ultérieures.

$FieldId$ et Var dénotent respectivement l'ensemble des noms d'attributs de classe et de variables locales de programmes (utilisées essentiellement dans le corps des méthodes).

2.3.1.2 Expressions

Le jeu d'expression, basé sur le système de type précédent, est essentiellement composé de trois sortes d'expressions : les expressions *booléennes* et *arithmétiques* (dans la production $BAExpression$) et les expressions *référentielles* (production $RefExpr$). Nous trouvons pour chaque sous-ensemble d'expressions les littéraux correspondants ($BoolLiteral$, $ArithLiteral$ et $RefLiteral$), et les opérations de base bien typées, binaires ($BinOp$) et unaires ($UnOp$). $Instanciation$ dénote l'instanciation de classe, qui distingue l'instanciation des objets simples et celles des threads.

$$
\begin{array}{lcl}
RefExpr & ::= & \texttt{this} \mid \texttt{null} \mid Var.FieldId \\
BoolLiteral & ::= & \texttt{true} \mid \texttt{false} \\
BAExpr & ::= & BoolLiteral \mid ArithLiteral \\
 & \mid & UOp\ BAExpr \mid BAExpr\ BOp\ BAExpr \\
Expression & ::= & Var \mid RefExpr \mid BAExpr
\end{array}
$$

Le jeu d'expression est volontairement allégé afin de clarifier l'exposé, en omettant de distinguer certains types arithmétiques (flottants notamment) et autres constructions plus compliquées. En outre, il faut remarquer que l'affectation, l'instanciation d'objet et l'appel de méthode ne sont pas considérés comme expressions dans ce contexte : la raison en est de pouvoir éviter les effets de bords dans les expressions. Les instructions correspondantes sont présentées dans le paragraphe suivant.

2.3.1.3 Instructions

$Instanciation$ dénote l'instanciation de classe, qui distingue la création des objets simples et celle des threads. $MethodCall$ dénote l'appel de méthode sur une expression dénotant une référence. $Assignment$ dénote les instructions d'affectation entre une variable, et une expression ou l'un des deux éléments précédents. $ControlStm$ permet d'introduire les instructions de contrôle conditionnel et itératif.

Declaration	::=	*TypeId Var*;
ControlStm	::=	`if(` *Expression* `)` *Stm* [`else` *Stm*]
	|	`while (` *Expression* `)` *Stm*
Instanciation	::=	`new` *ClassId*`(` *ParList* `)`
	|	`new` *ThreadId*`(` *ParList* `)`
MethodCall	::=	*RefExpr*`.`*MethodId*`(` *ParList* `)`
ParList	::=	*Expression* {, *Expression*}*
SynchStm	::=	`synchronized(` *RefExpr*`)` *Block*
	|	*RefExpr*`.wait(`*ArithLiteral*`);`
	|	*RefExpr*`.notifyAll();`
Assignment	::=	*Var*=*Expression*;
	|	*Var*=*Instanciation*; | *Var*=*MethodCall*;
Stm	::=	*Declaration* | *ControlStm*
	|	*Assignment* | *SynchStm*
	|	*MethodCall* | *RefExpr*`.start();`
	|	*Block*
Block	::=	`{`*Stm**`}`

Les instructions de synchronisation sur les objets sont distinguées par la production *SynchStm* : l'exécution d'un bloc synchronisé (instruction `synchronized`) et les appels des méthodes d'attente / notification (`wait()` et `notifyAll()`). Le paramètre passé à la méthode `wait()` spécifie le délai d'attente maximal autorisé, il est contraint à un littéral par la syntaxe afin de faciliter l'analyse statique du code par la suite. Le paragraphe 2.3.3 réglemente l'utilisation des constructions synchronisées du langage par les threads d'une application.

Le *démarrage* d'une instance de thread est effectué à l'aide de la méthode `start()`. Les appels à cette méthode sont limités à la phase d'initialisation d'une application, cette restriction est aussi détaillée au paragraphe 2.3.3.

Enfin, la production *Block* définit les instructions de bloc d'instructions.

De même que pour les expressions, les instructions sont simplifiées en n'introduisant pas certaines constructions qui relèvent du sucre syntaxique, notamment pour les structures conditionnelles et itératives où il est possible d'en considérer d'autres (telles que les boucles `for` ou les sélections `switch-case` *etc.*).

Dans le but de faciliter l'analyse statique des programmes, les restrictions suivantes sont apportées aux instructions définies par la syntaxe :

Invocation de méthode Comme nous l'avons indiqué plus haut, les invocations de méthodes sont considérées comme des instructions et non comme expressions dans la syntaxe. Ceci permet d'éviter tout effet de bords dans les expressions tels que les appels

imbriqués ou en cascade. Par exemple, soient $v \in RefExpr$ une variable dénotant une référence et $m, n \in MethodId$ deux noms de méthodes. Les appels `v.m().n()` et `m(n())` sont interdits et devront être remplacés par une série d'expressions écrites à l'aide de variables temporaires.

Conditions des structures de contrôle De la même manière, les conditions des structures de contrôle, itératives et conditionnelles, sont statiques. L'évaluation de ces conditions se limite aux opérations arithmétiques et booléennes et ne produit pas d'effets de bords (les appels de méthode ou instanciations d'objets sont interdits dans ces expressions).

Terminaison abrupte Toutes les instructions qui génèrent une terminaison abrupte (instructions de saut et exceptions *cf.* [GJSB00], paragraphe 14.1) sont écartées : ainsi, toute instruction termine en ayant effectué tous les pas de calcul dont elle est composée. La seule instruction de saut tolérée, dont nous parlons dans le paragraphe suivant, concerne les instructions de retour (`return`).

2.3.1.4 Déclaration de classes et threads

Nous distinguons dans cette définition syntaxique la déclaration des classes simples et des threads. $ClassDecl$ (respectivement $ThreadDecl$) représentent la déclaration abstraite d'une classe Java (respectivement d'un thread).

Une classe peut étendre une autre classe (clause `extends`), et se compose de la déclaration d'attributs ($FieldDecl$) et de méthodes ($MethodDecl$). Nous distinguons une classe particulière, $Class_{main}$, composée d'une méthode particulière `main()`, qui constitue le point d'entrée du programme. En outre, par souci de simplification et de clarification, nous ne considérons aucun modificateur de classe et d'attributs.

Un thread est aussi composé de la déclaration d'attributs si besoin, mais possède une méthode particulière $MethodDecl_{run}$ qui définit sa *logique* représentant les instructions qu'il exécute à son lancement.

$Program$ dénote un programme Java au sens où nous l'entendons : celui-ci est composé de la donnée d'un ensemble de déclarations de classes et/ou de threads, avec une classe particulière $ClassDecl_{main}$.

$$
\begin{array}{lcl}
MethodDecl & ::= & MethodId(\ [\, ParDeclList \,]\ )\ Body \\
ParDecList & ::= & TypeId\ Var\ \{, TypeId\ Var\}^* \\
Body & ::= & \{ Stm^*\ [\texttt{return}\ Expression]\ \} \\
FieldDecl & ::= & TypeId\ Var\ ; \\
ClassDecl & ::= & ClassId\ [\, \texttt{extends}\ ClassId \,]\ \{\ \{FieldDecl \mid MethodDecl\}^*\ \} \\
ThreadDecl & ::= & ThreadId\ [\, \texttt{extends}\ ClassId \,]\ \{\{\ FieldDecl\}^*\ MethodDecl_{run}\ \} \\
Program & ::= & ClassDecl_{main}\ \{\ ClassDecl | ThreadDecl\ \}^*
\end{array}
$$

La table 2.1 récapitule l'ensemble de la proposition de syntaxe abstraite retenu pour la modélisation des programmes que nous effectuons.

2.3.2 Restrictions au niveau des classes

Dans ce paragraphe sont détaillées les restrictions adoptée aux niveaux de classes dans ce profil. A savoir, borner la récursivité, autoriser la surcharge, écarter la redéfinition de méthode ainsi que le chargement dynamique de classes.

Récursivité

La récursivité pose un problème pour analyser le comportement des systèmes, notamment pour borner le temps d'exécution de certaines méthodes du programme. Il est donc essentiel de disposer pour tout cycle du graphe d'appel d'une borne supérieure qui limite la séquence d'appel correspondante. Afin de simplifier l'exposé, nous supposons que les programmes étudiés ne comportent pas de récursivité.

Surcharge de méthode

La surcharge de méthode en Java est la présence dans une même classe de plusieurs – au moins deux– méthodes ayant le même nom mais des signatures[4] deux à deux différentes. Les implémentations des méthodes surchargées sont aussi potentiellement différentes. Il est possible de connaître d'une manière statique la signature exacte de la méthode qui va être invoquée lors de l'exécution : c'est la signature la *plus spécifique* (au sens de [GJSB00] section 15.12.2.2), si elle existe, parmi les signatures applicables à l'invocation qui est choisie ; sinon, le programme est considéré incorrect et une erreur de compilation se produit. Ainsi, lors d'une invocation d'une méthode surchargée, il est possible de définir la méthode exacte qui sera lancée lors de l'exécution, à la redéfinition près. La surcharge n'affecte pas le déterminisme du programme et ne déroute pas, en elle même, l'analyse statique. La surcharge est donc autorisée dans ce cadre de travail.

Redéfinition de méthode et lien tardif

Une variable déclarée d'un type T peut contenir à l'exécution une référence vers tout objet instance d'une sous classe de T. Chaque sous classe peut redéfinir les méthodes de ses super-classes afin de réagir d'une manière spécifique à certains messages. C'est le type concret de l'objet référencé qui détermine lors de l'exécution quelle méthode va être invoquée parmi celles définies dans la hiérarchie de classes : ce procédé s'appelle *lien tardif*. Ce mécanisme fait partie des points forts de la programmation orienté objet mais pénalise les performances en

[4] La signature d'une méthode est composée de son nom et des types et positions de ses arguments (*cf.* [GJSB00] section 8.4.2).

$$
\begin{array}{lcl}
\mathit{RefExpr} & ::= & \texttt{this} \mid \texttt{null} \mid \mathit{Var}.\mathit{FieldId} \\
\mathit{BoolLiteral} & ::= & \texttt{true} \mid \texttt{false} \\
\mathit{BAExpr} & ::= & \mathit{BoolLiteral} \mid \mathit{ArithLiteral} \\
 & \mid & \mathit{UOp}\ \mathit{BAExpr} \mid \mathit{BAExpr}\ \mathit{BOp}\ \mathit{BAExpr} \\
\mathit{Expression} & ::= & \mathit{Var} \mid \mathit{RefExpr} \mid \mathit{BAExpr} \\
\mathit{Declaration} & ::= & \mathit{TypeId}\ \mathit{Var}\,; \\
\mathit{ControlStm} & ::= & \texttt{if(}\ \mathit{Expression}\ \texttt{)}\ \mathit{Block}\ [\texttt{else}\ \mathit{Block}] \\
 & \mid & \texttt{while (}\ \mathit{Expression}\ \texttt{)}\mathit{Block} \\
\mathit{ParList} & ::= & \mathit{Expression}\{,\mathit{Expression}\}^* \\
\mathit{Instanciation} & ::= & \texttt{new}\ \mathit{ClassId}\texttt{(}\ \mathit{ParList}\ \texttt{)} \\
 & \mid & \texttt{new}\ \mathit{ThreadId}\texttt{(}\ \mathit{ParList}\ \texttt{)} \\
\mathit{MethodCall} & ::= & \mathit{RefExpr}.\mathit{MethodId}\texttt{(}\ \mathit{ParList}\ \texttt{)} \\
\mathit{SynchStm} & ::= & \texttt{synchronized(}\ \mathit{RefExpr}\texttt{)}\ \mathit{Block} \\
 & \mid & \mathit{RefExpr}\texttt{.wait(}\mathit{ArithLiteral}\texttt{)}\,; \\
 & \mid & \mathit{RefExpr}\texttt{.notifyAll()}\,; \\
\mathit{Assignment} & ::= & \mathit{Var}\texttt{=}\mathit{Expression}\,; \\
 & \mid & \mathit{Var}\texttt{=}\mathit{Instanciation}\,; \mid \mathit{Var}\texttt{=}\mathit{MethodCall}\,; \\
\mathit{Stm} & ::= & \mathit{Declaration} \mid \mathit{ControlStm} \\
 & \mid & \mathit{Assignment} \mid \mathit{SynchStm} \\
 & \mid & \mathit{MethodCall}\,; \mid \mathit{RefExpr}\texttt{.start()}\,; \\
 & \mid & \mathit{Block} \\
\mathit{Block} & ::= & \texttt{\{}\mathit{Stm}^*\texttt{\}} \\
\mathit{MethodDecl} & ::= & \mathit{MethodId}\texttt{(}\ [\mathit{ParDeclList}]\ \texttt{)}\ \mathit{Body} \\
\mathit{ParDecList} & ::= & \mathit{TypeId}\,\mathit{Var}\{,\mathit{TypeId}\,\mathit{Var}\}^* \\
\mathit{Body} & ::= & \texttt{\{}\mathit{Stm}^*\ [\texttt{return}\ \mathit{Expression}]\ \texttt{\}} \\
\mathit{FieldDecl} & ::= & \mathit{TypeId}\ \mathit{Var} \\
\mathit{ClassDecl} & ::= & \mathit{ClassId}\,[\ \texttt{extends}\ \mathit{ClassId}\ ]\ \texttt{\{}\ \{\mathit{FieldDecl} \mid \mathit{MethodDecl}\}^*\ \texttt{\}} \\
\mathit{ThreadDecl} & ::= & \mathit{ThreadId}\,[\ \texttt{extends}\ \mathit{ClassId}\ ]\ \texttt{\{\{}\ \mathit{FieldDecl}\}^*\ \mathit{MethodDecl}_{run}\ \texttt{\}} \\
\mathit{Program} & ::= & \mathit{ClassDecl}_{main}\ \{\ \mathit{ClassDecl}|\mathit{ThreadDecl}\ \}^*
\end{array}
$$

TAB. 2.1 – Syntaxe abstraite du sous-ensemble considéré de Java

temps d'exécution et rend l'analyse statique onéreuse et complexe. Plusieurs travaux traitent l'élimination, quand elle est possible, ou l'optimisation du lien tardif dans les langages orienté objet. En [VH96] est proposée une représentation plus efficace des tables utilisée pour le lien dynamique afin de minimiser les délais lors de l'exécution. Afin de remplacer le lien tardif par des appels directs des méthodes [DGC95] propose une technique pour déterminer un ensemble minimal de classes susceptibles de recevoir un certain envoi de message en analysant statiquement le programme et la hiérarchie de classes.

L'indéterminisme introduit par le lien tardif ne peut être toléré dans le développement des systèmes critiques. Cependant, en effectuant une analyse globale du programme et en construisant la hiérarchie de classes, il est possible de considérer à chaque invocation de méthode toutes les redéfinitions possibles de celle-ci et ainsi prendre en compte tout les comportements possibles qui peuvent suivre cette invocation. Nous *ne retenons pas* cette option. En effet :

1. Prendre toutes les redéfinitions de toutes les méthodes à chaque point d'appel fait exploser la taille du modèle fini que nous désirons construire statiquement à partir du programme. Ce modèle serait, en plus, trop conservateur car constituant une sur-approximation trop large du comportement réel. Cela entraîne, en l'occurrence, une estimation trop pessimiste des pires temps d'exécution.
2. Il n'est plus possible d'effectuer une analyse modulaire du programme. En effet, à chaque extension de ce dernier il faut recalculer le graphe de hiérarchie de classe (propriété globale) et adapter toutes les analyses effectuées au préalable au niveau des sites d'appel.
3. L'analyse statique des programmes est dans ce cas plus complexe à concevoir et à mettre en oeuvre et plus gourmande en temps de calcul et espace mémoire.

En plus des problèmes cités plus haut, la redéfinition de méthode risque d'être une source d'introduction de code mort dans le programme (par exemple dans le cas de réutilisation de code où le code client redéfinit certaines méthodes sans utiliser les anciennes). Dans un travail dédié à la certification des programmes orienté objet pour l'avionique , Rierson [Cer99] décourage fortement la redéfinition de méthode pour les programmes soumis à la certification DO-178B [Eur92]. La redéfinition de méthode est aussi interdite par le sous-ensemble Spark [BDC02] du langage Ada. Vu les problèmes évoqués ci-dessus, nous nous alignons sur cette position qui nous semble indispensable pour écrire un code déterministe qui reste analysable avec un coût raisonnable. La redéfinition de méthode est donc *interdite* dans notre sous-ensemble du langage Java.

Chargement dynamique

Le langage Java permet d'ajouter à l'exécution de nouvelles classes à l'application par le mécanisme de *chargement dynamique de classe*. Le chargement se fait soit par le chargeur de classe de base de la machine virtuelle ou encore par un chargeur implémenté par le programme.

Ceci permet d'adapter le chargement aux besoins de l'application : chargement à partir du système de fichier, du réseau ou même à partir de données forgées par l'application. Le chargement dynamique complique la tâche d'analyse et affecte la prédictibilité d'une application. Nous relevons deux problèmes essentiels :

1. Il est difficile de prédire le temps de chargement. La difficulté provient de différents facteurs. Chaque chargeur possède un père à qui il délègue systématiquement la requête de chargement avant d'effectuer lui même l'opération (au cas ou la classe a déjà été chargée par le chargeur parent). Ce modèle de recherche par délégation (appelé *chaîne de responsabilité* [GHJV93]) est fortement dynamique et difficile à borner d'une manière précise en temps d'exécution. D'autres parts, le temps de chargement dépend fortement du médium dans lequel est stocké le code de la classe à charger : fichier compressé, réseau, données en mémoire *etc.*
2. Le comportement introduit par les nouvelles classes chargées dans l'application n'est pas connu apriori. Cette opération doit se faire en respectant les analyses effectuées au préalable telles que les temps d'exécution et l'usage des objets partagés. Sinon, une analyse complémentaire doit être effectuée à l'exécution afin de valider le nouveau comportement introduit.

Le chargement dynamique de classe est un mécanisme intéressant notamment pour la mise à jour à chaud des applications temps-réel. La difficulté introduite par le chargement dynamique reste cependant rédhibitoire pour l'élaboration d'une méthodologie d'analyse avec des coûts raisonnables. Nous écartons le chargement dynamique de notre sous-ensemble de Java.

2.3.3 Restrictions au niveau des threads

Dans ce paragraphe nous présentons les restrictions sur l'architecture concurrente de l'application et sur les opérations effectuées par les threads. Nous considérons un scénario d'exécution en trois phases : *initialisation*, *mission* et *terminaison*. Une application est constituée d'ensembles statiques de threads et objets partagés, qui doivent rester invariants en phase mission. La communication inter-thread est limitée aux objets partagés et au mécanisme attente/notification. Nous définissons les deux ensembles suivants :

- $\Theta = \{th_i\}_{1 \leq i \leq n}$ l'ensemble des threads en phase mission. Il est à noter que le thread d'initialisation qui exécute la méthode `main()` est lancé au démarrage de l'application et termine avant la phase mission. Ce thread n'est pas pris en compte dans Θ.
- $\Omega = \{O_i\}_{1 \leq i \leq m}$ l'ensemble des objets partagés. Lors de la phase d'initialisation, les références des objets de Ω sont passés aux threads selon la logique de l'application.

Nous présentons les phases d'exécution, les propriétés des threads et les règles qui régissent les opérations de communication et accès aux objets partagés.

Phases d'exécution

Nous adoptons un cycle d'exécution des applications identique à celui adopté dans le projet *Expresso*, inspiré des profil Ravenscar-Ada et Ravenscar-Java. L'exécution d'une application se fait en deux phases essentielles, phase *initialisation* puis phase *mission*. Une phase *terminaison* est ajoutée pour le test et le débogage. Les phases d'exécution sont illustrées dans la figure 2.1.

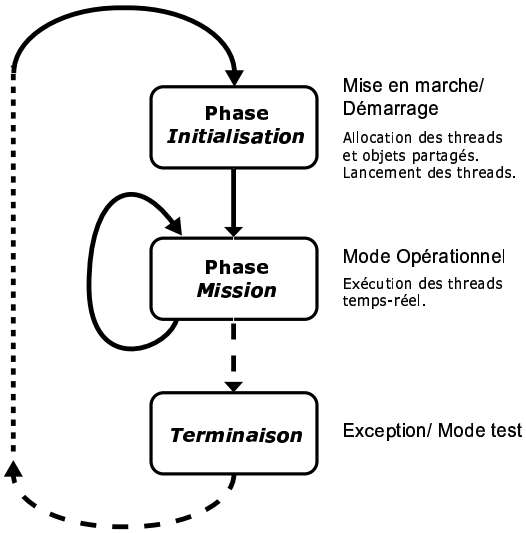


FIG. 2.1 – Phases d'exécution d'une application temps-réel

- **Phase Initialisation** — Cette phase correspond au lancement de l'application et l'invocation de la méthode `main`. Toutes les allocations de threads et objets partagés doivent être faites dans cette phase. Cette phase est exclusivement mono-threadée. La fin d'exécution de la méthode `main` entraîne la fin d'initialisation et le lancement de la phase mission.
- **Phase Mission** — Tout les threads lancés dans la phase d'initialisation commencent au même temps dans la phase mission. Seuls les objets locaux peuvent être alloués par les threads dans cette phase. Les objets crées par un thread lors de la phase mission ne peuvent être transmis à d'autres threads. La phase mission ne termine jamais, sauf en phase de test ou pour des raisons de maintenance.
- **Terminaison** — Cette phase est dédiée à la procédure de test et mise au point.

Propriétés des threads

Les threads sont des objets actifs dotés de méthodes d'activation, notées `start()` dans notre grammaire. A chaque thread est associée une méthode appelée *logique du thread* exécutée

à chacune de ses arrivées. La fin de l'exécution de la logique du thread correspond à la fin de son cycle. Dans notre grammaire la méthode `run()` de la classe du thread dénote sa logique. Il est possible, en général, d'associer dynamiquement une logique à un thread donné ; ce procédé est accepté dans ce travail s'il est effectué lors de la phase d'initialisation. Il est interdit d'attribuer et changer la logique d'un thread en phase mission.

Les threads sont des activités cycliques et qui ne terminent jamais. Chaque thread est donc soit périodique, soit sporadique. Un thread périodique est caractérisé par sa période T fixée lors de son initialisation et invariable durant son exécution. Un thread sporadique est caractérisé par la borne minimale et maximale du temps séparant deux arrivées successives du thread, respectivement T^l et T^u. Pareillement, les bornes T^l et T^u sont fixées lors de l'initialisation et ne changent pas durant l'exécution du thread.

Le mécanisme de transfert asynchrone de contrôle (*Asynchronous Transfert of Control*) est un mécanisme utilisé essentiellement pour l'implémentation de gardes-fou, basés sur un délai d'expiration ou un évènement externe, sur un bloc d'instructions donné. Bien que ce mécanisme a généralement pour vocation de limiter le temps d'exécution d'un ensemble d'instructions, il est difficile de borner le temps entre le moment de l'activation et le transfert effectif du contrôle. En effet, l'interruption de l'exécution nécessaire au transfert de contrôle ne peut survenir à tout point d'exécution du thread, notamment dans les sections critiques au risque de corrompre les données partagées de l'application. Dans ce cas, le transfert est reporté. Les temps de délai du transfert de contrôle dépendent donc étroitement du contexte à l'exécution. D'autre part, le transfert asynchrone de contrôle entraîne une terminaison abrupte de l'exécution des instructions et complique l'analyse du flot de contrôle. Nous ne prenons pas en compte le transfert asynchrone de contrôle dans ce travail.

Les instructions d'interruption, destruction et arrêt de thread ne sont pas autorisés. Un thread ne termine jamais et son flot d'exécution ne peut être affecté par un autre thread. De ce fait, les méthodes `join()`, `stop()` et `destroy()` de la classe `java.lang.Thread` ou toutes autres méthodes ayant une sémantique similaire ne doivent pas être utilisées.

Objets partagés et communications

La communication entre les différents threads obéit à des règles strictes. Les threads ne peuvent interagir que par le biais des objets partagés fixés lors de la phase de l'initialisation et via le mécanisme d'attente / notification sur ces objets. La coordination entre les threads obéit aux règles suivantes :

- Tous les accès en lecture et écriture aux objets partagés doivent s'effectuer exclusivement par le biais d'instructions ou méthodes synchronisées.
- Durant la phase mission, un thread ne peut transmettre les objets qu'il crée à d'autres threads, notamment à travers les méthodes d'accès de leurs attributs. De même, les

références des objets créés par les threads ne peuvent être affectées aux attributs des objets globaux, faute de quoi ces derniers sont susceptibles d'être accessibles par plusieurs threads.

- L'utilisation de la méthode `notify()` est interdite. Contrairement à la méthode `notifyAll()` qui émet une notification par transmission générale, la méthode `notify()` notifie au plus un thread en attente de notification. Le choix de ce dernier n'est pas spécifié : il dépend de l'implémentation de la machine virtuelle utilisée ou de l'ordonnanceur mis en place.
- la méthode `wait()` sans délai d'expiration est interdite. Les threads doivent utiliser à la place la méthode avec délai d'expiration afin de pouvoir borner les temps d'attente. Le délai d'expiration ne doit pas dépendre du contexte d'exécution ; il doit être fixé et connu à priori.

2.4 Discussion

Dans ce chapitre nous avons proposé un ensemble de règles et restrictions qui définissent un sous-ensemble du langage Java pour la programmation des systèmes temps-réel. Ces restrictions concernent les aspects du langage qui compromettent la prédictibilité du flot de contrôle et du temps d'exécution au niveau des instructions, des classes et des threads. Ce sous-ensemble peut sembler restrictif car il élimine un bon nombre d'aspects dynamiques du langage, comme la redéfinition de méthode et le chargement dynamique de classe. Les restrictions adoptées restent toutefois comparables aux profils temps-réel pratiqués en industrie, tels que les règles Misra [MIS98] pour le langage C et le sous-ensemble Spark [BB03] du langage Ada.

Le restrictions présentées dans ce chapitre définissent un sous-ensemble du langage qui nous permet une construction statique des modèles des systèmes temps-réel. Le chapitre suivant présente la construction des modèles fonctionnels à partir des instructions des programmes Java conforme à ce sous-ensemble.

Chapitre 3

Modélisation des Instructions du Langage

Au chapitre précédent, nous avons défini un sous-ensemble du langage Java afin de permettre l'analyse et l'extraction statique de modèle à partir du logiciel applicatif d'un système temps-réel. Il est ici question de formaliser le modèle mathématique utilisé pour représenter les instructions des programmes qui respectent ce sous-ensemble (notamment la syntaxe de la table 2.1), et de mettre en oeuvre le procédé de construction des modèles.

Les modèles utilisés sont basés sur les systèmes de transitions que nous enrichissons, dans le contexte de la programmation concurrente en Java, par des contextes de synchronisation rendant compte, à chaque état du modèle, de l'utilisation des ressources partagées.

La notion centrale d'observabilité régit la construction du modèle : une instruction déclarée comme observable doit figurer dans le modèle, étant jugée pertinente par rapport aux propriétés qu'il faut montrer. En ce sens, l'observabilité paramètre la construction du modèle.

Les modèles des instructions sont construits par raffinements successifs. Nous nous proposons dans ce chapitre d'éclaircir la construction du modèle relatif à une instruction donnée du programme. En ce sens, la définition proposée est structurelle : à chaque instruction est associé un modèle élémentaire, qu'il est possible de raffiner suivant un critère d'observabilité. Le raffinement des transitions impliquant des actions observables est présenté sous forme de règles de réécriture respectant la sémantique des instructions dans le langage Java.

Pour une instruction donnée, nous montrons que sa modélisation possède la propriété importante de convergence : paramétrée par le critère d'observabilité, la construction que nous proposons conduit d'une part, toujours au même modèle raffiné, quel que soit l'ordre d'application des règles de réécriture ; et d'autre part, termine toujours, ce qui est naturellement le cas dans le contexte d'une instruction d'un langage de programmation. Les modèles des instructions d'un programme serviront de base, au chapitre suivant, pour construire les modèles temporisés des threads. Ces derniers, associés aux modèles de leurs interactions, forment la

base du modèle d'une application temps-réel.

La Section 3.1 présente une formalisation du modèle utilisé, basé sur la notion de système de transition et de contexte de synchronisation. La construction du modèle à partir d'une instruction, présentée dans la section 3.2, établit le lien entre programme et modèle en capturant la notion de raffinement de modèle à l'aide du concept de réécriture. Finalement, la section 3.3 présente les algorithmes nécessaires à la construction des modèles.

3.1 Modèles et opérations sur les modèles

Les modèles que nous utilisons pour modéliser les instructions Java sont basés sur des systèmes de transition. Ces modèles sont des automates constitués d'états représentant de manière abstraite un état du programme, liés par des transitions qui déterminent quand il est possible de passer d'un état à un autre.

Les modèles que nous utilisons étendent les systèmes de transitions en ajoutant un *contexte de synchronisation* : à chaque état est associé des informations concernant les objets partagés verrouillés et éventuellement un objet sur lequel se fait une attente de notification.

Les labels utilisés au sein des systèmes de transitions sont dans un premier temps explicités, en faisant le lien avec les programmes modélisés. Les contextes de synchronisation sont ensuite expliqués notamment en rapport avec les instructions de synchronisation et de mise en attente propres à Java. L'ensemble de ces éléments permet ensuite de formaliser les modèles que nous adoptons : des opérations importantes (concaténation et réécriture de transition) sont ensuite présentées dans l'esprit de la construction des modèles à partir de la donnée d'une instruction.

3.1.1 Labels

Les transitions du modèle sont décorées par des labels. Ceux-ci servent à matérialiser les transitions modélisant une instruction, ou une primitive de la machine virtuelle Java, que l'on désire marquer ou observer dans des algorithmes d'analyse ultérieurs. Trois catégories répartissent les labels que nous considérons dans nos systèmes de transition :

Labels de calcul $\mathcal{L}_{exec}$: Cet ensemble est composé de deux labels : $beginCompute$ marque le début d'un pas de calcul ; et $endCompute\,\langle E^l, E^u\rangle$ (avec $E^l, E^u \in \mathbb{N} : E^l \leq E^u$) sa fin, en bornant la durée d'exécution de ce calcul (E^l étant la borne inférieure et E^u la borne supérieure).

Labels de synchronisation $\mathcal{L}_{synch}$: Cet ensemble de labels marque les transitions relatives aux actions de synchronisation sur des objets partagés, de même que le mécanisme d'attente / notification. Pour un objet $O \in \Omega$ et un entier $W \in \mathbb{N}_{>0}$, on notera par les labels :

- $p\,O$ le verrouillage de O

- $v\,O$ le déverrouillage de O
- $!\,O$ la notification sur O
- $?\,O$ l'attente de notification sur O
- $timeout\,\langle W \rangle$ l'expiration du délai d'attente de notification fixé à W unités de temps

Labels d'instruction $\mathcal{L}_{stm}$: Ces labels reprennent toutes les instructions du programme, sous la forme qui est définie par la production *Stm* dans la table 2.1. Nous distinguons le sous-ensemble $\mathcal{L}^c_{stm} \subset \mathcal{L}_{stm}$ qui regroupe les *instructions composées* du programme. Ce sont les instructions dont l'exécution entraîne l'exécution d'autres instructions (qu'elles contiennent syntaxiquement) ou des primitives de la machine virtuelle (typiquement les instructions de synchronisation sur les objets partagés). Les instructions composées sont exactement celles conforment aux règles de production *ControlStm*, *MethodCall*, *Block* et *SynchStm* de notre syntaxe.

De même, nous introduisons un label τ qui ne dénote aucune action particulière, mais que l'on veut exhiber sur une transition.

Définition 3.1 *(Labels)*

Un label $l \in \mathcal{L}$ est un élément pris dans la partition suivante : $\mathcal{L}_{stm}$ est l'ensemble des labels d'instructions ; $\mathcal{L}_{synch}$ est l'ensemble des labels de synchronisation ; $\mathcal{L}_{exec}$ est l'ensemble des labels de calcul, et $\{\tau\}$.

$$\mathcal{L} = \mathcal{L}_{stm} \cup \mathcal{L}_{synch} \cup \mathcal{L}_{exec} \cup \{\tau\}$$

L'ensemble des instructions composées est noté $\mathcal{L}^c_{stm}$.

Les fonctions $stm2lab : Stm \to \mathcal{L}$ et $lab2stm : \mathcal{L} \to Stm$ permettent de faire trivialement le lien entre instructions du programme et labels. Ainsi, toutes les fonctions définies sur les instructions *Stm* (respectivement sur les labels $\mathcal{L}$) pourront facilement être redéfinies sur les labels (respectivement sur les instructions). Nous confondrons désormais les deux.

3.1.2 Système de Transitions

Les systèmes de transitions sont un modèle simple et formel qui permet de décrire les applications dans le cadre de la vérification. Les systèmes de transition servent de base à la modélisation que nous proposons pour les applications temps-réel écrites en Java, et seront enrichis par la suite par des contextes de synchronisation pour capturer l'ensemble des informations pertinentes sur les programmes proposés.

Un système de transitions est la donnée d'un ensemble d'états abstrayant les états concrets de l'application modélisée, où il est possible de passer d'un état à un autre en empruntant

une transition qui modélise les étapes d'exécution nécessaires pour pouvoir accéder à l'état d'arrivée.

Définition 3.2 *(Système de Transition)*
Un système de transition est la donnée d'un triplet $TS = (S, L, T)$ *dans lequel :*
- S *est un ensemble fini d'états ;*
- $L \subseteq \mathcal{L}$ *est un sous-ensemble des labels que nous avons définis plus haut ;*
- $T \subseteq S \times L \times S$ *est une relation de transition.*

Une transition $(s, l, s') \in S \times L \times S$ sera notée sous la forme imagée $s \xrightarrow{l} s'$: s en est l'état d'origine (ou de départ) ; s' est l'état de fin (ou d'arrivée). Pour un état $s \in S$, on définit deux fonctions *ingoing* et *outgoing*, désignant l'ensemble des transitions rentrantes et sortantes relatives à s, dans la relation de transition T. Elles sont définies par $ingoing(s) = \{(s', l, s) | s' \xrightarrow{l} s\}$ et $outgoing(s) = \{(s, l, s') | s \xrightarrow{l} s'\}$.

Nous définissons une opération de *renommage* sur les états, qui remplace, dans une transition, un état par un autre état, en redéfinissant proprement les transitions entrantes et sortantes de l'état renommé.

Le renommage dans une relation de transition d'un état s par un état s' est calculé à partir de la relation de transition : chaque occurrence de l'état s, qu'il soit à l'origine d'une transition ou à sa fin, est remplacée par s'.

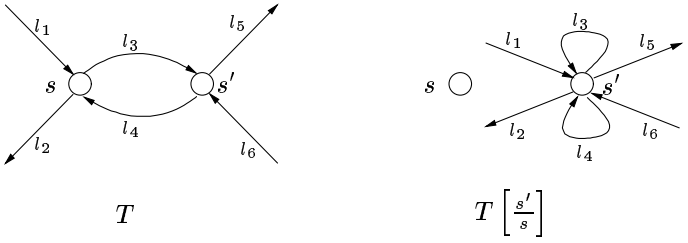


FIG. 3.1 – Exemple de renommage d'état : s est renommé par s'.

Définition 3.3 *(Renommage d'état)*
Soit $TS = (S, L, T)$ *un système de transition et* $s, s' \in S$ *deux états du système. Le* renommage de s par s' *dans* T *est la nouvelle relation de transition* T' *notée*

$$T' = T\left[\frac{s'}{s}\right]$$

telle que $\forall (s_1, l, s_2) \in T \,.\, (s'_1, l, s'_2) \in T'$ *où* $s'_i = \begin{cases} s_i \text{ si } s_i \neq s \\ s' \text{ sinon} \end{cases} \quad i = 1, 2$

La figure 3.1 montre un exemple de renommage dans une relation de transition T de l'état s par l'état s'.

Le renommage des états dans les relations de transitions possède des propriétés caractéristiques remarquables. Nous en citons trois qui nous seront importantes dans le schéma de construction de modèle.

Propriété 3.3.1 *Soient (S, L, T), $(S, L, T') \in \mathcal{TS}$ deux systèmes de transitions sur le même ensemble d'états et le même ensemble de labels, et $s, s', s'', s_1, s_2 \in S$ des états.*

1. *Le renommage d'un état par lui-même ne change pas la relation de transition.*

$$T\left[\frac{s}{s}\right] = T$$

2. *Le renommage d'états distincts est commutatif .*

$$s' \neq s'' \Rightarrow T\left[\frac{s_1}{s'}\right]\left[\frac{s_2}{s''}\right] = T\left[\frac{s_2}{s''}\right]\left[\frac{s_1}{s'}\right]$$

   Il est à noter que cette propriété reste valable pour $s_1 = s_2$.

3. *Le renommage dans une union de relations de transitions est égal à l'union de relations après renommage.*

$$(T \cup T')\left[\frac{s}{s'}\right] = T\left[\frac{s}{s'}\right] \cup T'\left[\frac{s}{s'}\right]$$

3.1.3 Contexte de synchronisation

Le mécanisme de gestion des objets partagés dans Java passe par l'utilisation des verrous, implicitement associés à chaque objet (qu'il soit un objet simple ou un thread). La modélisation des applications Java doit donc prendre en compte, de manière simple mais néanmoins fidèle, l'utilisation de ce mécanisme dans les programmes.

Les *contextes de synchronisation* modélisent les deux manières complémentaires de gestion d'un verrou :

Synchronisation Lorsqu'une instruction `synchronized` est rencontrée, il est nécessaire de représenter l'obtention du verrou au sein de cette instruction, et son relâchement à sa fin. Puisque les appels synchronisés peuvent être imbriqués (soit par des instructions syntaxiquement imbriquées, soit par le biais d'appels de méthodes qui contiennent des blocs synchronisés), il est possible d'avoir plusieurs objets sur lesquels un état est synchronisé.

Attente / Notification Lorsque après avoir obtenu un objet partagé, il n'est pas possible de l'exploiter (typiquement quand son état ne correspond pas à la logique des calculs à effectuer), il est possible de le libérer afin d'en accorder l'accès aux threads concurrents. C'est le but de la méthode `wait()`. Il est à noter qu'à chaque état, il n'est possible d'attendre une notification que sur un unique objet.

Ces observations donnent lieu à une structure rattachée à chaque état du système de transition : un contexte de synchronisation modélise d'une part l'ensemble des objets sur lesquels porte une synchronisation, et d'autre part l'éventuel unique objet sur lequel une attente a lieu.

Définition 3.4 *(Contexte de Synchronisation)*

Soit $(S, L, T) \in \mathcal{TS}$. *Un* contexte de synchronisation *est une fonction* Σ *qui associe à chaque état* $s \in S$ *du système de transition un couple* (Σ^+, Σ^-) *: à l'état* s, Σ^+ *représente l'ensemble des objets verrouillés ;* Σ^- *est l'ensemble contenant au plus un objet sur lequel se fait une attente de notification et qui, par conséquent, ne peut être verrouillé par aucun thread possédant le contrôle.*

$$\Sigma : S \to \mathcal{P}(\Omega) \times \mathcal{P}(\Omega)$$

On pourra, pour simplifier l'écriture d'un contexte dans un état $s \in S$ donné, écrire $(\Sigma^+(s), \Sigma^-(s))$ ou $(\Sigma^+, \Sigma^-)(s)$ suivant l'usage conventionnel, pour désigner le contexte propre à l'état s. Le contexte de synchronisation *vide*, noté $\Sigma_\emptyset$ est le contexte qui associe à chaque état de S le couple $(\emptyset, \emptyset)$: $\forall s \in S \,.\, \Sigma_\emptyset(s) = (\emptyset, \emptyset)$.

