VERBATIM

Projet exploratoire RNRT 2005-2006

Sous projet SP4
Analyses statiques pour la
validation de codes
Livrable Lot2, Version 1.0

Abstraction de programmes guidée par les propriétés

Auteurs: B.d'Ausbourg et G.Durrieu
ONERA/DTIM - Consortium Verbatim
mailto:ausbourg@cert.fr,durrieu@cert.fr
http://www.onera.fr

Réseau National de Recherche en Télécommunications - Agence Nationale de la Recherche

Table des matières

Chapitre 1

Introduction

L'objectif du sous-projet 4, dans le cadre du projet Verbatim, est de permettre la validation de programmes d'interaction multi-modaux. Une application de navigation géographique, l'application *Pages Jaunes*, développée au laboratoire CLIPS/IMAG, constitue l'étude de cas retenue dans le contexte de ce projet.

Un programme tel que celui présenté par cette application comporte différents volets remplissant différentes fonctions. Il comporte également un certain nombre d'imports, provenant de librairies Java, et pour lesquels on ne dispose pas du source Java.

Si l'on cherche, dès lors, à construire la formalisation d'un tel programme, il est nécessaire au préalable de répondre à deux objectifs :

- disposer d'un code complet qui déclare le code de toutes les classes et de toutes les méthodes invoquées dans le programme ;
- ne retenir, dans la formalisation produite, que les éléments d'intérêt concernant l'interaction.

Ces deux objectifs plaident donc pour un travail de transformation opéré sur le code de l'application fourni afin d'en produire une forme de modèle d'exécution. Ce modèle d'exécution est également fourni sous la forme d'un programme Java. Ce dernier peut être vu comme une abstraction de l'application originale et décrit le comportement interactif encodé dans cette application. On décrira, dans une phase ultérieure du projet, comment un tel modèle d'exécution peut être obtenu à travers une analyse fouillée du code Java de l'application.

On fait cependant ici l'hypothèse qu'un tel programme Java, constituant le modèle d'exécution de l'application, a pu être produit. Le problème est donc de savoir et de pouvoir construire une formalisation de ce programme. C'est l'expression formelle de ce modèle d'exécution qui sera vérifiée au regard de spécifications exprimant les propriétés que l'on souhaite voir satisfaites par l'application ou qui sera utilisée pour générer des scénarios de test de cette application.

L'idée retenue dans ce projet est de construire une formalisation en se laissant guider par la propriété que l'on cherche à vérifier ou à tester sur l'application. C'est le programme décrivant le modèle d'exécution Java qui est formalisé. C'est ce modèle qui sera vérifié. Le problème est celui de la formalisation de codes Java.

Ce rapport a pour objet de présenter un ensemble de techniques auxquelles on a recours dans le projet pour d'une part formaliser un programme Java et d'autre part obtenir un modèle formel suffisamment réduit quoique conservant l'ensemble des informations relatives

à la propriété que l'on cherche à vérifier ou tester. Ces techniques ont pour une part été implantées dans la suite d'outils de Bandera que nous présentons ici.

Dans le cadre du projet, c'est une extension et une adaptation de ces outils qui sont envisagées pour, en particulier :

- intégrer des outils d'abstraction et de construction d'un modèle d'exécution Java à partir de codes Java d'une application ;
- traiter l'ensemble des constructions Java introduites dans les modèles d'exécution produits ;
- opérer des opérations de slicing suffisamment efficaces pour réduire la taille des formalisations du modèle d'exécution ;
- permettre la génération de scénarios de tests.

Ce rapport présente donc un ensemble d'outils et de techniques utilisés dans le cadre de ce projet et dévolus à la construction d'une formalisation (en Promela dans le cadre de cette étude) d'un code Java. Dans le cadre de cette étude, ce code Java sera le modèle d'exécution de l'application interactive multimodale. Il présente également un ensemble d'outils et de techniques permettant de formaliser les propriétés ou plus généralement les spécifications auxquelles le modèle formel produit sera confronté. Enfin il décrit les techniques qui seront implantées dans Bandera pour réaliser des opérations de slicing efficaces sur les programmes Java en fonction des propriétés attendues, en permettant ainsi de réduire la formalisation produite.

Chapitre 2

Présentation générale de Bandera

La boîte à outils Bandera est constituée d'un ensemble intégré de composants d'analyse, de transformation et de visualisation de programmes conçus pour permettre l'expérimentation de la vérification de propriétés de modèles sur du code source Java. Bandera prend en entrée un code source Java ainsi qu'une spécification écrite dans le langage de spécification temporelle de Bandera, et produit un modèle du programme et de la spécification dans le langage d'entrée d'un parmi plusieurs outils de vérification de modèles existants (dont Spin, dSpin, SMV, JPF). Quand un vérificateur de modèles produit une trace d'erreur, Bandera la remonte au niveau du code source et permet à l'usager d'avancer dans le code le long de la trace en visualisant la valeur des variables et l'état interne des objets Java.

On supposera ici le lecteur familier des concepts généraux de la compilation et de la vérification de modèles, et possédant la compréhension de base des logiques temporelles telles que LTL [MP91] ou CTL [CGL94]. La connaissance d'outils de vérification de modèles particuliers tels que SPIN ou SMV n'est pas nécessaire, encore que la familiarité avec de tels outils ne puisse qu'aider à la compréhension des traitements internes de Bandera. Pour une courte description de Bandera et des raisons motivant son architecture, voir [CDH+00]. Pour une vue d'ensemble de la logique temporelle et des techniques de vérification de modèles, voir [HR04].

2.1 Les difficultés de la vérification de modèles de codes sources

Les applications informatiques modernes nécessitent de plus en plus des systèmes logiciels concurrents ou distribués et d'une grande fiabilité. Malheureusement, les techniques actuelles de validation de logiciel, telles que test et inspection, sont impuissantes à fournir le haut degré de confiance exigé par ces systèmes, en raison de la taille et de la complexité de ces derniers mais aussi à cause des difficultés fondamentales posées par le raisonnement sur des séquences d'états ou d'événements constituant un comportement concurrent.

Les techniques de vérification de modèles (maintenant largement utilisées pour la vérification de matériel) apparaissent prometteuses pour l 'établissement de propriétés comportementales cruciales des logiciels complexes, car elles permettent de vérifier automatiquement qu'un modèle abstrait d'un système de transitions à états finis se conforme ou non à une propriété portant sur des séquences d'états ou d'événements. Si le modèle ne satisfait pas la propriété, le vérificateur de modèles exhibe un contre-exemple un chemin dans les transitions du modèle qui viole la propriété - qu'on peut utiliser pour localiser et corriger le défaut correspondant

du logiciel.

Malgré ces aspects prometteurs, il existe, semble-t-il, quatre problèmes empêchant à l'heure actuelle que la technologie de la vérification de modèles soit appliquée avec succès au domaine du logiciel.