Durant la construction récursive du modèle de l'application, en particulier lors de synchronisations imbriquées, il est nécessaire d'ajouter successivement des verrous à un contexte de synchronisation. Cet ajout doit se faire en respectant la sémantique de chacun des ensembles constituant le contexte de synchronisation : en effet, à chaque point du programme Java, il n'est pas possible de se synchroniser sur un objet qui est en attente. Ainsi, l'opération d'addition d'un ensemble de ressources n'ajoute que les objets qui ne sont pas en attente de notification.

Définition 3.5 *(Ajout d'objets partagés)*

Soient $(S, L, T) \in \mathcal{TS}$ *un système de transition,* $s \in S$ *un état du système et* $K \subset \Omega$ *un ensemble d'objets partagés et* Σ *un contexte de synchronisation. L'ajout par l'opérateur add de* K *à un contexte* Σ *est le nouveau contexte* $\Sigma' = \Sigma \, add \, K$ *défini pour tout* $s \in S$ *par :*

$$(\Sigma'^+, \Sigma'^-)(s) = (\Sigma^+(s) \cup (K \setminus \Sigma^-(s)), \Sigma^-(s))$$

Cette opération prend son sens lors du raffinement des modèles par réécriture que nous détaillons dans la suite. L'exemple 3.2 montre l'utilité de cette opération dans le raffinement des modèles.

L'opérateur *add* est insensible à l'ordre choisi pour l'ajout d'objets à un contexte de synchronisation.

Propriété 3.5.1 *Soit $K_1, K_2 \subseteq \Omega$ et Σ un contexte de synchronisation. On a*

$$(\Sigma \, add \, K_1) \; add \; K_2 = (\Sigma \, add \, K_2) \; add \; K_1 = \Sigma \; add \; (K_1 \cup K_2)$$

3.1.4 Modèle

Les modèles que nous utilisons pour modéliser les instructions sont basés sur les systèmes de transition. Chaque état du système de transition correspond à une abstraction d'un état de l'application et chaque transition représente l'exécution d'une ou plusieurs instructions transformant un état en un autre. Le contexte de synchronisation permet d'attacher une information aux états du modèle sur les objets partagés utilisés.

Définition 3.6 *(Modèle)*

Un modèle *est un sextuplet $(S, L, T, s_b, s_e, \Sigma) \in \mathcal{M}$ composé des éléments suivants :*

- *(S, L, T) forme un système de transition sur l'ensemble d'états S, de labels L et défini par les transitions de T ;*
- *$s_b, s_e \in S$ sont des états respectivement de* début *et de* fin *;*
- *$\Sigma : S \to \mathcal{P}(\Omega) \times \Omega$ est un contexte de synchronisation.*

Par commodité d'écriture, lorsqu'il n'est pas utile de préciser le contenu du système de transition, on pourra noter un modèle $(TS, s_b, s_e, \Sigma) \in \mathcal{M}$.

Un modèle est dit *valide* si l'état de début n'admet aucune transition entrante et l'état de fin aucune transition sortante (c'est-à-dire si $ingoing(s_b) = outgoing(s_e) = \emptyset$) ; et si le contexte de synchronisation est vide pour les états de début et de fin (c'est-à-dire si $\Sigma(s_b) = \Sigma(s_e) = (\emptyset, \emptyset)$).

Exemple 3.1 *La figure 3.2 illustre un exemple de modèle où l'état $s_b = s_1$ est l'état de début et $s_e = s_5$ est l'état de fin. L'ensemble de labels est $L = \{!File, ?File, timeout\,\langle 20\rangle$, `open() ;`, `write() ;`, `checkAccess() ;`, `report() ;`, `close() ;`, $\tau\}$. La fonction Σ est définie comme suit :*

- *$\Sigma = \Sigma_\emptyset$ sur $\{s_1, s_2, s_3, s_4, s_5, s_6, s_{23}, s_{24}\}$,*
- *$\Sigma(s) = (\{File\}, \emptyset)$ pour $s \in \{s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{16}, s_{17}, s_{18}, s_{19}, s_{20}, s_{21}, s_{22}, s_{23}\}$,*
- *$\Sigma(s) = (\emptyset, \{File\})$ pour $s \in \{s_{14}, s_{15}\}$.*

Le procédé de construction de modèles à partir du source Java suit une définition sémantique et modulaire. Nous présentons deux opérations sur lesquels repose cette construction.

La première opération est la concaténation de modèles : étant donné deux modèles, leur concaténation branche les deux modèles en faisant la connexion entre l'état de fin du premier et l'état de début du second. Cette opération permet de construire, à partir de deux modèles, un modèle représentant la mise en séquence des éléments modélisés.

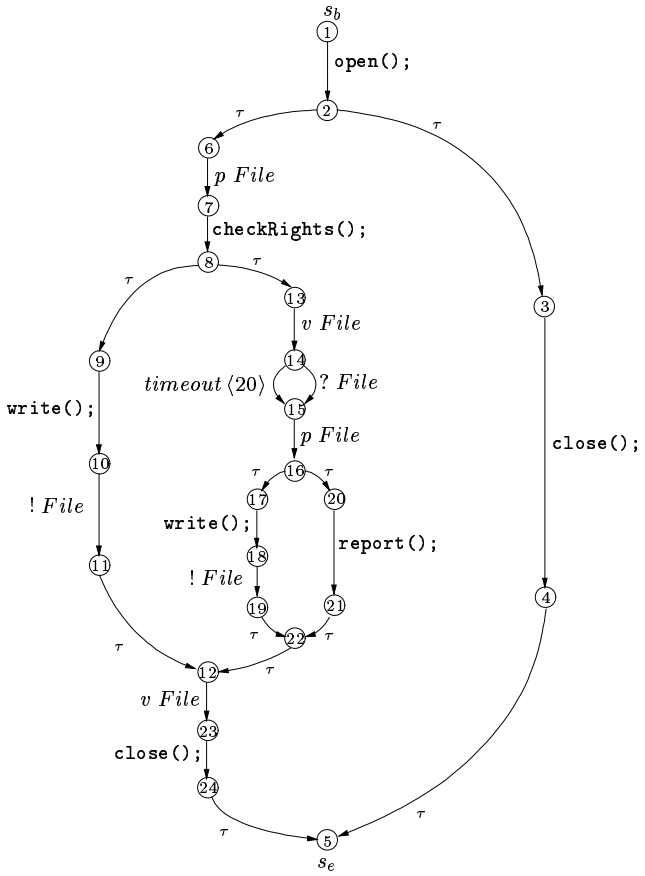


FIG. 3.2 – Exemple d'un modèle.

Définition 3.7 *(Concaténation de modèles)*

Soient $M_1 = (S_1, L_1, T_1, s_b^1, s_e^1, \Sigma_1)$ *et* $M_2 = (S_2, L_2, T_2, s_b^2, s_e^2, \Sigma_2) \in \mathcal{M}$ *deux modèles. La concaténation* $M_1 \cdot M_2$ *est un modèle* $M = (S, L, T, s_b, s_e, \Sigma) \in \mathcal{M}$ *défini à partir de* M_1 *et* M_2 *par*

- *L'ensemble des états est l'union des états précédents, sans l'état de début de* M_2 *; l'état de début* s_e *est l'état de début de* M_1*, l'état de fin* s_e *celui de* M_2 *:*

$$S = (S_1 \cup S_2) \backslash \{s_b^2\} \quad s_b = s_b^1 \quad s_e = s_e^2$$

- *L'ensemble des labels est l'union des ensembles de labels de chaque modèle*

$$L = L_1 \cup L_2$$

- *La relation de transition est définie comme l'union des relations de transitions de chaque*

modèle dans laquelle on renomme l'état s_b^2 de début de M_2 par l'état s_e^1 de fin de M_1 :

$$T = (T_1 \cup T_2)\left[\frac{s_e^1}{s_b^2}\right]$$

– *Le contexte de synchronisation Σ coïncide avec Σ_1 sur les états S_1 de M_1 et avec Σ_2 sur les états S_2 de M_2 :*

$$\Sigma(s) = \begin{cases} \Sigma_1 & si\ s \in S_1 \\ \Sigma_2 & si\ s \in S_2 \backslash \{s_b^2\} \end{cases}$$

La concaténation à partir de modèles valides produit toujours des modèles valides. Cette opération sera utilisée dans la suite pour modéliser une séquence d'instructions. La figure 3.3 montre un exemple de concaténation de modèles.

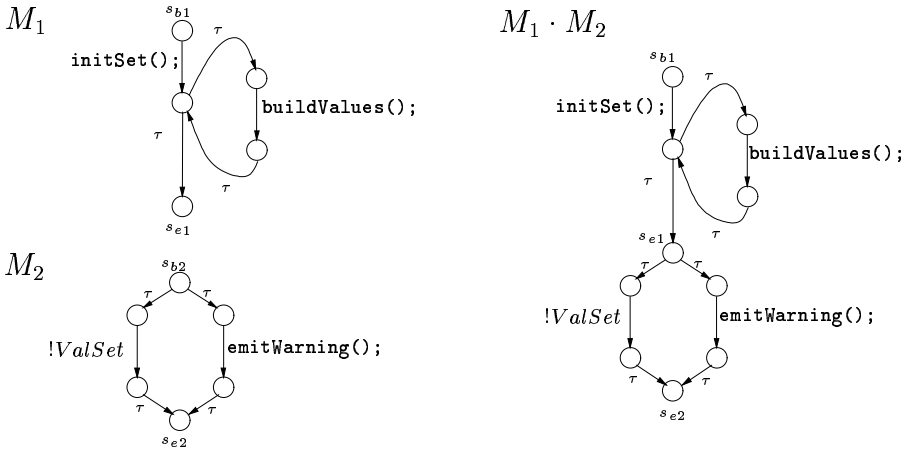


FIG. 3.3 – Exemple de concaténation de modèles.

La deuxième opération est la réécriture dans un modèle M d'une transition t par un modèle M'. Il s'agit de remplacer dans M la transition t en connectant en son lieu le modèle M'. Cette opération permet de raffiner un modèle lorsqu'une transition peut être vue comme une macro-transition qu'il est nécessaire à un moment d'expliciter.

Définition 3.8 *(Réécriture d'une transition par un modèle)*

Soient deux modèles $M = (S, L, T, s_b, s_e, \Sigma)$ et $M' = (S', L', T', s'_b, s'_e, \Sigma') \in \mathcal{M}$ valides et $t = (s_1, l, s_2) \in T$ une transition de M. La réécriture de t par M' dans M est le nouveau modèle $M'' = (S'', L'', T'', s''_b, s''_e, \Sigma'') \in \mathcal{M}$, noté $M'' = M\left[\frac{M'}{t}\right]$ et défini par

– *L'ensemble des états S'' est l'union des états de M et M' sans les états s_1 et s_2. Les états de debut et de fin, respectivement s''_b et s''_e, sont ceux de M.*

$$S'' = (S \cup S')\backslash\{s_1, s_2\} \quad s''_b = s_b \quad s''_e = s_e$$

- *La relation de transition T'' est l'union des relations de transition $T\backslash\{t\}$ et de T' dans laquelle on renomme s_1 par s'_b et s_2 par s'_e.*

$$T'' = ((T\backslash\{t\}) \cup T') \left[\frac{s'_b}{s_1}\right] \left[\frac{s'_e}{s_2}\right]$$

- *Le nouveau contexte de synchronisation Σ'' est défini à partir des contextes de synchronisation de M et M' de la manière suivante :*

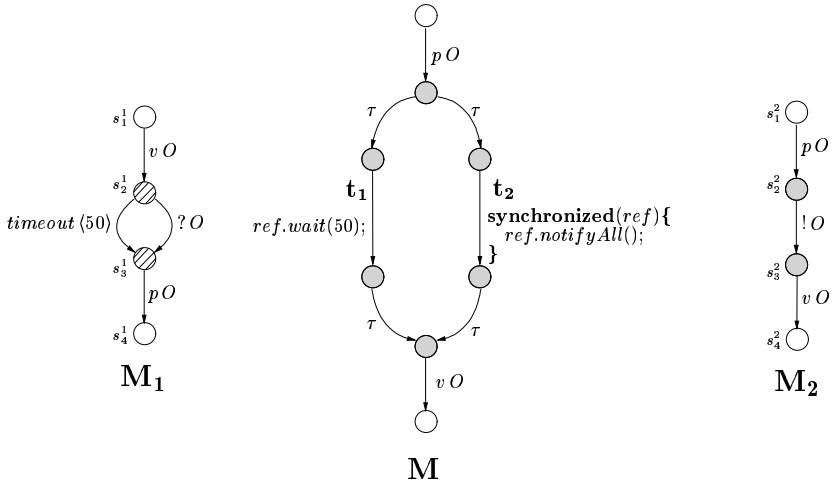
$$\Sigma''(s) = \begin{cases} \Sigma(s) & si\ s \in S\backslash\{s_1, s_2\} \\ \Sigma'(s)\, add\, \Sigma^+(s_1) & si\ s \in S' \end{cases}$$

Il est facile de remarquer que la réécriture d'une transition d'un modèle valide par un modèle valide produit un modèle valide. En effet d'après la construction, l'état de départ (respectivement d'arrivée) du modèle réécrit est un état de départ (respectivement d'arrivée) d'un modèle valide ; et le contexte de synchronisation de ces mêmes états ne change pas durant la construction : le modèle final est donc valide. L'exemple 3.2 illustre l'opération de réécriture de transition.

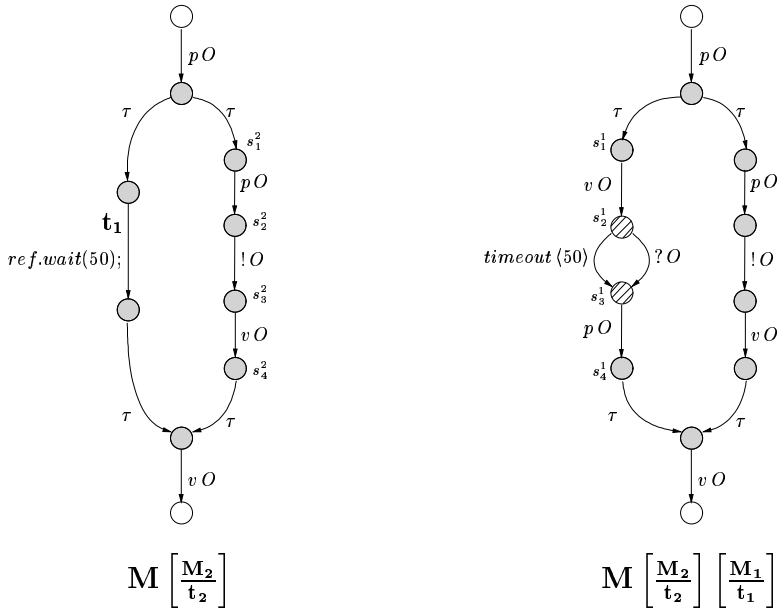
Exemple 3.2 *Considérons les modèles M, M_1 et M_2 illustrés à la figure 3.4, (a). Le contexte de synchronisation est représenté sur le schéma de la manière suivante : les états grisés possèdent le verrou de l'objet O : si s est un tel état et $\Sigma = (\Sigma^+, \Sigma^-)$ le contexte de synchronisation du modèle contenant s, alors $\Sigma^+(s) = \{O\}$ et $\Sigma^- = \emptyset$. Les états hachurés sont ceux où une attente de notification se fait sur O, la mémorisation de cette information se fait par le biais du Σ^- : pour s un tel état, on a $\Sigma^+(s) = \emptyset$ et $\Sigma^-(s) = \{O\}$. Les autres états ont un contexte de synchronisation vide.*

Considérons la réécriture de la transition t_2 par M_2 dans M. L'état dont est issu t_2 est synchronisé sur O. Lors de la réécriture, l'objet O est ajouté, via l'opérateur add, au contexte de synchronisation sur les états de M_2. Il en résulte que les états s_1^2 et s_4^2 sont synchronisés sur O dans $M\left[\frac{M_2}{t_2}\right]$ (grisés sur le schéma) ; d'autres parts, la synchronisation imbriquée sur O en s_2^2 et s_3^2 ne change pas le contexte de synchronisation dans ces états.

La réécriture de la transition t_1 dans le modèle résultant par M_1 est tout aussi intéressante : l'objet O est exclu de synchronisation sur les états s_2^1 et s_3^1 (à cause d'une attente de notification dans ces états). Lors de la réécriture, comme dans le premier cas, l'objet O est ajouté au contexte de synchronisation sur les états de M_1 par add. L'objet est effectivement ajouté aux états s_1^1 et s_4^1, mais il est "absorbé" par Σ_1^- (contexte de synchronisation de M_1) sur les états s_2^1 et s_3^1. De plus, ces états conserverons les valeurs de leurs contextes de synchronisation dans le modèle final. Cette réécriture montre l'utilité de l'opérateur add dans la réécriture, notamment lors d'une construction modulaire des modèles.



(a) Modèles M, M_1 et M_2.



(b) Réécriture successive des transitions t_1 et t_2 du modèle M.

FIG. 3.4 – Exemple de réécriture de transition. En (a) sont représentés les modèles M, M_1 et M_2 ; les réécritures $M\left[\frac{M_2}{t_2}\right]$ et $M\left[\frac{M_2}{t_2}\right]\left[\frac{M_1}{t_1}\right]$ sont représentées en (b). Un état s est représenté : grisé si $\Sigma(s) = (\{O\}, \emptyset)$ et hachuré si $\Sigma(s) = (\emptyset, \{O\})$. Les états sans décoration ont des contextes de synchronisation vides.

La réécriture d'une transition par un modèle possède deux propriétés intéressantes. La première établit le fait que la réécriture dans un modèle de deux transitions distinctes donne lieu au même modèle indépendemment de l'ordre dans lequel elle est appliquée. La seconde propriété établit le fait que la réécriture d'une transition par un modèle où l'on réécrit une autre transition par un troisième modèle, donne le même résultat. D'une manière informelle, ceci montre que la réécriture de transition est indépendante du sens dans lequel elle est appliquée, que l'on commence du bas vers le haut ou de haut en bas.

Proposition 3.1 *(Propriétés de la réécriture de transitions)*
Soient $(S_i, L_i, T_i, s_b^i, s_e^i, \Sigma_i) \in \mathcal{M}$ *trois modèles (avec* $i = 1, \ldots, 3$*), et soient* $t_1, t_1' \in T_1$ *et* $t_2 \in T_2$ *des transitions sur les différents modèles.*

1. *La réécriture de deux transitions* distinctes *d'un même modèle est indépendante de l'ordre dans lequel elle est appliquée.*

$$M_1 \left[\frac{M_2}{t_1}\right] \left[\frac{M_3}{t_1'}\right] = M_1 \left[\frac{M_3}{t_1'}\right] \left[\frac{M_2}{t_1}\right]$$

2. *La réécriture successive d'une transition d'un modèle peut se faire soit en réécrivant la transition par un modèle réécrit, soit en réécrivant la transition, puis en réécrivant le modèle obtenu.*

$$M_1 \left[\frac{M_2 \left[\frac{M_3}{t_2}\right]}{t_1}\right] = M_1 \left[\frac{M_2}{t_1}\right] \left[\frac{M_3}{t_2}\right]$$

Preuve:
Il suffit de calculer le système de transition et la fonction de contexte de synchronisation des deux modèles de part et d'autre de l'égalité. L'identification terme à terme des deux modèles est immédiate en utilisant la propriété 3.5.1 de la commutativité de l'opérateur *add*, et la propriété 3.3.1 du renommage d'état dans l'union de deux relations de transition.

□

3.2 Construction de modèle

Nous passons à la construction effective des modèles à partir de la donnée d'une instruction d'un programme Java. Cette construction est présentée sous la forme d'un *système de réécriture* sur les modèles introduits précédemment.

Nous présentons dans un premier temps la notion d'environnement statique, qui introduit en sus du modèle comportemental et pour l'ensemble du programme considéré, des informations supplémentaires sur sa structure et sa sémantique qui ne sont pas représentées de manière syntaxique. Ensuite, nous associons aux instructions du langage des modèles sémantiques qui,

associés à la notion d'observabilité, permettent de mettre en place un système de réécriture sur les modèles. Ce dernier possède les propriétés nécessaires pour définir, d'une manière unique, le modèle d'une instruction donnée.

3.2.1 Environnement statique

La modélisation que nous proposons se base sur des informations qui ne sont pas totalement contenues dans la syntaxe du programme. Nous avons besoin d'associer aux instructions d'un programme donné des informations *dynamiques* liées à des instructions particulières de la syntaxe. Ces informations sont données par l'utilisateur ou, quand cela est possible, calculées statiquement à partir du contexte des instructions dans le programme.

Ces informations sont données sous forme de fonctions qui associent à une instruction du programme une donnée typée. Bien évidemment, ces fonctions sont partielles, car elles ne concernent qu'un sous-ensemble d'instructions de la syntaxe.

Nombre maximal d'itérations La fonction $iterations : Stm \to \mathbb{N}$, définie pour les instructions itératives, renvoie le nombre d'itérations maximal qu'il est possible d'y effectuer. Cette information est nécessaire pour la construction statique de modèle, notamment pour borner les temps d'exécution.

Délais d'expiration d'attente La fonction $timeout : Stm \to \mathbb{N}$, définie pour les instructions d'appel de méthode correspondant à la mise en attente de notification (invocation de la méthode `wait()`), renvoie le délai maximal d'attente. Ce délai correspond à l'évaluation du paramètre spécifié dans l'instruction d'appel de la méthode. D'après les restrictions établies dans le sous-ensemble de Java considéré (*cf.* paragraphe 2.3.3), les délais d'attente sont fixés et ne dépendent pas des données du programme. Afin de faciliter la détermination des délais d'une manière statique, il sont fixés à des littéraux par la syntaxe et peuvent donc être extraits à partir du code.

Objets référencés Afin de pouvoir effectuer une construction statique de modèle à partir du code source, nous avons besoin de déterminer statiquement les objets référencés par certaines expressions de type référence (règle de production $RefExpr$ dans la syntaxe). Conformément à la restriction avancée dans le chapitre 2 paragraphe 2.3.3, d'une part chaque objet partagé est un singleton (instance unique de sa classe) et d'autre part les instructions de synchronisation et d'attente et notification ne sont pas autorisées à être effectuées sur des expressions ayant comme type la racine de la hiérarchie des classes du langage Java (à savoir `java.lang.Object`). Il est donc possible de déterminer statiquement, et d'une manière exacte, l'objet partagé sur lequel est appliquée une instruction synchronisée à partir de l'expression utilisée, en examinant simplement son type. Nous définissons la fonction partielle $object : RefExpr \to \Omega$ permettant d'identifier les objets partagés sur lesquels s'opèrent les instructions sus-mentionnées.

Temps d'exécution La fonction partielle $exec : MethodDecl \rightarrow \mathbb{N} \times \mathbb{N}$ associe à certaines méthodes du programme les limites inférieures et supérieures (dans cet ordre) bornant leurs temps d'exécutions. Les méthodes sur lesquelles la fonction *exec* est définie sont appelées *méthodes de calcul* ; elles ont une sémantique particulière, détaillée au paragraphe 3.2.2.

Conformément aux sous-ensemble du langage Java considéré (*cf.* paragraphe 2.3.2), la redéfinition de méthode est interdite ; seule la surcharge est autorisée. Il est donc possible de déterminer statiquement la déclaration d'une méthode à partir de sa signature (l'ensemble des signatures et des déclarations de méthode sont en bijection dans notre cadre de travail). La fonction *exec* sera donc appliquée indifféremment aux déclarations et aux signatures des méthodes. Afin de simplifier la présentation, nous confondons, quand le contexte le permet, signature et nom de méthode. La fonction *exec* peut donc être définie par $exec : MethodId \rightarrow \mathbb{N} \times \mathbb{N}$.

L'ensemble de ces fonctions compose un *environnement statique* rattaché au programme qu'il s'agit de modéliser.

Définition 3.9 *(Environnement statique)*

On appellera environnement statique $\sigma \in Env$ *d'un programme, le quadruplet de fonctions* $(iterations, timeout, object, exec)$ *définies sur les instructions du programme.*

3.2.2 Modèles sémantiques des instructions

Nous introduisons la fonction $smodel : Stm \rightarrow \mathcal{M}$ qui définit les *modèles sémantiques* des instructions du langage. Le modèle sémantique représente la relation entre l'état initial et l'état final de l'exécution d'une instruction et est construit à partir des *constituants immédiats* de l'instruction. Ces derniers peuvent être des constituants syntaxiques (dans ce cas ce sont alors des instructions Java conformes à la syntaxe de la table 2.1) et/ou des primitives de la machine virtuelle qui ne sont pas présentes au niveau syntaxique mais qui font partie de la sémantique de certaines instructions (par exemple le verrouillage des objets). La définition de la fonction *smodel* se fait de manière structurelle, c'est-à-dire par énumération des différents cas possibles d'instructions.

Nous commençons par introduire un *modèle élémentaire* utilisé dans la définition des modèles sémantiques. Etant donné un label $l \in \mathcal{L}$, son modèle élémentaire est simplement constitué d'une unique transition entre l'état de début et de fin, étiquetée par l.

Définition 3.10 *(Modèle élémentaire)*

La fonction $\mu : \mathcal{L} \rightarrow \mathcal{M}$ *associe à chaque label* $l \in \mathcal{L}$ *un modèle élémentaire* $M \in \mathcal{M}$ *composé des seuls états de début et de fin dont les contextes de synchronisation sont vides, et*

d'une unique transition entre ces états, décorée par l.

$$\mu(l) = (\{s_b, s_e\}, \{l\}, \{(s_b, l, s_e)\}, \Sigma_\emptyset)$$

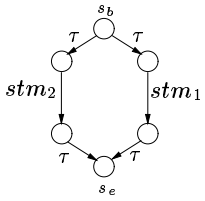
En particulier, un modèle élémentaire dont le label est τ sera appelé τ-modèle.

Les instructions non-composées se voient associer un modèle élémentaire dont le label correspond à l'instruction. Pour une instruction non-composée *stm* le modèle sémantique est défini par $smodel(stm) = \mu(stm)$. Pour les instructions composées, telles que les appels de méthode et les blocs d'instructions, la construction est un peu plus compliquée. Les modèles sémantiques associés sont énumérés dans les paragraphes qui suivent. Nous donnerons, pour illustrer les réécritures, une schématisation de la construction des modèles associés aux différentes instructions composées.

Instruction conditionnelle La sémantique de l'instruction conditionnelle est traduite par un choix non-déterministe suivant la valeur de la condition.

$smodel(if(b)\ stm_1\ else\ stm_2) = (TS, s_b, s_e, \Sigma)$

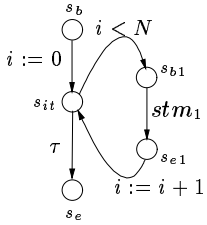
- $S = \{s_b, s_e, s_{b1}, s_{e1}, s_{b2}, s_{e2}\}$
- $T = \{s_b \xrightarrow{\tau} s_{b1},\ s_{b1} \xrightarrow{stm_1} s_{e1},\ s_{e1} \xrightarrow{\tau} s_e,\ s_b \xrightarrow{\tau} s_{b2},\ s_{b2} \xrightarrow{stm_2} s_{e2},\ s_{e2} \xrightarrow{\tau} s_e\}$
- $\Sigma = \Sigma_\emptyset$



Instruction itérative Afin de définir le modèle sémantique des instructions de boucle, nous utilisons la fonction de l'environnement statique *iterations*. Soit N le nombre maximal d'itérations indiqué par la fonction *iterations*. Afin d'obtenir une représentation compacte, le modèle sémantique est défini à l'aide d'une macro-notation en utilisant une variable entière $i \in \{1 \ldots N\}$ qui décrit l'ensemble d'itération de la boucle.

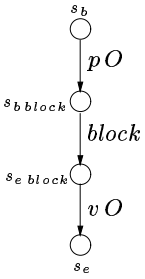
$smodel(\ while(b)\ stm_1\ ) = (TS, s_b, s_e, \Sigma)$

- $S = \{s_b, s_e, s_{b1}, s_{e1}, s_{it}\}$
- $T = \{s_b \xrightarrow{i:=0} s_{it},\ s_{it} \xrightarrow{i<N} s_{b1},\ s_{b1} \xrightarrow{stm_1} s_{e1},\ s_{e1} \xrightarrow{i:=i+1} s_{it},\ s_{it} \xrightarrow{\tau} s_e\}$
- $\Sigma = \Sigma_\emptyset$



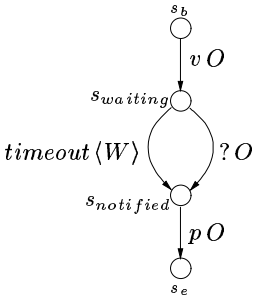
Instruction de synchronisation Une instruction de synchronisation $stm = synchronized(ref)\ block$ permet d'exécuter le bloc d'instructions *block* avec l'objet référence par *ref* verrouillé. Cet objet est déterminé à l'aide de l'environnement statique par $O = object(ref)$. Le modèle sémantique représente les actions de verrouillage et de déverrouillage, respectivement, par les transitions étiquetées par $p\,O$ et $v\,O$. La fonction de contexte de synchronisation Σ associe l'objet O aux états de début et de fin de la transition étiquetée par le bloc d'instruction.

$smodel(\ synchronized(ref)\ block\ ) = (TS, s_b, s_e, \Sigma)$
- $S = \{s_b, s_e, s_{b\,block}, s_{e\,block}\}$
- $T = \{s_b \xrightarrow{p\,O} s_{b\,block}, s_{b\,block} \xrightarrow{block} s_{e\,block}, s_{e\,block} \xrightarrow{v\,O} s_e\}$
- $\Sigma(s_{b\,block}) = \Sigma(s_{e\,block}) = (\{O\}, \emptyset),\ \Sigma(s_b) = \Sigma(s_e) = (\emptyset, \emptyset)$



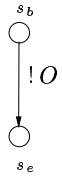
Instruction de mise en attente Une instruction de mise en attente de notification $stm = ref.wait(litt)$; est caractérisée par l'objet partagé sur lequel se fait l'attente et le délai d'expiration. Ces informations sont déterminées, comme pour l'instruction `synchronized`, à l'aide de l'environnement statique. Soit $O = object(ref)$ et $W = timeout(stm)$. Le modèle sémantique détaille les actions suivantes : le verrou de l'objet O est relâché pour aboutir à l'état d'attente $s_{waiting}$. De cet état, deux actions possibles amènent à l'état *notifié* $s_{notified}$: un dépassement de délai W modélisé par la transition étiquetée $timeout\,\langle W \rangle$, ou une notification qui vient se synchroniser avec la transition étiquetée par $?\,O$ (la modélisation de la communication entre les threads est détaillée au chapitre 4). L'exécution se termine par l'acquisition de nouveau du verrou de l'objet via la transition étiquetée par $p\,O$.

$smodel(\ ref.wait(litt);\ ) = (TS, s_b, s_e, \Sigma)$
- $S = \{s_b, s_e, s_{waiting}, s_{notified}\}$
- $T = \{s_b \xrightarrow{v\,O} s_{waiting}, s_{waiting} \xrightarrow{?\,O} s_{notified}, s_{waiting} \xrightarrow{timeout\langle W \rangle} s_{notified}, s_{notified} \xrightarrow{p\,O} s_e\}$
- $\Sigma(s_{waiting}) = \Sigma(s_{notified}) = (\emptyset, \{O\}),\ \Sigma(s_b) = \Sigma(s_e) = (\emptyset, \emptyset)$



Instruction de notification La notification se modélise simplement par une transition portant le label $!O$, où $O = object(ref)$. Cette action se synchronise avec toutes les attentes de notification (*cf.* instruction de mise en attente) qui sont actives à ce moment.

$smodel(\ ref.notifyAll();\ ) = (TS, s_b, s_e, \Sigma)$
- $S = \{s_b, s_e\}$
- $T = \{s_b \xrightarrow{!O} s_e\}$
- $\Sigma = \Sigma_\emptyset$



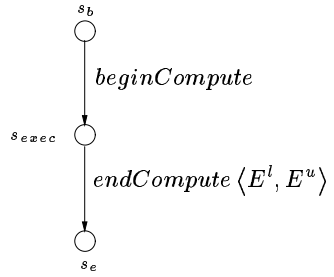
Bloc d'instructions Les blocs d'instructions peuvent soit être vides, soit contenir une ou plusieurs instructions. Le modèle d'un bloc d'instruction est la concaténation des modèles élémentaires formés par les instructions qui constituent le bloc. Les modèles sémantiques relatifs à chaque cas sont les suivants :

- $smodel(\{\ \}) = \mu(\tau)$
- $smodel(\{stm\}) = \mu(stm)$
- $smodel(\{stm_1 ... stm_k\}) = \mu(stm_1) \cdot ... \cdot \mu(stm_k)$ pour $k \geq 2$

Instructions d'appel de méthode Les instructions d'appel de méthode sont celles conformes à la règle de production *MethodCall* de notre syntaxe. Nous distinguons les méthodes dites *méthodes de calcul*. Ces méthodes sont assimilées à leurs temps d'exécution. Le modèle sémantique de l'appel d'une méthode de calcul se réduit au temps d'exécution requis pour traiter la totalité des instructions de son corps. Cette catégorie de méthodes est identifiée par la fonction partielle $exec : MethodId \rightarrow \mathbb{N} \times \mathbb{N}$ qui associe aux méthodes de calcul les bornes supérieure et inférieure du leurs temps d'exécution. Les autres invocations de méthode ont pour modèle sémantique celui du corps de leurs déclarations. Le modèle sémantique d'une instruction d'appel de méthode $ref.methodId()$; est donné suivant deux cas :

1. La fonction *exec* est définie pour *mehodId* et $exec(methodId) = (E^l, E^u)$. Le modèle sémantique comprend une action de début et de fin de calcul, respectivement *beginCompute* et $endCompute\,\langle E^l, E^u\rangle$. L'état s_{exec} correspond à l'état où la méthode est exécutée .

   $smodel(\,ref.methodId();\,) = (TS, s_b, s_e, \Sigma)$
   - $S = \{s_b, s_{exec}, s_e\}$
   - $T = \{s_b \xrightarrow{beginCompute} s_{exec},\ s_{exec} \xrightarrow{endCompute\langle E^l, E^u\rangle} s_e\}$
   - $\Sigma = \Sigma_\emptyset$

2. La fonction *exec* n'est pas définie en *methodId*. Dans ce cas, le modèle sémantique de l'invocation de la méthode est celui du bloc d'instructions qui constitue son corps. L'instruction **return** à la fin de la liste d'instructions du corps de la méthode n'est pas prise en compte : l'affectation potentielle de la valeur de retour n'est pas représentée. Le modèle sémantique est donc défini par $smodel(ref.methodId();) = smodel(Body_{methodId})$ où $Body_{methodId}$ est le corps de la déclaration de la méthode (déterminé statiquement grâce à l'interdiction de la redéfinition de méthode).

Il est à noter que les modèles sémantiques ainsi construits pour les instructions composées sont des modèles valides.

3.2.3 Prédicats d'observabilité

La modélisation est basée sur la notion d'*observabilité* définie sur les actions du programme. En effet, les modèles issus d'applications réelles souffrent toujours de l'explosion combinatoire : la nature même des programmes Java mène à des modèles conséquents, même abstraits. L'idée est donc d'établir un certain nombre d'actions (instructions particulières du programme, actions de synchronisation ou pas de calcul) que l'on souhaite conserver (observer), et qui seront donc représentées dans le modèle. Les autres instructions, non pertinentes pour les propriétés

que l'on souhaite démontrer sur le programme, sont simplement écartés et ne figurent pas dans le modèle. L'observabilité est ainsi un paramètre de la modélisation de l'application, qui sert à moduler sa construction.

Nous définissons le prédicat *obs* sur l'ensemble des labels $\mathcal{L}$ afin de désigner les instructions et actions du programme que l'on désire observer. Ce prédicat caractérise les actions dont il est nécessaire de garder trace dans un modèle.

Exemple 3.3 *Afin d'observer les opérations de verrouillage/déverrouillage des objets, le prédicat obs est défini tel que* $obs(l) = vrai$ *pour tout* $l \in \mathcal{L}_{sync}$. *Un autre exemple est celui d'observer les appels de certaines méthodes du programme. Ainsi, pour observer par exemple les appels des méthodes* `write()`, `open()` *et* `close()` *d'un programme, il suffit de définir obs tel que* $obs(rexp.methodId();) = vrai$ *pour* $methodId \in \{$`write`, `open`, `close`$\}$[1].

Il est important de déterminer si l'exécution d'une instruction composée, qui n'est pas elle même observable, entraîne l'exécution d'une action observable. Nous définissons le prédicat *hasObs* afin de caractériser les instructions composées dont l'exécution fait intervenir des actions observables.

Définition 3.11 *(Prédicat hasObs)*

Soient $\mathcal{L}$ *un ensemble de labels et obs un prédicat sur* $\mathcal{L}$. *Le prédicat hasObs définit sur* $\mathcal{L}$ *est vrai pour un label* $l \in \mathcal{L}$ *si et seulement si* l *correspond à une instruction composée et dont le modèle sémantique contient un label* l' *tel que* $obs(l')$ *ou* $hasObs(l')$.

$$hasObs(l) = \neg obs(l) \wedge (l \in \mathcal{L}^{c}_{stm} \wedge \bigvee_{l' \in smodel(l)} obs(l') \vee hasObs(l'))$$

D'une manière informelle, une instruction composée contient une action observable si elle n'est pas à son tour observable et si l'un des labels de son modèle sémantique est observable ou entraîne une action observable. L'exemple 3.4 illustre le calcul du prédicat *hasObs* sur une instruction d'un programme.

Exemple 3.4 *Considérons le code donné dans la table 3.1. Pour cet exemple adoptons le prédicat d'observabilité défini par* $obs(l) = vrai$ *pour* $l \in \mathcal{L}_{sync}$ *et* $obs(l) = faux$ *pour tout autre* $l \in \mathcal{L}$. *Soit* stm_i *l'instruction figurant à la i-ème ligne. Intuitivement, l'instruction* stm_7 *– correspondant à la structure de contrôle* `while`*– comprend un appel à la méthode* `update()` *de* `FilterBuffer` *qui comprend elle-même une instruction synchronized à la ligne 22, donc une action observable. Ceci est conforme au résultat obtenu en évaluant* $hasObs(stm_7)$. *En effet, en appliquant la définition de hasObs, nous obtenons après une première simplification :* $hasObs(stm_7) = hasObs(stm_9) \vee hasObs(stm_{10}) \vee hasObs(stm_{11})$. *D'autre part,* $hasObs(stm_{10}) =$

[1] En réalité, l'observabilité d'une méthode particulière se base sur sa signature et non uniquement sur son identificateur. Cet abus de notation est utilisé pour simplifier l'exposé.

```
1  class Transfer {
2    void actuate(InBuffer inBuffer,
3             FilterBuffer fBuffer){
4      int i, MAX; float x;
5      MAX=inBuffer.size();
6      i=0;
7      while(i<MAX){
9         x = inBuffer.get(i);
10        fBuffer.update(x);
11        i++;
12     }
13 }

14 class InputBuffer {
...
15   void get(i){
16     return m_aValues[i];
17   }
18 }
```

```
19 class FilterBuffer {
...
20   void update(float x){
21     if(x>=FILTER_THRESHOLD){
22       synchronized(this) {
23         put(x);
24       }
25     } else;
26   }
27   void put(float val){
28     m_Val=val;
29   }
30 }
```

TAB. 3.1 – Extrait d'une application Java : filtre de données.

$hasObs(stm_{22})$. *L'instruction* stm_{22} *étant une instruction synchronized qui comprend dans son modèle sémantique (*cf. *section 3.2.2) les actions observables, selon obs, de verrouillage et déverrouillage de l'objet instance de* `FilterBuffer`. *Nous avons donc le résultat attendu :* $hasObs(stm_7) = hasObs(stm_{22}) = vrai$.

3.2.4 Réécriture de modèle

La construction du modèle d'une instruction est présentée sous la forme d'un système de réécriture : une première étape construit un modèle élémentaire qui est ensuite raffiné par réécriture suivant l'observabilité de l'instruction.

Nous commençons par présenter succinctement les systèmes de réécriture : les caractéristiques essentielles (confluence, terminaison et convergence) et les propriétés que nous utiliserons ensuite dans la construction. Ensuite, la convergence de la réécriture des modèles est établie : la modélisation est d'une part confluente (quelle que soit la manière dont on raffine le modèle, on obtient un modèle final identique) et d'autre part, elle termine (quelle que soit l'instruction de départ, il arrive un moment où il n'est plus possible de réécrire le modèle). La convergence assure alors que le modèle final obtenu par réécriture à partir du modèle élémentaire est unique quelle que soit la succession des réécritures choisies.

3.2.4.1 Rappels sur les systèmes de réécriture

Les systèmes de réécriture sont un moyen de formaliser un calcul par transformation progressive d'objets (qui peuvent être des formules, des graphes, des chaînes, *etc.*).

Un grand nombre de concepts et de propriétés des systèmes de réécriture peuvent être définis et étudiés indépendamment des objets qu'ils manipulent. Ceci présente l'avantage de séparer les propriétés abstraites du système de réécriture des propriétés relatives à la structure des objets manipulés.

Un système de réécriture est la donnée d'un ensemble muni d'une relation binaire. Deux éléments sont en relation s'il est possible de passer du premier au second par un pas de réécriture.

Définition 3.12 *(Système de réécriture)*
Soit A un ensemble fini quelconque. On appelle système de réécriture *le couple $(A, \Longrightarrow)$, où $\Longrightarrow \subseteq A \times A$ est une relation binaire sur A.*

On appellera dans ce contexte, un élément de A un *terme*. Nous noterons par commodité, un élément $(a, a') \in \Longrightarrow$ par $a \Longrightarrow a'$.

Puisque $\Longrightarrow \subseteq A \times A$ est une relation binaire sur A, il est possible de définir l'application itérative de $\Longrightarrow$ sur elle-même (pour $n \in \mathbb{N}$) lorsque plusieurs éléments peuvent se réécrire en cascade :

Réécriture $\Longrightarrow^n$ en n pas Pour $a, b \in A$, a se réécrit en b en n pas si $n \geq 1$ et il existe une suite de termes $\{a_i\}_{0 \leq i \leq n}$ telle que $a_0 = a$, $a_n = b$ et $a_i \Longrightarrow a_{i+1}$, $i = 0 \ldots n-1$, ou $n = 0$ et $a = b$. On notera $a \Longrightarrow^n b$.

Clôtures On définit les clôtures *réflexive* ($\Longrightarrow^=$) et *réflexive transitive* ($\Longrightarrow^*$) par
- $a \Longrightarrow^= b$ si $a = b \vee a \Longrightarrow b$
- $a \Longrightarrow^* b$ si $\exists k \in \mathbb{N} : a \Longrightarrow^k b$.

On dira qu'un terme a est en *forme normale* s'il ne peut pas se réécrire (*ie.* si $\nexists b \in A \,.\, a \Longrightarrow b$), et qu'il a une forme normale s'il peut se réécrire en une forme normale (*ie.* si $\exists b$ forme normale $. \, a \Longrightarrow^* b$).

La définition suivante caractérise les relations de réécriture d'après les propriétés qu'elles possèdent. Le lemme qui suit établit les liens entre ces différentes caractéristiques.

Définition 3.13 *(Terminaison et propriétés de confluence)*

Normalité $\Longrightarrow$ *est* normale *si chaque terme admet une forme normale.*

Terminaison $\Longrightarrow$ termine *s'il n'existe pas de terme qui peut se réécrire indéfiniment.*

Sous-commutativité $\Longrightarrow$ *est* sous-commutative *si pour chaque terme qui peut se réécrire en un pas soit en b, soit en c, il est possible de réécrire en un ou zéro pas b et c en un même terme unique d.*

$$si \; \forall a \in A, \exists b, c \in A : a \Longrightarrow b \wedge a \Longrightarrow c \; alors \; \exists d \in A : b \Longrightarrow^= d \wedge c \Longrightarrow^= d$$

Confluence $\Longrightarrow$ *est* confluente *si chaque terme a qui peut se réécrire en b ou en c, permet de réécrire b et c en un terme d.*

$$si\ \forall a \in A, \exists b, c \in A : a \Longrightarrow^* b \wedge a \Longrightarrow^* c\ alors\ \exists d \in A : b \Longrightarrow^* d \wedge c \Longrightarrow^* d$$

Convergente $\Longrightarrow$ *est* convergente *si elle termine et est confluente.*

Lemme 3.1 *(Caractéristiques des relations de réécriture)*

1. *Si* $\Longrightarrow$ termine *alors elle est* normale.
2. *Si* $\Longrightarrow$ *est sous-commutative alors elle est confluente.*
3. *Si* $\Longrightarrow$ *est confluente alors toute forme normale est unique.*

Preuve:

Nous indiquons quelques éléments de preuve :

1. Puisque $\Longrightarrow$ termine, tout terme se réécrit en une forme normale; donc chaque terme possède une forme normale.
2. La preuve se fait par une double induction : la première établit que si $\Longrightarrow$ est sous-commutative, tout terme $a \in A$ est caractérisé par si $a \Longrightarrow b \wedge a \Longrightarrow^* c$ alors $\exists d$: $b \Longrightarrow^* d \wedge c \Longrightarrow^* d$; la seconde établit le fait que cette propriété de la sous-commutativité implique la confluence.
3. Par l'absurde, supposons que $a \in A$ possède deux formes normales a_1, a_2. Puisque $\Longrightarrow$ est confluente, $\exists a', a'' : a_1 \Longrightarrow^* a' \wedge a_2 \Longrightarrow^* a''$. Puisque a_1, a_2 sont des formes normales, on a $a_1 = a'$ et $a_2 = a''$, d'où $a' = a''$.

□

3.2.4.2 Transformation des modèles par réécriture

Nous définissons un système de réécriture basé sur la réécriture de transition et paramétré par un prédicat d'observabilité. La relation de réécriture a pour but de raffiner (*ie.* réécrire) les transitions étiquetées par des instructions composées où *hasObs* est vrai. Le système de réécriture est noté $(\mathcal{M}, \Longrightarrow)_{obs}$ où *obs* est un prédicat d'observabilité. En gardant les notations habituelles sur les modèles, il vient :

Définition 3.14 *(Réécriture de modèle)*

Soit obs un prédicat d'observabilité. $(\mathcal{M}, \Longrightarrow)_{obs}$ *est le système de réécriture sur les modèles paramétré par obs tel que pour tout* $M, M' \in \mathcal{M}$ *et* T *la relation de transition de* M*, on a :*

$$M_1 \Longrightarrow M_2$$

ssi

$$\exists\ t = s \xrightarrow{l} s' \in T\ .\ hasObs(l) \wedge M' = M\left[\frac{smodel(l)}{t}\right]$$

Lorsque le prédicat d'observabilité est clairement défini, on simplifiera la notation du système de réécriture en $(\mathcal{M}, \Longrightarrow)$. D'après la définition de $hasObs$, la réécriture d'une transition étiquetée par $l \in \mathcal{L}$ a lieu quand il désigne une instruction composée qui contient strictement une action observable. La réécriture remplace la transition en question par le modèle sémantique associé à l'instruction composée.

La relation de réécriture implique seulement les instructions qui contiennent des actions observables : il est donc possible de réécrire de plusieurs manières et dans un ordre quelconque le modèle initial. Nous montrons que cette relation est convergente : d'une part elle termine, et d'autre part, elle est confluente, ce qui signifie que l'on peut effectuer les réécritures dans n'importe quel ordre en aboutissant toujours au même modèle normal.

Lemme 3.2 *(Sous-commutativité)*
La relation de réécriture $\Longrightarrow$ est sous-commutative.

Preuve:

Soit $M, M_1, M_2 \in \mathcal{M}$ tels que $M \Longrightarrow M_1$ et $M \Longrightarrow M_2$. Il s'agit alors de démontrer qu'il existe $M' \in \mathcal{M}$ qui vérifie $M_1 \Longrightarrow^{=} M'$ et $M_2 \Longrightarrow M'$.

Par définition de $\Longrightarrow$, M_1, M_2 sont obtenus à partir de M par réécriture de transition. Il existe donc $t_1 = s_1 \xrightarrow{l_1} s_1', t_2 = s_2 \xrightarrow{l_2} s_2' \in T$ deux transitions de M telles que $M_1 = M\left[\frac{M_{l_1}}{t_1}\right]$ et $M_2 = M\left[\frac{M_{l_2}}{t_2}\right]$, où $M_{l_1} = smodel(l_1)$ et $M_{l_2} = smodel(l_2)$. On a alors deux cas de figure :

$t_1 = t_2$ On obtient alors $M_1 = M_2$. Il suffit alors de poser $M' = M_1 = M_2$ pour obtenir le résultat (d'après la définition de $\Longrightarrow^{=}$).

$t_1 \neq t_2$ La preuve est symétrique. Soit $M' = M_1\left[\frac{M_{l_2}}{t_2}\right]$ (cette réécriture est légitime puisque $t_2 \in T_1$, étant au départ une transition de M qui n'a pas été modifiée dans la réécriture) : on a donc $M' = M\left[\frac{M_{l_1}}{t_1}\right]\left[\frac{M_{l_2}}{t_2}\right]$. D'après la propriété 3.1, cette égalité donne $M' = M\left[\frac{M_{l_2}}{t_2}\right]\left[\frac{M_{l_1}}{t_1}\right]$ ce qui correspond à $M' = M_2\left[\frac{M_{l_1}}{t_1}\right]$.

$\Longrightarrow$ est bien sous-commutative.

□

Théorème 3.1 *(Convergence)*
$\Longrightarrow$ est convergente.

Preuve:

Il s'agit de démontrer que $\Longrightarrow$ est confluente et termine.

- **Confluence** — Le lemme 3.2 établit la sous-commutativité de $\Longrightarrow$; d'après le lemme 3.1, $\Longrightarrow$ est donc confluente.
- **Terminaison** — Puisque le nombre d'instructions du programme est fini, et que la récursivité est prohibée dans le type de programme considéré, il ne peut y avoir de suite infinie de réécriture d'un modèle.

□

Puisque la réécriture sur les modèles est convergente, et qu'il est possible d'associer à chaque instruction de la syntaxe un modèle unique, nous pouvons définir une *fonction model* qui associe à chaque instruction le modèle normal unique qui lui correspond. Cette fonction construit, à partir d'une instruction du langage, un modèle de cette instruction contenant toutes les actions observables engagées par son exécution.

Définition 3.15 *(Modèle model d'une instruction)*

Soit obs un prédicat d'observabilité. La fonction $model_{obs} : Stm \rightarrow \mathcal{M}$*, paramétrée par obs, associe à chaque instruction stm du programme son modèle M défini par la forme normale unique dans* $(\mathcal{M}, \Longrightarrow)_{obs}$ *du modèle élémentaire* $\mu(stm)$.

Le modèle d'une instruction est obtenu par raffinement à l'aide de la règle de réécriture (paramétrée par *obs*), à partir de son modèle élémentaire $\mu(stm)$. La fonction *model* est donc définie sur l'ensemble des instructions du programme, et est bien définie : le théorème 3.1 assure la convergence de la réécriture, donc l'unicité des formes normales.

3.3 Algorithmes et exemple

Dans ce paragraphe sont présentés deux algorithmes simples pour le calcul du prédicat *hasObs* et la fonction de modélisation des instructions *model*. Les deux algorithmes supposent défini un prédicat d'observabilité *obs*. L'exemple 3.5 illustre le déroulement des algorithmes proposés.

Algorithme 3.1 : Prédicat $hasObs(stm)$

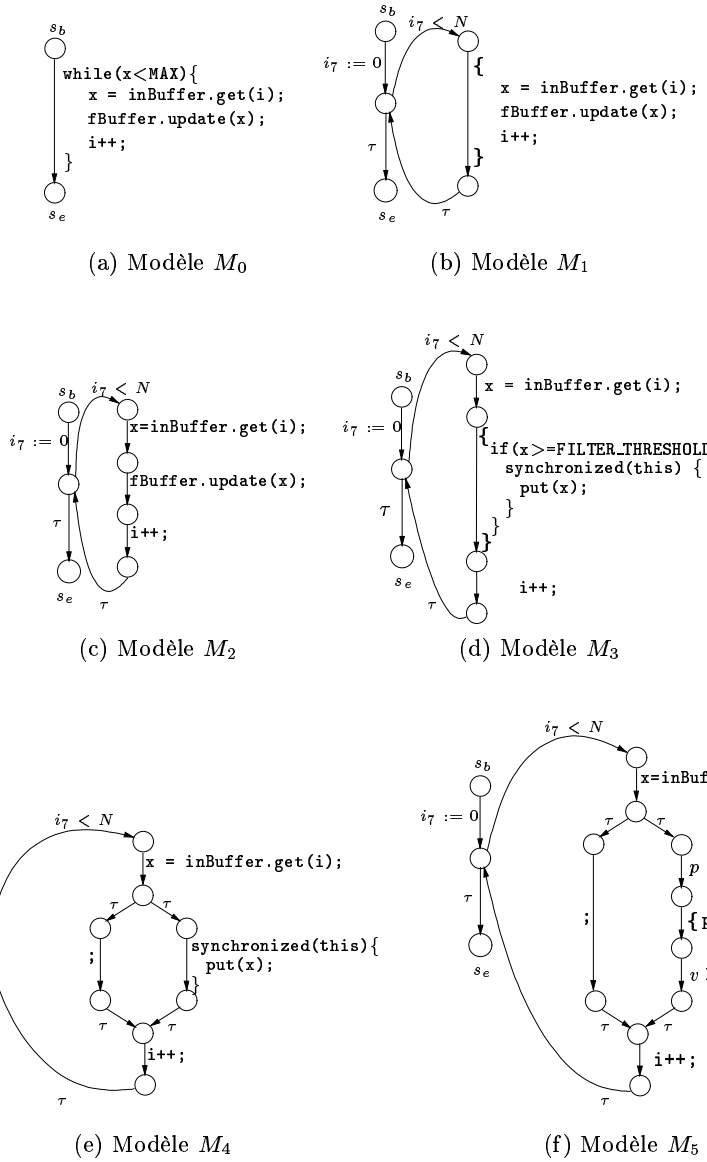
Entrées :
stm : instruction du programme
Sorties :
Valeur booléenne du prédicat

```
1  si stm ∈ L^c_stm ∧ ¬obs(stm) alors
2  |  M := smodel(stm)
3  |  pour t = (s, l, s') ∈ M faire
4  |  |  si obs(l) alors
5  |  |  |  retourner vrai
6  |  |
7  |  |  sinon si hasObs(l) alors
8  |  |  |  retourner vrai
9  |  |  finsi
10 |  finpour
11 finsi
12 retourner faux
```

Prédicat $hasObs(stm)$ — L'algorithme 3.1 calcule d'une manière récursive le prédicat $hasObs$ pour une instruction stm du programme. Conformément à la définition, la condition de la ligne 1 de l'algorithme indique que $hasObs$ ne peut être vrai que pour stm instruction composée et non observable. Dans ce cas, le modèle sémantique est construit. Si un des labels du modèle est observable, l'algorithme retourne $vrai$ à la ligne 5. Sinon l'algorithme est appliqué de nouveau à ce label, à la ligne 7.

Fonction $model_{obs}(stm)$ — L'algorithme 3.2 construit le modèle d'une instruction stm du programme en calculant la forme normale du modèle élémentaire $\mu(stm)$. Les modèles M et M' contiennent le même modèle à chaque début d'itération de la boucle de la ligne 5. A chacune des itérations de cette boucle, est appliquée une réécriture de transition. Chaque pas correspond au calcul d'une dérivation directe ; en effet on a à la ligne 7 $M' \Longrightarrow M'\left[\frac{smodel(l)}{t}\right]$. A la fin des itérations, à la ligne 9, le modèle M' est dérivé du modèle M par la réécriture de *toutes* les transitions de M où une règle de réécriture est possible : $M \Longrightarrow^* M'$. Si à ce point $M = M'$, il n'y a plus de règles de réécriture applicables sur M : c'est la forme normale de $\mu(stm)$.

Exemple 3.5 *Reprenons l'exemple du programme de la table 3.1. Soit* $M_0 = \mu(stm_7)$ *le modèle élémentaire contenant une unique transition étiquetée par la septième instruction du programme. Considérons le même prédicat d'observabilité obs défini dans l'exemple 3.4*



(a) Modèle M_0 (b) Modèle M_1

(c) Modèle M_2 (d) Modèle M_3

(e) Modèle M_4 (f) Modèle M_5

FIG. 3.5 – Exemple de modélisation d'une instruction. Le modèle initial est $M_0 = \mu(stm_7)$ où stm_7 est l'instruction de la ligne 7 de l'exemple table 3.1. Les modèles sont tels que $M_0 \Longrightarrow M_1 \Longrightarrow M_2 \Longrightarrow M_3 \Longrightarrow^* M_4 \Longrightarrow M_5$. A noter que $iterations(stm_7) = N$, la boucle a pour compteur i_7. b_{21} dénote `(x>=FILTER_THRESHOLD)`.

Algorithme 3.2 : Fonction de modélisation des instruction du programme $model_{obs}(stm)$

Données :
- *obs* : prédicat d'observabilité
- *stm* : instruction du programme

Résultat :
Modèle de l'instruction *stm*

```
1  M := nil
2  M' := μ(stm)
3  tant que M ≠ M' faire
4  |  M := M'
5  |  pour t = (s, l, s') ∈ M faire
6  |  |  si hasObs(l) alors
7  |  |  |  M' := M' [smodel(l) / t]
8  |  |  finsi
9  |  finpour
10 fintq
11 retourner M
```

(ie. *seules les actions de synchronisation sur les objets sont observables)* et rappelons que $hasObs(stm_7) = hasObs(stm_{10}) = hasObs(stm_{22}) = vrai$. *Soit* $iterations(stm_7) = N$ *le nombre d'itérations de la boucle. La figure 3.5 illustre les différents pas de transformation. Par exemple, nous avons* $M_0 \Longrightarrow M_1$ *; en effet* $M_1 = M_0 \left[\frac{smodel(stm_7)}{(s_b, stm_7, s_e)}\right]$ *et* $hasObs(stm_7) = vrai$. *De la même manière, nous avons* $M_1 \Longrightarrow M_2$, $M_2 \Longrightarrow M_3$ *et* $M_4 \Longrightarrow M_5$. *Le modèle* M_4 *quant à lui n'est pas une dérivation directe de* M_3, *mais d'un modèle intermédiaire dérivé de* M_3 *omis afin d'alléger le schéma. Il est à noter que tout les labels de* M_5 *vérifient* $\neg hasObs$. *Par conséquent aucune règle de réécriture n'est applicable et il n'est donc pas possible d'obtenir de dérivations de* M_5. *Le modèle* M_5 *est donc une* forme normale *de* M_0. *Par conséquent* $M_5 = model_{obs}(stm_7)$.

Chapitre 4

Modélisation des Applications Java Temps-Réel

Le chapitre précédent montre comment construire des système de transition et des fonctions de contexte de synchronisation qui modélisent les instructions concrètes d'un programme. Ces modèles sont construits selon des critères d'observabilité donnés, et reflètent les aspects fonctionnels (non temporisés) de l'exécution des instructions. Dans ce chapitre, nous présentons une méthodologie pour la construction du modèle temporisé d'une application temps-réel. Celui-ci est obtenu en ajoutant des contraintes temporelles au modèle fonctionnel du logiciel applicatif et en modélisant les interactions et synchronisations entre les différents threads.

Les modèles temporisés adoptés sont des systèmes de transitions temporisés avec urgences [BST98, Bor98, BS00]. Un modèle temporisé est construit, séparément, pour chaque thread d'un système à partir des instructions de sa logique. La construction des modèles des threads est détaillée à la section 4.1. La communication et la synchronisation des threads est modélisée par des fonctions de priorité. Les priorités, étudiées en [AGS00, GS02, GS03], offrent un cadre unifié pour exprimer des propriétés fonctionnelles, telles que l'exclusion mutuelle, et non fonctionnelles, telles que les politiques d'ordonnancement. La section 4.2 présente la construction du modèle temporisé d'un système temps-réel en composant les modèles de ses threads et en ajoutant les couches de communication et synchronisation, modélisées par des fonctions de priorité. Finalement, un exemple détaillé est présenté à la section 4.3.

4.1 Modélisation des threads

Chaque thread est modélisé par un système de transition temporisé avec des priorités. Ce modèle est construit à partir des instructions du corps de sa logique et des différentes contraintes temporelles. Nous commençons par présenter le modèle temporisé et les notations utilisées dans ce chapitre au paragraphe 4.1.1. Ensuite, nous détaillons notre méthodologie de

construction des modèles temporisés au paragraphe 4.1.2.

4.1.1 Système de transition temporisé avec priorités

Les systèmes de transition temporisés avec urgences (ou avec échéances) [BST98, BS00] sont une généralisation des automates temporisés. La différence essentielle réside dans la formulation de l'urgence dans ces modèles. Dans les automates temporisés, l'urgence est spécifiée d'une manière implicite avec des contraintes temporelles associés aux états, appelés *invariants*. Si l'invariant est (ou devient) faux, le temps ne peut plus s'écouler dans l'état associé, et une transition discrète franchissable (*ie.* dont la garde est évaluée à *vrai*) issue de cet état est prise. Dans les systèmes de transition temporisé, un type d'urgence est associé à chaque transition. Cette information détermine quand une transition franchissable devient urgente : dans ce cas le temps ne peut plus s'écouler et une transition discrète doit être prise.

Les définitions 4.2 et 4.3 définissent respectivement les systèmes de transition temporsiés et lerus sémantiques. L'exemple 4.1 donne une instance simple de notre modèle.

Préliminaires Soit $X = (x_1, ..., x_n)$ un vecteur de variables d'état mesurant la progression du temps appelées *horloges*. L'ensemble des valuations de X est noté V et la valuation de X est notée par un vecteur $v = (v_1, ..., v_n) \in V$. Les prédicats sur V sont appelés *X-contraintes*. Dans ce travail, nous optons pour une interprétation discrète du temps, l'ensemble des valuations V est donc isomorphe à $\mathbb{N}^n$. La notation $v + d$ où $v \in V$ et $d \in \mathbb{N}$ dénote le vecteur $(v_i + d)_{1 \leq i \leq n}$.

Définition 4.1 *(Prédécesseur temporel)*

Etant donnée une X-contrainte p, *on appelle* prédécesseur temporel *de* p *la* X-contrainte $\Diamond p$ *définie pour tout* $v \in V$ *par*

$$\Diamond p(v) = \exists t \geq 0 \,.\quad p(v + t)$$

Système de transition temporisé

Définition 4.2 *(Système de transition temporisé)*

Un système de transition temporisé[1]*, ou simplement système temporisé,* $TTS = (TS, X, h)$ *se compose de :*

- *Un système de transition* $TS = (S, L, T)$,
- *Un vecteur* X *d'horloges,*
- *Une fonction d'étiquetage* h *qui associe à chaque transition de* T *le triplet* $h(\,(s, l, s')\,) = (s, (l, g, u, r), s')$ *appelé* transition temporisée. *Le label de cette dernière* (l, g, u, r) *est appelé* label temporisé *et est défini comme suit :*
  - g *est une* X*-contrainte appelée* garde.

[1] En anglais *Timed Transition System*

- *u est le type d'urgence de la transition : $u \in \{\epsilon, \delta, \lambda\}$. Les symboles ϵ, δ et λ désignent respectivement les types d'urgence* urgent, délayable *et* paresseux.
- *$r : V \to V$ est une fonction appelée* remise à zéro *de la transition. Pour $v \in V$, $r(v) = v'$ est telle que chaque composante $v'_i \in \{v_i, 0\}$ pour tout $0 \leq i \leq n$.*

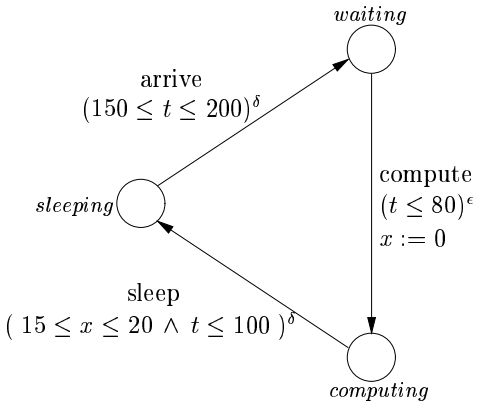


FIG. 4.1 – Exemple de système de transition temporisé.

Exemple 4.1 *La figure 4.1 montre un exemple simple d'un système de transition temporisé modélisant un thread cyclique. L'horloge t mesure le temps écoulé depuis l'arrivée du thread, elle est remise à zéro en prenant la transition étiquetée par arrive partant de l'état sleeping vers waiting. Cette transition est gardée par $(150 \leq t \leq 200)^{\delta}$: la transition peut être franchie pour les valuation de t entre 150 et 200 ; ces valeurs indiquent, respectivement, les bornes inférieure et supérieure des temps d'arrivée du thread. La transition étant de type* délayable, *elle doit être prise au plus tard quand la valuation de t atteint 200. L'horloge x est utilisée afin de mesurer le temps de séjour dans l'état de contrôle computing. Elle est remise à zéro par la transition de type* urgent *étiquetée par compute. La garde $t \leq 80$ indique le retardement maximal autorisé à l'état waiting sans violer l'échéance du thread. Finalement, la transition étiquetée par sleep ne peut être franchie que pour une valuation de x entre 15 et 20 ; ce sont les valeurs de séjour minimum et maximum dans l'état computing. La contrainte $t \leq 100$ exprime l'échéance du thread.*

Définition 4.3 *(Sémantique des systèmes de transition temporisés)*

Un état d'un système de transition temporisé $TTS = (TS, X, h)$, $TS = (S, T, L)$, est une paire (s, v) où $s \in S$ est un état de contrôle et $v \in V$ une valuation des horloges. Le système de transition temporisé définit une relation de transition $\to^{TTS} \subseteq (S \times V) \times (L \cup \mathbb{N}) \times (S \times V)$.

Soit $(s, v) \in S \times V$, on a

- $(s, v) \xrightarrow{l}{}^{TTS} (s', r(v))$ ***si*** $g(v) = vrai$ *où* $(s, l, s') \in T$ *et* $h(\,(s, l, s')\,) = (s,\, (l, g, u, r)\,, s')$.

– $(s,v) \xrightarrow{d}{}^{TTS} (s, v+d)$ ***ssi***

$$\bigwedge_{(s,l_i,s_i)\in T} \begin{cases} \neg(\Diamond g_i)(v) \vee (\Diamond g_i)(v+d) & si \quad u=\delta \\ \forall d'\ 0 \leq d' < d\,.\,\neg g_i(v+d') & si \quad u=\epsilon \\ vrai & si \quad u=\lambda \end{cases} \tag{4.1}$$

où $h(\,(s,l_i,s_i)\,) = (s,\ (l_i,g_i,u_i,r_i)\,,s_i)$.

Le modèle sémantique d'un TTS est un système de transition dont les transitions sont de deux catégories : celles étiquetées par des labels de L représentent un changement d'état de contrôle. Celles étiquetées par des entiers représentent l'écoulement du temps.

Un changement d'état de contrôle se fait en franchissant une transition du TTS si la garde g de cette transition est évaluée à *vrai* pour la valuation courante du vecteur d'horloges. Le passage d'un état de contrôle à un autre se fait instantanément, seules les horloges affectées par r sont changée.

Une transition étiquetée par un entier d représente l'écoulement de d unités de temps au même état de contrôle. L'écoulement du temps sur un état de contrôle n'est possible que si *aucune* transition sortante de cet état ne devient *urgente*. L'équation 4.1 qualifie la condition sous laquelle le temps peu progresser selon le type d'urgence. Le franchissement d'une transition *délayable* peut être retardé tant que le temps écoulé ne désactive pas la garde à jamais. Une transition de type *urgent* est, comme sont noms l'indique, urgente aussitôt que sa garde est vraie. Une transition *paresseuse* n'est jamais urgente.

Fonctions de priorité

Une fonction de priorité est une fonction sur les états de contrôle et les valuations des horloges. Chaque couple formé par un état de contrôle et une valuation du vecteur d'horloges lui est associé un ordre de priorité. Les ordres de priorités restreignent le comportement du système en prenant une transition plus prioritaire à une autre transition moins prioritaire. Les priorités forment un outil de restriction flexible et composable qui réduit le comportement du système sans en modifier la structure. C'est aussi un moyen élégant qui permet de formuler d'un manière unifiée des aspects fonctionnels et non fonctionnels des systèmes.

Définition 4.4 *(Ordre de priorité)*

Soit $TTS = (TS, X, h)$ *un système de transition temporisé avec* $TS = (S, L, T)$. *Considérons la relation* $\prec \subseteq T \times \{0,\infty\} \times T$. *On note* $t_1 \prec_0 t_2$ *pour* $(t_1, 0, t_2) \in \prec$ *et* $t_1 \prec_\infty t_2$ *pour* $(t_1, \infty, t_2) \in \prec$. *La relation* $\prec$ *est un* ordre de priorité *si pour toutes transitions* t_1, t_2, t_3,

– $\prec_0$ *et* $\prec_\infty$ *sont des ordres partiels sur* T

– *si* $t_1 \prec_\infty t_2$ *alors* $t_1 \prec_0 t_2$

– *si* $t_1 \prec_0 t_2 \wedge t_2 \prec_\infty t_3$ *alors* $t_1 \prec_\infty t_3$

Définition 4.5 *(Restriction des gardes)*

Soit $TTS = (TS, X, h)$ un système de transition temporisé. Soit $\prec$ un ordre de priorité défini sur TTS et s un état de contrôle. Pour toute transition t issue de s, la restriction de sa garde g par l'ordre de priorité est notée $g/\prec$ et est définie comme suit

$$g/\prec = g \wedge \neg \bigvee_{t' \in outgoing(s)} \begin{cases} \Diamond g' & si\ t \prec_\infty t' \\ g' & si\ t \prec_0 t' \\ faux & sinon \end{cases}$$

Où g' dénote la garde de la transition t'.

Définition 4.6 *(Fonction de priorité)*

Une fonction de priorité *définie sur un système de transition temporisé $TTS = (TS, X, h)$ est un ensemble fini de couples $pr = \{(C^i, \prec^i)\}_i$ où*

- *$\prec^i$ dénote un ordre de priorité,*
- *C^i est un prédicat sur $S \times V$. C^i associe à chaque état de contrôle $s \in S$ une X-*contrainte*, notée $C^i(s)$. Les contraintes d'état vérifient si $i \neq j$ alors $C^i \wedge C^j = faux$.*

Pour un état (s, v) du système, la fonction de priorité défini un ordre de priorité comme suit

$$pr(s, v) = \begin{cases} \prec^i & si\ \exists i\,.\, C^i(s, v) \\ \emptyset & sinon \end{cases}$$

La fonction de priorité pr est dite bien définie *si pour tout état $(s, v) \in S \times V$, $pr(s, v)$ est un ordre de priorité.*

Définition 4.7 *(Systèmes temporisés avec priorités)*

Soit $TTS = (TS, X, h)$ un système temporisé et pr une fonction de priorité. Un système temporisé avec priorité, noté (TTS, pr), est un système temporisé $TTS' = (TS', X, h')$ avec $TS' = (S, L, T')$ tel que :

- *$T' = T \setminus \{(s, l, s') \in T \mid \exists v \in V\,.\, pr(s, v)$ n'est pas un ordre de priorité $\}$*
- *Pour tout $t \in T'$ si $h(t) = (s, (l, g, u, r), s')$ alors $h'(t) = (s, (l, g', u, r), s')$ avec*

$$g' \quad = \quad g \wedge \bigwedge_{\substack{(C, \prec) \in pr \\ \prec \neq \perp}} (\neg C(s) \vee g/\prec)$$

$$pr = \{(s_0 \wedge (x \geq 1)\ ,\ \{t_2 \prec_\infty t_3, t_2 \prec_0 t_1\})\ ,\ (s_0 \wedge (x \geq 2)\ ,\ \{t_3 \prec_\infty t_1\})\}$$

$$\begin{aligned} g_2' &= g_2 \wedge (\neg(x \geq 1) \vee (g_2 \wedge \neg(\Diamond g_3 \vee g_1))) \\ &= \neg(x \geq 1) \vee \neg(x \geq 1) \vee \neg(x \geq 1) \\ &= (x < 1) \vee (3 < x < 9) \end{aligned}$$

$$\begin{aligned} g_3' &= g_3 \wedge (\neg(x \geq 2) \vee g_3 \wedge \neg\Diamond g_1) \\ &= (3 \geq x \geq 1) \wedge x < 2 \\ &= (x = 1) \end{aligned}$$

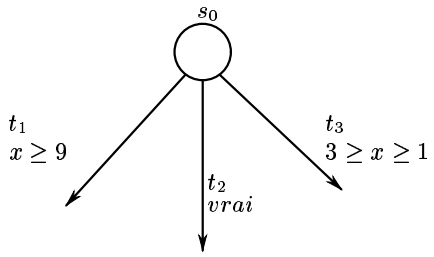


FIG. 4.2 – Exemple de restriction des gardes par application d'une fonction de priorité.

Exemple 4.2 *Soit le système de transition temporisé avec priorité (TTS, pr), s_0 un état de contrôle de TTS , t_1, t_2 et t_3 trois transitions issues de s_0 et pr la fonction de priorité définie par $pr = \{(s_0 \wedge (x \geq 1)\ ,\ \{t_2 \prec_\infty t_3, t_2 \prec_0 t_1\})\ ,\ \ (s_0 \wedge (x \geq 2)\ ,\ \{t_3 \prec_\infty t_1\})\}$. Les gardes des transitions sont respectivement $g_1 = (x \geq 9)$, $g_2 = vrai$ et $g_3 = (3 \geq x \geq 1)$. Les transitions sont illustrées dans le schéma 4.2. L'application de pr à l'état s_0 engendre la restriction de la garde de la transition g_2 en $g_2' = (x < 1) \vee (3 < x < 9)$. Intuitivement, la transition t_2 peut être prise soit lorsque la condition de l'application de l'ordre de priorité ne s'applique pas* ie. *$x < 1$; ou lorsque la transition t_3 ne peut jamais être prise ($x > 3$) et la transition t_1 n'est pas activée ($x < 9$). La garde de la transition g_3 est restreinte d'une manière similaire en $g_3' = (x = 1)$.*

Les fonctions de priorité sont conçues d'une manière modulaire, en modélisant séparément une propriété différente du système. La composition des différentes fonctions de priorité donne la fonction de priorité qui sera utilisée pour restreindre le comportement du système. Pour ce faire, nous introduisons l'opérateur de composition des fonctions de priorités $\oplus$.

Définition 4.8 *(Opérateur $\oplus$)*

Soient les ordres de priorité $\prec^1$, $\prec^2$ définis sur les labels du système temporisé $TTS = (TS, X, h)$. Est noté $\prec^1 \oplus \prec^2$ le plus petit ordre de priorité, s'il existe, tel que $\prec^1 \cup \prec^2 \subseteq \prec^1 \oplus \prec^2$.

L'opérateur $\oplus$ est surchargé pour les fonctions de priorité. Soient pr_1 et pr_2 deux fonctions de priorité définies sur TTS, la fonction de priorité $pr_1 \oplus pr_2$ est définie par

$$\forall (s, v) \in S \times V\ (pr_1 \oplus pr_2)(s, v) = pr_1(s, v) \oplus pr_2(s, v).$$

Remarque 4.1 *$\prec^1 \oplus \prec^2$ existe si $\prec^1 \cup \prec^2$ est un ordre de priorité* ie. *il n'y a pas de circuit qui apparaît dans l'union. $\perp$ est un élément absorbant pour $\oplus$. L'opérateur partiel $\oplus$ défini sur les priorités est associatif et commutatif* [Gößl01a, BGS00].

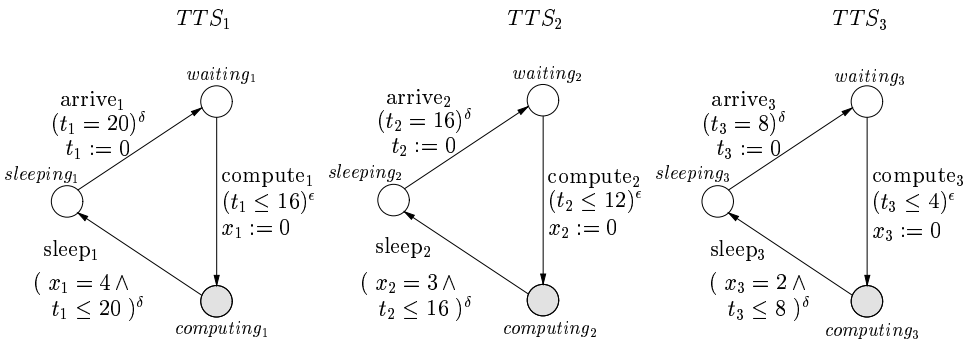


FIG. 4.3 – Trois processus périodiques partageant la même ressource.

Exemple 4.3 *Nous considérons trois threads périodiques représentés par les systèmes de transitions temporisés TTS_1, TTS_2 et TTS_3 illustrés dans la figure 4.3. Les trois threads partagent une ressource en exclusion mutuelle aux états $\{computing_i\}_{1\leq i\leq 3}$ grisés sur le schéma. L'exclusion mutuelle et la politique d'admission sont modélisées d'une manière incrémentale par des fonctions de priorité. L'exclusion mutuelle peu être garantie par le biais de fonctions de priorité* [Gößl01a, BGS00]. *Afin de modéliser l'exclusion mutuelle dans cet exemple, la fonction de priorité pr_{mutex} est définie par : $pr_{mutex} = \{(waiting_i \wedge computing_j\,,\, compute_i \prec_\infty sleep_j)\}_{i,j\in\{1,..,3\}}$. Intuitivement, afin de préserver l'exclusion mutuelle, les transitions menant vers des états critiques sont moins prioritaires que les actions qui sortes de ces états.*

Considérons la politique de résolution de conflit sur la ressource FIFO *pour cet exemple. La ressource est donc accordée au premier à la demander parmi les thread en attente. La règle de priorité pour cette politique est la suivante :*

$$pr_{fifo} = \{ \quad (t_i < t_j\,,\, compute_i \prec_0 compute_j)\,, \\ (t_i = t_j \wedge i < j\,,\, compute_i \prec_0 compute_j) \quad \}$$

En d'autres termes, la ressource est donnée au thread l'ayant le plus "attendu". Le deuxième couple relève l'indéterminisme en appliquant un ordre statique entre les threads en cas d'égalité entre les temps d'attente.