- **Le problème de l'explosion combinatoire**. C'est le problème posé par la croissance exponentielle en taille du modèle à états finis à mesure que le nombre de composants du système augmente. Il existe un certain nombre de méthodes permettant de freiner l'explosion combinatoire du nombre d'états lors de l'analyse de certains types de systèmes, et ces méthodes se sont révélées suffisamment efficaces pour rendre abordable l'analyse de beaucoup de systèmes matériels. Malheureusement, les systèmes logiciels tendent à avoir des états plus complexes que les composants matériels, et il faut donc les abstraire encore plus efficacement pour obtenir des modèles qui puissent être manipulés.
- **Le problème de la construction des modèles**. On touche ici au problème du lien à établir entre les constructions produites par les développeurs de logiciels et celles prises en comptes par les outils de vérification actuels. La plupart des développements sont faits à l'aide de langages de programmation généraux (par exemple C, C++, Java ou Ada), mais la majorité des outils de vérification prennent en entrée des langages de spécification se caractérisant par la simplicité de leur sémantique (par exemple les algèbres de processus ou les machines d'états). Pour pouvoir utiliser un outil de vérification sur un programme réel, le développeur doit extraire un modèle mathématique abstrait du comportement pertinent du programme, et spécifier ce modèle dans le langage d'entrée de l'outil de spécification. Ce processus est source d'erreurs et coûteux en temps.
- **Le problème de la spécification des exigences**. Il s'agit ici de la difficulté qu'il y a à exprimer les exigences relatives au logiciel dans les langages de spécification temporelle des outils de vérification de modèles existants. Bien que les langages de spécification de propriétés des vérificateurs de modèles soient bâtis sur des logiques temporelles d'une grande élégance théorique, les usagers et même les chercheurs éprouvent des difficultés à les utiliser pour exprimer de façon précise des propriétés relatives à des séquences d'événements complexes. Et, une fois écrites, ces dernières se révèlent souvent difficiles à lire et à déboguer.

  En outre, les langages de spécification associés aux outils de vérification de modèles sont conçus pour permettre l'expression de propriétés de modèles mathématiques plutôt que celles d'un code source logiciel. La plupart des spécifications logicielles font référence à des caractéristiques des programmes telles que points de contrôle (par exemple l'entrée ou la sortie d'une méthode), variables locales ou variables d'instance, accès à des tableaux, de référence d'objets imbriqués. Mais les outils actuels fournissent peu ou pas d'aide pour établir un pont entre les caractéristiques du code source et leur représentation dans les modèles correspondants.

  Ceci signifie que l'usager est souvent contraint d'exprimer la spécification dans les termes de la représentation des caractéristiques du programme dans le modèle et non dans les termes du code source lui-même. L'usager doit donc comprendre ces représentations, qui sont habituellement fortement optimisées, pour exprimer ses spécifications de façon précise. De plus, les représentations peuvent changer selon les optimisations qui ont été utilisées lors de la génération du modèle. Des difficultés plus ardues encore apparaissent si l'on veut modéliser le dynamisme habituellement présent dans les logiciels orientés objets : les composants correspondant aux objets ou aux fils (threads) créés dy-

namiquement sont ajoutés dynamiquement à l'espace d'états en cours d'exécution. Ces composants sont anonymes, dans le sens où ils ne sont souvent pas directement liés aux variables apparaissant dans le programme source. Le défaut de noms de composants fixés au niveau du code source rend difficile l'écriture de spécifications décrivant les propriétés dynamiques de ces composants : de telles propriétés ne peuvent être exprimées qu'en fonction de la représentation du ń tas ż (heap) dans le modèle.

– **Le problème de l'interprétation des résultats**. Lorsqu'une propriété se révèle fausse lors de la vérification d'un gros modèle (et les systèmes logiciels produisent le plus souvent de très gros modèles), la trace correspondant au contre-exemple produit par le vérificateur peut comprendre plusieurs centaines ou même plusieurs milliers d'étapes[1]. Confronter manuellement ces contre-exemples au code est extrêmement fastidieux pour différentes raisons. En premier lieu, la longueur de la trace est telle que cela peut prendre des heures pour la parcourir. En second lieu, la trace d'erreur est exprimée dans les termes des représentations de bas niveau du modèle, qui peuvent être fortement optimisées. On se trouve ici avec le problème inverse de celui mentionné pour la spécification des propriétés : l'analyste doit comprendre la représentation dans le modèle de caractéristiques complexes des programmes pour pouvoir projeter précisément la trace d'erreur au niveau source. Une ń étape ż dans le programme source correspond habituellement à dix étapes dans la représentation du modèle de bas niveau.

Bandera tente de fournir différents types d'aides, sous forme d'outillages, permettant dÇaborder les différents problèmes énumérés ci-dessus, mais chacun d'entre eux est fondamental, et on ne voit pour l'instant pas clairement quelles sont exactement les technologies et les formes d'aides et d'outillages les mieux adaptés à leur résolution. Le but ultime du projet Bandera n'est donc pas de fournir une solution définitive aux problèmes énoncés ci-dessus, mais plutôt de fournir plusieurs formes d'aide et d'outillage à l'intérieur d'une infrastructure ouverte permettant plus facilement l'expérimentation de nouvelles techniques.

2.2 Domaines d'utilisation de Bandera

Bandera est en premier lieu utilisable pour vérifier automatiquement des propriétés sur des codes sources Java par vérification de modèles. La force de la vérification de modèles réside dans sa capacité à détecter des fautes logicielles subtiles résultant d'entrelacements non prévus lors de l'exécution concurrente de fils. On peut aussi l'utiliser pour vérifier des assertions, des pré-conditions et des post-conditions, ou de simples invariants de données.

Il reste cependant que la vérification de modèles fonctionnera toujours mieux pour la vérification de propriétés de contrôle que de propriétés relatives aux données. Plus la propriété implique de données, plus grand devient l'espace d'états, et donc plus grandes aussi deviennent les ressources nécessaires à la vérification des modèles.

La vérification de modèles a été utilisée pour vérifier des propriétés relatives à des objets tels que piles et files d'attente. Par exemple, on a pu l'utiliser pour montrer que le code d'une pile maintient effectivement un ordre LIFO [DP98]. Il semble toutefois impossible d'utiliser la vérification de modèles pour prouver qu'un algorithme de tri est correct, la validité d'un

1. Par exemple, dans une expérimentation de vérification de modèles de logiciel menée par des chercheurs de la NASA Ames et de Honeywell, la vérification par SPIN de propriétés de l'ordonnanceur du système d'exploitation temps-réel DEOS a produit des contre-exemples allant jusqu'à 2700 pas de long [PVE+00]

tri étant une propriété orientée données impliquant plusieurs quantifications et différentes structures de données. Pour vérifier ce type de propriété, il faut utiliser d'autres méthodes formelles, comme la preuve de théorèmes. De la même façon, d'autres problèmes de vérification, comme la vérification d'erreurs sur les bornes de tableaux, la saturation de tampons ou la déréférence de pointeurs nuls sont probablement mieux traités par d'autres formes d'analyse de flot de donnée.

Bandera offre différents outillages permettant de découper des programmes Java et leur appliquer des abstractions de données. Ces techniques sont également intéressantes pour l'analyse et le test. Bandera effectuant ces opérations sous forme de transformations de source à source, on peut en récupérer le résultat pour s'en servir d'entrée pour, par exemple, du test. On peut enfin incorporer un nouvel outil de vérification de modèles, à utiliser sur du code source Java, simplement en adjoignant à Bandera un traducteur du BIR dans le langage d'entrée du vérificateur.

2.3 Architecture de Bandera

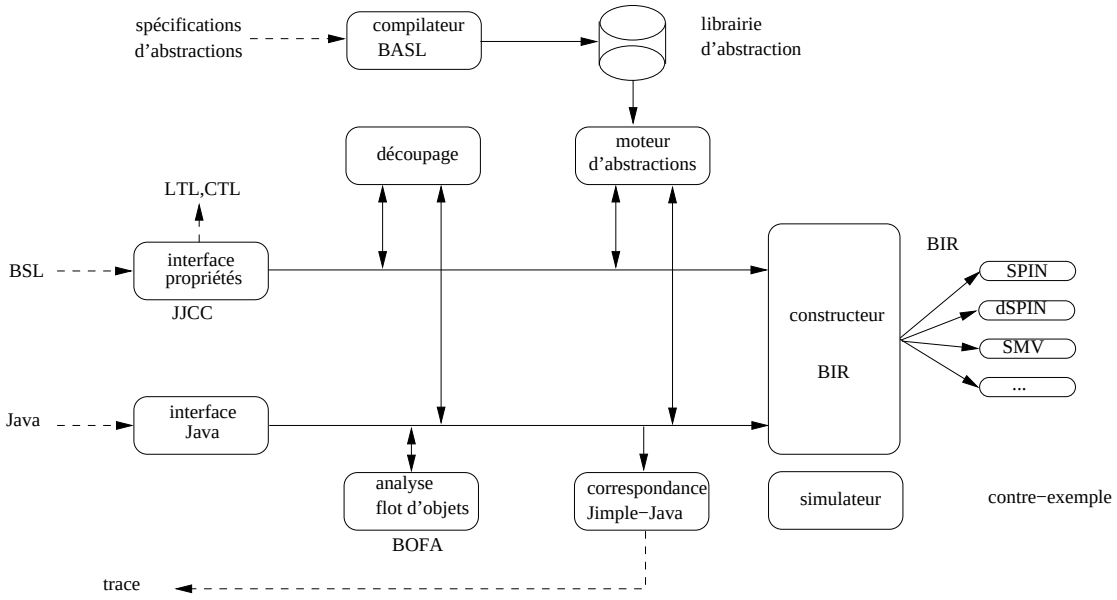

FIG. 2.1 – *Architecture de Bandera*

La figure 2.1 représente l'architecture de Bandera.

2.3.1 Infrastructure Java et représentation intermédiaire

Bandera est construit au-dessus de l'environnement de compilation Java Soot développé par le groupe Sable de Laurie Hendren à l'université McGill [VRHS+99]. Dans l'environnement Soot, les programmes Java sont traduits dans un langage intermédiaire appelé Jimple. Jimple a été au départ développé en tant que langage cible d'un décompilateur Java (c'est-à-dire que Soot fournit un outil permettant de décompiler es fichiers de classe Java (bytecode) en Jimple)

. Jimple est donc essentiellement un langage de graphes de flot de contrôle où:

- les instructions sont de forme code à trois adresses (la manipulation explicite de piles inhérentes aux instructions de la Machine Virtuelle Java a été supprimée par introduction de variables intermédiaires) ;
- les différentes constructions de Java telles qu'invocations de méthodes ou instructions synchronisées sont représentées par leur contrepartie dans la machine virtuelle (par exemple `invokevirtual`, `monitorenter` et `monitorexit`).

2.3.2 Frontal Java

À partir du prototype initial du groupe Soot, le groupe Bandera a développé son propre frontal Java, appelé JJJC (Java-to-Jimple-to-Java Compiler), traduisant Java en Jimple. JJJC construit aussi des structures de données permettant au Mappeur Jimple-Java d'aller et venir entre Java et Jimple. Par exemple, JJJC peut aussi décompiler du Jimple qui a été transformé par les composants de découpage (slicing) et d'abstraction de Bandera pour redonner du Java.

Le découpage et l'abstraction sous Bandera peuvent donc être considérés comme des opérations de source à source. Cela est utile lorsqu'on désire utiliser d'autres outils de vérification, travaillant sur Java au niveau source, en conjonction avec Bandera. Le Mappeur Jimple-Java peut aussi être invoqué durant le processus consistant à projeter un contre-exemple produit par le vérificateur de modèles au niveau source.

2.3.3 Spécification de propriétés

Les propriétés du code source à vérifier sont écrites dans le Langage de Spécification de Bandera (BSL). BSL est basé sur un ensemble de profils de spécifications temporelles à champs [DAC99], permettant à l'usager d'exprimer des spécification sous forme ń anglaise stylisée ż. Ces profils sont essentiellement des macros paramétrées pouvant être instanciées dans une ou plusieurs logiques temporelles, telles que LTL ou CTL. Ce système de profils aborde donc le problème de l'expression de spécifications, mentionné à la section précédente, en fournissant à l'usager des structures temporelles couramment utilisées dans les spécifications.

Les spécifications BSL sont paramétrées par des observables (prédicats portant sur l'état du programme) définis dans le code source Java par l'intermédiaire de la notation commentaire Javadoc. Du point de vue théorique, les spécifications BSL sont instanciées sous forme d'assertions et de formules en logique temporelle, et les observables déclarés par l'usager sont les propositions primitives apparaissant dans ces formules.

Le frontal de propriétés de la figure 2.1 appelle le frontal Java afin d'extraire tous les observables définis dans un code source donné, il vérifie le type des observables déclarés et instancie la spécification BSL dans une logique temporelle particulière, en traduisant les observables utilisés dans la spécification d'entrée dans la représentation de bas niveau. Cette dernière étape aborde donc le problème du hiatus entre les différents niveaux de représentation, en traduisant automatiquement les propriétés décrites en termes de caractéristiques du code source en représentation de bas niveau optimisée.

2.3.4 Analyse statique

Aux fins d'optimisation, Bandera effectue un certain nombre d'analyses classiques sur les données et le contrôle. En particulier l'Analyse de Flot d'Objets Bandera (BOFA) collecte statiquement des informations sur les objets pouvant entrer dans chaque expression du programme à l'exécution. Dans une configuration typique, BOFA associera un ensemble de jetons t1, Ä ,tn à une expression particulière e, un jeton étant une paire (C, s) où C est un nom de classe et s est le site d'allocation, où l'objet a été créé. BOFA est, on le voit, analogue aux analyses de type pointeur utilisées pour les langages impératifs traditionnels, ou aux analyses par fermetures utilisées pour les langages de programmation fonctionnels. BOFA s'exécute habituellement en mode insensible au contexte et insensible aux flots, mais est conçu de façon à permettre d'accroître la précision selon différents axes.