4.1.2 Construction du modèle d'un thread

Dans ce paragraphe est présentée la construction du système de transition temporisé avec priorités qui modélise un thread donné d'une application. Nous commençons par rappeler certaines hypothèses établies au chapitre 2 et introduire les notations adoptées dans cette section :

- Pour une application donnée, les threads ont un nombre fini et connu à priori. Ils sont tous créés et initialisés lors de la phase d'initialisation et sont tous actifs lors du lancement de la phase mission. Rappelons $\Theta = \{th_i\}_{1\leq i\leq n}$, $n \in \mathbb{N}$ l'ensemble des threads.

Dans toute la suite, th_i dénote un thread de l'application où $i \in \{1 \dots n\}$. La logique du thread th_i est notée run_i, c'est la méthode qui contient les instructions exécutées par le thread lors de son arrivée.

- Les threads de l'application sont des activités cycliques, périodiques ou sporadiques, qui ne terminent jamais. T_i^l et T_i^u dénotent respectivement les bornes inférieure et supérieure qui délimitent la durée entre deux arrivées successives (*inter-arrival time*) ; si th_i est périodique alors $T_i^l = T_i^u = T_i$ période du thread. A chaque thread est associée une échéance notée D_i.

Les threads sont modélisés par des systèmes de transition temporisés avec priorités. Le modèle du thread th_i est noté (TTS_i, pr_i) avec $TTS_i = (TS_i, X_i, h_i)$ et $TS_i = (S_i, L_i, T_i)$.

Au paragraphe 4.1.2.1 est montrée la construction du système de transition TS_i à partir du modèle du corps de la logique du thread run_i, selon la méthode introduite au chapitre 3. Au paragraphe 4.1.2.2, le système de transition est complété par les contraintes temporelles. Finalement, au paragraphe 4.1.2.4 est définie la fonction de priorité pr_i afin de modéliser l'attente introduite par les éventuels appels à la méthode `wait()`.

4.1.2.1 Construction du système de transition

Chaque arrivée d'un thread th_i correspond à l'invocation de la méthode run_i correspondante. Après avoir terminé l'exécution, le thread dort jusqu'à sa prochaine arrivée. Le système de transition du modèle du thread $TS_i = (S_i, L_i, T_i)$ est obtenu à partir du modèle du corps de la méthode run_i logique du thread. La démarche est schématisée dans la figure 4.4.

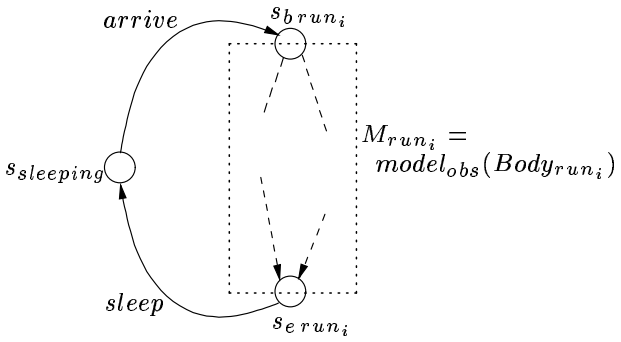


FIG. 4.4 – Construction du système de transition du modèle d'un thread à partir la méthode *run* associée.

En premier lieu est construit le modèle du corps de la logique du thread run_i. Soit $Body_{run_i}$ le corps de la méthode run_i et notons $M_{run_i} = model_{obs}(Body_{run_i})$. Le prédicat d'observabilité *obs* est choisi de manière à inclure dans le modèle les actions de verrouillage des ressources, l'attente/notification introduite par les méthodes `wait()` et `notifyAll()`, et l'invocation des

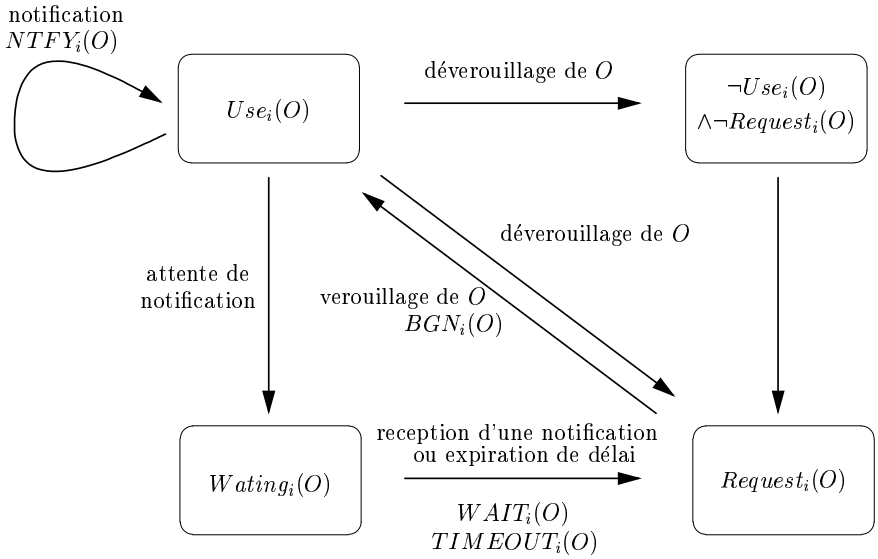


FIG. 4.5 – Actions et états caractérisant l'utilisation d'un objet partagé O par un thread th_i.

méthodes de calcul, identifiés par la fonction *exec*. Le prédicat *obs* est défini par $obs(l) = (l \in (\mathcal{L}_{sync} \cup \mathcal{L}_{exec}))$.

Soit $M_{run_i} = (TS_{run_i}, s_{brun_i}, s_{erun_i}, \Sigma_{run_i})$ avec $TS_{run_i} = (S_{run_i}, L_{run_i}, T_{run_i})$. Nous définissons $TS_i = (S_i, L_i, T_i)$ par :

- $S_i = S_{run_i} \cup \{s_{sleeping}\}$ est ajouté l'état de sommeil du thread
- $L_i = L_{run_i} \cup \{arrive, sleep\}$ sont ajoutées les actions d'arrivée et de sommeil
- $T_i = T_{run_i} \cup \{s_{sleeping} \xrightarrow{arrive} s_{brun_i}\,,\ s_{erun_i} \xrightarrow{sleep} s_{sleeping}\}$ les transitions ajoutées caractérisent le caractère cyclique du thread.

Afin de caractériser les états et transitions liés à l'usage des objets partagés et à l'attente / notification dans TTS_i, nous notons pour tout $O \in \Omega$:

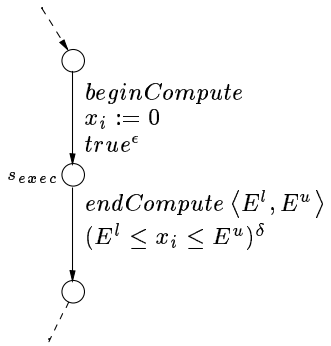
- $Waiting_i(O) = \{s \in S_i | \exists s' \in S_i\,.\,(s, ?O, s') \in T_i\}$ l'ensemble des états d'attente
- $Use_i(O) = \{s \in S_i \,|\, O \in \Sigma_i^+(s)\}$ états de TTS_i où le thread th_i *possède* le verrou de l'objet O.
- $Request_i(O) = \{s \in S_i \,|\, s \notin Use_i(s) \wedge \exists (s, p\,O, s') \in T_i\}$ états où le thread demande le verrou de l'objet O.
- $TIMEOUT_i(O) = \{t \in T_i \,|\, t = (s, timeout\,\langle W \rangle, s')\}$ l'ensemble des transitions de dépassement de délai
- $WAIT_i(O) = \{t \in T_i \,|\, t = (s, ?\,O, s')\}$ l'ensemble des transitions d'attente de notification
- $NTFY_i(O) = \{t \,|\, t = (s, !\,O, s') \in T_i\}$ l'ensemble des transitions de notification
- $BGN_i(O) = \{t \,|\, t = (s, p\,O, s') \in T_i \wedge s \notin Use_i(O)\}$ l'ensemble de transitions qui mènent de $Request_i(O)$ à $Use_i(O)$.

La figure 4.5 illustre les différents ensembles d'états et de transitions définis ci-dessus.

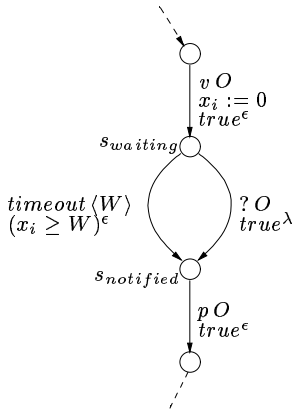
4.1.2.2 Système de transition temporisé associé à un thread

Dans ce paragraphe sont définies les horloges de contraintes temporelles relatives au système de transition temporisé $TTS_i = (TS_i, X_i, h_i)$ qui modélise le thread th_i. Deux horloges sont définies : la première notée x_i mesure les temps d'exécution correspondant aux méthodes de calculs, la deuxième est notée t_i et mesure le temps total depuis l'arrivée du thread. Les deux horloges définissent l'ensemble des horloges du système de transition temporisé $X_i = \{x_i, t_i\}$. Les contraintes temporelles sont associées aux transitions par la fonction d'étiquetage $h_i : (s, l, s') \mapsto (s, (l, g, u, r)s')$. Nous définissons la fonction d'étiquetage selon le label l de la transition non temporisé.

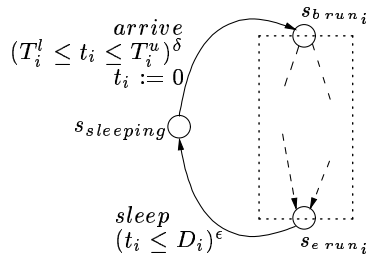
Nous dénombrons trois types de contraintes temporelles dues à l'exécution du programme, l'environnement de l'application et les échéances temps-réel.



(a) Contraintes liées au temps d'exécution.



(b) Attente de notification par invocation de `wait(`W`)`.



(c) Contraintes dues aux temps d'arrivée et échéances temps-réel.

FIG. 4.6 – Etiquetage des transitions non temporisées par les contraintes temporelles.

Contraintes dues à l'exécution du programme Les temps d'exécution caractérisent le déroulement des différents calculs de l'application et sont relatifs à la plate-forme cible sur laquelle l'application est exécutée. Dans notre cadre de travail, ces temps correspondent à l'exécution de méthodes de calcul aux quelles sont associées leurs durées d'exécution par la fonction *exec*. Cette information est présente sur le système de transition TS_i par le biais du modèle sémantique de l'invocation des méthodes de calcul (*cf.* paragraphe 3.2.2). L'étiquetage par la fonction h_i de ces transitions est illustré par le schéma (a) de la figure 4.6. Le temps de calcul est modélisé par un temps de séjour dans l'état $s_{executing}$. L'horloge x_i est mise à zéro en franchissant la transition qui mène à cet état ; la transition sortante est quant à elle gardée par la contrainte délayable $(E^l \leq x_i \leq E^u)^\delta$ afin de permettre un séjour au moins égal à E^l et au plus égal à E^u, valeur pour laquelle la transition devient urgente.

En plus des méthodes de calcul, l'appel à la méthode `wait()` introduit un délai d'attente borné pour le thread qui l'exécute. Conformément à la syntaxe abstraite présentée au chapitre 2, nous considérons les appels ayant un délai d'attente (*timeout*) fini et non nul. Sur le modèle temporisé de l'appel à cette méthode, typiquement O`.wait(`W`)` représenté en (b) figure 4.6, le passage du thread de l'état d'attente à l'état notifié peut se faire de deux manière :

- En atteignant le délai d'attente mesuré dans le modèle par l'horloge x_i, la transition étiquetée par $timeout\langle W\rangle$ devient active et urgente dès que la valuation de x_i atteint W. Ceci est représenté par la garde $(x_i \geq W)^\epsilon$.
- La prise de la transition étiquetée par $?O$, ce qui correspond à la notification par un autre thread. Cette transition est active dès l'arrivée à l'état d'attente, mais n'est jamais urgente. En effet, cette action ne peut se faire que par le concours du thread offrant l'action de notification, en se synchronisant. La synchronisation entre l'action d'attente et notification est présentée à la section 4.2.

Contraintes dues à l'environnement de l'application L'environnement de l'application est la source des stimuli auxquels elle réagit. La vitesse de l'environnement dicte et sert à dimensionner la vitesse à laquelle doit tourner l'application. C'est la dynamique de l'environnement (*ie.* sa vitesse de changement) qui caractérise les temps d'arrivée des threads. Soient T_i^l et T_i^u, respectivement, les bornes inférieure et supérieure qui délimitent la durée entre deux arrivée successives du thread th_i. Nous associons à la transition d'arrivée du thread (étiquetée par *arrive*) une garde de type délayable $(T_i^l \leq t_i \leq T_i^u)^\delta$. Cette contrainte modélise le fait que le thread arrive d'une manière incontrôlable pour une valuation de t_i entre T_i^l et T_i^u. L'horloge t_i est remise à zéro en prenant la transition d'arrivée afin de décompter le temps écoulé entre deux arrivées successives. La figure 4.6 montre l'étiquetage des transitions d'arrivée.

Contraintes d'échéances Dans un système temps-réel, les résultats de calcul, logiquement corrects, ne sont valables que s'ils sont délivrés "à temps", avant une certaine échéance. Les échéances sont des contraintes temporelles définies à la conception du système temps-réel en

Label l	**Label temporisé** (g, u, r)
begincompute	$(vrai, \epsilon, x_i \mapsto 0)$
endcompute $\langle E^l, E^u \rangle$	$(E^l \leq x_i \leq E^u, \delta, Id)$
arrive	$(T_i^l \leq t_i \leq T_i^u, \delta, t_i \mapsto 0)$
sleep	$(t_i \leq D_i, \epsilon, Id)$
$?O \quad O \in \Omega$	$(vrai, \lambda, Id)$
autres labels	$(vrai, \epsilon, Id)$

TAB. 4.1 – Fonction d'étiquetage h_i du système de transition temporisé TTS_i.

fonction de son environnement. Ces contraintes temporelles constituent l'engrenage qui lie la vitesse d'exécution de l'application et celle de son environnement. Une échéance D_i est associée à chaque thread th_i. L'échéance est une contrainte sur le temps maximal que doit mettre un thread depuis son arrivée pour terminer un cycle d'exécution. La transition de sommeil du thread (étiquetée par *sleep*) est donc gardée par une contrainte de type urgent $(t_i \leq D_i)^\epsilon$. L'horloge t_i mesure l'écoulement du temps depuis l'arrivée du thread et doit avoir une valuation inférieure à D_i. Au paragraphe 4.2.2, nous montrons comment rétropropager la contrainte d'échéance sur les gardes du modèle.

Les différentes contraintes temporelles associées aux transitions du système de transition temporisé TTS_i par la fonction d'étiquetage $h_i : (s, l, s') \mapsto (s, (l, g, u, r)s')$ sont résumées dans la table 4.1.

4.1.2.3 Rétropropagation des échéances

Afin de construire un modèle temporisé correct, il faut spécifier, d'une manière explicite à chaque état, comment on avance en fonction de la progression du temps. La spécification de la progression du modèle en fonction du temps élimine les attentes injustifiées des *états intermédiaires*, c'est-à-dire les états qui séparent la remise à zéro d'une horloge de l'état où cette horloge est contrainte par la garde d'une transition sortante. La négligence de cet aspect de la modélisation peut introduire des blocages dans le modèle. Il faut dire explicitement à chaque état de contrôle d'un système de transition temporisé la durée de séjour maximale possible, le temps maximal pour lequel le système peut "attendre". Ceci pour être sûr qu'un blocage détecté lors de la vérification n'est pas dû à l'incomplétude de la spécification temporelle du modèle.

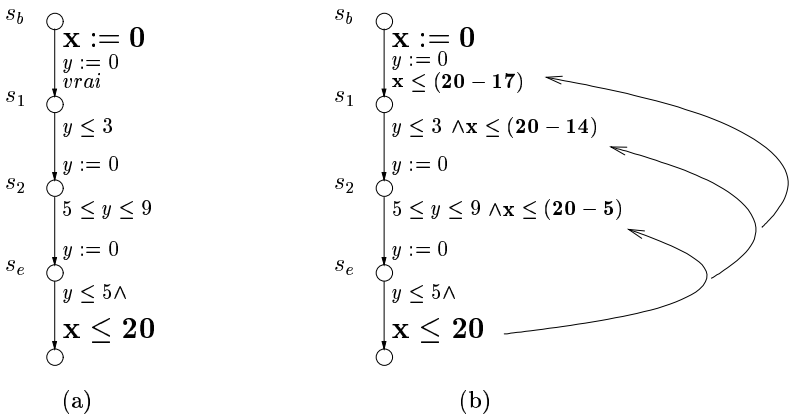


FIG. 4.7 – Exemple de rétropropagation de contrainte temporelle. (a) et (b) représentent, respectivement, le système de transition temporisé avant et après la rétropropagation de la contrainte $x \leq 20$.

Exemple 4.4 *Considérons l'exemple illustré dans la figure 4.7 (a). L'horloge x est mise à zéro à l'issue de l'état s_b et est testée par la garde $x \leq 20$ à l'état s_e. Par ailleurs, nous avons des temps de séjours, dictés par l'horloge y, dans les états s_1, s_2 et s_e avec les bornes supérieures, respectives, de 3, 9 et 5 unités de temps. La rétropropagation de la contrainte renforce les gardes des transitions précédentes. Ceci a pour effet de dicter le temps de séjour autorisé dans les états s_b, $s_1$et s_2 pour respecter la garde $x \leq 20$ à l'état s_e. Le résultat de cette opération est montré dans la partie (b) de la figure 4.7.*

Pour construire correctement des modèles temporisés des threads, les contraintes temporelles sont adéquatement rétropropagées. Considérons TTS_i modèle temporisé du thread th_i. Par construction, l'horloge t_i décompte le temps écoulé entre l'arrivée du thread (transition étiquetée par *arrive*) jusqu'à la fin de son cycle (transition étiquetée par *sleep*). Les différents calculs effectués par le thread sont, quant à eux, mesuré par l'horloge x_i comme nous l'avons décrit dans le paragraphe précédent. L'échéance temps-réel est modélisée par la contrainte temporelle $t_i \leq D_i$ qui garde la transition de fin de cycle du thread. Cette contrainte est rétropropagée dans le modèle. La rétropropagation restreint les actions d'accès aux ressources du système, à savoir, les actions de calcul *beginCompute* et l'acquisition des objets partagés par les actions du type pO, $O \in \Omega$. L'exemple 4.5 donne une illustration du mécanisme de rétropropagation, détaillé dans les paragraphes qui suivent.

Exemple 4.5 *La figure 4.8 - (a) présente le système de transition temporisé TTS_1 qui modélise un thread périodique de période 100 et d'échéance 85. La rétropropagation de la contrainte d'échéance $t_1 \leq 85$ dans le modèle donne lieu au système de transition temporisé (b) de la même figure.*

La rétropropagation revient à calculer, à partir d'une échéance globale, *l'échéance relative* pour chaque état de contrôle s, notée $D_{rel}(s)$. L'échéance relative est utilisée afin de restreindre les gardes des transitions sortantes de s par la contrainte $t_i \leq D_{rel}(s)$. D'une manière informelle, l'échéance relative à l'état s dénote le temps maximal, mesuré par l'horloge t_i, au delà duquel le séjour à l'état s risque de compromettre l'échéance globale du thread. La rétropropagation limite donc le temps de séjour à l'état s en renforçant les gardes des transitions sortantes par la contrainte temporelle $t_i \leq D_{rel}(s)$.

4.1.2.4 Fonction de priorité associée au système de transition temporisé

Nous associons au système de transition temporisé modèle du thread une fonction de priorité qui contrôle l'attente introduite par l'appel de la méthode `wait()`. La notification d'un thread est effectuée, soit par dépassement du délai d'attente, en prenant la transition étiquetée par $timeout\,\langle W \rangle$ notée $t_{timeout}$, ou par notification via la transition étiquetée par $?\,O$ notée t_{wait}.

L'action $?\,O$ ne peut se faire que par le concours d'un tierce thread qui offre l'action de notification $!\,O$. L'action $?\,O$ est une action qui ne peut se produire spontanément, la transition t_{wait} ne doit donc être engagée qu'en se synchronisant avec une action de notification offerte par un autre thread. En l'absence de synchronisation avec une notification extérieure, l'attente du délai matérialisée par $t_{timeout}$ doit donc prévaloir sur la transition t_{wait}. La fonction de priorité pr_i est définie afin de modéliser cette restriction. La synchronisation est l'interaction inter-thread liée à la notification est quant à elle étudiée au paragraphe 4.2.

La fonction de priorité qui modélise la précédence de l'attente de délai à l'attente de notification pour un objet $O \in \Omega$ est définie par :

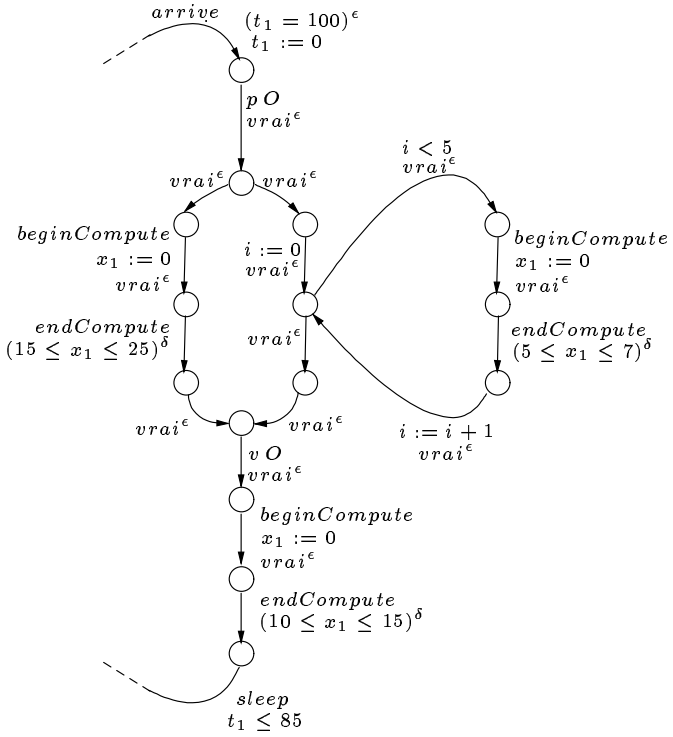
$$pr^{O}_{wait,i} = \{\, (\, Waiting_i(O)\ ,\ WAIT_i(O) \prec_{\infty} TIMEOUT_i(O)\,)\, \}$$

La fonction de priorité pr_i est la composition des fonctions de priorités pour tous les objets partagés. Elle s'écrit :

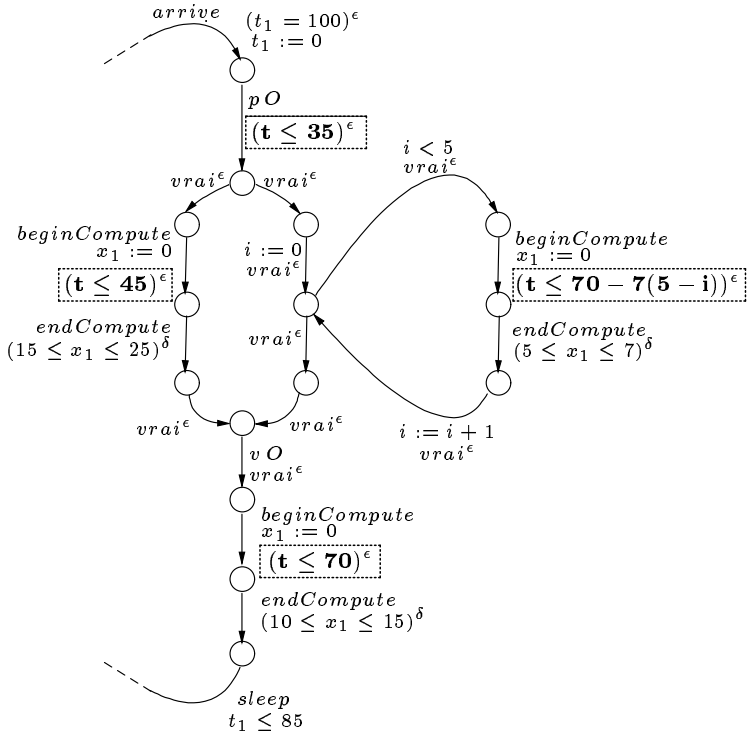
$$pr_i = \bigoplus_{O \in \Omega} pr^{O}_{wait,i}$$

4.2 Modélisation de l'application

Dans ce qui précède, nous avons montré comment construire, d'une manière modulaire, l'ensemble des systèmes de transition temporisés $\{(TTS_i, pr_i)\}_{1 \leq i \leq n}$ qui modélisent des threads. Dans ce paragraphe, nous allons montrer comment construire le système de transition temporisé avec priorité (TTS, pr) qui modélise l'application, en composant les modèles des threads et en modélisant leurs interactions.



(a) TTS_1 avant rétropropagation des contraintes.



(b) TTS_1 après rétropropagation des contraintes.

FIG. 4.8 – Exemple de rétropropagation des échéances sur le modèle d'un thread.

4.2.1 Composition des systèmes temporisés avec priorités

Nous adoptons la composition parallèle des systèmes de transition temporisés par l'opérateur de composition commutatif et associatif $||$ défini en [BS00]. Nous surchargeons cet opérateur pour la composition des systèmes de transition temporisés avec priorité d'une manière analogue a celle présentée en [BGS00, AGS02].

Définition 4.9 *(Composition des systèmes temporisés avec priorité)*

Nous considérons $n \geq 2$ *systèmes temporisés de la forme* $TTS_i = (TS_i, X_i, h_i)$, $i \in \{1, \ldots, n\}$ *avec* $TS_i = (S_i, L_i, T_i)$. *Soit* $\Lambda \in \mathcal{P}(\bigcup_{i=1}^{m} T_i)$ *tel que si* $\mathbf{t} \in \Lambda$ *alors* $\mathbf{t}$ *contient au plus un élément de chaque ensemble de transition et* $|\mathbf{t}| \geq 2$. *La composition parallèle des systèmes temporisés avec priorité* (TTS_i, pr_i) $i \in \{1, \ldots, n\}$ *est un système temporisé avec priorité* (TTS, pr) *noté* $||_\Lambda\{(TTS_i, pr_i)\}_{1 \leq i \leq n}$ *où* $TTS = (TS, X, h)$ *et* $TS = (S, L, T)$ *tels que :*

- $TTS = ||_\Lambda\{TTS_i\}_{1 \leq i \leq n}$ *où* $||$ *est l'opérateur de composition des systèmes temporisés défini en* [BGS00].
- $pr = \bigoplus_{i=1}^{n} pr_i \oplus pr_\Lambda$ *où* pr_Λ *est la règle de* progrès maximal *définie par*

$$\begin{aligned} pr_\Lambda \quad = \quad & \{(vrai, \{t \prec_\infty \mathbf{t}\}) \,|\, t \in \mathbf{t} \wedge \mathbf{t} \in \Lambda\} \quad \oplus \\ & \{(vrai, \{\mathbf{t}_1 \prec_\infty \mathbf{t}_2\}) \,|\, \mathbf{t}_1 \subset \mathbf{t}_2 \wedge \mathbf{t}_1, \mathbf{t}_2 \in \Lambda\} \end{aligned}$$

Il est à noter que dans le système de transition temporisé produit de la composition, la synchronisation des transition se fait en respectant deux principes importants :

- **Activation et urgence** étant donné une transition $\mathbf{t} \in \Lambda$ une synchronisation des transitions $\{t_k\}_{k \in K}$ où $K \subseteq \{1 \ldots n\}$, la transition $\mathbf{t}$ est active quand *toutes* les transitions t_k sont actives. La transition $\mathbf{t}$ devient urgente quand elle est active et au moins l'une des transitions t_k devient urgente.
- **Progrès maximal** la règle de priorité pr_Λ garantit le progrès maximal [HR95, AGS02] : les transitions de synchronisation sont prises au détriment des transition qui les composent. De plus, parmi les transition de synchronisation actives la transition maximale, au sens de l'inclusion, est prioritaire.

4.2.2 Construction du modèle de l'application

L'application est modélisée par un système de transition temporisé avec priorités (TTS, pr). Nous adoptons une approche par couches : dans un premier temps, nous allons définir l'ensemble des transitions synchronisées Λ, la fonction de priorité pr_{comm} et effectuer la composition des modèles des threads TTS_i. Ensuite, la relation de priorité pr_{mutex} sera définie pour assurer l'exclusion mutuelles entre les sections critiques des threads. Le modèle de l'application sera finalement obtenu en composant les modèles des threads et les fonctions de priorités.

4.2.2.1 Communication

Au paragraphe 4.1.2.4 nous avons associé à chaque TTS_i une fonction de priorité pr_i qui donne la priorité, pour chaque état d'attente de notification sur $O \in \Omega$, à la transition d'expiration de délai (étiquetée par *timeout*) par rapport à la transition d'attente notification (étiquetée par $?O$). Cette règle a pour but "d'empêcher" les transitions d'attente de notification d'être franchies sans être synchronisées avec une transition de notification d'un autre thread. Dans ce paragraphe, nous montrons comment construire l'ensemble des transitions synchronisées Λ, introduit dans la définition 4.9, afin de synchroniser les transitions d'attente avec les transitions de notification correspondantes pour tous les threads.

Rappelons qu'une communication par attente / notification sur un objet $O \in \Omega$ fait intervenir une unique transition de notification synchronisée avec *toutes* les transitions d'attente de notification, sur ce même objet, qui sont disponibles. Afin de prendre en compte tous les cas possibles, une transition synchronisée est définie pour chaque combinaison de transitions d'attente / notification. La transition synchronisée faisant participer le plus de transitions d'attente a une plus grande priorité selon la fonction de priorité du progrès maximal pr_Λ (définition 4.9). Par exemple, soient th_1, th_2 et th_3 trois threads tels que th_1 émet une notification sur un objet partagé O et th_2 et th_3 effectuent une attente de notification sur ce même objet. Soient TTS_1, TTS_2 et TTS_3 les modèles respectifs de ces threads avec t^1_{notify} la transition de notification dans TTS_1, t^2_{wait} et t^3_{wait}, respectivement, les transitions d'attente dans TTS_2 et TTS_3. Trois actions synchronisées sont possibles : $t^1_{notify}|t^2_{wait}$, $t^1_{notify}|t^3_{wait}$ et $t^1_{notify}|t^2_{wait}|t^3_{wait}$. La fonction de priorité de progrès maximal stipule dans ce cas :

$$t^1_{notify} \prec_\infty \{t^1_{notify}|t^2_{wait}\,,\, t^1_{notify}|t^3_{wait}\} \prec_\infty t^1_{notify}|t^2_{wait}|t^3_{wait}$$

Ceci garantit que la transition faisant intervenir le plus de threads en état d'attente est choisie. L'exemple 4.6 donne une illustration plus détaillée de ce principe.

Soient $j \in \{1,\ldots,n\}$ et $O \in \Omega$ un objet partagé. Rappelons l'ensemble des transitions d'attente de notification $WAIT_j(O)$ et l'ensemble des transitions de notification $NTFY_j(O)$. Soit l'ensemble Γ_j contenant les ensembles de transitions ayant chacun au plus une transition d'attente de notification de chaque thread, sauf celles du thread th_j. Cet ensmeble s'écrit $\Gamma_j = \left\{\gamma \,|\, \gamma \in \mathcal{P}\left(\bigcup_{1\le i\le n}^{i\neq j} WAIT_i(O)\right) \wedge \forall i = 1...n,\, i \neq j.\, |\gamma \cap WAIT_i(O)| \le 1\right\}$. Les transitions synchronisées qui représentent la notification par le thread th_j peuvent alors s'écrire : $WN_j(O) = \{\mathbf{t} \,|\, \mathbf{t} = \{t\} \cup \gamma\,,\, t \in NTFY_j(O) \wedge \gamma \in \Gamma_j\}$. D'une manière informelle, chaque élément de $WN_j(O)$ contient une transition de notification du thread th_j avec toutes les combinaisons, non vides, des transitions d'attente de notification des autres threads. L'ensemble de toutes les actions synchronisées pour l'objet O en découle : $WN(O) = \bigcup_{1\le j\le n} WN_j(O)$.

L'ensemble Λ regroupe tous les ensembles de transitions synchronisées pour tous les objets

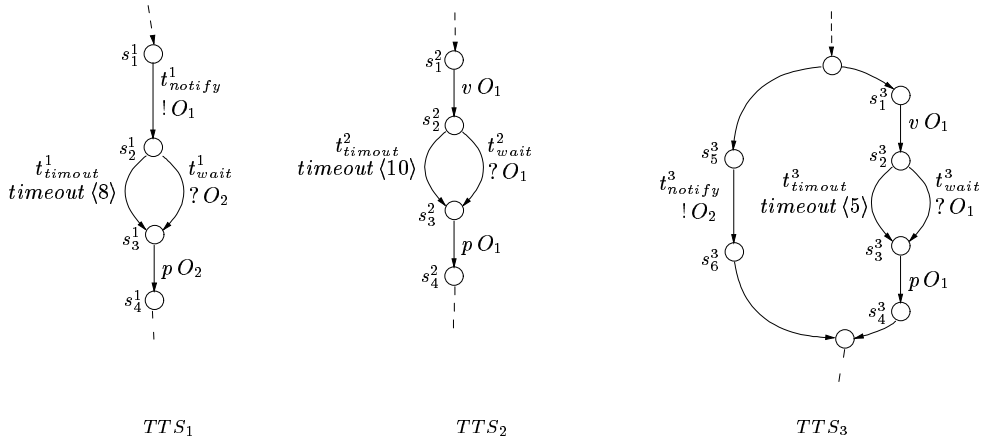


FIG. 4.9 – Trois thread avec une communication par attente/notification sur deux objets.

partagés de l'application. Ainsi :

$$\Lambda = \bigcup_{O \in \Omega} WN(O)$$

Exemple 4.6 *Considérons les systèmes de transition temporisés TTS_1, TTS_2 et TTS_3 représentés dans le figure 4.9. Afin de construire l'ensemble des transitions synchronisées Λ, on construit pour chaque thread l'ensemble $WN_i(O_j)$ qui regroupe les transitions synchronisées ayant chacune une transition de notification de TTS_i et une combinaison de transitions d'attente de notification des autres threads (une au plus par thread). Pour notre exemple, nous avons :*

- $WN_1(O_1) = \{t^1_{notify} | t^2_{wait}\ ,\ t^1_{notify} | t^3_{wait}\ ,\ t^1_{notify} | t^2_{wait} | t^3_{wait}\}$ *et* $WN_1(O_2) = \emptyset$.
- $WN_2(O_1) = WN_2(O_2) = \emptyset$. *Aucune notification dans* TTS_2.
- $WN_3(O_1) = \emptyset$ *et* $WN_3(O_2) = \{t^3_{notify} | t^1_{wait}\}$

L'ensemble des transitions de synchronisations se déduit directement en faisant l'union $\Lambda = \{t^3_{notify} | t^1_{wait}\ ,\ t^1_{notify} | t^2_{wait}\ ,\ t^1_{notify} | t^3_{wait}\ ,\ t^1_{notify} | t^2_{wait} | t^3_{wait}\}$.

La priorité de communication en découle d'après la définition 4.9 de la composition des systèmes de transitions temporisés avec priorités comme la composition des priorités des threads et celle du progrès maximal :

$$pr_{comm} = \bigoplus_{i=1}^{n} pr_i \oplus pr_\Lambda$$

Exemple 4.7 *Reprenons le système de l'exemple 4.6. A chaque thread th_i est associée une fonction de priorité pr_i qui empêchent les transition d'attente d'être prises sans synchronisation (cf. paragraphe 4.1.2.4). Dans cet exemple, ces fonctions s'écrivent simplement comme suit :* $pr_i = \{(\ s^i_2\ ,\ t^i_{wait} \prec_\infty t^i_{timeout}\ )\}$ *pour* $i = 1 \ldots 3$. *La fonction de priorité qui assure le progrès maximal est déduite de* Λ *calculé dans l'exemple précédent :*

$pr_\Lambda = \Big\{(vrai, \{t^1_{notify} | t^2_{wait} \prec_\infty t^1_{notify} | t^2_{wait} | t^3_{wait}\ ,\ t^1_{notify} | t^3_{wait} \prec_\infty t^1_{notify} | t^2_{wait} | t^3_{wait}\}),$

$(vrai, \{t^3_{notify} \prec_\infty t^3_{notify} | t^1_{wait}\ ,\ t^1_{notify} \prec_\infty t^1_{notify} | t^2_{wait}\ ,\ t^1_{notify} \prec_\infty t^1_{notify} | t^3_{wait}\ ,$
$t^1_{notify} \prec_\infty t^1_{notify} | t^2_{wait} | t^3_{wait}\})\Big\}$.

La fonction de priorité de communication pr_{comm} se déduit dans ce cas par l'union des quatre fonctions de priorité calculées ci-dessus.

4.2.2.2 Exclusion mutuelle

Chaque objet $O \in \Omega$ est doté d'un verrou dont disposent les threads afin d'y interdire tout accès concurrent en exclusion mutuelle. Les objets sont des ressources non-préemptables : une fois le verrou d'un objet est obtenu par un thread, il n'est relâché que si ce dernier le libère d'une manière explicite. L'exclusion mutuelle restreint le comportement du système en organisant l'accès aux ressources partagés. Nous modélisons l'exclusion mutuelle entre les threads par une fonction de priorité notée pr_{mutex}. Dans ce paragraphe nous montrons comment construire pr_{mutex} en composant les fonctions de priorités qui modélisent l'exclusion mutuelle sur chaque objet partagé de Ω.

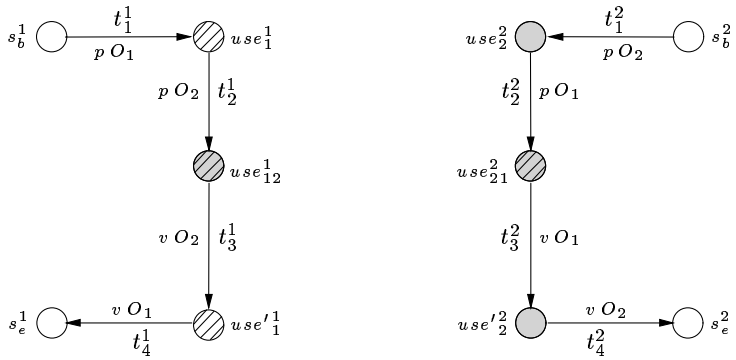


FIG. 4.10 – Exclusion mutuelle sur deux ressources partagées entre deux threads.