2.3.5 Approche de la construction de modèles

L'approche de Bandera en matière de construction de modèles consiste à engendrer un modèle pour chaque propriété à vérifier. Cette approche est basée sur l'observation que, étant donné une propriété spécifique Õ, une grande partie du programme n'a aucune influence sur Õ. Bandera emploie donc optimisations et abstractions pour retirer du programme les éléments ne concernant pas Õ et donc engendrer un modèle rendu très compact. Il faut noter que cette approche, adaptée à chaque cas, est généralement impraticable si on génère les modèles à la main. On pourrait objecter qu'engendrer un modèle spécifique pour chaque propriété induit un surcoût important. Il est toutefois fréquent que la vérification d'une propriété particulière Õ sans adaptation du modèle s'avère infaisable en raison du coût exponentiel de la vérification.

La philosophie de Bandera, en outre, est de concevoir l'adaptation de telle sorte que le coût de cette adaptation soit très inférieur au coût de la vérification (en d'autres termes, le temps consacré à l'adaptation doit être en pratique plus court que le temps nécessaire à la vérification).

2.3.6 Découpage(slicing)

Bandera utilise aussi bien le découpage de programme que l'abstraction de données pour adapter les modèles. L'outil de découpage de programmes de Bandera prend en entrée tous les observables indiqués pour la propriété d'entrée φ. Ces observables peuvent faire référence à des variables ou à des points de contrôle particuliers du programme. La sémantique de ces caractéristiques du programme doit être préservée afin de vérifier correctement φ, mais tous les autres éléments du programme n'influençant pas la sémantique des caractéristiques observables peuvent être éliminées dans le modèle engendré.

L'outil de découpage de programmes construit un graphe de dépendances du programme, représentant plusieurs formes de dépendances, et engendre un programme exécutable résiduel (le découpage du programme) où tous les éléments n'influençant pas l'exécution des observables dans φ ont été retirés. Le programme découpé est créé au niveau Jimple, mais il peut aussi être décompilé en Java par JJJC. Pour plus de détails, le lecteur est renvoyé aux formalisations de l'approche Bandera du découpage conduit par les propriétés [HDZ00], et des notions de dépendances dans les programmes nécessaires pour découper les constructions parallèles de Java [HCD+99].

2.3.7 Interprétation abstraite

L'outil d'abstraction de Bandera fournit une aide automatique pour la réduction de modèles par l'intermédiaire de l'abstraction de données. Ceci est utile lorsqu'une spécification φ à vérifier ne dépend pas des valeurs concrètes traitées par le programme, mais ne dépend que des *propriétés* de ces valeurs. Par exemple, une application peut stocker un ensemble d'éléments dans un vecteur, mais si la propriété à vérifier ne dépend que de la présence d'un élément particulier dans le vecteur, on peut abstraire un grand nombre d'états du vecteur en un petit ensemble : *ItemInVector,ItemNotInVector*.

L'usager guide le processus d'abstraction en liant les variables à des entrées d'une *librairie d'abstractions*. Les entrées de la librairie sont indexées par type concret, et chaque entrée implémente une version abstraite de son type concret correspondant. Chaque abstraction de la librairie est définie en utilisant le Langage de Spécification d'Abstraction de Bandera (BASL).

Une spécification BASL pour une abstraction d'un type de base consiste en la déclaration d'un ensemble fini de jetons abstraits, d'une fonction d'abstraction qui relie chaque valeur concrète Java à un jeton abstrait et d'une opération abstraite pour chaque opération sur le type concret.

Un format à base de règles incorporant de la reconnaissance de profils (*pattern matching*) simplifie la définition des opérations abstraites. Pour les abstractions des types de base, le compilateur BASL permet à l'usager de fournir une spécification BASL abrégée contenant seulement les jetons abstraits et la fonction synthétiser automatiquement la version abstraite d'opérations telles que +, -, etc. Ceci rend extrêmement facile la définition de nouvelles abstractions des types de base. Étant donné une spécification BASL (éventuellement synthétisée), le compilateur BASL engendre une classe Java contenant les méthodes implémentant la fonction d'abstraction définie ainsi que les versions abstraites des opérateurs concrets appropriés. La classe Java est conservée en librairie, et est liée aux reste du code lors de la génération du modèle, si l'usager a choisi cette abstraction particulière.

Les abstractions étant incorporées en prenant en compte chaque variable, deux variables de même type concret peuvent avoir des types abstraits différents. Par exemple, si *I1* et *I2* sont deux abstractions du type `int`, la variable `int x` peut être liée à *I1* et la variable `int y` à *I2*. Après que l'usager ait choisi une abstraction pour quelques variables clé, significatives pour la propriété à vérifier, une phase d'inférence de type propage cette information dans tout le programme et infère le type d'abstraction pour les variables restantes et pour chaque expression du programme. L'inférence de type informe aussi l'usager lorsqu'une erreur dans le type d'abstraction apparaît.

Une fois l'inférence de type abstrait exécutée, le moteur d'abstraction transforme le code source en une version spécialisée dans laquelle toutes les opérations concrètes et tous les tests sur les objets pertinents (par exemple appels de méthodes sur la classe du vecteur) sont remplacés par leurs versions abstraites manipulant des jetons représentant les valeurs abstraites *ItemInVector,ItemNotInVector*. De l'information étant perdue lors du processus d'abstraction, les opérations et tests reposant sur de l'information perdue ne peuvent plus être complètement déterminées dans le programme abstrait. Par exemple, dans l'exemple sur le vecteur, la longueur du vecteur abstrait ne peut être déterminée. Tout test conditionnel consultant la longueur du vecteur doit être transformé en choix non déterministe entre la branche `vrai` et la branche `faux`. Comme pour le découpage, le code abstrait est représenté au niveau Jimple mais JJJC peut le décompiler en Java.

2.3.8 Génération finale de modèles

Le terminal Bandera est analogue à un générateur de code, prenant en entrée le programme découpé et abstrait et produisant les représentations spécifiques des vérificateurs cibles. Les composants terminaux communiquent par l'intermédiaire de la Représentation Intermédiaire Bandera (BIR), intermédiaire entre les représentations orientées compilation et les représentations orientées vérification. Comme le montre la figure 2.1, le terminal comporte un composant fixe appelé BIRC(*Bandera Intermediate Representation Constructor*) prenant en entrée une forme restreinte de Jimple et produisant du BIR. Pour chaque vérificateur pris en compte, il existe aussi un traducteur qui prend en entrée la représentation du programme en BIR et génère l'entrée pour ce vérificateur. Actuellement les traducteurs pour SPIN, dSPIN, JPF et SMV sont opérationnels, et des traducteurs pour FDR2, XMC et le vérificateur de modèles SAL de Stanford, en cours de conception, sont en cours de développement.

BIR est un langage à commandes gardées pour la description de systèmes à transitions d'états. Le but principal du BIR est de fournir une interface simple pour l'écriture de traducteurs pour les vérificateurs cibles. Pour utiliser Bandera avec un nouveau vérificateur, il suffit d'écrire un traducteur du BIR vers le langage d'entrée du vérificateur. Le style du BIR à base de commandes gardées est bien adapté aux langages d'entrée des vérificateurs actuels.

Le BIR contient certaines constructions évoluées afin de faciliter la modélisation de certaines des caractéristiques de Java, telles que les fils, les verrous, et une forme bornée de l'allocation de ń tas ż. Plutôt que de choisir une implémentation de ces constructions et de les retirer du BIR (par exemple, représentation des verrous sous forme de variables booléennes), on permet au traducteur d'implémenter ces constructions de la façon la plus efficace du point de vue du langage d'entrée du vérificateur. Le BIR fournit aussi d'autres informations pouvant aider les traducteurs à élaborer des modèles plus compacts. Par exemple, une commande gardée peut être étiquetée *invisible*, ce qui indique qu'elle peut être exécutée avec son successeur de façon atomique. L'ensemble des variables locales qui sont vivantes à chaque point de contrôle peut être spécifié (les variables mortes peuvent être affectées d'une valeur fixée pour SPIN ou laissées non contraintes pour SMV).

Chapitre 3

Spécification : formalisation de propriétés dans Bandera

On donne ici des informations sur la façon dont les spécifications sont construites à l'aide de la syntaxe et des outils Bandera. On commence par décrire la syntaxe utilisée pour les spécifications en Bandera, c'est-à-dire le Langage de Spécification Bandera (BSL). On donne ensuite un aperçu du système de profils de spécifications, puis un aperçu des outils permettant de spécifier, parcourir et manipuler des spécifications en Bandera.

On approfondit ici les motivations de BSL et les raisons ayant orienté sa conception, et on discute sa syntaxe et sa sémantique.

3.1 Introduction

On a donné en introduction les raisons ayant conduit à la définition d'un langage de spécification propre, par rapport à l'option consistant à ce que les développeurs expriment leurs exigences dans le langage de spécification de l'un des outils de vérification de modèles sous-jacents (par exemple la logique temporelle linéaire [LTL] de SPIN ou la logique calculatoire arborescente [CTL] de NuSMV).

On décrit ici la conception et l'implémentation du Langage de Spécification Bandera (BSL), un langage évolué indépendant de tout outil de vérification de modèles, permettant d'exprimer des propriétés temporelles sur les actions et les données de programmes Java. Ce langage aborde les problèmes énoncés en introduction et aide à surmonter les difficultés qu'on rencontre lorsqu'on spécifie les propriétés d'un logiciel évoluant dynamiquement. Par exemple, si l'on considère une propriété d'une implémentation de tampon borné spécifiant que quand un tampon est plein, il finira par se vider, il y a plusieurs difficultés lorsqu'on met cette spécification sous une forme pouvant être vérifiée automatiquement, notamment :

- définir le sens de plein dans l'implémentation ;
- quantifier le temps pour s'assurer qu'un tampon plein se vide finalement ;
- quantifier toutes les instances de tampons bornés dynamiquement créées dans un programme.

BSL sépare tous ces aspects et les traite à l'aide de sous-langages spécialisés, comme le

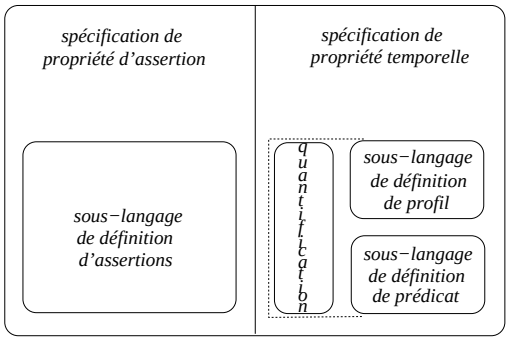

FIG. 3.1 – *Organisation du BSL*

montre la figure 3.1.

- Un sous-langage d'*assertion* permet aux développeurs de définir des contraintes sur le contexte des programmes dans une notation familière de type assertion. Les assertions peuvent être sélectivement validées ou invalidées de façon à ce que l'on puisse aisément identifier un sous-ensemble d'assertions pour la vérification. Bandera exploite cette possibilité en optimisant les modèles engendrés (par découpage et abstraction) spécifiquement pour les assertions choisies.

- un sous-langage de *propriétés temporelles* fournit les moyens de définir des prédicats sur les points de contrôle Java courants (par exemple l'invocation et le retour de méthode) et sur les données Java (notamment les fils et objets dynamiquement créés). Ces prédicats deviennent les propositions de base des spécifications temporelles. Le langage de spécification temporelle n'est pas basé sur une logique particulière, mais plutôt sur un ensemble de profils de spécifications temporelles à champs développé dans de précédents travaux [DAC99]. Ce langage de profils est extensible et permet la création de librairies de profils spécifiques d'un domaine.

  Interagissant aussi bien avec les aspects prédicat qu'avec les aspects profils de BSL on trouve un outil de quantification puissant permettant aux spécifications temporelles d'être quantifiées sur tous les objets ou fils d'une classe particulière. La quantification fournit un moyen de nommer les données potentiellement anonymes, et ce type d'aide se révèle crucial pour l'expression de raisonnements sur des objets dynamiquement créés.

Les assertions et les prédicats sont inclus dans le code source sous forme de commentaires *Javadoc*. Ceci permet l'extraction et le parcours par les développeurs d'une documentation HTML et, dans certains cas spéciaux de traitement par *doclet*, cela fournit une infrastructure bien outillée pour l'écriture de spécifications.

Même si BSL est basé sur Java, et, jusqu'à un certain point résulte du désir de s'abstraire de l'ensemble des outils de vérification de modèles finals utilisés par Bandera, les idées générales incorporées dans ce langage peuvent être utiles dans tout environnement dédié à la vérification de modèles de code source.

3.2 Objectifs de la conception

BSL est le dernier d'une série de langages et de systèmes qui ont été créés pour aider à la spécification de propriétés temporelles de systèmes logiciels. L'expérience d'un précédent langage de propriétés simple pour des logiciels Ada [DPC98] et avec un système de profils de spécifications temporelles [DAC99] a permis de dégager les critères de conception suivants :

1. Le langage de spécification doit cacher les complications de la logique temporelle, en s'appuyant sur des profils de codage communs.
2. Même si les détails de la logique temporelle doivent être cachés, les usagers experts doivent pouvoir écrire des spécifications (ou des fragments de spécification) directement en LTL (par exemple).
3. Le langage doit prendre en compte les styles de spécification déjà utilisés par les développeurs, comme les assertions, les pré-conditions et les post-conditions, ...
4. Les constructions de la spécification elles-mêmes doivent être proches du code. Les spécifications référencent des caractéristiques du code (par exemple, noms de variables, noms de méthodes, points de contrôle) ; les principaux éléments de la spécification sont situés dans le code, de façon à ce qu'ils puissent facilement être lus, parcourus et maintenus.
5. Finalement, et c'est peut-être le plus important, le langage doit contenir des aides au raisonnement sur les caractéristiques utilisées dans les logiciels réels, comme création dynamique d'objets ou de fils, définition d'interface, exceptions, etc. Ces caractéristiques impliquent souvent des structures allouées dans le ń tas ż, atteignables en suivant des chaînes de références, mais elles sont réellement anonymes quand on considère les champs statiques et les variables locales d'un programme Java

Le critère (1) dérive de l'expérience avec le Projet de Profils de Spécification. BSL fournit un outillage à base de macros implémentant les profils de [DAC99]. Conformément au critère (2), il permet aussi aux usagers experts d'écrire leurs propres profils ou de coder directement leurs spécifications en logique temporelle.