La méthodologie de synthèse des priorités pour l'exclusion mutuelle a été introduite en [BGS00]. Intuitivement, l'approche consiste à donner la priorité aux actions issues des états critiques d'un thread utilisant un objet sur les actions des autres threads qui demandent l'accès à ce même objet. Afin d'illustrer ce concept, considérons l'exemple de la figure 4.10. Dans cet exemple, deux threads th_1 et th_2 se partagent deux objets O_1 et O_2. Le thread th_1 se synchronise d'abord sur O_1 puis sur O_2 ; tandis que th_2 effectue d'abord une synchronisation sur O_2 puis sur O_1 (sur le schéma, les états où l'objet O_1 est verrouillé sont décorés par des lignes obliques, ceux où O_2 est verrouillé sont grisés). La fonction d'exclusion mutuelle sur l'objet O_1 est donnée par :

$pr^{O_1}_{mutex} = \{\quad ((use^1_1 \vee use^1_{12} \vee use'^1_1) \wedge use^2_2\ ,\ t^2_2 \prec_\infty \{t^1_2,\ t^1_3,\ t^1_4\})\ ,$
$\qquad\qquad (use^2_{21} \wedge s^1_b\ ,\ t^1_1 \prec_\infty t^2_3)\quad \}$

D'une manière analogue, la fonction de priorité qui modélise l'exclusion mutuelle pour l'objet O_2 s'écrit :

$$pr^{O_2}_{mutex} = \{ \quad ((use^2_2 \vee use^2_{21} \vee use'^2_2) \wedge use^1_1 \,,\, t^1_2 \prec_\infty \{t^2_2,\, t^2_3,\, t^2_4\}) \,, \\ (use^1_{12} \wedge s^2_b \,,\, t^2_1 \prec_\infty t^1_3) \quad \}$$

La fonction de priorité pr_{mutex} est la composition des fonctions de priorité relatives à l'exclusion mutuelle sur O_1 et O_2 :

$$\begin{aligned} pr_{mutex} = \quad & pr^{O_1}_{mutex} \oplus pr^{O_2}_{mutex} \\ pr_{mutex} = \{ \quad & (\,(use^1_{12} \vee use'^1_1) \wedge use^2_2 \,,\, t^2_2 \prec_\infty \{t^1_2,\, t^1_3,\, t^1_4\}\,), \\ & (\,(use^2_{21} \vee use'^2_2) \wedge use^1_1 \,,\, t^1_2 \prec_\infty \{t^2_2,\, t^2_3,\, t^2_4\}\,), \\ & (\,use^1_{12} \wedge s^2_b \,,\, t^2_1 \prec_\infty t^1_3\,), \\ & (\,use^1_{12} \wedge s^2_b \,,\, t^2_1 \prec_\infty t^1_3\,), \\ & (\,use^1_1 \wedge use^2_2 \,,\ \text{cycle}\ ) \quad \} \end{aligned}$$

La composition des deux fonctions de priorité donne lieu à *cycle* dans l'ordre de priorité aux états $use^1_1 \wedge use^2_2$. En effet, la composition des ordres de priorités $\{t^2_2 \prec_\infty \{t^1_2,\, t^1_3,\, t^1_4\}\} \oplus \{t^1_2 \prec_\infty \{t^2_2,\, t^2_3,\, t^2_4\}\}$ comporte un cycle constitué par $t^2_2 \prec_\infty t^1_2$ et $t^1_2 \prec_\infty t^2_2$. Ceci indique un blocage potentiel dans l'état $(use^1_1,\, use^2_2)$. En effet, dans cet état du système, le thread th_1 est synchronisé sur l'objet O_1 et tente de se synchroniser sur l'objet O_2, le thread th_2 est bloqué d'une manière symétrique en l'attente de O_1. Cet exemple illustre la capacité des priorités à donner un moyen de diagnostic intéressant pour l'analyse de blocage du système.

Nous commençons par définir les fonctions de priorité relatives à chaque objet. Soit $O \in \Omega$ un objet partagé. L'exclusion mutuelle sur l'objet O est assurée par la fonction de priorité :

$$pr^{O}_{mutex} = \bigoplus_{i,j \in \{1,..,n\}} \{(Use_i(O) \wedge Request_j(O) \,,\, BGN_j(O) \prec_\infty Use_i(O) \circ)\} \tag{4.2}$$

L'exclusion mutuelle sur tous les objets de l'application est donc garantie par la fonction de priorité :

$$pr_{mutex} = \bigoplus_{O \in \Omega} pr^{O}_{mutex} \tag{4.3}$$

4.2.2.3 Modèle de l'application

Nous définissons le modèle de l'application par la composition parallèle des systèmes de transition temporisés synchronisés sur les actions de communication et restreint par les fonctions de priorité de communication et d'exclusion mutuelle sur les objets partagés. Il en découle :

$$(TTS, pr) = (||_\Lambda \{TTS_i\}_{1 \leq i \leq n}, pr_{comm} \oplus pr_{mutex})$$

La section suivante donne un exemple détaillé de la construction du modèle temporisé d'une application.

4.3 Exemple

Dans ce paragraphe nous présentons les différentes étapes de construction du modèle temporisé d'une application Java multi-threadée temps-réel écrite en RTSJ et conforme aux restrictions énoncées au chapitre 2. Le paragraphe 4.3.1 présente l'application étudiée. Les systèmes de transition temporisés et les règles de priorité relatives à la communication et l'exclusion mutuelle sont établis au paragraphe 4.3.2.

4.3.1 Présentation de l'application

L'application *PutterGetter* étudiée ici est un exemple de producteur / consommateur. L'application se compose de deux threads $\Theta = \{putter, getter\}$ qui agissent sur deux objets partagés : $\Omega = \{ioBuffer, dataBuffer\}$.

Objets partagés

- $ioBuffer$ instance du type $IOBuffer$, est le tampon d'entrée / sortie associé à un matériel de communication qui lie l'application à son environnement. La taille du tampon est de 512 octets. Les méthodes *get*() et *put*() permettent, respectivement, d'enlever et d'insérer un élément dans le tampon. La méthode *send*() émet le contenu du tampon. La méthode *receive*() rafraîchitle contenu du tampon avec les données reçues. L'émission et réception ne doivent se faire qu'après avoir rempli le tampon.
- $dataBuffer$ instance du type $DataBuffer$, est un tampon de données qui sert à stocker les données reçues dans le $ioBuffer$. Sa taille est de 4096 octets. De la même manière, les méthodes *put*() et *get*() permettent d'insérer, respectivement retirer, un élément du tampon.

Threads

- Le thread *putter* est périodique de période $T_{putter} = 5300\,ms$; son échéance est $D_{putter} = 4000\,ms$. Ce thread s'occupe de la mise à jour du tampon de données $dataBuffer$. Il commence par rafraîchir le contenu du tampon de communication en invoquant la méthode *receive*() sur $ioBuffer$ puis tente de transférer son contenu à $dataBuffer$. Si ce dernier ne contient pas assez d'espace libre, *putter* se met en attente de notification pour $1000\,ms$; si à son réveil (par notification ou par expiration du temps d'attente) l'espace requis est disponible, il effectue la mise à jour, sinon il termine son cycle d'exécution. Le code de la logique du thread `handlePeriod()` est présenté à la table 4.2, dans le type `Putter`.
- Le thread *getter* est périodique de période $T_{getter} = 4100\,ms$; son échéance est $D_{getter} = 3000\,ms$. Il opère d'un manière similaire à celle du thread *putter*, mais dans le sens "inverse". Il commence par vérifier la présence de données dans le tampon $dataBuffer$ et transfère celles-ci dans le tampon $ioBuffer$; après le transfert il émet une notification.

Si $dataBuffer$ ne contient pas suffisamment de données pour remplir $ioBuffer$, le thread termine son cycle. Le code de la logique du thread `handlePeriod()` est présenté à la table 4.2, dans le type `Getter`.

```
public class Getter extends
  PeriodicNoHeapRealTimeThread {
    ...
  public void handlePeriod() {
    synchronized (dataBuffer) {
      synchronized (ioBuffer) {
        int datasize = dataBuffer.getSize();
        int iosize = ioBuffer.getPacketSize();
        if (datasize - iosize >= 0) {
          int i = 0;
          while (i < iosize) {
            int value = dataBuffer.get();
            ioBuffer.put(value);
            i++;
          }
          ioBuffer.send();
        }
        dataBuffer.notifyAll();
      }
    }
  }
    ...
}
```

```
public class Putter extends
  PeriodicNoHeapRealTimeThread {
    ...
  public void handlePeriod() {
    synchronized (dataBuffer) {
      synchronized (ioBuffer) {
        int datasize = dataBuffer.getSize();
        int iosize = ioBuffer.getPacketSize();
        int dataMaxSize = dataBuffer.getMaxSize();
        if (datasize + iosize >= dataMaxSize) {
          try {
            dataBuffer.wait(1000);
          } catch (InterruptedException e) {}
        }
        if (datasize + iosize < dataMaxSize) {
          ioBuffer.receive();
          int i = 0;
          while (i < iosize) {
            int value = ioBuffer.get();
            dataBuffer.put(value);
            i++;
          }
        }
      }
    }
  }
    ...
}
```

TAB. 4.2 – Extraits du code des threads *getter* (`Getter.java`) et *putter* (`Putter.java`). Le code complet est donné à l'annexe A.

4.3.2 Construction du Modèle

La construction du modèle de l'application est effectuée d'une manière modulaire. En premier lieu sont construits les systèmes de transition temporisés avec priorité $(TTS_{getter}, pr_{getter})$ et $(TTS_{putter}, pr_{putter})$ qui modélisent, respectivement, les threads *getter* et *putter*. Ensuite sont construites les fonctions de priorité qui modélisent la communication et l'exclusion mutuelle. Finalement, Le modèle du système est obtenu par composition.

Modélisation des threads

Les figures 4.11 et 4.12 illustrent, respectivement, les systèmes de transition temporisés TTS_{getter} et TTS_{putter} extraits à partir du code de l'application à l'aide de notre outil *JediTool* (présenté au chapitre 5).

Les structures de contrôle sont extraites en appliquant l'algorithme présenté en 3.3. Les informations sur les temps d'exécution et le nombre maximal d'itération des boucles sont fournis, respectivement, par les fonctions *exec* (calculée à partir des annotations dans le code source) et *iterations* (spécifiée par l'utilisateur). La fonction *exec* est définie dans la table 4.3 (les temps sont exprimés en ms).

$$
\begin{aligned}
&exec(\,IOBuffer.\_send()\,) = (30, 45)\\
&exec(\,IOBuffer.\_receive()\,) = (15, 20)\\
&exec(\,BaseBuffer.put()\,) = (3, 5)\\
&exec(\,BaseBuffer.get()\,) = (2, 4)\\
&exec(\,BaseBuffer.isFull()\,) = (1, 2)\\
&exec(\,BaseBuffer.isEmpty()\,) = (1, 2)\\
&exec(\,BaseBuffer.getSize()\,) = (1, 1)\\
&exec(\,BaseBuffer.getMaxSize()\,) = (1, 1)
\end{aligned}
$$

TAB. 4.3 – Fonction *exec* définie à partir des annotations *Javadoc* du code source.

Par abus de notation, nous définissons la fonction *iterations* sur les états de contrôle d'itération correspondants aux instructions de boucle dans le source. La fonction est définie par $iterations(s^3_{getter}) = 128$ et $iterations(s^6_{putter}) = 128$. Il est à noter que ces valeurs, dictées par l'utilisateur, proviennent de la spécification du programme : les itérations se font sur les éléments du tampon $ioBuffer$ qui est de taille 512 octets (128 valeurs de type *int*).

Le temps d'expiration du délai d'attente est extrait du code source à partir du littéral passé en argument à l'instruction d'invocation de la méthode $wait()$. Le temps d'attente vaut $1000\,ms$.

Les systèmes de transition sont complétés par les boucles modélisant l'arrivée périodique des threads : les états de sommeil s^{15}_{getter} et s^{17}_{putter}, les transitions d'arrivée (étiquetées par *arrive*) t^{16}_{getter} et t^{20}_{putter}, et les transitions de sommeil (étiquetées par *sleep*) t^{17}_{getter} et t^{21}_{putter}. Les contraintes temporelles sont ajoutées au systèmes de transition et les échéances des threads sont rétropropagées conformément à la démarche présentée aux paragraphes 4.1.2 et 4.2.2.

Les fonctions de priorité pr_{getter} et pr_{putter} se calculent simplement à partir des ensembles d'états d'attente et des ensembles de transitions d'attente de notification et d'expiration de délai :

- $Waiting_{putter}(dataBuffer) = \{s^3_{putter}\}$,
- $WAIT_{putter}(dataBuffer) = \{t^3_{putter}\}$ et
- $TIMEOUT_{putter}(dataBuffer) = \{t^4_{putter}\}$.

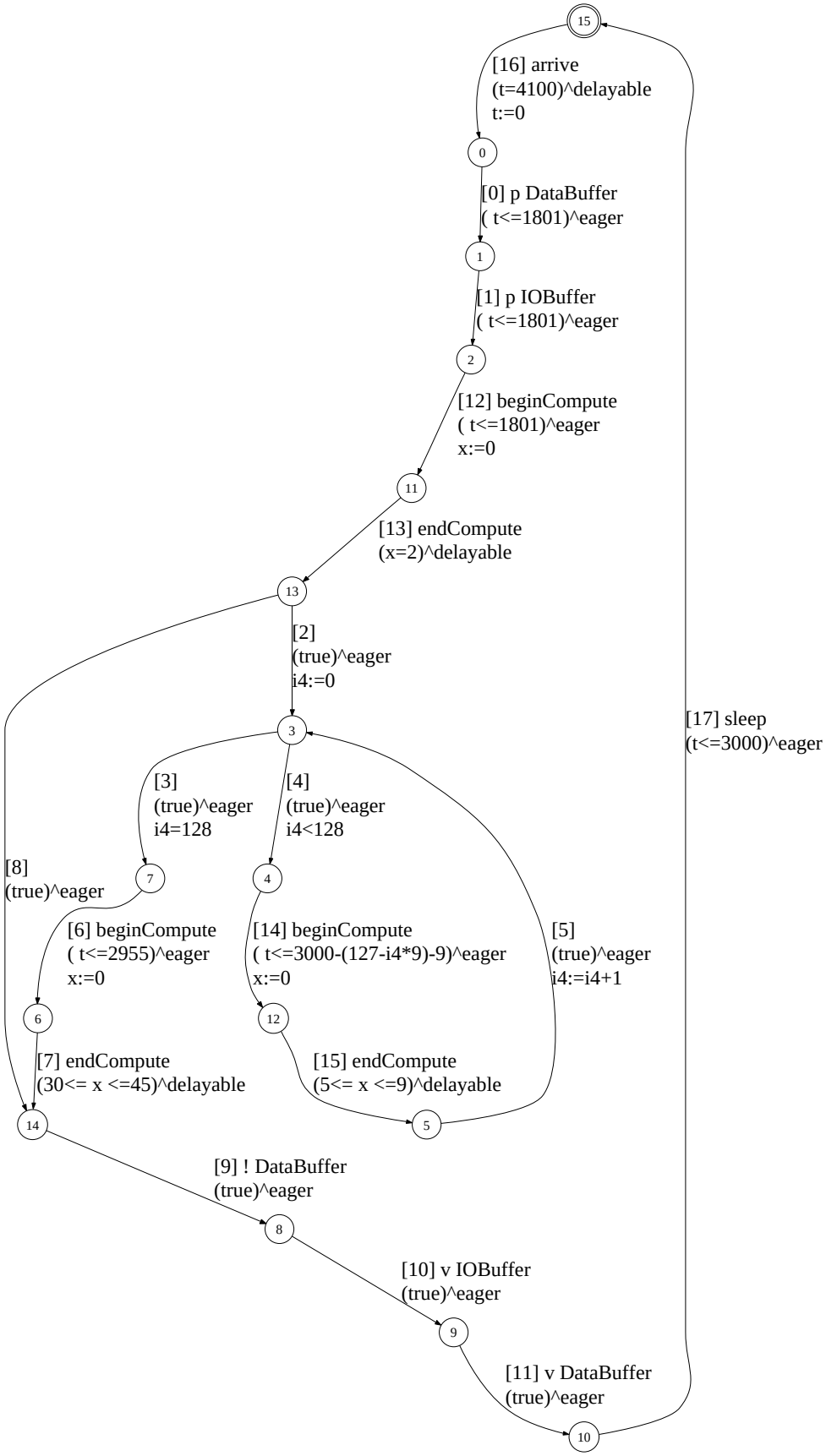


FIG. 4.11 – Système de transition temporisé TTS_{getter} (généré automatiquement avec *Jedi-Tool*).

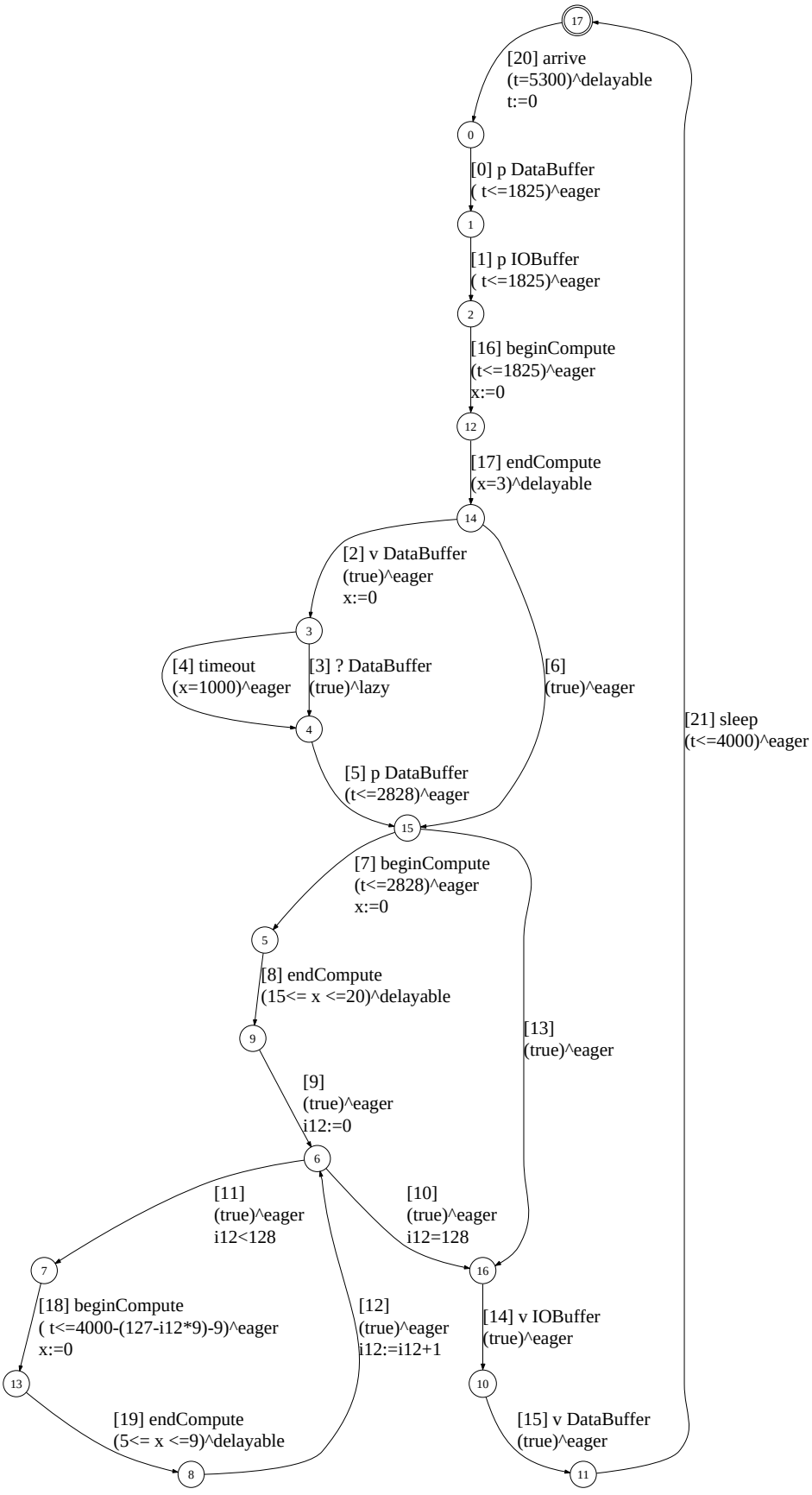


FIG. 4.12 – Système de transition temporisé TTS_{putter} (généré automatiquement avec *Jedi-Tool*).

La fonction de priorité associé au thread *putter* s'en déduit par :

$$pr_{putter} = pr^{dataBuffer}_{wait,putter} = \{\ (s^3_{putter}\ ,\ t^3_{putter} \prec_\infty t^4_{putter})\ \}$$

Le thread *getter* n'effectue pas d'attente de notification, la fonction de priorité pr_{getter} est vide.

Modélisation de la communication

Seul le thread *getter* émet une notification, on a $NOTFY_{getter}(dataBuffer) = \{t^{14}_{getter}\}$. L'ensemble de synchronisation se déduit par $\Lambda = \{t^9_{getter} | t^3_{putter}\}$. La fonction de priorité pour le progrès maximal en découle :

$$pr_\Lambda = \left\{(s^{14}_{getter} \wedge s^3_{putter}, \{\ t^3_{putter} \prec_\infty t^9_{getter} | t^3_{putter}\ ,\ t^9_{getter} \prec_\infty t^9_{getter} | t^3_{putter}\ \})\right\}$$

La fonction de priorité qui modélise la communication est définie par $pr_{comm} = pr_\Lambda \oplus pr_{getter} \oplus pr_{putter}$. Ainsi :

$$\begin{aligned} pr_{comm} = \quad & \left\{(s^{14}_{getter} \wedge s^3_{putter}, \{\ t^3_{putter} \prec_\infty t^9_{getter} | t^3_{putter}\ ,\ t^9_{getter} \prec_\infty t^9_{getter} | t^3_{putter}\ \})\right. \\ & \left.(s^3_{putter}\ ,\ t^3_{putter} \prec_\infty t^4_{putter})\right\} \end{aligned}$$

Modélisation de l'exclusion mutuelle

L'exclusion mutuelle est modélisée par la fonction de priorité pr_{mutex}. Cette fonction est construite, conformément au paragraphe 4.2.2, à partir des contextes de synchronisation Σ_{putter} et Σ_{getter} obtenus lors de la modélisation des instructions du programme. Les ensembles d'états et transitions de synchronisation sur les objets sont obtenus automatiquement à l'aide de notre outil d'analyse *JediTool*.

Objet $dataBuffer$**, thread** *getter*

- $Use_{getter}(dataBuffer) = \{s^1_{getter}, s^2_{getter}, s^3_{getter}, s^4_{getter}, s^5_{getter}, s^6_{getter}, s^7_{getter}, s^8_{getter}, s^9_{getter}, s^{11}_{getter}, s^{12}_{getter}, s^{13}_{getter}, s^{14}_{getter}\}$
- $Request_{getter}(dataBuffer) = \{s^0_{getter}\}$
- $Use_{getter}(dataBuffer)\circ = \{t^k_{getter}\}_{1\leq k\leq 15}$
- $BGN_{getter}(dataBuffer) = \{t^0_{getter}\}$

Objet $ioBuffer$**, thread** *getter*

- $Use_{getter}(ioBuffer) = \{s^2_{getter}, s^3_{getter}, s^4_{getter}, s^5_{getter}, s^6_{getter}, s^7_{getter}, s^8_{getter}, s^{11}_{getter}, s^{12}_{getter}, s^{13}_{getter}, s^{14}_{getter}\}$
- $Request_{getter}(ioBuffer) = \{s^1_{getter}\}$

- $Use_{getter}(ioBuffer)\circ = \{t^2_{getter}, t^3_{getter}, t^4_{getter}, t^5_{getter}, t^6_{getter}, t^7_{getter}, t^8_{getter}, t^9_{getter}, t^{10}_{getter}, t^{12}_{getter}, t^{13}_{getter}, t^{14}_{getter}, t^{15}_{getter}\}$
- $BGN_{getter}(ioBuffer) = \{t^1_{getter}\}$

Objet $dataBuffer$**, thread** $putter$

- $Use_{putter}(dataBuffer) = \{s^1_{putter}, s^2_{putter}, s^5_{putter}, s^6_{putter}, s^7_{putter}, s^8_{putter}, s^9_{putter}, s^{10}_{putter}, s^{12}_{putter}, s^{13}_{putter}, s^{14}_{putter}, s^{15}_{putter}, s^{16}_{putter}\}$
- $Request_{putter}(dataBuffer) = \{s^0_{putter}, s^4_{putter}\}$
- $Use_{putter}(dataBuffer)\circ = \{t^1_{putter}, t^2_{putter}, t^6_{putter}, t^7_{putter}, t^8_{putter}, t^9_{putter}, t^{10}_{putter}, t^{11}_{putter}, t^{12}_{putter}, t^{13}_{putter}, t^{14}_{putter}, t^{15}_{putter}, t^{16}_{putter}, t^{17}_{putter}, t^{18}_{putter}, t^{19}_{putter}\}$
- $BGN_{putter}(dataBuffer) = \{t^0_{putter}, t^5_{putter}\}$

Objet $ioBuffer$**, thread** $putter$

- $Use_{putter}(ioBuffer) = \{s^2_{putter}, s^3_{putter}, s^4_{putter}, s^5_{putter}, s^6_{putter}, s^7_{putter}, s^8_{putter}, s^9_{putter}, s^{12}_{putter}, s^{13}_{putter}, s^{14}_{putter}, s^{15}_{putter}, s^{16}_{putter}\}$
- $Request_{putter}(ioBuffer) = \{s^1_{putter}\}$
- $Use_{putter}(ioBuffer)\circ = \{t^2_{putter}, t^3_{putter}, t^4_{putter}, t^5_{putter}, t^6_{putter}, t^7_{putter}, t^8_{putter}, t^9_{putter}, t^{10}_{putter}, t^{11}_{putter}, t^{12}_{putter}, t^{13}_{putter}, t^{14}_{putter}, t^{16}_{putter}, t^{17}_{putter}, t^{18}_{putter}, t^{19}_{putter}\}$
- $BGN_{putter}(ioBuffer) = \{t^1_{putter}\}$

Les fonctions de priorité $pr^{dataBuffer}_{mutex}$ et $pr^{ioBuffer}_{mutex}$ se déduisent directement des définitions précédentes par le biais de la formule 4.2. Finalement, nous obtenons la fonction de priorité qui modélise l'exclusion mutuelle sur les objets partagés par composition :

$$pr_{mutex} = pr^{ioBuffer}_{mutex} \oplus pr^{dataBuffer}_{mutex}$$

Analyse des fonctions de priorité

En analysant la fonction de priorité pr_{mutex} construite au paragraphe précédent on constate un cycle dans l'ordre de priorité à l'état $(s^4_{putter}, s^{16}_{getter})$. A cet état, on a :

- $t^5_{putter} \prec_\infty t^6_{getter}$,
  car $t^5_{putter} \in BGN_{putter}(dataBuffer)$ et $t^6_{getter} \in Use_{getter}(dataBuffer)\circ$
- $t^6_{getter} \prec_\infty t^5_{putter}$
  car $t^6_{getter} \in BGN_{getter}(ioBuffer)$ et $t^5_{putter} \in Use_{putter}(ioBuffer)\circ$

Ce cycle dans la fonction de priorité révèle un blocage potentiel dans le système dû à l'acquisition croisée des objets partagés $ioBuffer$ et $dataBuffer$ par les deux threads. Ce blocage a été difficile à détecter durant le test du système. De plus, l'ordre d'acquisition des verrous des objets semble être le même dans les deux logiques des threads : d'abord $dataBuffer$ puis $ioBuffer$ dans les deux cas. C'est l'instruction d'attente de notification qui inverse cet ordre dans le thread $putter$ et crée le blocage. L'analyse des fonctions de priorité offre un moyen

efficace pour la détection de telles situations de blocage, cette tâche a été automatisée dans notre outil *JediTool*.

Afin de pallier le problème de blocage détecté, plusieurs solutions sont possibles. Une première solution serait de modifier le programme afin d'éviter le verrouillage croisé des objets. Par exemple, déplacer le début de synchronisation sur l'objet $ioBuffer$ dans la logique du thread *putter* de la ligne 6 à la ligne 15 (*cf.* code à la table 4.2). Dans ce cas, l'exécution de l'attente de notification n'inverse pas l'ordre de verrouillage des objets et le blocage disparaît. Toutefois, il n'est pas toujours possible de modifier le code sans entraver au bon fonctionnement et à la spécification du système.

Une deuxième solution est de mettre en exclusion mutuelle, d'un côté, les états où le thread *getter* possède le verrou de $dataBuffer$ et demande celui de $ioBuffer$; et d'autre côté, les états où le thread *putter* relâche le verrou de $dataBuffer$ et garde le verrou de l'objet $ioBuffer$. Cela revient à mettre en exclusion mutuelle les états $\{s^3_{putter}, s^4_{putter}\}$ avec l'état s^1_{getter}, rendant ainsi l'état de blocage $(s^4_{putter}, s^1_{getter})$ inaccessible. La fonction de priorité qui assure l'exclusion mutuelle sur ces états s'écrit :

$$\begin{aligned} pr_{deadlock} = \quad & \{((s^3_{putter} \vee s^4_{putter}) \wedge s^0_{getter} \, , \, t^0_{getter} \prec_\infty \{t^3_{putter}, t^4_{putter}, t^5_{putter}\}), \\ & (s^{14}_{putter} \wedge s^1_{getter} \, , \, t^2_{putter} \prec_\infty t^1_{getter})\} \end{aligned}$$

La fonction de priorité $pr_{mutex} \oplus pr_{deadlock}$ assure l'exclusion mutuelle et évite le blocage potentiel détecté dans le modèle du système. Le modèle du système (TTS, pr) est finalement obtenu en composant les fonctions de priorité obtenues :

$$(TTS, pr) = (TTS_{putter} ||_\Lambda TTS_{getter}, pr_{comm} \oplus pr_{mutex} \oplus pr_{deadlock})$$

Chapitre 5

Implantation de *JediTool*

5.1 Motivation

JediTool est un outil de modélisation automatique des applications temps-réel concurrentes écrites en Java. Il implémente la méthodologie et les algorithmes que nous avons présentés dans les chapitres précédents. A partir du code source et d'un ensemble d'informations complémentaires, l'outil construit automatiquement le modèle temporisé de l'application. *JediTool* implémente les opérations suivantes :

- Construction automatique des systèmes de transition temporisés et des règles de priorité qui modélisent chaque thread.
- Génération automatique des fonctions de priorité qui modélisent la communication inter-thread pr_{comm} (mécanisme d'attente/notification).
- Génération automatique des fonctions de priorité qui modélisent l'exclusion mutuelle sur les objets partagés pr_{mutex}. De plus, *JediTool* permet l'analyse de ces fonctions afin de détecter les blocages potentiels.
- Exportation des modèles générés vers les outils d'analyse et de vérification, notamment la plate-forme de vérification IF-2 [BGM02].

JediTool est intégré comme composant (*plug-in*) dans l'environnement de développement à code ouvert Eclipse [OTI03] de IBM.

La section 5.2 présente le principe de fonctionnement de l'outil. Ensuite, la section 5.3 explique l'utilisation de *JediTool* dans l'environnement Eclipse. Finalement, la section 5.4 présente l'architecture interne de l'outil et les choix technologiques effectués.

5.2 Présentation de *JediTool*

JediTool prend comme entrée le code source de l'application formé par les différentes unités de compilation (fichiers `.java`) et un fichier de configuration qui contient la liste des threads et objets partagés créés lors de la phase d'initialisation (fichier `.jxml`) ; ce fichier contient aussi les descriptions temporelles (temps d'arrivées et échéance) et fonctionnelles (logiques) des threads. Après génération du modèle de l'application, l'utilisateur peut choisir d'effectuer les analyses fournies par l'outil et/ou traduire le modèle dans des formats appropriés aux outils de vérification. Le fonctionnement et le contexte de l'outil sont illustrés dans la figure 5.1.

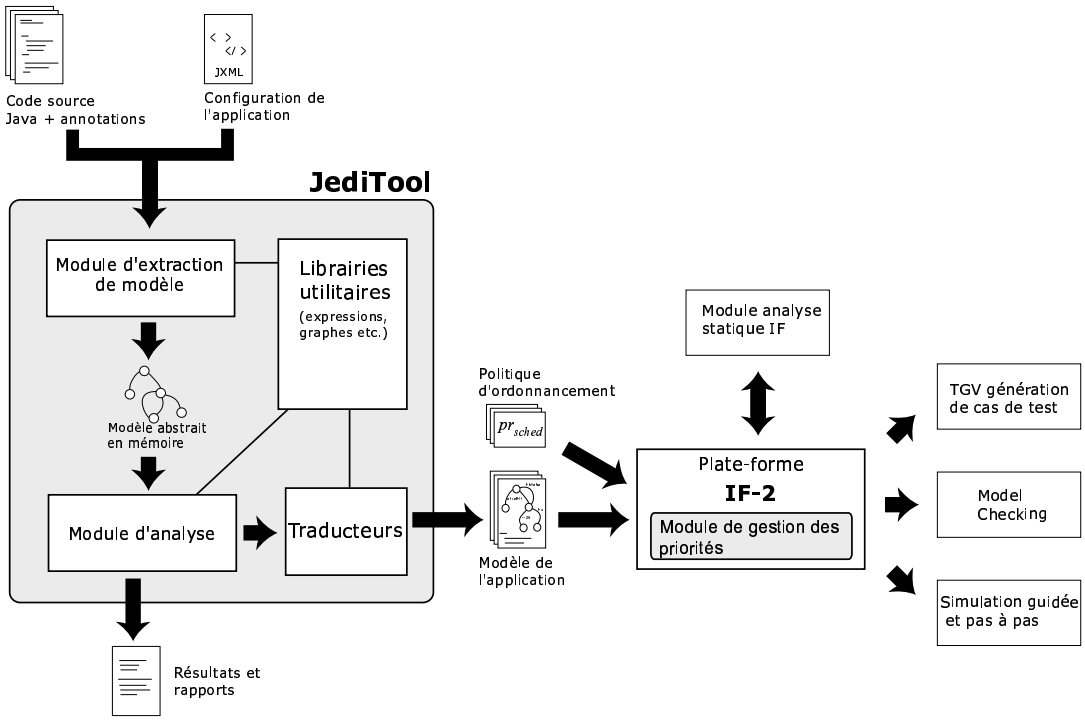


FIG. 5.1 – Schéma général d'organisation et contexte de *JediTool*

La construction du modèle d'une application se fait suivant différentes phases chacune correspondant à un étage de *JediTool*.

Construction des systèmes de transition temporisées

Le module d'extraction de modèle construit d'une manière modulaire une représentation en mémoire des systèmes de transition temporisés qui modélisent chaque thread. Il conserve au niveau de cette représentation la correspondance entre les états du modèle et les positions associées dans le source afin de retrouver dans le code les traces d'exécution ou d'erreurs éventuelles, obtenues en analysant le modèle. En premier lieu, chaque logique associée à un thread

déclarée dans le fichier de configuration est localisée dans le source. L'unité de compilation correspondante est compilée et son arbre abstrait est généré à l'aide des composants dédiés à Java dans Eclipse. Les visiteurs du module d'extraction de *JediTool* visitent alors les arbres abstraits et génèrent à la volée des systèmes de transition décorés par les temps d'exécution et par l'usage des objets partagés. Chaque instruction d'appel de méthode rencontrée entraîne la recherche, la compilation et la modélisation de l'unité de compilation qui contient la méthode appelée. Les systèmes de transition sont obtenus en appliquant les règles de réécritures définies au chapitre 3. A la fin de cette étape, on obtient un modèle en mémoire qui sera complété et analysé par le module d'analyse.

Rétropropagation des échéances et génération et analyse des fonctions de priorité

Le module d'analyse se charge essentiellement des tâches suivantes :

- Rétropropagation des échéance des threads,
- Synthèse des fonctions de priorité qui modélisent la communication par attente / notification,
- Synthèse des fonctions de priorité qui modélisent l'exclusion mutuelle,
- Analyse des fonctions de priorité afin de détecter les blocages potentiels.

Afin de générer les fonctions qui modélisent la communication, les ensembles de transitions synchronisées sont calculés conformément à la démarche présentée en 4.2.2.1. Les fonctions de priorité pour l'exclusion mutuelle sont obtenues à partir des fonctions de contextes de synchronisation générées par le module d'extraction. Les fonctions de priorité sont analysées afin de détecter les cycles dans les ordres de priorité et identifier les états où ils surviennent. Chaque blocage potentiel détecté est défini par l'ensemble des états de contrôle du système où il risque de se produire ainsi que les positions exacte dans le source et les chaînes d'appel effectuées par chaque thread conduisant à ces états (rappelons qu'une position dans le source peut être atteinte par un thread donné par plusieurs chemins possible dans le graphe d'appel). Les échéances des threads sont rétropropagées afin de contraindre les gardes des transitions des systèmes de transitions temporisés comme présenté au paragraphe 4.2.2.

Exportation du modèle et génération de rapports

Une fois le modèle temporisé d'une application est construit, *JediTool* offre la possibilité de le traduire dans les formats d'outils de vérification et de générer différents types de rapports et informations.

Nous avons implémenté l'exportation vers le format de l'outil *Prometheus* [Gößl01b]. Grâce à cette connexion, les systèmes de transitions temporisés générés par *JediTool* peuvent être analysés afin de vérifier certaines propriétés structurelles telles que l'absence de cycles zénons

et la vivacité. Nous avons aussi implémenté l'exportation des modèles vers la boîte à outils IF-2 que nous avons étendue afin de spécifier les fonctions de priorités. Ces modifications et la connexion avec *JediTool* sont détaillées dans le paragraphe suivant.

JediTool comprend aussi plusieurs traducteurs qui rapportent différentes informations sur les modèles et les résultats d'analyse. Parmi les formats et informations générés, on a :

- Fichiers de liaison entre les états de contrôle des modèles aux positions dans le code source. Ceci permet d'identifier les contre-exemples produits par la vérification.
- Fonctions de contexte de synchronisation et les actions d'accès et utilisation des objets partagés.
- Séquences d'appel de méthode pour chaque thread. Cette information sert à générer les graphes d'appels exactes pour chaque thread ainsi que les diagrammes de séquence UML.
- Description détaillée des états de blocage potentiels détectés lors de l'analyse des fonctions de priorités.
- Pire temps d'exécution des threads. L'outil fournit aussi un rapport sur le pire temps d'exécution restant à chaque état de contrôle du modèle de chaque thread.
- Exportation vers le format *DOT* [GN00] pour la visualisation et l'impression (les figures 4.12 et 4.11 sont des exemples des fichiers produits).

Ces informations sont produites sous forme de fichiers texte. De plus, nous avons implémenté des interfaces utilisateurs afin de permettre de visualiser ces informations dans l'environnement Eclipse pour faciliter leur présentation et permettre de les retrouver directement dans le source d'une manière interactive.

Extension de la plate-forme IF-2 et connexion avec *JediTool*

La boîte à outils IF-2 regroupe un ensemble d'outils pour la modélisation et l'analyse des systèmes temps-réel. Ces outils sont structurés autour de trois niveaux de description :

1. *Niveau Spécification*
   La boîte à outils IF-2 accepte des langages de description standards des systèmes temps-réel, tels que UML [BFG+99, OGO04] et SDL [Tel93]. Ces descriptions sont transformées vers le langage intermédiaire IF qui leur sert de sémantique.
2. *Niveau Langage Intermédiaire*
   Ce niveau correspond au langage IF (*Intermediate Format*) où les systèmes sont modélisés par des systèmes de transition temporisés communicants. Des outils d'analyse statiques s'appliquent à ce niveau notamment pour la réduction de nombre d'horloges, le *slicing* des variables et l'élimination du code mort [BFG00]. La description IF sert à la génération de cas de test à travers la connexion avec l'outil TGV[FJJV97]. Différents outils de validation de la boîte à outil permettent d'effectuer une exploration exhaustive, guidée ou aléatoire de la description IF-2 qui donne accès à son modèle sémantique.

3. *Niveau Sémantique*
   Le modèle sémantique correspond aux exécutions de la description IF d'un système. Cette sémantique est représentée par un système de transition étiqueté (LTS) qui peut être analysé par des outils de vérification, tels que les outils CADP [JHA+96].

La boîte à outils IF-2 a été utilisée avec succès pour traiter des systèmes de taille réelle [JG01, BFG00].

JediTool vient compléter la boîte à outils IF-2 comme module frontal (*front-end*) dédié aux applications temps-réel développées avec la technologie Java. Afin de réaliser l'intégration de notre outil, nous avons étendu le langage IF et le simulateur afin de supporter les fonctions de priorité. L'extension que nous avons implémentée est similaire à celle que nous avons effectuée sur les premières versions de la boîte à outils [Nak01]. La grammaire du langage intermédiaire a été étendue afin d'intégrer la spécification des fonctions de priorité. Les fonctions de priorités ont été ajoutées à la description du système conformément au productions suivantes :

$$
\begin{array}{rl}
\mathit{priorityRules} ::= & \textbf{priorityrules} \\
& \quad \{\mathit{priorityRule}\}* \\
& \textbf{endpriorityrules;} \\
\mathit{priorityRule} ::= & \mathit{ruleId} : \mathit{processId}_1 < \mathit{processId}_2 \ \textbf{if}\ \mathit{bexp};
\end{array}
$$

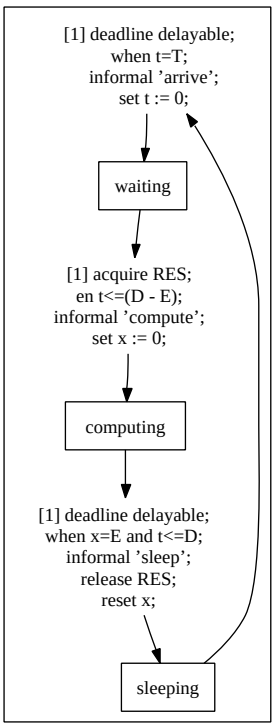
La règle de production *priorityRules* définit la section dédiée aux fonctions de priorités. La règle de production *priorityRule* est identifiée par son nom unique *ruleId* est donne la priorité au processus $\mathit{processId}_1$ sur le processus $\mathit{processId}_2$ si l'expression *bexp*, expression booléenne du langage, est vraie. L'exemple suivant illustre l'usage des priorités dans le langage IF.

Exemple 5.1 *Reprenons l'exemple 4.3 donné au chapitre 4 et présentant trois threads partageant une ressource commune selon la politique d'admission* FIFO. *Afin de simplifier la présentation de l'exemple, considérons une période et un temps d'exécution identique pour les trois threads. Une description du système en IF ainsi qu'une représentation graphique sont données à la table 5.1. L'exclusion mutuelle sur la ressource* `RES` *partagée aux états computing est garantie par les actions* `acquire RES` *et* `release RES` *correspondant respectivement à l'acquisition et à la libération de la ressource. Afin de modéliser la politique d'admission* FIFO, *il suffit d'ajouter à la description du système la déclaration de fonction de priorité suivante :*

```
prorityrules
 fiforule : x < y   if    (({Process}y) instate waiting)
                          and (({Process}x) instate waiting)
                          and ( (timeval ({Process}x).t) < (timeval ({Process}y).t) );
endpriorityrules;
```

En plus des extensions effectuées sur l'analyseur lexical, syntaxique et sur l'arbre abstrait du langage, nous avons modifié le module de d'exploration de la boîte à outils afin d'obtenir

```
system fifo;
const D = 3;
const T = 5;
const E = 2;
resource RES;
process Process(3);
   var t clock public;
   var x clock;
   state sleeping #start;
      deadline delayable;
      when t=T;
      informal "arrive";
      set t :=0;
      nextstate waiting;
   endstate;
   state waiting;
      deadline eager;
      acquire RES;
      when t<=D-E;
      informal "compute";
      set x :=0;
      nextstate computing;
   endstate;
   state computing;
      deadline delayable;
      when x=E and
                t<=D;
      informal "sleep";
      release RES;
      reset x;
      nextstate sleeping;
   endstate;
endprocess;
endsystem;
```



TAB. 5.1 – Modèle IF-2 du système et représentation graphique du modèle d'un thread.

le modèle sémantique correcte lors de l'exploration de l'espace d'état du système restreint par les fonctions de priorité. Grâce à ces extensions, les modèles générés par *JediTool* sont traduit directement vers le langage intermédiaire IF.

5.3 Utilisation de *JediTool*

Afin d'illustrer l'utilisation de *JediTool*, reprenons l'application *PutterGetter* présentée au paragraphe 4.3 du chapitre 4 (le code source complet de l'application est donné à l'annexe A). Dans ce paragraphe sont présentés les étapes de modélisation et analyse de l'application avec *JediTool*.

Annotations temporelles

Afin de modéliser les calculs effectués par chaque threads, l'outil construit la fonction *exec* qui associe à certaines méthodes du programme, appelées méthodes de calcul, un temps minimum et maximum d'exécution (cf. paragraphe 3.2.2). Dans la pratique, cette information est calculée soit par profilage de l'application sur la plate-forme cible soit par analyse du code compilé (natif [dVCF$^+$02] ou bytecode [BBW00a]). *JediTool* collecte cette information au niveau des commentaires Javadoc des méthodes. Le temps minimum, noté E^l, et maximum,

noté E^u, d'exécution est indiqué par une entrée spéciale Javadoc comme suit : `@exec` E^l E^u. A titre d'exemple, nous avons les annotations suivantes dans le source de l'application *PutterGetter* :

```
/**                                  |  /**
 * @exec 30 45                       |   * @exec 2  4
 */                                  |   */
private native void                  |  public int get() { ...
_send();                             |
```

Fichier de configuration et génération du modèle

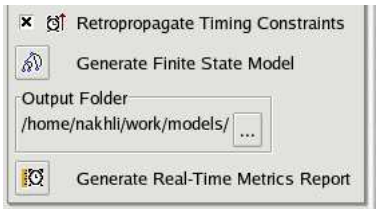
Le fichier de configuration dénombre les threads et les objets partagés actifs en phase mission d'une application, les caractéristiques temporelles et les logiques des threads y sont rapportées. La grammaire DTD qui définit les éléments accepté par les fichiers JXML est présentée à l'annexe A. L'application *PutterGetter* se compose de deux threads périodiques et deux objets partagés. Le fichier de configuration de l'application est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<Application id="PutterGetterApplication">
<RealTimeThread id="getter" ubound="4100" lbound="4100" deadline="3000">
      <logic  signature="()V"
            method="net.sf.jeditool.examples.Getter.handlePeriod"/>
</RealTimeThread>
<RealTimeThread id="putter" ubound="5300" lbound="5300" deadline="4000">
      <logic  signature="()V"
            method="net.sf.jeditool.examples.Putter.handlePeriod"/>
</RealTimeThread>                                                          10
<SharedObject id="ioBuffer"
          type="net.sf.jeditool.examples.IOBuffer"/>
<SharedObject id="dataBuffer"
          type="net.sf.jeditool.examples.DataBuffer"/>
</Application>
```

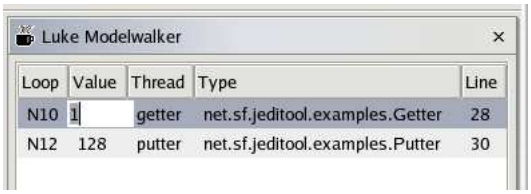
L'édition des fichiers JXML se fait par un éditeur à pages multiples apporté par *JediTool* : une page pour l'édition textuelle et une deuxième page pour l'édition par interface graphique.

L'outil *JediTool* génère le modèle d'une application pour un fichier de configuration donné choisi par l'utilisateur. Il est permis d'avoir plusieurs fichiers JXML par application afin de tester/analyser plusieurs configurations. Une fois un fichier de configuration sélectionner, l'utilisateur peut lancer la génération de modèle par un menu contextuel ou par l'interface graphique dédiée (respectivement (d) et (e), figure 5.2). Une fois le modèle en mémoire généré, l'utilisateur doit éventuellement définir le nombre d'itération de certaines boucles du programme et

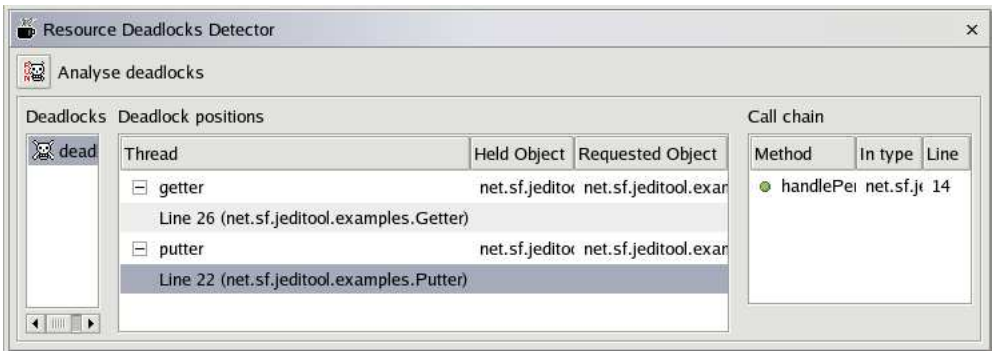
effectuer par la suite l'analyse des fonctions de priorités et l'exportation du modèle vers les outils de vérification. Ces opérations sont détaillées dans les paragraphes qui suivent.



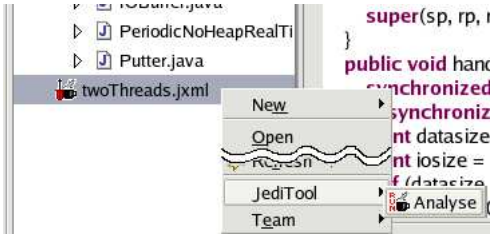
(a) Exportation du modèle



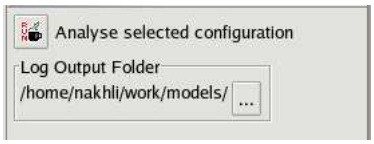
(b) Bornes supérieures des boucles



(c) Fenêtre de détection de blocages



(d) Menu contextuel pour la génération de modèle



(e) Interface de génération de modèle

FIG. 5.2 – Aperçu de quelques fenêtres de *JediTool*.

Définition des bornes des boucles

Afin de construire le système de transition qui modélise les instructions de la logique d'un thread, il faut disposer de la fonction *iterations* définie en 3.2.2. Cette fonction indique le nombre d'itérations dans chaque instruction de boucle *stm* dont l'exécution entraîne l'exécution d'une action observable ; en d'autres termes $hasObs(stm) = vrai$. *JediTool* détecte automatiquement les instructions sur lesquelles la fonction *iterations* est définie et demande

à l'utilisateur de fournir les valeurs d'itérations correspondantes. Une interface graphique présente une table où sont indiqués pour chaque instruction de boucle le thread qui exécute l'instruction, l'unité de compilation et la ligne exacte où se elle se situe. En cliquant sur un élément de la table, un éditeur Java s'ouvre, montre et sélectionne l'instruction de boucle correspondante permettant de la localisée exactement dans le source.

Analyse des fonctions de priorité

JediTool dispose d'un module d'analyse qui permet de détecter les états de contrôle où surviennent des cycles dans les fonctions de priorité ; ces cycles représentent une incohérence dans la spécification et un blocage potentiel dans le système. Une interface graphique de *JediTool* permet de présenter les blocages éventuels et permet à l'utilisateur de les retrouver directement dans le source d'une manière interactive.

Les blocages sont organisés par schéma. Un schéma de blocage est identifié par les threads qui y sont impliqués et les objets partagés sur lesquels les threads se bloquent mutuellement. Chaque schéma lui correspond une ou plusieurs positions de blocage concrètes dans le source pour chaque thread (au moins une position par thread). L'interface graphique fournie par *JediTool* dédiée à la présentation des blocages est présente trois tables : la première table affiche les schémas de blocage. Une fois un schéma est sélectionné dans cette table, la deuxième table présente la liste des threads et objets partagés sur lesquels se fait le blocage. La table affiche pour chaque thread l'objet verrouillé et celui demandé lors du blocage. L'utilisateur peut affiner la présentation en sélectionnant l'un des threads pour afficher les positions exactes dans le source où le thread se bloque. En sélectionnant l'une de ces dernières, la troisième table affiche la chaîne d'appel exacte effectuée par le thread depuis sa logique jusqu'au point de blocage.

En sélectionnant un élément de la deuxième ou troisième table, *JediTool* actionne un éditeur Java de l'environnement pour afficher et sélectionner l'instruction Java correspondant à cet élément (une instruction de synchronisation pour la deuxième table, une instruction d'appel de méthode pour la troisième table). La fenêtre d'analyse de blocage est illustrée en (c) figure 5.2 où est affiché le blocage détecté dans l'application *PutterGetter*.

Exportation du modèle

Après sélection d'un fichier de configuration et génération du modèle correspondant, l'outil permet de transcrire ce modèle dans différents formats. *JediTool* utilise des composants appelés traducteurs qui parcourent le modèle en mémoire pour cette tâche. L'interface utilisateur d'exportation du modèle, présentée en (a) figure 5.2 permet de lancer tout les traducteurs installés dans *JediTool* et de produire leurs fichiers en sortie dans le répertoire indiqué par l'utilisateur. Ce dernier a le choix de faire figurer ou non la rétropropagation des échéances des threads. Il est aussi possible à partir de cette interface de générer et afficher les métriques

temps-réel classiques dans un éditeur fourni par l'outil : pires temps d'exécution et retardement (*lateness*) par thread, le retardement maximal et le facteur d'utilisation du processeur (dans le cas de déploiement sur une machine mono-processeur).

5.4 Architecture et choix technologiques

JediTool est totalement écrit en Java, il est composé de 76 unités de compilation contenant plus de 8,000 lignes de code effectives[1] (hors commentaires et Javadoc). Bien que la prise en main des librairies de la plate-forme Eclipse et du processus d'intégration aient demandé un effort supplémentaire, l'intégration de *JediTool* au sein d'Eclipse a été bénéfique sur plusieurs plans :

- L'architecture d'Eclipse est basée sur les composants (*plug-in*). Ceux-ci possèdent des interfaces bien définies et offrent des points d'extension à l'usage des autres composants. Ceci nous a permis une réutilisation élevée et a réduit le coût de développement. L'environnement offre un ensemble intéressant de fonctionnalités simples à utiliser et étendre ; comme l'accès et gestion des fichiers, le mécanisme de sélection, la localisation, le système d'aide, la gestion des préférences *etc.* La disponibilité du source a facilité cette intégration.
- Bien que la plate-forme Eclipse dispose de composants dédiés à divers langages de programmation (C/C++, Cobol, Eiffel *etc.*), le langage Java reste le plus fourni en outils et composants. *JediTool* repose principalement sur l'ensemble de composants JDT (*Java Development Tooling*) pour le traitement des programmes Java. Ces composants facilitent la recherche, la navigation et l'édition du code source. De plus, JDT dispose de son propre compilateur Java (conforme aux JCK[2] 1.3 et 1.4). Ceci nous a permis de compiler programmatiquement le source et d'accéder à certaines structures internes du compilateur telles que les arbres abstrait des unités de compilation, les marqueurs d'erreur et les la résolution de liens entre les types.

L'architecture de *JediTool* ainsi que les connexions avec les différents composants de la plate-forme hôte Eclipse sont illustrées dans la figure 5.3 et commentées dans les paragraphes qui suivent.

Module de construction — Il prend comme entrée le fichier de configuration JXML et le code source de l'application. Nous avons choisi d'utiliser la technologie XML[3] pour le fichier

[1] L'estimation de l'effort de développement par COCOMO [Boe02] de *JediTool* est de 1,76 homme/années (21.17 homme/mois)[Whe04].

[2] JCK (*Java Compatibility Kit*) est un ensemble de tests de conformité auxquels il est impératif de se soumettre afin d'utiliser le label et marque déposée *"Java Compatible"* de Sun Microsystems.

[3] XML (*eXtensible Markup Language*) format textuel dérivé de SGML largement utilisé pour l'échange de données notamment sur Internet.

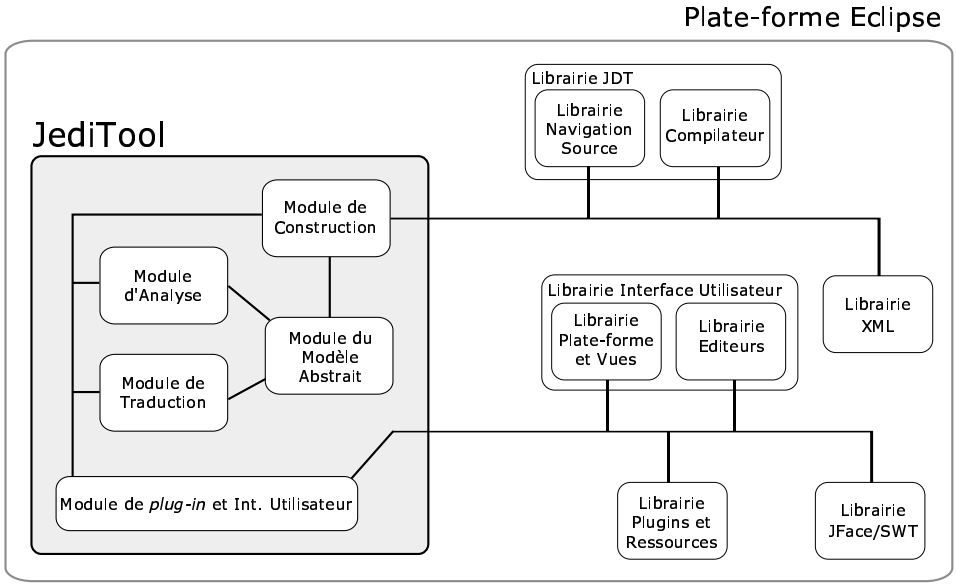


FIG. 5.3 – Architecture de *JediTool*

de configuration afin de profiter des outils et librairies de traitement existants : construction et manipulation de l'arbre abstrait du document et validation du format. Nous avons spécifié la grammaire du format JXML en DTD[4], le fichier correspondant est en annexe A. Le choix du format XML nous a facilité le développement et intégration d'un éditeur à pages multiples permettant une édition textuelle et par interface graphique du fichier de configuration.

L'exploration du code source Java est effectuée en utilisant les composants JDT pour compiler les unités de compilation à traiter. Nous avons développé nos propres visiteurs [GHJV93] de l'arbre abstrait syntaxique. Ces visiteurs construisent à la volée, lors de la visite de l'arbre abstrait, les modèles des instructions en utilisant les opérations offertes par le module du modèle abstrait.

Module du modèle Abstrait — Le module dédié au modèle abstrait permet la création et manipulation des systèmes de transition temporisés. Il offre des fonctionnalités pour ajouter et supprimer des états et transitions dans un modèle et d'attacher à ces derniers différentes sortes d'informations telles que les contraintes temporelles, les fonctions de contexte de synchronisation et les liens avec le code source. Ce module implémente aussi les opérations de concaténation et réécriture des systèmes de transition, telles qu'elles sont décrites dans le chapitre 3.

Les pires temps d'exécution ainsi que les valeurs des contraintes temporelles rétropropagées dans les modèles des threads sont paramétrés par les valeurs que prend la fonction *iterations*

[4]DTD (*Document Type Definition*) format pour la définition des blocs de données autorisés pour un fichier XML.

qui détermine le nombre d'itérations dans les boucles des modèles. Afin de ne pas refaire la rétropropagation des contraintes à chaque modification par l'utilisateur des valeurs de la fonction *iterations*, la rétropropagation se fait par des expressions formelles paramétrées par les bornes des boucles. Pour ce faire, le module du modèle abstrait fournit un ensemble de classe pour la représentation et la manipulation des expressions arithmétiques formelles (copie, simplification, évaluation *etc.*). Une fois calculées, ces expressions sont évaluées pour l'ensemble des valeurs fournies par l'utilisateur lors de l'exportation du modèle.

Ce module contient aussi d'autres classes utilitaires utilisées par les autres modules de *JediTool*, telles que les graphe d'appel et les classes représentant les fonctions de priorité utilisées dans le module d'analyse.

Module d'analyse — Le module d'analyse offre une interface programmatique pour la génération et analyse de des fonctions de priorité et la rétropropagation des échéances.

A ce niveau se trouvent les classes permettant la génération des fonctions de priorité à partir des actions de communication par attente / notification et des fonctions de contexte de synchronisation. Une fois les fonctions de priorités générées, ce module permet d'analyser ces dernières afin de détecter les circuits éventuels. L'analyse écarte certains faux blocages, notamment ceux associés à des cycles qui se produisent dans des états du système protégés par l'exclusion mutuelle. Les blocages potentiels détectés sont représentés par des classes contenant les informations sur les états du modèle où ils interviennent, les positions exactes dans le code source de l'application et le chemin emprunté par chaque thread dans le graphe d'appel jusqu'au point de blocage.

Le module d'analyse implémente un algorithme pour la rétropropagation d'échéance pour les systèmes de transition temporisé générés. Chaque garde d'une transition donnée est renforcée par une contrainte qui traduit le temps de séjour maximum possible dans l'état d'où est issue la transition sans violer l'échéance globale du thread. La rétropropagation se fait par une exploration en arrière du système de transition temporisé. Les pires temps d'exécution sont calculés pour chaque état de contrôle du modèle d'un thread. La restriction des gardes se fait en utilisant la librairie de manipulation d'expressions implémentée dans le module du modèle abstrait.

Module de traduction — Le module de traduction contient un ensemble de classes responsables de l'exportation des informations internes en mémoire produites par *JediTool* en d'autres formats afin de les visualiser, les éditer ou effectuer des analyses complémentaires. *JediTool* comprend deux catégories principales de traducteurs. La première catégorie concerne l'exportation du modèle de l'application au format d'entrée des outils d'analyse et vérification. La seconde catégorie est dédiée à la génération des informations liant le modèle de l'application au code source ; ceci peut s'avérer très utile pour identifier une trace d'exécution du modèle dans le code. L'API fournie par *JediTool* facilite l'implémentation de nouveaux traducteurs.

Un traducteur doit, soit implémenter une interface prévue à cet effet ; soit hériter d'une classe générique de traduction et implémenter uniquement les méthodes gabarit (*Template Methods* [GHJV93]) qui lui sont spécifiques, ce qui réduit l'effort d'implémentation.

Nous avons implémenté des traducteurs pour l'exportation des modèles générés dans les formats de IF-2 et Prometheus. Ce module génère aussi un ensemble de rapports qui lient les états de contrôle du modèle au code source et les séquences d'appels et de synchronisation sur les objets partagés pour chaque thread.

Module *plug-in* et interface utilisateur — Nous avons ajouté une catégorie dédiée à *JediTool* dans le gestionnaire de fenêtres d'Eclipse. Cette catégorie regroupe trois fenêtre principales : la fenêtre de génération de modèle à partir d'un fichier de configuration, la fenêtre qui contrôle la traduction du modèle et la fenêtre pou la détection et traçage des blocages. *JediTool* utilise les librairies JFace et SWT d'Eclipse pour implémenter les éléments de son interface utilisateur graphique.

L'explorateur de paquetage fournit par JDT a été étendu afin d'acceuillir les fichiers de configurations JXML. Pour ce faire, nous avons ajouté les décorateurs de ressource et les contributeurs de barre d'action à l'environnement. *JediTool* interagit aussi avec l'éditeur Java afin de le positionner, de révéler et sélectionner les instructions.

L'installation et mise à jour de *JediTool* se fait en utilisant la centrale de mise à jour d'Eclipse (*Update Center*). En utilisant cette technologie, il est possible d'installer ou mettre à jour *JediTool* facilement en quelques secondes en se connectant à Internet et sans quitter l'environnement. Pour ce faire, nous avons configuré un site d'installation sur Internet selon le protocole de mise à jour d'Eclipse. Il est cependant possible d'installer *JediTool* manuellement pour les machines ne disposant pas de connexion à Internet. Afin d'alléger l'exposé, l'explication détaillée de la technologie et de la marche à suivre pour l'installation de *JediTool* est disponible en [Nak05].

Chapitre 6

Applications

6.1 Synthèse d'ordonnanceur - Chaîne d'outils Expresso

Au cours du projet *Expresso*, nos partenaires industriels ont opté pour restreindre l'ordonnancement des systèmes temps-réel dans le cadre du projet à un ordonnancement préemptif basé sur des priorités fixes ; précisément *Rate Monotonic Scheduling* [LL73, SKG91] avec une gestion des ressources par PIP ou PCP [SRL90, RSLR95]. Cette approche d'ordonnancement a pour seule vocation de garantir le respect des échéances temps-réel des différents threads. Ce choix peut s'avérer restrictifs et insuffisant pour des applications où l'on souhaite garantir certaines qualités de service (QoS) ou opter pour une gestion de ressources plus flexible. Les QoS peuvent être d'un intérêt majeur pour les systèmes embarqués, notamment les contraintes sur l'usage de la mémoire et de l'énergie, souvent limitées dans ces systèmes. Nous avons proposé un mécanisme de synthèse d'ordonnanceur basé sur l'application, moins restrictif que les politiques d'ordonnancement classiques, qui garantie les échéances temps réels et permet d'exprimer les contraintes de QoS.

L'approche que nous avons adoptée est la suivante [KNY03] : en premier lieu, les modèles des différents threads de l'application, où apparaissent les temps de calculs et actions de synchronisation et communication, sont générés par *JediTool*. En suite, le module de synthèse analyse les modèles obtenus et produit les contraintes qui définissent l'ordonnanceur. De plus, *JediTool* a été étendu pour instrumenter le code source de l'application étudiée. Cette instrumentation permet de mettre à jour les structures de données de l'ordonnanceur nécessaires à l'évaluation des contraintes d'ordonnancement à l'exécution. Le code instrumenté de l'application et l'ordonnanceur obtenu sont compilés en natif, à l'aide du compilateur *TurboJ* [Ins] de *Silicomp RI*, et déployés sur un RTOS sur lequel sont installés l'exécutif Java de Expresso (*Java Run-Time System*)[RFTF+03]. La chaîne d'outils est illustrée dans le figure 6.2.

6.1.1 Extraction du modèle et instrumentation du source

Les modèles des threads sont extraits à partir du code source à l'aide de *JediTool* et exportés dans le format utilisé par le module de synthèse d'ordonnanceur grâce à un traducteur dédié (*cf.* paragraphe 5.4). Les actions retenues dans le modèle sont les synchronisations sur les objets, la communication et les temps d'exécution.

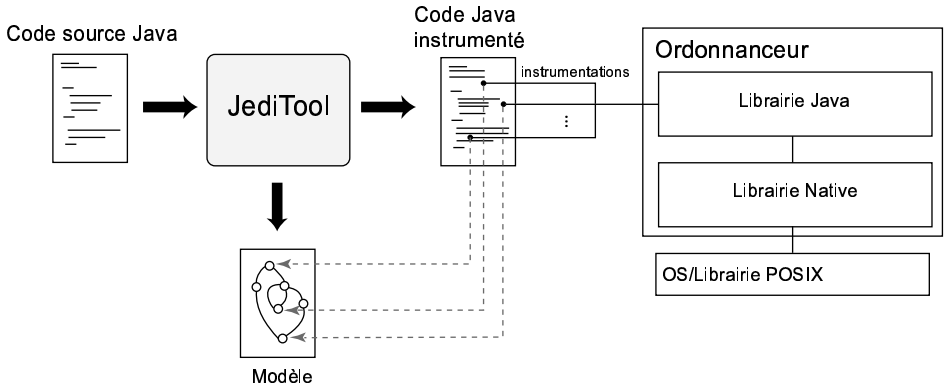


FIG. 6.1 – Extraction du modèle de l'application et instrumentation du code source Java par *JediTool*.

Lors de l'exécution de l'application, l'ordonnanceur détermine l'ensemble des threads éligibles pour l'exécution pour un état donné de l'application. Pour ce faire, l'ordonnanceur évalue un prédicat sur l'état du système et calcule l'ensemble des threads dont l'exécution ne risque pas d'entraîner un blocage ou une violation des contraintes de la QoS. Afin de pouvoir disposer des informations concernant l'état de l'application à l'exécution, le code source de l'application est instrumenté de manière à mettre à jour des structures de données de l'ordonnanceur. L'instrumentation est effectuée en appliquant des scriptes d'instrumentation (*patches*) générés par *JediTool* en même temps que les modèles. Les structures de données de l'ordonnanceur renseignent sur l'état de l'application par rapport au modèle généré par notre outil et utilisé par la synthèse d'ordonnanceur. Notamment, chaque position du programme qui correspond à un changement d'état de contrôle est instrumenté par un appel à une primitive de l'ordonnanceur afin d'indiquer l'état courant du thread. Les données de l'ordonnanceur sont mises à jour à l'aide d'une interface programmatique Java utilisée par le code d'instrumentation. Le processus d'extraction de modèle et d'instrumentation du code est illustré dans la figure 6.1.

6.1.2 Synthèse de l'ordonnanceur

Le module de synthèse d'ordonnanceur effectue une exploration exhaustive de l'espace d'état afin de détecter les états de blocage et de violation de la QoS. Il génère les contraintes qui décrivent, pour un état donné, les threads pouvant être exécutés sans mener à un état

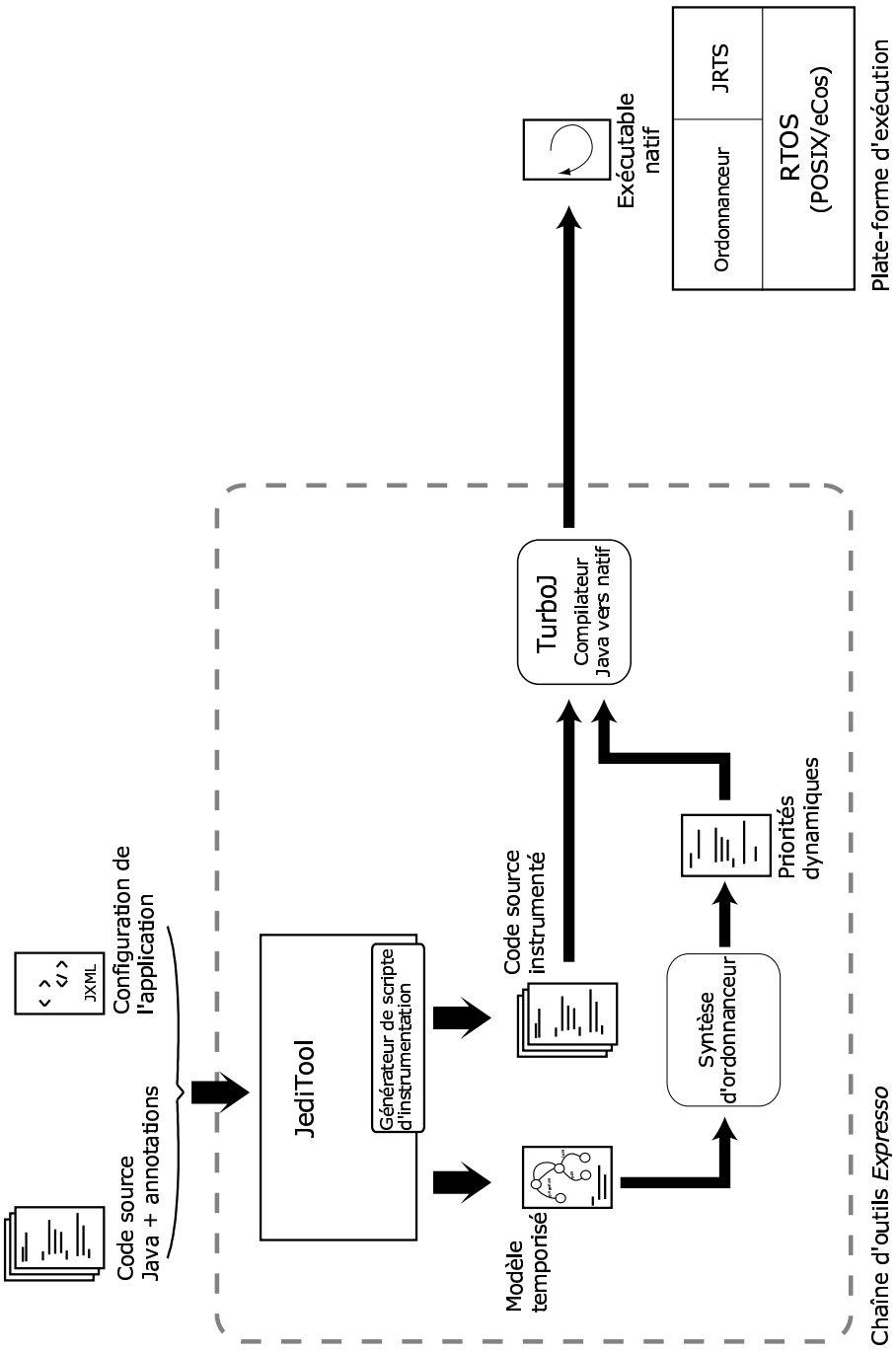


FIG. 6.2 – Génération d'ordonnanceur dirigé par l'application. De Java vers code natif compilé pour un système d'exploitation temps réel conforme à POSIX

du système qui viole les "bonnes propriétés" : absence de blocage, respect de contraintes temporelles et de la QoS.

Fonctionnement de l'ordonnanceur

Lors de l'exécution, l'ordonnanceur prend le contrôle de l'application à chacune des actions suivantes :

- Accès aux ressources : actions de verrouillage et déverrouillage des objets
- Attente de notification : appels à la méthode `wait()`, avec ou sans délai d'expiration
- Terminaison du cycle d'un thread : elle est signalée par l'invocation de la méthode `waitForNextPeriod()` dans le cadre du projet Expresso
- Arrivée d'un thread : ceci est implémenté par le déclenchement d'une alarme du système d'exploitation temps-réel
- Notification par `notify()` et `notifyAll()`. Le système n'est pas réordonnancé lors de la notification. L'ordonnancement a lieu lors le concours pour la reprise de verrou par les threads notifié.

Parmi les threads prêt à être exécutés (*ie.* ceux qui ne sont pas bloqués par l'attente d'une ressource, d'une notification ou qui dorment), l'ordonnanceur choisit un sous-ensemble $\mathcal{S}_{exec}$ dont l'exécution n'entraîne pas le blocage du système (dû aux verrouillage des objets partagés ou à un dépassement d'échéance). Cet ensemble est restreint par une dernière sélection Q_{exec} qui prends parmi les threads de $\mathcal{S}_{exec}$ ceux qui respectent les contraintes de QoS. Comme la spécification de Java laisse le choix du thread à notifier libre à l'implémentation pour un appel à `notify()`, l'ordonnanceur opère d'une manière analogue aux cas précédent : il choisit les ensemble S_{notif} et Q_{notif} parmi les threads à notifier afin de satisfaire les contraintes d'exécution et de QoS.

Les ensembles $\mathcal{S}_{exec}$, Q_{exec}, $\mathcal{S}_{notif}$ et Q_{notif} sont calculés à l'exécution en évaluant des contraintes exprimées les états et horloges du système ; ces contraintes sont générées lors de la synthèse de l'ordonnanceur.

Constructions des contraintes d'ordonnancement

Les contraintes générées sont de la forme $C_{th}(s, v)$, où s un état du système, v une valuation des horloges et th un thread. Si $C_{th}(s, v)$ est évaluée à $vrai$ lors de l'exécution, le thread th est exclu de l'ensemble des thread à exécuter ; d'une manière informelle : *si* $C_{th}(s, v)$ *alors* ne pas exécuter th. Cela revient à définir la fonction de priorité $\{C_{th}(s, v)\ ,\ \{t_{th} \prec_0 t_{th'} |\ t_{th} \in T_{th} \wedge t_{th'} \in T_{th} \wedge th' \in \Theta \wedge th' \neq th\}\}$. La synthèse des contraintes se fait en explorant les états atteignables du modèle du système. Ceci est obtenu en combinant les modèle des threads, extrait à partir du source par *JediTool*, combinés avec un modèle de l'ordonnanceur [KNY03]. Une fois les états de blocage identifiés, une exploration en arrière est effectuée à partir de ceux-ci jusqu'à atteindre un état où l'ordonnanceur intervient en faisant le choix d'un thread

à exécuter. L'ensemble de threads éligibles à l'exécution à cet état est donc restreint afin d'éviter de choisir les threads menant au blocage. Si l'ensemble des threads éligibles se trouve vide après restriction, l'état est marqué comme étant état de blocage et la procédure et réitérée.

6.1.3 Exemples d'application

Nous avons appliqué la chaîne d'outils mise en oeuvre dans le projet *Expresso* à une application de contrôle d'un bras de robot [KNY03]. Cette application, introduite en [LV98], consiste en un bras du robot programmé pour prendre des objets sur un tapis roulant, les entreposer sur une étagère pour les transférer ensuite dans un panier. Le système comporte les threads suivants :

- `SensorReader` effectue la lecture de plusieurs capteurs et actualise les tampons associés. Sa période est de 24ms et son temps d'exécution est de 1ms.
- `TrajectoryControl` lit les commandes dans un tampon et délivre en conséquence les positions au contrôleur bas-niveau du bras (le thread `Controller`) par le biais les actions *set-points*. Dans l'absence de commande il termine. L'exécution prends 5ms à 6ms.
- `Lifter` est un thread périodique de période 40ms. Il supervise le mouvement du bras pour l'acquisition des objets sur le tapis. Avant de terminer, il active le thread `Putter` et informe le thread `TrajectoryControl`.
- `Putter` commande le mouvement du bras pour le transfert des objets de l'étagère tampon à leurs destination finale dans le panier. son temps d'exécution est de 4ms à 8ms.
- `Controller` délivre les instructions au bras du robot. Il est périodique de période 16ms.

Le modèle de l'application ainsi que les scripts d'instrumentation ont été extraits avec succès par *JediTool* à partir du code Java annoté. Une représentation graphique est donnée figure 6.4.

Les fonctions de priorités synthétisées pour l'application du bras du robot sont larges et complexes, et donc humainement illisibles. Nous présentons les priorités générées sur une application plus simple, donné à la figure 6.3, afin d'en illustrer le principe.

Contraintes générées

Après analyse du modèle, les contraintes qui décrivent l'éligibilité des threads sont calculées selon le procédé expliqué plus haut. Afin de contenir l'explosion combinatoir, ces contraintes sont calculée par étapes : d'abords sont générées les contraintes pour les blocages fonctionnels, sans prendre compte le temps. Puis les contraintes pour les blocages temporels et finalement ceux relatifs aux QoS.

Les contraintes obtenues pour les blocages fonctionnels sont présentées dans la table 6.1. En effet, ces contraintes empêchent l'application d'atteindre les états de blocage $(W2, R3, \ .\ )$ où les threads `Writer` et `User` se trouvent bloqué par prise croisée de verrous, ceux de L et V. Ces

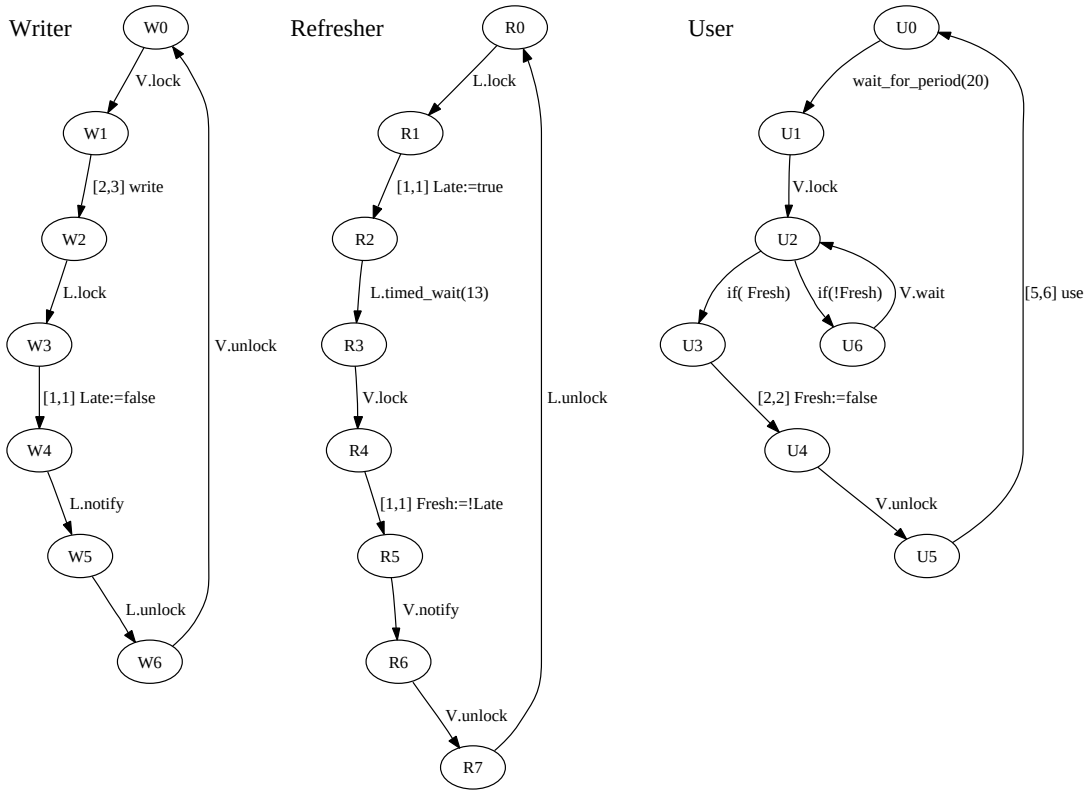


FIG. 6.3 – Modèle de l'application *Writer-Refresher-User*.