En ce qui concerne le critère (3), la conception et la présentation de BSL ont été influencées par différents langages d'assertions Java comme iContract [12]. Dans chacun de ces langages, les développeurs écrivent leurs spécifications dans des commentaires Java en utilisant l'outil Javadoc. C'est aussi une manière efficace de traiter le critère (4) : on peut créer une documentation HTML pour BSL, que l'on peut parcourir, et la proximité physique des commentaires et du code encourage la maintenance régulière.

BSL ne traite pas actuellement toutes les caractéristiques des programmes mentionnées par le critère (5) (on ne parle pas de spécifier des propriétés sur les exceptions) mais de progrès significatifs ont été effectués dans la manipulation des données et fils alloués dans le ń tas ż.

3.3 Le sous-langage d'assertions

Objectifs. Le mécanisme des assertions fournit au programmeur une technique adéquate pour spécifier des contraintes sur l'espace de données d'un programme, devant être valides quand le contrôle atteint un endroit particulier. En programmation C ou C++, les assertion sont habituellement directement intégrées au code source par l'intermédiaire de la macro assert, l'emplacement de l'assertion étant donné par la position de l'appel de la macro dans le code source.

En raison des possibilités offertes par Java pour l'extraction de documentation HTML à partir de commentaires dans le code Java par l'intermédiaire de la technologie Javadoc, plusieurs mécanismes d'assertions Java connus, comme iContract [Kra98], aident à la définition d'assertions dans les commentaires en tête des méthodes Java. BSL adopte aussi cette approche. Par exemple, dans l'assertion BSL:

```
/**
 * @assert
 *   PRE PositiveBound: (b>0);
 */
```

la contrainte sur les données est (b>0) et le point de contrôle est spécifié par l'occurrence de l'étiquette @assert PRE dans l'en-tête de documentation de la méthode pour laquelle on veut poser cette assertion ; la contrainte doit être valide chaque fois que le contrôle passe dans la première instruction exécutable de la méthode.

Syntaxe et sémantique informelle. La syntaxe des assertions est donnée ci-dessous :

```
<assertions> ::= @assert <assertion-set-name>? <comment>* <assertion>*

<assertion-set-name> ::= <java-id> | <assertion-set-name> . <java-id>
<label>              ::= <java-id>
<assertion-name>     ::= <java-id>

<assertion>
 ::= PRE<assertion-name> : <exp> ; <comment>*
   | LOCATION '[' <label> ']' <assertion-name> : <exp> ; <comment>*
   | POST <assertion-name> : <exp> ; <comment>*
```

Les ensembles d'assertions sont définis en utilisant l'étiquette Javadoc `@assert` dans l'en-tête de documentation des méthodes ou constructeurs. Chaque ensemble d'assertions défini par une étiquette `@assert` peut recevoir un nom optionnel, et ce nom est utilisé avec le nom de chaque assertion dans l'ensemble pour identifier les assertions de façon unique, pour la valider ou l'inhiber. Si le nom d'ensemble est omis, on utilise le nom entièrement qualifié de la méthode correspondante comme nom pour l'ensemble d'assertions. Le nom optionnel peut être suivi de zéro ou plus commentaires Java.

Outre les assertions de pré-condition illustrées plus haut, BSL prend en compte les assertions d'emplacement et les assertions de post-condition.

`LOCATION [ <label> ] <assertion-name> : <exp>` est satisfaite si l'expression `<exp>` est vraie quand le contrôle est à l'instruction Java étiquetée par `<label>` dans la méthode correspondante. `POST <assertion-name> : <exp>` est satisfaite si `<exp>` est vraie immédiatement après l'exécution de l'une des instructions `<return>` de la méthode ou après l'exécution de la dernière instruction de la méthode s'il n'y a pas d'instruction `<return>`. L'expression `<exp>` peut faire référence à la valeur retournée par la méthode en utilisant l'identificateur réservé de Bandera : `$ret`.

Il existe différentes autres règles de bonne formation associées à la portée des variables qu'on ne discutera pas ici. Par exemple, une assertion de pré-condition ne peut référencer des variables locales de la méthode puisqu'elles n'ont pas été initialisées quand le *bytecode* du

corps de la méthode commence à s'exécuter, et assertion portant sur une étiquette ne peut référencer que les variables dans la portée du point où l'étiquette considérée apparaît dans le code.

Une fois les assertions déclarées, on peut présenter un ensemble d'assertions à Bandera comme une *spécification d'assertions* à vérifier par modèle. Les assertions violées font l'objet d'un rapport à l'usager par l'intermédiaire d'une trace traversant le programme jusqu'à l'endroit où la condition sur les données est violée.

Aspect implémentation. Comme pour les autres outils Java d'assertions, l'implémentation d'une assertion BSL fonctionne comme un préprocesseur : elle transforme le code source et intègre chaque assertion valide en utilisant la méthode `Bandera.assert(boolean)` de la librairie Bandera. Il faut un petit peu plus de travail pour établir la correspondance d'étiquettes appropriée avec les assertions `LOCATION` et pour calculer la valeur de la variable `$ret` pour les assertions `POST`, mais sinon la transformation est directe. On peut aussi inscrire des assertions en dur directement avec `Bandera.assert(boolean)`, mais ce n'est pas conseillé. Quand Bandera génère un modèle pour SPIN, chaque appel à `Bandera.assert` est traduit en instructions `ASSERT` de Promela.

3.4 Le sous-langage de spécification temporelle

Bien que les assertions fournissent une façon pratique d'écrire des contraintes sur les données en des points particuliers du programme, elles ne sont pas assez puissantes pour permettre la spécification directe de relations temporelles intéressantes entre actions du système. De telles propriétés étant fréquemment requises dans les systèmes concurrents, les vérificateurs de modèles prennent en général en compte un langage de spécification de propriétés temporelles, basé par exemple sur LTL ou CTL.

Ces langages de spécification temporelle subsument les assertions dans le sens où toute assertion *(l, c)* où l est un emplacement et c une condition sur les données à cet endroit peut être codée sous forme d'une propriété temporelle : le long de tous les chemins, il doit être vrai dans tout état s que si le point de contrôle de s est l alors c est valide. Toutefois, les vérificateurs de modèles tels que SPIN prennent en compte les assertions parce qu'on peut les vérifier beaucoup plus efficacement que les propriétés temporelles générales.

Sous-langage de définition de prédicats. On introduit les prédicats BSL en utilisant deux schémas de classification différents. Le premier schéma classifie un prédicat `p` comme étant indépendant de l'emplacement (la satisfaction du prédicat est indépendante de la valeur du compteur ordinal) ou comme dépendant de l'emplacement (`p` référence un emplacement particulier du programme et donc sa satisfaction dépend de la valeur du compteur ordinal).