```
(* no-deadlocks *)
define(‘Refresher_DEADLOCKS’,‘ (false
 \/ ((W2=Writer) /\ (R2_Relock=Refresher) /\ (U1=User))
 \/ ((W2=Writer) /\ (R2_Relock=Refresher) /\ (U5=User))
 \/ ((W2=Writer) /\ (R2_Relock=Refresher) /\ (U6=User))
 \/ ((W2=Writer) /\ (R2_Relock=Refresher) /\ (U6_Relock=User))
)’)
define(‘Writer_DEADLOCKS’,‘ (false
 \/ ((W0=Writer) /\ (R3=Refresher) /\ (U1=User))
 \/ ((W0=Writer) /\ (R3=Refresher) /\ (U5=User))
 \/ ((W0=Writer) /\ (R3=Refresher) /\ (U6=User))
 \/ ((W0=Writer) /\ (R3=Refresher) /\ (U6_Relock=User))
)’)
```

TAB. 6.1 – Contraintes pour l'élimination des bloquages dûs aux verrouillages des ressources.

```
(* no-timelocks *)
define('Refresher_TIMELOCKS','(false
\/ ((W0=Writer) /\ (Writer_clock0=Writer_Clock) /\ (R0=Refresher)
      /\ (Refresher_clock0=Refresher_Clock) /\ (U1=User)
      /\ (User_clock0=User_Clock) /\ (11=(Global_Clock%20))
      /\ (11=User_Clock_response))
\/ ...
```

TAB. 6.2 – Contraintes pour assurer le respects des échéances.

contraintes empêchent ces threads d'être pris par l'ordonnanceur aux états précédents ceux du blocage, et rendre par conséquent les états de blocage inatteignables. Ainsi, le thread `Writer` est exclu aux l'état $(W0, R3, \ .\ )$ et le thread `User` est exclu aux états $(W2, R2_{Relock}, \ .\ )$.

La table 6.2 présente un extrait des contraintes calculées en prenant en compte le temps. Elle sont calculées à partir du modèle réduit par les contraintes de blocage présentées ci-dessus afin contenir l'explosion combinatoire.

Compilation et déploiement

Les applications étudiées ont été compilées avec succès et déployées sur le système d'exploitation temps réel *eCos* [Tho00]. Le temps système requis pour l'ordonnancement (évaluation des contraintes d'ordonnancement et sélection du thread à exécuter) ne dépasse pas les 0.66μs sur une machine dotée d'un Pentium II cadencé à 330Mhz. Ces délais expérimentaux sont relativement faibles et restent acceptables pour un bon nombre de domaines d'application.

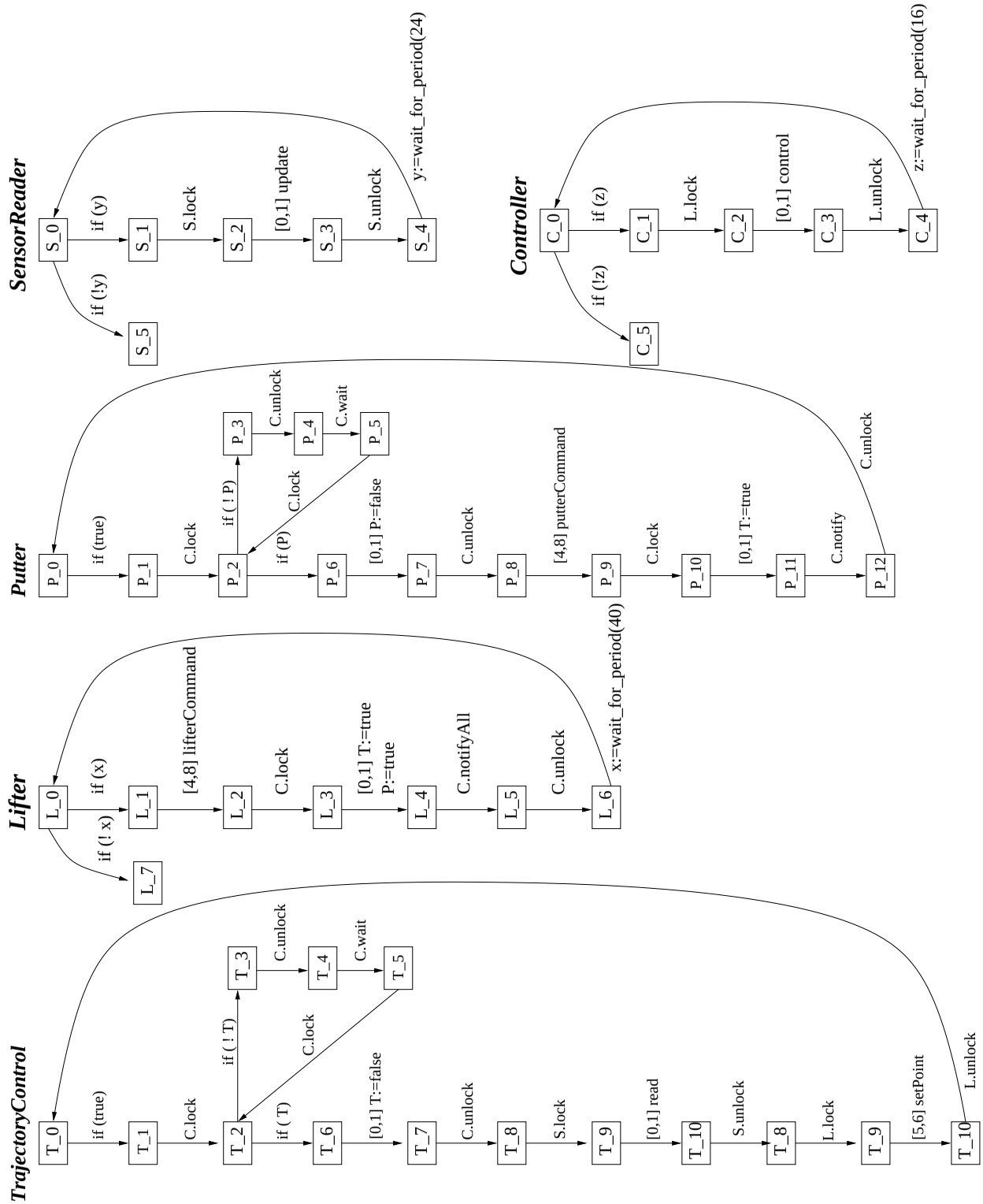


FIG. 6.4 – Modèle pour la synthèse d'ordonnanceur extrait du code Java par *JediTool*.

6.2 Robot K9 Rover de la NASA

Le rover K9 [BBF+01] est un prototype mettant en oeuvre les nouvelles technologies dont l'agence spatiale américaine *NASA* désire équiper ses robots pour les prochaines missions d'exploration de la planète Mars. Le K9 a donc pour vocation l'expérimentation et validation des choix technologiques et leurs intégrations dans une plate-forme commune qui doit respecter les exigences strictes propres aux missions spatiales.

6.2.1 Description générale

Le rover K9, illustré par la figure 6.5, dispose d'un châssis sur-élevé équipé de six roues avec un système de suspension spécial. Huit caméras, pour l'acquisition vidéo et la photographie, sont disposées à l'avant, arrière et sur le bras télescopique de l'engin. La transmission et réception des différents signaux se fait par le biais de deux antennes latérales. Un panneau solaire installé sur le haut de la plate-forme fourni l'énergie nécessaire à l'ensemble de l'électronique embarquée sur le robot. La partie électronique se compose d'une carte mère équipée d'un processeur Intel Pentium MMX cadencé à 166Mhz, 128Mo en mémoire vive, différents port E/S (SCSI, USB et RS232), une carte Ethernet et une entrée vidéo. Le tout communique via les bus PCI et ISA de la carte mère. Ce dispositif électronique est complété par un système logiciel qui se charge d'interpréter et exécuter les séquences d'opérations envoyées par la station terrestre. C'est sur ce système logiciel que porte notre intérêt, il est présenté dans le paragraphe suivant.

FIG. 6.5 – Le Robot K9.

6.2.2 Système logiciel

Le système logiciel du rover K9 est composé de deux unités essentielles. Un *planificateur* embarqué transforme les directives issues de la station terrestre en une séquence d'actions appelée *plan*. Ce plan est ensuite interprété par la deuxième unité appelée *exécutif*. Le schéma de fonctionnement du rover est illustré dans la figure 6.6.

- **Planificateur** — son rôle est de transformer les objectifs transmis par la station terrestre en un plan exécutable. Le planificateur dispose d'un modèle qui décrit les contraintes physiques du rover et les conditions nécessaire à son bon fonctionnement. Le planificateur définit l'ordre d'exécution des objectifs et construit le plan adéquat en respectant les ressources du systèmes et les conditions temporelles des objectifs. Le planificateur et le modèle associé ont été étudiés et testés en conditions réelles dans [DLM03].
- **Plan** — les plans sont formulés en CRL (*Contingent Rover Language Executive*). Un plan est composé de noeuds structurés de manière hiérarchique qui décrivent une sé-

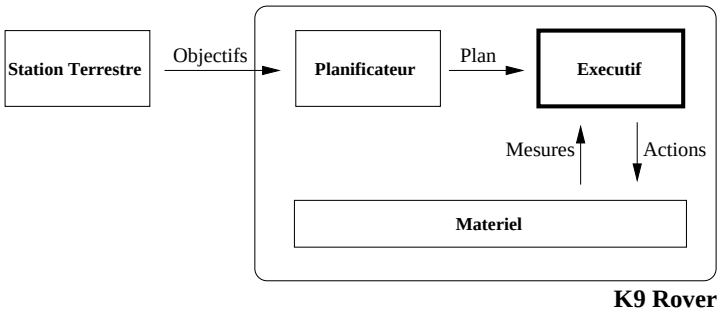


FIG. 6.6 – Traitement des ordres de la stations terrestre par le K9 Rover.

quence d'actions. Chaque noeud représente une tâche à exécuter, un bloc de noeuds ou une structure de branchement. Le plan est strictement séquentiel : il ne contient pas de boucles ni des instructions concurrentes. Chaque noeud peut s'accompagner de conditions temporelles, relatives ou absolues, ou sur l'état du système (niveau d'énergie, coordonnées *etc.*).

– **Exécutif** — l'exécutif du rover interprète le plan fourni par le planificateur et contrôle les mécanismes de mouvements et les différents capteurs. L'exécutif s'assure de vérifier les différentes contraintes à l'exécution comme la disponibilité des ressources et l'état du système. L'exécutif supervise l'exécution du plan et effectuent le choix entre les différentes options possibles en fonction des mesures effectuées. Le cas échéant, les situations d'échec sont reporté à la station terrestre. En [ABBO04] est présenté un modèle ainsi qu'une sémantique opérationnelle du CRL sont donnés en terme d'automates temporisés ; la traduction des plans a été automatisée mais le modèle de l'exécutif a été extrait à la main.

6.2.3 Architecture de l'exécutif

Nous disposons d'un prototype en Java de l'exécutif écrit par les ingénieurs de *NASA Ames*. Le code source a été traduit d'une première version en C++, il consiste en 8000 lignes de code disposé en 81 unités de compilation. L'implémentation ne repose sur aucune librairie ou extension temps-réel du langage Java, la concurrence est implémentée avec les threads Java de base.

L'architecture de l'exécutif est représentée dans la figure 6.7. Elle comporte six threads organisés comme suit :

– `Executive` est le thread central de l'application. Il interprète le plan et coordonne l'exécution avec les autres threads.
– `PlanWatcher` fournit le plan du planificateur à l'exécutif.
– `ActionExecution` est responsable d'exécuter les commandes au niveau matériel.
– `ExecConditionCheckerThread` vérifie les contraintes et conditions sur l'état du rover.

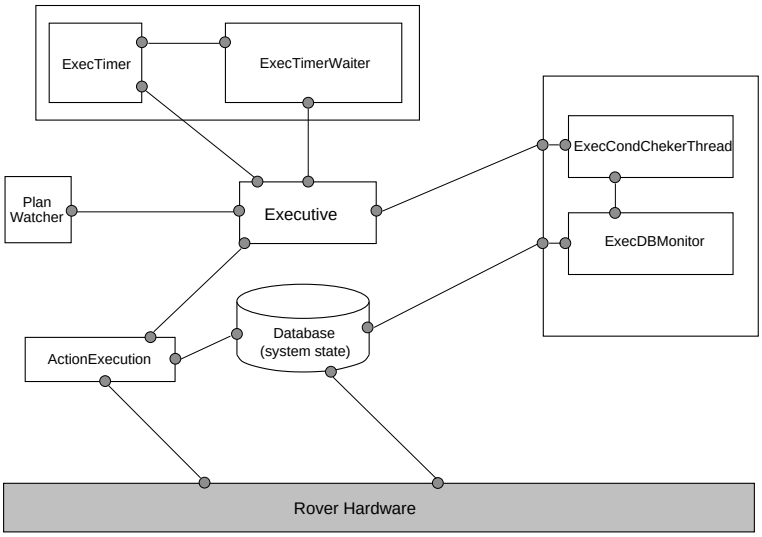


FIG. 6.7 – Architecture de l'exécutif.

- `ExecDBMonitor` surveille l'état du rover et met à jour les variables associée à ce dernier à chaque changement.
- `ExecTimer` et `ExecTimerWaiter` gèrent les contraintes temporelles indiquée sur les noeuds du plan.

L'exclusion mutuelle entre les threads est implémentée en utilisant les objets partagés, mais aussi les verrous des objets associés au threads : chaque thread dispose des références des objets représentant les autres threads et effectue les différentes sections critiques en se synchronisant sur ces références. Les objets sur lesquels les threads se synchronisent sont donc les instances (tous singletons) des classes suivantes : `Database`, `ExecConditionChecker`, `ActionExecution`, `ExecTimer`, `ExecTimerWaiter` et `Executive`.

6.2.4 Analyse de de l'exécutif du K9 Rover

L'implémentation de l'exécutif du K9 Rover qui nous a été livrée par la NASA ne repose sur aucune librairie temps-réel et n'inclut pas d'informations sur les temps d'exécutions ou les échéances des threads. Cependant, *JediTool* a été capable de générer le modèle de l'application et en analyser les blocages dus aux synchronisations sur les objets partagés. Le modèle de l'exécutif généré par *JediTool* comporte au total 483 états et 562 transitions, répartis par thread selon le tableau suivant :

Thread	Nb états	Nb transitions
`Executive`	330	366
`PlanWatcher`	4	3
`ExecTimer`	28	36
`ExecTimerWaiter`	26	33
`ExecConditionCheckerThread`	44	60
`ExecDBMonitor`	15	16
`ActionExecution`	36	48

Blocages potentiels détectés

A l'aide de *JediTool* nous avons généré et analysé la fonction de priorité pr_{mutex} qui modélise l'exclusion mutuelle sur les objets partagés de l'application. Le module d'analyse de l'outil nous a permis de relever deux schémas de blocage potentiels dans l'exécutif du K9 Rover.

Le premier schéma de blocage est causé par la synchronisation des threads `Executive` et `ExecTimerWaiter` sur les objets instances de `Executive` et `ExecTimer`. Le schéma correspond à la situation suivante :

- Le thread `Executive` détient le verrou de `Executive` et demande celui de `ExecTimer`,
- Le thread `ExecTimerWaiter` détient le verrou de `ExecTimer` et demande celui de `Executive`.

JediTool donne un recensement exhaustif des positions dans le code source ainsi que la chaîne d'appel associée qui donnent lieu à ces situations de blocage. Le tableau 6.4 dénombre les positions correspondant au schéma mentionné ci-dessus.

Le deuxième schéma de blocage détecté, est dû à une situation analogue :

- Le thread `ExecConditionCheckerThread` est synchronisé sur l'objet `ExecConditionChecker` et demande l'objet `Database`,
- Le thread `ExecDBMonitor`, inversement, tient le verrou de `Database` et attend d'acquérir le verrou de `ExecConditionChecker`.

La description des blocages est donnée dans la table 6.3.

6.3 Synthèse automatiques des régions d'allocation

Ce paragraphe décrit un travail annexe que nous avons effectué au cours du projet Expresso. Ce travail concerne la gestion de la mémoire au sein des programmes Java et la problématique posée par une gestion manuelle de celle-ci par le mécanisme d'allocation par régions. Nous avons réutilisé certains modules de *JediTool* pour élaborer un prototype d'outil capable de générer automatiquement et d'une manière sûre le code nécessaire à une gestion par région de la mémoire pour les programmes Java temps-réel.

Thread `ExecConditionCheckerThread` Objet verrouillé : `ExecConditionChecker` Objet demandé : `Database`

– Position : exec/Executive.java, Ligne 1215. Chaîne d'appel :

Méthode appelée	Site d'appel	
	Type	Ligne
exec.ExecConditionCheckerThread.run()	-	-
exec.ExecConditionChecker. internalConditionCheckerLoop()	exec. ExecConditionCheckerThread	14
exec.ExecConditionChecker. waitForConditions()	exec. ExecConditionChecker	366
exec.ExecConditionChecker. findMetConditions(...)	exec. ExecConditionChecker	172

Thread `ExecDBMonitor` Objet verrouillé : `Database` Objet demandé : `ExecConditionChecker`

– Position : exec/ExecConditionChecker.java, ligne 389. Chaîne d'appel :

Méthode appelée	Site d'appel	
	Type	Ligne
exec.ExecDBMonitor.run()	-	-
exec.ExecConditionChecker. internalDBMonitorLoop()	exec ExecDBMonitor	15

TAB. 6.3 – Positions de blocage entre les threads `ExecConditionCheckerThread` et `ExecDBMonitor` par prise de ressources croisée sur les objets `ExecConditionChecker` et `Database`.

L'usage du ramasse-miettes pour une gestion automatique de la mémoire compromet la prédictibilité des temps d'exécution d'une application. Plusieurs propositions de ramasse-miettes temps-réel ont été avancées [Hen98, HGB92], leurs temps de réponses et/ou consommation de mémoire restent cependant difficiles à borner. Afin de pallier à ce problème, la spécification RTSJ propose une gestion de mémoire *par régions*. Une région est associée programmatiquement à une unité logique de l'application (méthode, thread ou un bloc d'instructions) et est libérée à la fin de l'exécution de cette dernière. Cependant, cette programmation se heurte au problème des références pendantes (*dangling references*). Ce problème se produit lorsqu'un objet référence un autre objet appartenant à une région différente, et de durée de vie plus courte que la sienne. Cette situation est difficile à détecter par le test et par l'analyse statique du code. La gestion des régions reste donc une tâche difficile et source d'erreurs.

Nous proposons en [GNYZ04] une méthodologie pour instrumenté un programme Java où la mémoire est gérée par le ramasse-miettes par le code nécessaire à une gestion pro-

Thread `Executive`. Objet verrouillé : `Executive`. Objet demandé : `ExecTimer`.

– Position : exec/ExecTimer.java, ligne 311. Chaînes d'appel :

Méthode appelée	Site(s) d'appel	
	Type	Ligne(s)
exec.Executive.run()	-	-
exec.Executive.changePlan(exec.Node , int)	exec.Executive	66
exec.Executive.executePlan(boolean)	exec.Executive	1707
exec.Executive.executePlanbody()	exec.Executive	280/284
exec.Executive.executeCurrentNode()	exec.Executive	196
exec.Executive.waitForNodeStartWindow(...)	exec.Executive	1379
exec.ExecTimer.addTime(...)	exec.Executive	646

– Position : exec/ExecTimer.java, ligne 275. Chaînes d'appel :

Méthode appelée	Site(s) d'appel	
	Type	Ligne(s)
exec.Executive.run()	-	-
exec.Executive.changePlan(exec.Node , int)	exec.Executive	66
exec.Executive.executePlan(boolean)	exec.Executive	1707
exec.Executive.executePlanbody()	exec.Executive	280/284
exec.Executive.executeCurrentNode()	exec.Executive	196
exec.Executive.clearNodeTimeouts(exec.Node)	exec.Executive	1498/1384
exec.ExecTimer.removeTime(exec.Symbol)	exec.Executive	1259

– Position : exec/ExecTimer.java, ligne 286. Chaînes d'appel :

Méthode appelée	Site(s) d'appel	
	Type	Ligne(s)
exec.Executive.run()	-	-
exec.Executive.changePlan(exec.Node , int)	exec.Executive	66
exec.Executive.executePlan(boolean)	exec.Executive	1707
exec.Executive.executePlanbody()	exec.Executive	280/284
exec.ExecTimer.removeAllTimes()	exec.Executive	251

Thread `ExecTimerWaiter` Objet verrouillé : `ExecTimer`. Objet demandé : `Executive`.

– exec/ExecTimerWaiter.java, ligne 127. Chaîne d'appel :

Méthode appelée	Site d'appel	
	Type	Ligne
exec.ExecTimerWaiter.run()	-	-
exec.ExecTimerWaiter. runTimerWaitThreadInternal()	exec. ExecTimerWaiter	53
exec.ExecTimerWaiter. processWakeupTimes(...)	exec. ExecTimerWaiter	107

TAB. 6.4 – Positions de blocage entre les threads `Executive` et `ExecTimerWaiter` par prise de ressources croisée sur les objets `Executive` et `ExecTimer`.

grammatique, par régions. Nous avons implémenté notre approche en réutilisant le noyau de l'outil *JediTool*. Nous avons aussi développé une plate-forme de test afin d'évaluer différentes politiques de gestion d'allocation des objets au sein d'une même région.

6.3.1 Méthodologie

Nous appelons *site d'allocation* (respectivement *site d'appel*) une position dans le source contenant une instruction d'allocation `new` (respectivement une instruction d'appel de méthode). Un site d'allocation peut être responsable de la création d'un ou plusieurs objets. La récursivité et le lien tardif sont écartés du cadre de ce travail, le graphe d'appel est donc déroulé en un arbre d'appel où les noeuds correspondent aux sites d'appel.

D'une manière informelle, un objet instancié dans un site d'allocation d'une méthode est dit lui *échapper* si sa durée de vie dépasse l'exécution de la méthode. Un objet est dit *capturé* par une méthode s'il peut être collecté (au sens de [GJSB00]) à la fin de l'exécution de cette dernière. Les objets créés dans un site d'allocation donné, peuvent échapper et être capturé par différentes méthode appelantes. Ces dernières dépendent du chemin emprunté dans l'arbre d'appel. Le schéma 6.8 illustre ce phénomène : un objet produit par le site d'allocation 1 dans la méthode $m2$ est capturés dans la méthode $m1$ suivant la séquence d'appel $m0\,m1\,m2$, mais capturé par $m0$ suivant la séquence d'appel $m0\,m2$. Les objets produits par le site d'allocation 2 dans $m2$ sont toujours capturés par la méthode $m0$.

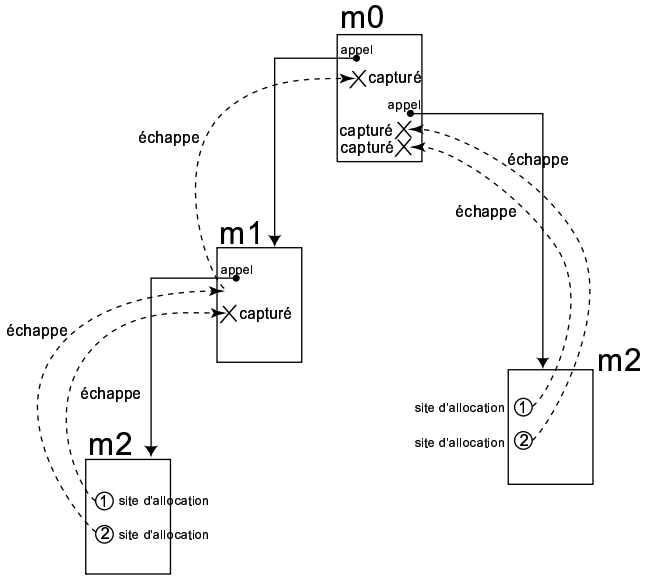


FIG. 6.8 – Exemple d'échappement/capture d'objet sur un arbre d'appel.

Nous proposons d'associer une région d'allocation par méthode, celle-ci sera alloué à l'entrée de la méthode et libérée à la fin de son exécution. Afin de pallier au problème des références

pendantes, un objet doit être alloué dans la région qui correspond à sa durée de vie : dans notre cas, c'est la région de la méthode qui le capture. En se basant sur l'analyse d'échappement des objets [Bla99, SR01, CGS+99], chaque site d'allocation lui ai associé la région de la méthode qui capture ses objets. Cette association est, comme nous l'avons illustré dans l'exemple de la figure 6.8, est paramétrée par la chaîne d'appel. Une première solution serait d'attribuer, statiquement, à tout les objets générés par un site d'allocation la région de plus longue durée de vie vers qui ils échappent. Une approche similaire a été présentée en [DC02] où l'analyse d'échappement a été effectuée par profilage du programme. Cette approche ne tient pas compte de la chaîne d'appel, dans l'exemple de la figure 6.8, les objets relatifs au site d'allocation 1 seraient donc alloués dans la région $r0$ associée à la méthode $m0$. Ceci engendre une occupation inutile de la mémoire (par exemple dans le cas de la chaîne d'appel $m0\,m1\,m2$) et a tendance à produire des régions volumineuses et de longue durée de vie, ce qui conduit à une sur-consommation de la mémoire.

Notre approche permet d'exploiter l'information liée à la chaîne d'appel afin d'allouer, dynamiquement, un objet donné dans la région de la méthode vers laquelle il échappe pour la séquence d'appel en cours. Pour ce faire, chaque site d'appel est instrumenté par l'information de capture des objets avant d'effectuer l'appel en question. Chaque allocation est instrumentée de manière à indiquer le site d'allocation impliqué.

6.3.2 Implémentation

L'analyse de pointeurs est effectuée en utilisant le compilateur *Flex* [AG, SR01]. A l'issue de cette phase nous disposons de l'information concernant la capture et échappement des sites d'allocations pour chaque chemin emprunté dans l'arbre d'appel (séquence de sites d'appel). Ensuite, le module de synthèse de régions, basé sur le noyau de *JediTool*, analyse le code source en détectant les sites d'allocations et les séquences d'appels associés. En se basant sur l'information de capture/échappement, ces sites sont instrumentés par des primitives de la librairie d'enregistrement et gestion de régions que nous avons implémentée. L'instrumentation se fait de la manière suivante :

- Une instruction de création et attribution d'une nouvelle région sera insérée au début de chaque méthode :
  `ScopedMemory.enter(new Region(`$method_{Id}$`)) ;` où $method_{Id}$ est l'identificateur de la méthode.
- L'appel d'une méthode m_{callee} correspondant à un site d'appel dans la méthode m_{caller} est précédé par l'enregistrement de la liste des sites d'allocations des objets capturés dans la méthode appelante m_{caller}. Cette information est enregistrée dans une table et servira à guider l'allocation des objets dans les méthodes appelées (noeuds du sous arbre de racine m_{caller}). Avant l'instruction d'appel à m_{callee} sera donc inséré l'enregistrement en question par l'instruction

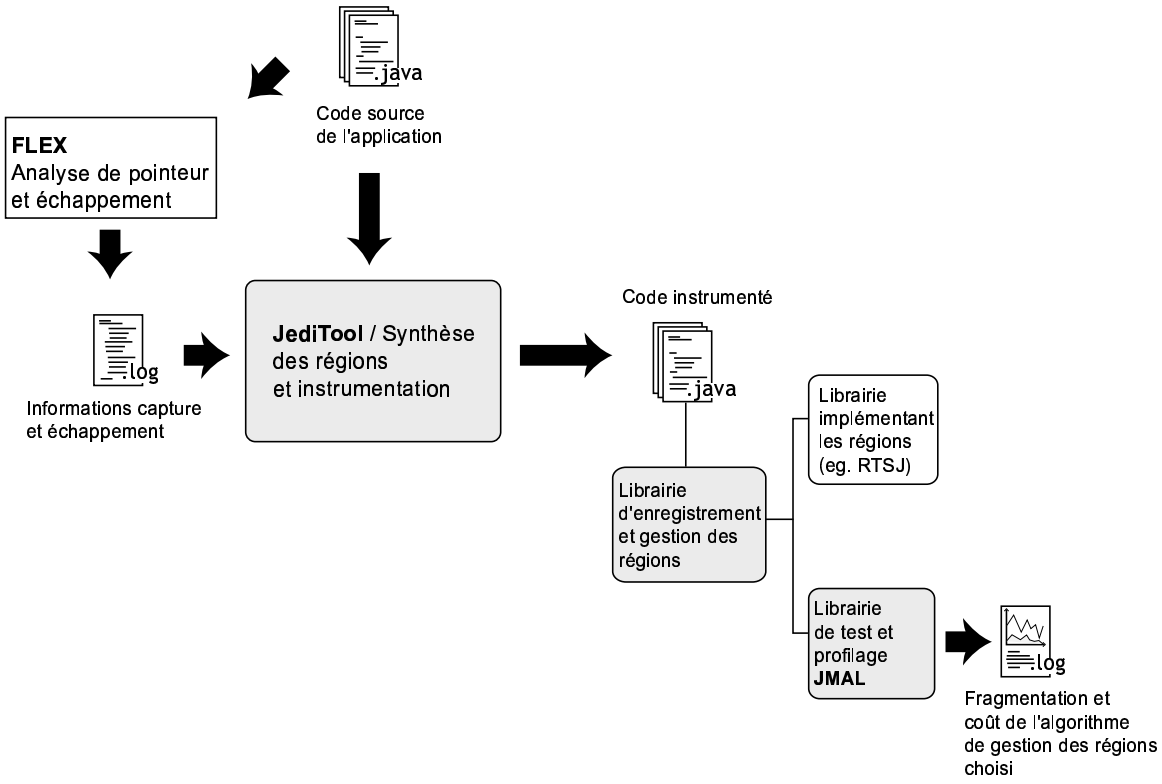


FIG. 6.9 – Implémentation de la synthèse des régions.

`ScopedMemory.determineAllocationSite(`$allocationSiteList$`);`

- L'expression `new` $Type()$ est remplacée par l'instruction `ScopedMemory.newInstance(`$Type$`.class,`$allocationSite_{Id}$`)`. Ainsi, le site d'allocation est identifié par $allocationSite_{Id}$ est la librairie de gestion d'allocation peut choisir dynamiquement dans quelle région alloué l'objet de type $Type$.
- A la fin de la méthode, la région qui lui est associée est libérée par l'instruction `ScopedMemory.exit();`.

A l'exécution, la librairie de gestion des régions décide de la région dans laquelle chaque objet sera effectivement alloué. Les allocations effectives sont déléguée à une libraire qui implémente un mécanisme d'allocation par région typiquement la RTSJ (cependant, le code d'instrumentation reste indépendant de celle-ci). Finalement, nous avons implémenté une librairie pour le profilage et simulation de la gestion interne des régions, appelée *JMAL* [NZ04]. Cette librairie permet d'expérimenter différentes stratégies de gestion des régions en simulant les algorithmes d'allocation, tout en allouant en réalité les objets sur le tas. JMAL permet de générer des courbes et rapports, pour une exécution donnée du programme, qui décrivent la fragmentation interne des régions et les temps d'allocation des objets.

Chapitre 7

Conclusion

7.1 Bilan

Nous avons présenté une méthodologie et un outil pour l'analyse fondée sur les modèles des systèmes temps-réel dans le cadre de la technologie Java.

Nous avons défini un sous-ensemble du langage Java adéquat à la construction statique de modèles formels. Nous avons établi un processus d'extraction de modèles fonctionnels à partir d'un logiciel applicatif conforme à ce sous-ensemble. Nous avons mis au point une méthodologie afin d'enrichir les modèles fonctionnels en rajoutant les contraintes temporelles et en modélisant la communication entre les différents composants concurrents d'un système temps-réel. Les retombées du travail que nous avons mené sont multiples :

- Au niveau du projet *RNTL-Expresso*, notre contribution a été double. Nous avons activement participé à l'élaboration du profil temps-réel adopté au sein du projet. De plus, nous avons monté avec succès une chaîne d'outils pour l'extraction automatique de modèle que nous avons employée pour la synthèse et implémentation d'ordonnanceur [KNY03]. Nous avons montré sur un ensemble d'exemples que les approches fondées sur les modèles sont une alternative intéressante et prometteuse aux politiques d'ordonnancement classiques.
- Malgré les limitations imposées sur certains aspects dynamiques du langage Java, l'outil *JediTool* permet de prendre en charge les aspects temporels —tels que les temps d'exécu-tion et échéances— qui caractérisent l'implémentation des systèmes temps-réel. *JediTool* implémente une démarche originale et unique pour la technologie émergeante Java temps-réel. A notre connaissance, aucun autre outil actuel dédié au langage Java ne prend en charge la modélisation et l'analyse des propriétés non fonctionnelles des systèmes.
- Notre contribution à travers ce travail a servi de base pour d'autres projets qui adoptent une approche fondée sur les modèles au laboratoire Verimag. Combaz et al. proposent en [CFLS] un modèle et un algorithme de synthèse de contrôleur pour les applications

multimédia afin de respecter à la fois les contraintes temps-réel dur et les contraintes temps-réel mou de QoS. En [AGY05, ABY05], Assayad et al. étendent le langage C — principalement avec des constructions pour exprimer la concurrence et l'interdépendance— dans lequel il modélisent un encodeur vidéo MPEG-4. Ce modèle a été raffiné afin de fournir une implémentation parallèle sur une machine multiprocesseur.

Nous avons défini un sous-ensemble du langage Java qui correspond à un compromis entre le confort de programmation et les besoins d'une construction statique des modèles. Nous avons fait le choix d'éliminer certaines fonctionnalités dynamiques du langage dont l'implantation est difficile à modéliser et/ou fortement dépendante de la plate-forme de déploiement. Par exemple, le lien tardif fait intervenir à l'exécution les structures de données et algorithmes pour la gestion des tables de répartition (*dispatching tables*) spécifiques à la machine virtuelle choisie. Pour ce sous-ensemble de Java nous avons défini des modèles sémantiques des instructions qui sont peu dépendants de l'implantation.

L'expressivité du sous-ensemble du langage établi reste acceptable et présente peu d'écarts par rapport aux normes actuelles pour le développement des systèmes temps-réel pour des langages de haut niveau, telles que Ravenscar Ada [DB98, BDV04]. A titre d'exemple, l'application témoin développée au cours du projet Expresso par *Thales Systèmes Aéroportés*, mettant en oeuvre un contrôleur de centrale inertielle d'un avion militaire, s'y conforme.

Nous avons proposé une méthodologie pour l'extraction de modèles fonctionnels à partir des instructions du logiciel applicatif. L'extraction se base sur les modèles sémantiques définis sur la syntaxe du langage et est guidée par les critères d'observabilité codés par des prédicats sur les instructions du programme. Ces derniers paramètrent la construction des modèles et déterminent la granularité des modèles construits. Les modèles ainsi obtenus sont augmentés par des contraintes temporelles qui découlent des temps de calcul, des échéances et des stimuli de l'environnement. Les contraintes temporelles sont explicitées sur tous les états du modèle de telle sorte qu'un blocage dû à la non satisfaction d'une garde temporisée implique une incohérence dans la spécification du système.

L'interaction des threads est modélisée par la synchronisation d'actions et des fonctions de priorité. Les fonctions de priorités sont construites automatiquement et d'une manière modulaire pour modéliser la communication par attente/notification et l'exclusion mutuelle sur les objets partagés. La composition des différentes règles de priorités ainsi construites forme la couche de synchronisation du modèle. Les priorités offrent une approche compositionnelle qui nous a permis non seulement de modéliser séparément chaque aspect de la coordination entre les threads, mais aussi de modéliser les politiques d'ordonnancement.

Nous avons réalisé l'outil *JediTool* où sont implantées les différentes phases de construction de modèles temporisés des systèmes temps-réel écrits en Java. *JediTool* est intégré dans l'environnement de développement universel à code ouvert *Eclipse* et repose sur le compilateur Java qui y est embarqué. Le code Java à traiter est compilé à la demande et les modèles fonctionnels

sont construits à la volée et avec un système de cache afin d'optimiser la mémoire et le temps de calcul. Ceci nous a permis de traiter avec succès des programmes de taille relativement grande (plus de 8000 lignes). *JediTool* bénéficie d'une mise à jour et installation facile et de l'ergonomie de l'interface utilisateur d'Eclipse [Nak05].

JediTool fournit un module pour l'analyse des fonctions de priorités qui modélisent la communication et l'exclusion mutuelle. Les priorités sont analysées progressivement : statiquement (par thread), dynamiquement (par état) et ensuite par vérification en exportant les modèles vers des outils de vérification. Certains faux contre-exemples sont éliminés au niveau de l'analyse dynamique des priorités en observant les conditions d'exclusion mutuelle aux états de blocage potentiels.

Nous avons intégré *JediTool* au sein de la boîte à outils IF-2 [BFG+99, BGM02, OGO04] développée à Verimag. Nous avons étendu cette dernière pour prendre en compte les fonctions de priorité au niveau de la spécification IF et lors de l'exploration de l'espace d'état des modèles. Grâce à notre extension, il est possible de composer les fonctions de priorité générées par *JediTool* avec les fonctions de priorité pour spécifier et vérifier des politiques d'ordonnancement sur le système. Ceci est possible sans apporter de changements sur le modèle initialement généré.

Bien que d'autres outils tels que Bandera [CDH+00], ESC/Java 2 [FLL+02] et JPF [VHBP00] sont consacrés à l'analyse des programmes Java, une différence fondamentale distingue notre outil. L'originalité de *JediTool* réside dans la prise en compte de la *dimension temps* dans les modèles. Ceci est indispensable, notamment afin de prendre en compte les caractéristiques de l'implantation des systèmes et des contraintes qui lient l'interaction des systèmes avec leurs environnements.

7.2 Perspectives

Le langage Java reste d'actualité et offre une base intéressante pour le développement des systèmes temps-réel dans le futur. Le projet *Mackinac* [Bol04] chez Sun Microsystems et les machines virtuelles *Jamaïca* [Sie02] chez Aicas et *PERC* [Aon05] chez Aonix, montrent l'intérêt que portent les industriels pour la plate-forme Java temps-réel.

L'analyse des propriétés non fonctionnelles pour la technologie Java reste cependant totalement absente des outils d'analyse actuels dédiés au langage. L'approche présentée dans cette thèse et l'outil *JediTool* tentent de répondre à ce besoin. Le travail que nous avons effectué peut se poursuivre dans plusieurs directions intéressantes.

Modéliser certaines données des programmes permet d'obtenir des modèles plus fins et moins conservatifs. Choisir les données à modéliser est une tâche difficile, il faut inclure suffisamment de données pour affiner l'analyse tout en évitant l'explosion combinatoire. Les modèles générés par JediTool peuvent servir de base pour un raffinement dirigé par les contre-

exemples (*lazy abstraction*) [SS99, BR02, HJMS02]. Les modèles sont alors étendus par les données qui influencent les tests des structures de contrôle menant vers les états qui violent les contraintes fonctionnelles et de qualité de service. Cela permet d'éliminer progressivement les faux contre-exemples en construisant des modèles de plus en plus fins.

Une alternative possible est d'utiliser l'ensemble des données impliquées dans les contre-exemples pour l'analyse à l'exécution et le test. Une analyse des modèles permet de déduire les domaines des valeurs pour les données impliquées dans les contre-exemples et ainsi guider la procédure de test pour couvrir les erreurs potentielles détectées par l'analyse. Ceci peut orienter les approches basées sur le test [BH03, HR04] afin de les rendre plus efficaces.

L'outil *JediTool* peut être amélioré sur plusieurs plans, notamment pour augmenter les performances en termes de la consommation de la mémoire et des ressources de calcul. D'autres parts, les annotations des temps d'exécution utilisées au niveau du code source sont spécifiques à la plate-forme d'implantation et difficiles à transcrire. On pourra interfacer *JediTool* avec des outils d'analyse statiques [BBW00b, BBMP00] et de profilage [dVCF$^+$02] pour déterminer automatiquement les pires temps d'exécution.

A un plus long terme, une perspective intéressante serait de synthétiser, à partir des diagnostics issus de l'analyse, du code pour la détection et la récupération d'erreur à l'exécution du système. Le code généré, associé au programme du système instrumenté par des points de contrôle (*check points*) [RS93, CD02], permet la détection et récupération des situations de blocage à l'exécution. Cette approche vient compléter le test et le raffinement des modèles pour traiter les contre-exemples qui ne peuvent être vérifiés à cause de l'explosion combinatoire lors de la vérification des modèles et l'incomplétude des cas de test.

Bibliographie

[ABBO04] Anahita Akhavan, Saddek Bensalem, Marius Bozga, and Eleni Orfanidou. Experiment on verification of a planetary rover controller. In *Proceedings of the 4th Intl. Workshop on Planning and Scheduling for Space, IWPSS'04, Darmstadt, Germany*, June 2004.

[ABD+95] N.C. Audsley, A. Burns, R.I. Davis, K. Tindell, and A.J. Wellings. Real-time system scheduling. In B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictably Dependable Computing Systems*, ESPRIT Basic Research Series, pages 41–52. Springer, 1995.

[ABY05] I. Assayad, V. Bertin, and S. Yovine. Parallel model analysis and implementation for MPEG-4 encoder. In *Embedded Processors for Multimedia and Communications II. 17th Annual Symposium IS&T/SPIE's Electronic Imaging*, San Jose, CA, USA, January 2005. SPIE Press.

[AFM+04] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times : a tool for schedulability analysis and code generation of real-time systems. In Peter Niebert and Kim G. Larsen, editors, *Proc. of FORMATS'03*, number 2791 in Lecture Notes in Computer Science, pages 60–72. Springer–Verlag, 2004.

[AG] MIT. Program Analysis and Compilation Group. The Flex compiler infraestructure. http ://www.flex-compiler.csail.mit.edu/.

[AGS00] K. Altisen, G. Gößler, and J. Sifakis. A methodology for the construction of scheduled systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems 2000 (FTRTFT'00)*, 2000.

[AGS02] K. Altisen, G. Goessler, and J. Sifakis. Scheduler Modeling Based on the Controller Synthesis Paradigm. *Journal of Real-Time Systems*, 23(1-2) :55–84, 2002. special issue on control-theoritical approaches to real-time computing.

[AGY05] I. Assayad, Ph. Gerner, and S. Yovine. Modelling, Analysis and Implementation of an On-line Video Encoder. In *The First International Conference on Distri-*

buted Frameworks for Multimedia Applications (DFMA'2005), Besancon, France, February 2005. IEEE Computer Society.

[Aon05] Aonix. PERC Real-Time Java Virtual Machine. http ://www.aonix.fr/perc.html, 2005.

[Aud91] N. C. Audsley. Resource control for hard real-time systems : A review. Technical Report YCS 159, Real-Time Systems Research Group, Departement of Computer Science, University of York, York, UK, 1991.

[BB91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9) :1270–1282, September 1991.

[BB03] John Barnes and J. G. Barnes. *High Integrity Software : The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[BBF+01] John L. Bresina, Maria Bualat, Michael Fair, Richard Washingtony, and Anne Wright. The K9 on-board rover architecture. In *ESA Workshop on "On-board autonomy"*, pages 17–19, October 2001.

[BBMP00] Iain Bate, Guillem Bernat, Greg Murphy, and Peter Puschner. Low-level analysis of a portable java byte code wcet analysis framework. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications*, pages 39–48, Dec. 2000.

[BBW00a] G Bernat, A Burns, and A Wellings. Portable worst-case execution time analysis using Java Byte Code. In *Euromicro RTS 2000. 12th Euromicro Conference on Real-Time Systems*, pages 81–88, Stockholm , Sweden, 2000.

[BBW00b] Guillem Bernat, Alan Burns, and Andy Wellings. Portable worst case execution time analysis using java byte code. In *Euromicro Conference on Real Time Systems (ECRTS00)*, pages 19–21, Stockholm, Sweden, June 2000.

[BCE+03] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[BCP+01] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys = Esterel + Kronos. A tool for verifying real-time properties of embedded systems. In *Conference on Decision and Control*. IEEE Control Systems Society, 2001.

[BDC02] Janet Barnes, Brian Dobbing, and Rod Chapman. On the principled design of object oriented programming languages for high integrity systems, 2002. Praxis Critical Systems Ltd.

[BDV04] Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the use of the ada ravenscar profile in high integrity systems. *Ada Lett.*, XXIV(2) :1–74, 2004.

[BFG+99] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. IF : An intermediate representation and validation environment for timed asynchronous systems. In *World Congress on Formal Methods (1)*, pages 307–327, 1999.

[BFG00] Marius Bozga, Jean-Claude Fernandez, and Lucian Ghirvu. Using Static Analysis to Improve Automatic Test Generation. In S. Graf and M. Schwartzbach, editors, *Proceedings of TACAS'00 (Berlin, Germany)*, LNCS, pages 235–250. Springer-Verlag, March 2000.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language : Design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.

[BG00] Greg Bollella and James Gosling. The real-time specification for Java. *Computer*, 33(6) :47–54, 2000.

[BGM02] M. Bozga, S. Graf, and L. Mounier. If-2.0 : A validation environment for component-based real-time systems. In K.G. Larsen Ed Brinksma, editor, *Proceedings of CAV'02*, volume 2404 of *LNCS*, pages 343–348, Copenhagen, Denmark, July 2002. Springer.

[BGS00] S. Bornot, G. Gößler, and J. Sifakis. On the construction of live timed systems. In S. Graf and M. Schwartzbach, editors, *Proc. TACAS 2000*, volume 1785 of *LNCS*, pages 109–126. Springer-Verlag, 2000.

[BH03] Sadek BenSalem and Klaus Havelund. Deadlock analysis of multi-threaded java programs. Submitted for publication, December 2003.

[Bla99] B. Blanchet. Escape analysis for object-oriented languages : application to Java. *ACM SIGPLAN Notices*, 34(10) :20–34, 1999.

[BLL+95] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, pages 232–243, 1995.

[Boe02] Barry W. Boehm. *Software Engineering Economics*. Springer-Verlag New York, Inc., 2002.

[Bol04] Greg Bollella. Controlling Physical Systems with Real-time Specification for Java – Combining Real-time and Non-real-time Operations on a Single Processor. http ://sunflash.sun.com/articles/75/4/feature/13057, May 2004.

[Bor98] Sébastien Bornot. *De la composition des systèmes temporisés*. PhD thesis, Université Joseph Fourier, 1998. En français.

[BPPS00] V. Bertin, M. Poize, J. Pulou, and J. Sifakis. Towards validated real-time software. In 12 th Euromicro Conference on Real-Time Systems, Stockholm, Sweden, June 2000, 2000.

[BR01] William S. Beebee, Jr. and Martin Rinard. An implementation of scoped memory for real-time Java. *Lecture Notes in Computer Science*, 2211 :289– ? ?, 2001.

[BR02] T. Ball and S. K. Rajamani. The SLAM project : Debugging system software via static analysis. In *ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 1–3, 2002.

[BS00] S. Bornot and J. Sifakis. An algebraic framework for urgency. *Information and Computation*, 163, 2000.

[BST98] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. *Lecture Notes in Computer Science*, 1536 :103–129, 1998.

[But97] Giorgio Buttanzo. *Hard Real-Time Computing Systems : Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[CCGR00] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV : A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4) :410–425, 2000.

[CD02] Sung-Eun Choi and Steven J. Deitz. Compiler support for automatic Checkpointing. In *Sixteenth annual international symposium on high performance computing systems and applications*, Moncton, NB, Canada, June 2002.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera : extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[CDHR00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 205–223, London, UK, 2000. Springer-Verlag.

[Cer99] Certification Authorities Software Team (CAST). Object-oriented technology (OOT) in civil aviation projects : Certification concerns. Federal Aviation Administration, 1999. available at http ://av-info.faa.gov/software/CAST/cast-8.rtf.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, 1986.

[CFLS] J. Combaz, J-C. Fernandez, Th. Lepley, and J. Sifakis. Fine Grain QoS Control for Multimedia Application Software. accepted for publication DATE05.

[CGS+99] J-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for Java. In *OOPSLA*, pages 1–19, 1999.

[Cha00] Roderick Chapman. Industrial experience with SPARK. *ACM SIGADA Ada Letters*, 20(4) :64–68, 2000.

[Cha02] Rod Chapman. MISRA C at SIL4 ? perspectives and alternatives. Praxis Critical Systems Limited, September 2002.

[CK04] David R. Cok and Joseph Kiniry. Esc/java2 : Uniting esc/java and jml. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.

[CW03] H. Cai and A. J. Wellings. Towards a high-integrity real-time java virtual machine. In *On the Move to Meaningfull Internet Systems 2003 : Workshop on Java Technologies for Real-Time and Embedded Systems*, volume LNCS 2889, pages 319–334. Springer, 2003.

[DAR04] DARPA, Purdue University, SUNY Oswego, University of Maryland and DLTech. The Ovm Project. http ://www.ovmj.org/, 2004.

[DB98] Brian Dobbing and Alan Burns. The Ravenscar tasking profile for high integrity real-time programs. In *Proceedings of the annual ACM SIGAda international conference on Ada*, pages 1–6, Washington D.C., USA, 1998. ACM Press.

[DC02] M. Deters and R. K. Cytron. Automated discovery of scoped memory regions for real-time Java. In *Proc. of the 3rd Int. symposium on Memory management*, pages 25–35. ACM Press, 2002.

[DE97] Sophia Drossopoulou and Susan Eisenbach. Java is Type Safe — Probably. In *11th European Conference on Object Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, pages 389–418. Springer-Verlag, June 1997.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101. Springer-Verlag, 1995.

[Dib02] Peter Dibble. *Real-Time Java Platform Programming.* Pearson Education, Prentice Hall PTR, March 2002.

[DIS99] Claudio Demartini, Radu Iosif, and Riccardo Sisto. dspin : A dynamic extension of spin. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 261–276. Springer-Verlag, 1999.

[DLM03] M Bernardine Dias, Solange Lemai, and Nicola Muscettola. A real-time rover executive based on model-based reactive planning. In *The 7th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, May 2003.

[dVCF+02] Vincent Colin de Verdière, Sébastien Cros, Christian Fabre, Romain Guider, and Sergio Yovine. Speedup prediction for selective compilation of embedded java programs. In *Proceedings of the Second International Conference on Embedded Software*, pages 227–239. Springer-Verlag, 2002.

[Eur92] European Organisation for Civil Aviation Electronics. *DO-178B : Software Considerations in Airborne Systems and Equipment Certification.* RTCA Inc., December 1992.

[FJJV97] Jean-Claude Fernandez, Claude Jard, Thierry Jeron, and Cesar Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1-2) :123–146, 1997.

[FLL+02] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 2002.

[GHJV93] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns : Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707 :406–431, 1993.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, Boston, Mass., 2000.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11) :1203–1233, 2000.

[GNYZ04] Diego Garbervetsky, Chaker Nakhli, Sergio Yovine, and Hichem Zorgati. Program instrumentation and run-time analysis of scoped memory in Java. In *Runtime Verification, RV'04*, Electronic Notes in Theoretical Computer Science, Elsevier Science, pages 95–110, Barcelona, Spain, 2004.

[GOO03] Susanne Graf, Ileana Ober, and Iulian Ober. D1.1.3 - The OMEGA Time Extensions. Deliverable of the IST-2001-33522 OMEGA project, Verimag, 2003.

[Gößl01a] G. Gößler. *Compositional Modelling of Real-Time Systems — Theory and Practice.* PhD thesis, Université Joseph Fourier, Grenoble, France, 2001.

[Gößl01b] G. Gößler. PROMETHEUS — a compositional modeling tool for real-time systems. In P. Pettersson and S. Yovine, editors, *Proc. Workshop RT-TOOLS 2001.* Technical report 2001-014, Uppsala University, Department of Information Technology, 2001.

[Gre02] J. Grebe. C/C++ Coding Standard Recommendations for IEC 61508, July 2002. White paper, www.exida.com.

[Gre04] Green Hills Software, Inc. Green Hills Optimizing C/C++/EC++ Compilers. http ://www.ghs.com/download/datasheets/c++_ec++compiler.pdf, 2004. Last accessed May 2005.

[GRF03] Ludovic Gauthier and Marc Richard-Foy. EXPRESSO Realtime Java for Safety and Mission Critical Embedded Systems Java API. http ://www.irisa.fr/rntl-expresso/, May 2003. Aonix.

[GS02] G. Goessler and J. Sifakis. Composition for component-based modeling. In *Proceedings of FMCO'02*, volume 2852 of *LNCS*, pages 443–466, November 2002.

[GS03] Gregor Goessler and Joseph Sifakis. Component-based construction of deadlock-free systems. In *Proceedings of FSTTCS'03*, volume 2914 of *LNCS*, pages 420–433, December 2003.

[GV03] B. Gebremichael and F.W. Vaandrager. Control synthesis for a smart card personalization system using symbolic model checking. In K.G. Larsen and P. Niebert, editors, *Proceedings First International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 2791 of *LNCS*, pages 6–7, Marseille, France, September 2003. Springer Verlag.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993.

[Hal98] N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Tenth International Conference on Computer-Aided Verification, CAV'98*, Vancouver (B.C.), June 1998. LNCS 1427, Springer Verlag.

[Har97] Johnson M. Hart. *Win32 Systems Programming.* Addison-Wesley Longman Publishing Co., Inc., 1997.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language Lustre (extended abstract). In *1st European Control Conference*, pages 1661–1665, Grenoble, July 1991.

[Hen98] R. Henriksson. *Scheduling garbage collection in embedded systems.* PhD thesis, Lund Institute of Technology, July 1998.

[HGB92] Jr. H. G. Baker. The treadmill : Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3) :66–70, March 1992.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5) :279–295, 1997.

[HR95] Matthew Hennessy and Tim Regan. A process algebra for timed systems. *Information and Computation*, 117(2) :221–239, Mars 1995.

[HR04] Klaus Havelund and Grigore Rosu. An Overview of the Runtime Verification Tool Java PathExplorer. *Formal Methods in System Design*, 24(2) :189–215, 2004.

[Ins] Silicomp Research Institute. Turbo J. Java to native compiler. http ://www.ri.silicomp.fr/adv-dvt/java/turbo/index.htm.

[Ins95] Institute of Electrical and Electronic Engineers, Inc. Information Technology — Portable Operating Systems Interface (POSIX) — Part : System Application Program Interface (API) — Amendment 2 : Threads Extension [C Language]. IEEE Standard 1003.1c–1995, IEEE, New York City, New York, USA, 1995. also ISO/IEC 9945-1 :1990b.

[Int05] International Organization for Standardization. Information technology – Programming languages – Guide for the use of the Ada Ravenscar Profile in high integrity systems. ISO/IEC TR 15942 :2000, February 2005.

[J-C00] J-Consortium's Real-Time Java Working Group. Real-Time Core Extensions. The Open Group http ://www.opengroup.org/, September 2000. Revision 1.0.14.

[JG01] Guoping Jia and Susanne Graf. Verification experiments on the mascara protocol. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 123–142. Springer-Verlag New York, Inc., 2001.

[JHA+96] J. -C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP : a protocol validation and verification toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 437–440, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[KLP+98] Hee-Hwan Kwak, Insup Lee, Anna Philippou, Jin-Young Choi, and Oleg Sokolsky. Symbolic schedulability analysis of real-time systems. In *RTSS*, pages 409–, 1998.

[KNY03] Christos Kloukinas, Chaker Nakhli, and Sergio Yovine. A methodology and tool support for generating scheduled native code for real-time Java application. In Rajeev Alur and Insup Lee, editors, *Third International Conference on Embedded Software (EMSOFT 2003)*, LNCS-2855, pages 274–289, Philadelphia, Pennsylvania, USA, October 2003.

[KRP+93] Mark H. Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael Harbour. *A practitioner's handbook for real-time analysis.* Kluwer Academic Publishers, 1993.

[KWK02] Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-java : a high integrity profile for real-time java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 131–140. ACM Press, 2002.

[LGLL91] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real time applications with Signal. *Proceedings of the IEEE*, 79(9) :1321–1336, September 1991.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1) :46–61, 1973.

[LV98] M. Lusini and E. Vicario. Static analysis and dynamic steering of time-dependent systems using time petri nets. Technical Report 28.98, Dip. Sistemi e Informatica, University of Florence, 1998.

[McM93] K. L. McMillan. *Symbolic model checking : An approach to the state explosion problem.* Kluwer Academic Publishers, 1993.

[MIS98] Motor Industry Software Reliability Association MISRA. *Guidelines For the Use Of The C Language In Vehicle Based Software.* MIRA, 1998.

[MTdR96] A. K. Mok, Duu-Chung Tsou, and R. C. M. de Rooij. The msp.rtl real-time scheduler synthesis tool. In *RTSS '96 : Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, page 118. IEEE Computer Society, 1996.

[Nak01] Chaker Nakhli. Spécification compositionnelle à l'aide de priorités dynamiques. Master's thesis, Département Génie Electrique – Ecole Nationale d'Ingénieurs de Tunis, Juin 2001.

[Nak05] Chaker Nakhli. JediTool website. http ://www-verimag.imag.fr/PEOPLE/nakhli/jeditool/, 2005.

[NY01] P. Niebert and S. Yovine. Computing efficient operation schemes for chemical plants in multi-batch mode. *European Journal of Control*, 2001. Hermes.

[NZ04] Chaker Nakhli and Hichem Zorgati. Java Memory ALlocation scope-oriented simulator. http ://jmal.sourceforge.net/, May 2004.

[Obj05a] Object Management Group. Profile for QoS and Fault Tolerance. Technical Report V1.0, Object Management Group, 2005.

[Obj05b] Object Management Group. UML Profile for Schedulability, Performance and Time. Technical Report V1.1, Object Management Group, 2005.

[OGO04] Iulian Ober, Susanne Graf, and Ileana Ober. Validating timed UML models by simulation and verification. *Accepted for publication in STTT, Int. Journal on Software Tools for Technology Transfer*, 2004.

[OGO05] Iulian Ober, Susanne Graf, and Ileana Ober. Validating timed UML models by simulation and verification. *Accepted for publication in STTT, Int. Journal on Software Tools for Technology Transfer, 2004*, 2005.

[Ope03] Open SystemC Initiative. SystemC 2.0.1 Language Reference Manual — Revision 1.0. Technical report, Open SystemC Initiative, 2003.

[OTI03] Inc. Object Technology International. Eclipse platform technical overview. http ://eclipse.org/whitepapers/eclipse-overview.pdf, February 2003.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th Symp. Foundations of Computer Science*, pages 46–57, 1977.

[PS91] Thomas Plum and Dan Saks. *C++ Programming Guidelines*. Plum Hall, Inc, 1991.

[RFTF+03] Richard-Foy, Talpin, Fabre, Londres, and Yovine. Rapport d'activité – 01 juin 2001 au 31 mai 2003. Technical report, Projet RNTL Expresso, 2003. In French.

[RJB98] Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[Rob05] Robust Software Engineering Group, NASA Ames Research Center. Java Path Finder. http ://javapathfinder.sourceforge.net/, 2005. Last accessed May 2005.

[RS93] Parameswaran Ramanathan and Kang G. Shin. Use of Common Time Base for Checkpointing and Rollback Recovery in a Distributed System. *IEEE Trans. Softw. Eng.*, 19(6) :571–583, 1993.

[RSLR95] R. Rajkumar, L. Sha, J.P. Lehoczky, and K. Ramamritham. *An Optimal Priority Inheritance Policy For Synchronization in Real-Time Systems*, chapter 11, pages 246–268. Prentice Hall, 1995.

[Sie02] Fridtjof Siebert. Bringing the full power of java technology to embedded real-time applications. In *MSy'02 Embedded Systems in Mechatronics*, Winterthur, Switzerland, October 2002.

[SKG91] Sha, Klein, and Goodenough. Rate monotonic analysis for real-time systems. In Kluwer Academic Publishers, editor, *Foundations of Real-Time Computing : Scheduling and Resource Management*, pages 129–155, Boston, MA, 1991.

[SR01] A. Salcianu and M. Rinard. Pointer and Escape Analysis for Multithreaded Programs. *ACM SIGPLAN Notices*, 36(7) :12–23, 2001.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9) :1175–1185, September 1990.

[SRS98] John A. Stankovic, Krithi Ramamritham, and Marco Spuri. *Deadline Scheduling for Real-Time Systems : Edf and Related Algorithms.* Kluwer Academic Publishers, 1998.

[SS99] Hassen Saïdi and Natarajan Shankar. Abstract and Model Check While You Prove. In *CAV '99 : Proceedings of the 11th International Conference on Computer Aided Verification*, pages 443–454. Springer-Verlag, 1999.

[Tec01] Pi Technology. A comparison of MISRA C testing tools, October 2001. Presented at the MISRA C Forum.

[Tel93] Telecommunication Standardization Sector of ITU. Specification and Description Language (SDL), ITU-T Recommendation Z. 100. Technical report, International Telecommunication Union., March 1993. This document can be ordered via http ://www.itu.ch.

[The99] The Requirements Working Group for Real-time Extensions for the Java(tm) Platform. Requirements for Real-time Extensions for the Java Platform. Technical

report, National Institute of Standards and Technology, September 1999. NIST Special Publication 500-243.

[The02] The Embedded C++ Technical Commitee. Embedded C++ specification. http ://www.caravan.net/ec2plus/, 2002. Last accessed May 2005.

[Tho00] Gary Thomas. eCos : An operating system for embedded systems. *Dr. Dobb's Journal of Software Tools*, 25(1) :66, 68–72, 74, Jan 2000.

[TY01] S. Tripakis and S. Yovine. Timing analysis and code generation of vehicle control software using taxys. ENTCS, 55(2) :174-183, 2001. Elsevier., 2001.

[VH96] Jan Vitek and R. Nigel Horspool. Compact dispatch tables for dynamically typed object oriented languages. In *Proceedings of the 6th International Conference on Compiler Construction*, pages 309–325. Springer-Verlag, 1996.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Java pathfinder - second generation of a java model checker, 2000.

[WBC+00] D. Weil, V. Bertin, E. Closse, M. Poize, P. Vernier, and J. Pulou. Efficient Compilatin of ESTEREL for Real-Time Embedded Systems. In *CASES'2000*, San Jose, November 2000.

[Whe04] David A. Wheeler. SLOCCount Version 2.26 software metrics tool. http ://www.dwheeler.com/sloccount/sloccount.html, August 2004.

[Yov97] S. Yovine. KRONOS : A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1-2), December 1997.

Annexe A

Sources des Exemples

Grammaire DTD du Format JXML