Par exemple, `EXP Full(this): (head == tail)` est un prédicat indépendant de l'emplacement. Cette forme de prédicat, appelée *prédicat expression*, est souvent utilisée pour définir des invariants de classes ou pour indiquer des états distingués (par exemple, tampon plein) dans une classe ou une donnée d'instance. Les prédicats d'expression ne référençant pas de points de contrôle particuliers dans les méthodes, on peut les définir exclusivement dans l'entête de documentation de classe.

Le second schéma de classification classifie les prédicats selon le type de champs ou de

code auxquels ils se réfèrent. Les *prédicats statiques* sont utilisés pour raisonner sur les champs `static` (variables de classe) ou sur les points de contrôle des méthodes `static`. Les *prédicats d'instance* sont utilisés pour raisonner sur les champs d'instances (les cellules mémoire apparaissant dans chaque objet d'une classe donnée). Par exemple le prédicat `Full` est un prédicat d'instance, parce qu'il fait référence à des données membres de l'instance (`BoundedBuffer` en l'occurrence).

Il existe une distinction syntaxique entre prédicats statiques et prédicats d'instance qui s'exprime lorsqu'on utilise le mécanisme de paramétrisation des prédicats BSL. Java utilisant l'identificateur `this` pour identifier l'objet receveur d'une méthode virtuelle, on suit un style analogue et on spécifie un prédicat d'instance en déclarant `this` comme étant un paramètre formel du prédicat ; un prédicat statique est un prédicat n'ayant pas `this` comme paramètre formel. Quand un prédicat d'instance est utilisé dans une spécification temporelle, le paramètre réel correspondant à `this` indique l'instance à laquelle le prédicat est appliqué.

Syntaxe et sémantique informelle. La syntaxe des définitions de prédicats en BSL est donnée ci-dessous.

```
<predicates>
  ::= @observable <predicate-set-name>?
      <comment>* <predicate>*

<predicate-set-name>
  ::= <java-id>
    | <predicate-set-name> . <java-id>
<label>              ::= <java-id>
<predicate-name>     ::= <java-id>

<predicate>
  ::= EXP <predicate-name>
      <params> : <exp> ; <comment>*
    | INVOKE <predicate-name>
      <params [: <exp>]? ; <comment>*
    | LOCATION '[' <label> ']' <predicate-name>
      <params> [: <exp>]? ; <comment>*
    RETURN <predicate-name>
      <params> [: <exp>]? ; <comment>*
<params>
  ::= ( [this ,]? <var-decls>? )
<var-decls>
  ::= <var-decls> , <java-type> <java-id>
```

Le mécanisme de définition des prédicats dans des commentaires Javadoc est identique à celui des assertions. Des ensembles de prédicats sont définis au moyen de l'étiquette Javadoc `@observable` dans l'en-tête de documentation pour les classes ou les méthodes. Chaque ensemble de prédicats défini par une étiquette `@observable` peut recevoir un nom optionnel, et ce nom est utilisé avec le nom de chaque prédicat dans l'ensemble pour identifier les prédicats de façon unique, de telle sorte qu'il puisse être référencé par une spécification de propriété temporelle. Si le nom d'ensemble est omis, on utilise le nom entièrement qualifié de la méthode

correspondante comme nom pour l'ensemble de prédicats. Le nom optionnel peut être suivi de zéro ou plus commentaires Java. Outre `this`, un prédicat peut avoir un nombre quelconque de paramètres. Actuellement, chaque paramètre de prédicat doit avoir un type de classe, c'est-à-dire que les paramètres ne peuvent être des scalaires. La raison est que la structure syntaxique de BSL ne permet comme arguments que des variables apparaissant dans la quantification des classes.

Outre les prédicats d'*expression* illustrés plus haut, BSL prend en compte les prédicats d'*invocation*, les prédicats d'*emplacement* et les prédicats *de retour*, qui sont tous des prédicats dépendant de l'emplacement définis dans l'en-tête de documentation d'une méthode. Un prédicat statique dépendant de l'environnement ne peut être déclaré que dans une méthode statique. De même, un prédicat d'instance dépendant de l'environnement ne peut être déclaré que dans une méthode d'instance. La syntaxe informelle et les règles de type pour chacun de ces prédicats sont données ci-dessous.

`EXP <predicate-name> <params> : <exp>` est vrai quand `<exp>` s'évalue à vrai compte tenu des paramètres `<params>`. Le passage de paramètres se fait par valeur comme en Java. La portée de l'expression <exp> est considérée statiquement par rapport au contexte Java de son point de déclaration.

`INVOKE <predicate-name> <params [: <exp>]` est vraie quand le contrôle est sur la première instruction de la méthode correspondante et quand `<exp>` est vraie compte tenu des paramètres `<params>` ; la valeur vraie est prise en l'absence de `<exp>`.

La sémantique de `LOCATION '[' <label> ']' <predicate-name> <params> [: <exp>]` est similaire à celle du prédicat `INVOKE`, à ceci près que le point de contrôle pertinent est l'instruction Java étiquetée par `<label>` dans la méthode correspondante.

La sémantique de `RETURN <predicate-name> <params> [: <exp>]` est similaire à celle du prédicat `INVOKE`, à ceci près que les points de contrôle pertinents sont ceux situés immédiatement après les instructions return de la méthode, ou après la dernière instruction de la méthode s'il n'y a pas d'instruction return. L'expression `<exp>` peut référencer la valeur retournée par la méthode par l'intermédiaire de l'identificateur réservé Bandera : `$ret`.

Les expressions Java prises en compte dans <exp> sont des expressions Java garanties sans effet de bord. On exclut donc à l'heure actuelle les affectations, les créations d'objets ou de tableaux, les invocations de méthodes.

Il existe différentes autres règles de bonne formation associées à la portée des variables qu'on ne discutera pas ici. Par exemple, un prédicat de retour ne peut faire référence à des variables locales de la méthode puisqu'il peut y avoir plusieurs instructions de retour pour la méthode, chacune ayant un ensemble de variables locales visibles différent.

Contraintes implicites. L'exécution de certains opérateurs Java/BSL pouvant produire des exceptions à l'exécution (par exemple, l'exception `NullPointerException`), les expressions contenant de tels opérateurs sont assorties de contraintes implicites interdisant l'apparition de ces exceptions (qui, sinon, pourraient interférer avec la vérification de modèles). Par exemple, l'expression de prédicat `x.f` est interprétée comme `(x != null) && x.f`. Si `x == null`, le prédicat est faux. Si l'analyse statique peut déterminer que ces exceptions n'apparaîtront jamais, ces contraintes peuvent être omises.

Spécification de profils temporels. Les automates et formalismes de logique temporelle, communément utilisés pour décrire les états et propriétés relatives aux séquences d'événements des systèmes concurrents, sont subtils et difficiles à utiliser correctement. Même les gens ayant une bonne expérience des logiques temporelles peuvent éprouver des difficultés à spécifier les exigences portant sur les logiciels dans le formalisme de leur choix. Il est donc intéressant de rendre ces formalismes pratiquement utilisables par des développeurs de logiciels.