```
<!ELEMENT Application
  (SharedObject*,
  (Thread|RealTimeThread)*)
>
<!ATTLIST Application
  id ID #REQUIRED
>

<!ELEMENT Thread (logic)>
<!ATTLIST Thread
  id ID #REQUIRED
>

<!ELEMENT SharedObject EMPTY>
<!ATTLIST SharedObject
  type CDATA #REQUIRED
>

<!ELEMENT RealTimeThread (logic)>
<!ATTLIST RealTimeThread
  id       ID #REQUIRED
  lbound  CDATA #REQUIRED
  ubound CDATA #REQUIRED
  deadline CDATA #REQUIRED
>

<!ELEMENT logic EMPTY>
<!ATTLIST logic
  method  CDATA #REQUIRED
```

```
  signature CDATA #REQUIRED
>
```

Fichier de configuration JXML du K9 Rover

```
<?xml version="1.0" encoding="UTF-8"?>

<Application id="K9ROVER">
<!--
************************
Executive Thread
************************
-->
<Thread id="executive">
  <logic
    signature="()V"
    method="exec.Executive.run"
  />
</Thread>
<!--
************************
PlanWatcher
************************
-->
<Thread id="planWatcher">
  <logic
    signature="()V"
    method="exec.PlanWatcher.run"
  />
</Thread>
<!--
************************
ExecTimer
************************
-->
<Thread id="timer">
  <logic
    signature="()V"
    method="exec.ExecTimer.run"
  />
</Thread>
<!--
************************
ExecTimerWaiter
************************
-->
<Thread id="timerWaiter">
  <logic
    signature="()V"
    method="exec.ExecTimerWaiter.run"
  />
</Thread>
<!--
************************
ExecConditionCheckerThread
************************
-->
<Thread id="condChecker">
  <logic
    signature="()V"
    method="exec.ExecConditionCheckerThread.run"
  />
</Thread>
<!--
************************
ExecDBMonitorThread
************************
-->
<Thread id="dbMonitor">
  <logic
    signature="()V"
    method="exec.ExecDBMonitor.run"
  />
</Thread>
<!--
************************
ActionExecution Thread
************************
-->
<Thread id="actionExec">
```

```
  <logic
    signature="()V"
    method="exec.ActionExecution.run"
  />

</Thread>

<!--
*****************************
Shared Objects
*****************************
-->

<SharedObject id="execLock"
        type="exec.Executive" />

<SharedObject id="condCheckerLock"
        type="exec.ExecConditionChecker" />

<SharedObject id="database"
            type="exec.Database" />

</Application>
```

Code source de l'application PutterGetter

```
package net.sf.jeditool.examples;

public class BaseBuffer {
  private int maxSize;
  private int[] internalArray;
  private int size;

  public BaseBuffer(int max) {
    maxSize = max;
    internalArray = new int[maxSize];
    size = 0;
  }

  /**
   * @exec 3 5
   */
  public void put(int value) {
    if (size < maxSize) {
      size++;
    }
    internalArray[size - 1] = value;
  }
  /**
   * @exec 2 4
   */
  public int get() {
    if (size > 0) {
      size--;
    }
    return internalArray[size];
  }
  /**
   * @exec 1 2
   */
  public boolean isFull() {
    return size == maxSize - 1;
  }
  /**
   * @exec 1 2
   */
  public boolean isEmpty() {
    return size == 0;
  }
  /**
   * @exec 1 1
   */
  public int getSize() {
    return size;
  }
  /**
   * @exec 1 1
   */
  public int getMaxSize() {
    return maxSize;
  }

}
```

```
package net.sf.jeditool.examples;

public class DataBuffer {
  private static DataBuffer instance;
  public static void initInstance(){
    instance=new DataBuffer();
  }
  public static DataBuffer instance(){
    return instance;
  }
  private BaseBuffer buffer;
  private DataBuffer(){
    buffer=new BaseBuffer(1024);
  }
  public int get() {
    return buffer.get();
  }
  public int getMaxSize() {
    return buffer.getMaxSize();
  }
  public int getSize() {
    return buffer.getSize();
  }
  public boolean isEmpty() {
    return buffer.isEmpty();
  }
  public boolean isFull() {
    return buffer.isFull();
  }
  public void put(int value) {
    buffer.put(value);
  }
}
```

```
package net.sf.jeditool.examples;

public class IOBuffer {
  private static IOBuffer instance;
  public static void initInstance() {
    instance = new IOBuffer();
  }
  public static IOBuffer instance() {
    return instance;
  }
  private BaseBuffer buffer;
  private IOBuffer() {
    buffer = new BaseBuffer(128);
  }
  public int get() {
    return buffer.get();
  }
  public int getPacketSize() {
    return buffer.getMaxSize();
  }
  public int getSize() {
    return buffer.getSize();
  }
  public boolean isEmpty() {
    return buffer.isEmpty();
  }
  public boolean isFull() {
    return buffer.isFull();
  }
  public void put(int value) {
    buffer.put(value);
  }
  public void send() {
    _send();
  }
  public void receive() {
    _receive();
  }

  /**
   * @exec 30 45
   */
  private native void _send();

  /**
   * @exec 15 20
   */
  private native void _receive();

}
```

```
package net.sf.jeditool.examples;
import javax.realtime.MemoryArea;
import javax.realtime.NoHeapRealtimeThread;
import javax.realtime.ReleaseParameters;
import javax.realtime.SchedulingParameters;
```

```
abstract public class PeriodicNoHeapRealTimeThread
  extends NoHeapRealtimeThread {

  public PeriodicNoHeapRealTimeThread(
    SchedulingParameters sp,
    ReleaseParameters rp,
    MemoryArea ma) {
    super(sp, rp, ma);
  }

  public void run() {
    while (true) {
      handlePeriod();
      waitForNextPeriod();
    }
  }
  /**
   * Thread's logic method
   */
  abstract public void handlePeriod();
}
```

```
package net.sf.jeditool.examples;
import javax.realtime.MemoryArea;
import javax.realtime.ReleaseParameters;
import javax.realtime.SchedulingParameters;
public class Getter extends
  PeriodicNoHeapRealTimeThread {
  private DataBuffer dataBuffer;
  private IOBuffer ioBuffer;
  public Getter(
    SchedulingParameters sp,
    ReleaseParameters rp,
    MemoryArea ma) {
    super(sp, rp, ma);
  }
  public void handlePeriod() {
    synchronized (dataBuffer) {
      int datasize = dataBuffer.getSize();
      int iosize = ioBuffer.getPacketSize();
      if (datasize - iosize < 0) {
        try {
          dataBuffer.wait(1000);
        } catch (InterruptedException e) {}
      }
      if (datasize - iosize >= 0) {
        synchronized (ioBuffer) {
          int i = 0;
          while (i < iosize) {
            int value = dataBuffer.get();
            ioBuffer.put(value);
            i++;
          }
          ioBuffer.send();
        }
        dataBuffer.notifyAll();
      }
    }
  }
  public void init(
    DataBuffer databuffer,
    IOBuffer iobuffer) {
    dataBuffer = databuffer;
    ioBuffer = iobuffer;
  }
}
```

```
package net.sf.jeditool.examples;
import javax.realtime.MemoryArea;
import javax.realtime.ReleaseParameters;
import javax.realtime.SchedulingParameters;
public class Putter extends
  PeriodicNoHeapRealTimeThread {
  private DataBuffer dataBuffer;
  private IOBuffer ioBuffer;
  public Putter(
    SchedulingParameters sp,
    ReleaseParameters rp,
    MemoryArea ma) {
    super(sp, rp, ma);
  }
  public void handlePeriod() {
    synchronized (dataBuffer) {
      synchronized (ioBuffer) {
        int datasize = dataBuffer.getSize();
        int iosize = ioBuffer.getPacketSize();
        int dataMaxSize = dataBuffer.getMaxSize();
        if (datasize + iosize >= dataMaxSize) {
          try {
            dataBuffer.wait(1000);
          } catch (InterruptedException e) {}
```

```
      }
      if (datasize + iosize < dataMaxSize) {
        ioBuffer.receive();
        int i = 0;
        while (i < iosize) {
          int value = ioBuffer.get();
          dataBuffer.put(value);
          i++;
        }
      }
      dataBuffer.notifyAll();
    }
  }
 }
 public void init(
  DataBuffer databuffer,
  IOBuffer iobuffer) {
  dataBuffer = databuffer;
  ioBuffer = iobuffer;
 }
}
```

```
package net.sf.jeditool.examples;

import javax.realtime.ImmortalMemory;
import javax.realtime.PeriodicParameters;
import javax.realtime.PriorityParameters;
import javax.realtime.RealtimeThread;
import javax.realtime.RelativeTime;
import javax.realtime.VTMemory;
public class InitialisationPhase {
 public static void main(String[] args) {
  final Starter starterThread = new Starter();
  starterThread.start();
 }

}
class Starter extends RealtimeThread {
 public void run() {
  ImmortalMemory.instance().enter(new Runnable() {
   public void run() {
    /*
     * Initialize shared objects
     */
    DataBuffer.initInstance();
    IOBuffer.initInstance();
    /*
     * Memory areas
     *   minimum size 10kB
     *   maximum size 10kB
     */
    VTMemory getterMemoryArea =
     new VTMemory(10 * 1024, 10 * 1024);
    VTMemory putterMemoryArea =
     new VTMemory(10 * 1024, 10 * 1024);
    /*
     * Periodic scheduling parameters for thread Putter
     *   Period                5300ms
     *   Start time            0 ms
     *   Deadline              4000ms
     *   Cost                  null
     *   Overrun handler       null
     *   Deadline miss handler null
     */
    final PeriodicParameters putterPeriodicParameters =
     new PeriodicParameters(
      new RelativeTime(0L, 0),
      new RelativeTime(5300L, 0),
      null,
      new RelativeTime(4000L, 0),
      null,
      null);
    /*
     * Periodic scheduling parameters for thread Getter
     *   Period              4100ms
     *   Start time          0 ms
     *   Deadline            3000ms
     *   Cost                null
     *   Overrun handler     null
     *   Deadline miss handler null
     */
    final PeriodicParameters getterPeriodicParameters =
     new PeriodicParameters(
      new RelativeTime(0L, 0),
      new RelativeTime(4100L, 0),
      null,
      new RelativeTime(3000L, 0),
      null,
      null);

    /*
     * Create threads
     */
```

```
final Putter putterThread =
  new Putter(
    new PriorityParameters(10),
    putterPeriodicParameters,
    putterMemoryArea);
final Getter getterThread =
  new Getter(
    new PriorityParameters(10),
    getterPeriodicParameters,
    getterMemoryArea);
/*
 * Assign global objects
 */
putterThread.init(
  DataBuffer.instance(),
  IOBuffer.instance());
getterThread.init(
  DataBuffer.instance(),
  IOBuffer.instance());
/*
 * Start threads
 */
putterThread.start();
getterThread.start();
    }
  };
 }
}
```

Résumé

Construire des modèles qui représentent fidèlement les systèmes temps-réels est indispensable pour l'analyse de leurs propriétés fonctionnelles et temporelles. Dans ce travail, nous présentons une méthodologie de modélisation pour les systèmes temps-réels dans le contexte de la technologie Java.

Le modèle abstrait de l'implantation du système temps-réel est obtenu par la restriction du logiciel applicatif avec les informations temporelles appropriées. Nous définissons un sous-ensemble du langage Java correspondant à un compromis entre d'une part l'expressivité et le confort de programmation, et d'autre part la construction statique de modèles formels. En premier lieu, un modèle fonctionnel est extrait à partir du logiciel. Ce modèle est ensuite augmenté par les informations temporelles qui caractérisent l'implémentation du système telles que celles relatives à la plate-forme de déploiement, la dynamique de l'environnement et les échéances temps-réel. Nous adoptons une approche compositionnelle basée sur les automates temporisés – qui modélisent les unités concurrentes du système – avec des fonctions de priorité. Ces dernières jouent un rôle important, elles interviennent dans la modélisation des couches de synchronisation et d'ordonnancement.

Notre méthodologie de modélisation a été implémentée dans l'outil *JediTool* pour l'extraction du modèle temporisé d'un système temps-réel à partir du logiciel applicatif annoté par les informations temporelles. *JediTool* est intégré dans l'environnement de développement à code ouvert Eclipse et est connecté à la boîte à outils de vérification IF-2.

Abstract

Building accurate models of complex real-time systems is essential for their functional and timing analysis. In this work, we present a modeling methodology for real-time systems written in Java.

An abstract model of the real-time system implementation is obtained by restricting application software with appropriate timing information. We establish a subset of the Java language corresponding to a trade-off between expressiveness and static formal model extraction. First, a functional model is extracted from application source code. Then, this model is extended with timing information relative to the implementation, such as the constraints implied by the execution platform, the environment dynamics and the real-time deadlines. We adopt a compositional approach based on timed automata – which model system's concurrent units – with priority functions. The latter play a central role as they are used to model synchronizing and scheduling layers.

Our modeling methodology was implemented in the JediTool used for timed model extraction from application source code annotated with timing information. JediTool is designed as a plug-in in the integrated development environment Eclipse and is connected to the IF-2 validation toolbox.