Pour aborder ce problème, il a été, lors d'un précédent travail [DAC98b], identifié un petit nombre de classes d'exigences temporelles apparaissant fréquemment, par exemple l'invariance, la réponse, et les propriétés de précédence. On désigne ces classes sous le terme de *profils de spécification* ; il y a cinq profils de base :

- les propriétés *universelles* requièrent que l'argument soit vrai durant toute l'exécution ;
- les propriétés d'*absence* requièrent que l'argument ne soit jamais vrai durant l'exécution ;
- es propriétés d'*existence* requièrent que l'argument soit vrai en un certain point de l'exécution ;
- les propriétés de *réponse* requièrent qu'une occurrence désignée d'état ou d'événement soit suivie d'une occurrence d'un autre état ou événement désigné dans l'exécution ;
- les propriétés de *précédence* requièrent qu'une occurrence désignée d'état ou d'événement se produise toujours avant la première occurrence d'un autre état ou événement désigné.

En outre, certains profils de chaîne permettent la construction de séquences de relations de réponse et de précédence dépendantes. Un site web [DAC98a] présente l'ensemble actuel de huit profils et leurs variations ainsi que leur traduction dans cinq formalismes de spécification temporelle courants, parmi lesquels LTL et CTL.

Lors du développement de ce système de profils, il a paru utile de distinguer les profils d'états ou d'événements nécessaires des régions de l'exécution d'un système dans lesquelles ces profils sont requis. La *portée d'un profil* définit une variation du profil de base dans laquelle la vérification du profil peut être inhibée dans certaines régions de l'exécution. Il y a cinq portées de base :

- *globalement* pour toute l'exécution du système ;
- *après* la première occurrence d'un état ou d'un événement ;
- *avant* la première occurrence d'un état ou d'un événement ;
- *entre* deux états ou événements ;
- *durant* un intervalle, ou bien *après* un état ou un événement et `jusqu'à` la prochaine occurrence d'un autre état ou événement ou bien *durant* tout le reste de l'exécution si cette occurrence ne se produit pas.

Dans des travaux ultérieurs [DAC99], il a été vérifié que ces profils de spécification étaient représentatifs d'une large majorité des exigences temporelles écrites par les chercheurs et les usagers des outils de vérification de modèles à états finis. Plus de 600 spécifications de propriétés ont été étudiées, et il a été constaté que plus de 94% étaient des instances de ces profils ; il est aussi intéressant de noter que plus de 70% des propriétés étaient soit des propriétés universelles soit des propriétés de réponse.

L'approche basée sur les profils présente un certain nombre d'avantages. Elle peut écourter la période dÇapprentissage en présentant un plus petit ensemble de concepts aux développeurs de spécifications. Ces profils peuvent être exprimés dans presque tous les langages de spécification associés aux outils de vérification existants, et donc les profils offrent un haut niveau

d'indépendance. Enfin, des techniques d'optimisation de la construction de modèles à états finis peuvent exploiter l'information disponible sur la structure des profils.

BSL fait son travail en offrant un langage de structure anglaise comme source pour les profils, illustré par les ensembles de règles 4 et 5 de la syntaxe spécifiée ci-dessous :

```
<temporal>
  ::= <id> : <quantifier>* <pattern>

<quantifier> ::= forall [ <ids> : <java-type> ] .

<ids> ::= <id> | <ids> , <id>

<pattern>
  ::= <pred-expr> is universal <scope>
   |  <pred-expr> is invariant <scope>
   |  <pred-expr> is absent <scope>
   |  <pred-expr> exists  <scope>
   |  <pred-expr> precedes  <scope>
   |  <pred-expr> leads to <scope>
   |  <pred-expr> responds to <scope>

<scope> ::= globally
         |  before <pred-expr>
         |  after <pred-expr>
         |  between <pred-expr> and <pred-expr>
         |  after <pred-expr> until <pred-expr>

<pred-expr> ::= { <expr> }

<expr> ::= <predicate-name> ( <args>? )
        |  ( <pred-expr> )
        |  ! <pred-expr>
        |  <pred-expr> && <pred-expr>
        |  <pred-expr> || <pred-expr>
        |  <pred-expr> -> <pred-expr>

<args> ::= <id>
        |  <args> , <id>
```

Les synonymes usuels sont pris en compte. Par exemple, on peut exprimer ainsi la propriété déjà citée à propos des tampons bornés : quand un tampon est plein, il finira par se vider de la façon suivante :

```
FullToNonFull: forall[b:BoundedBuffer].
  {Full(b)} leads to {!Full(b)} globally
```

On traduira ensuite cette formulation pour le vérificateur choisi. Par exemple, pour SPIN:

```
[] ( {Full(b)} -> <> {!Full(b)} )
```

Spécifications de propriétés d'instances de classes. Une des principales difficultés lors de la spécification de propriétés de programmes Java réside dans le nommage des objets constituant l'état du système. La majorité des états du système est constituée du ń tas ż alloué et référencé par l'intermédiaire de variables locales par navigation le long de chaînes de références. On cherche en général à énoncer des propriétés valables pour toutes les instances d'une classe, ou bien de distinguer les instances d'une classe par observation de leur état. Une façon d'y arriver est d'offrir la possibilité d'énoncer des propriétés pour toutes les instances d'une classes créées lors d'une exécution.

La syntaxe de la *quantification d'instance de classe* universelle est donnée par les règles ci-dessus. BSL offre cette capacité par un mécanisme indépendant du vérificateur spécifique utilisé pour raisonner sur la propriété définie dans la portée du quantificateur. On y arrive en particularisant le modèle en fonction des quantificateurs utilisés et en intégrant la propriété à vérifier dans un profil de spécification garantissant qu'elle sera vérifiée seulement pour les objets du domaine du quantificateur.

Pour la clarté de la chose, on décrira la particularisation du modèle en termes de transformations du programme source et non en termes de transformations de la représentation intermédiaire Bandera. Cette transformation nécessite l'usage du non-déterminisme dans le modèle, ce que Java ne permet pas ; aussi on introduit une méthode statique `Bandera.choose`, traduite par Bandera en choix non déterministe entre ses arguments à l'entrée du vérificateur de modèles. On décrira aussi l'intégration d'une propriété LTL quantifiée.

Quantification d'instance de classe universelle. La quantification universelle se traduit comme suit, pour une formule quantifiée :

```
forall[var:QuantifierClass].(var)
```

1. Pour la variable liée `var` dans la quantification, on introduit un champ statique `var` de type `QuantifierClass` dans la classe publique `BoundVariables`. Ceci crée une variable d'état globale pour chacun de ces champs dans le modèle à états finis du programme.
2. Pour chaque champ de `BoundVariables` on définit un prédicat `var_selected` qui est vrai quand `var` n'a pas la valeur `null`.
3. À la fin de chaque constructeur de `QuantifiedClass` on introduit le fragment de code suivant :

   ```
   if(BoundVariables.var == null)
     if (Bandera.choose(true,false)) BoundVariables.var = this ;
   ```

   où `Bandera.choose(true,false)` introduit un choix non déterministe entre `true` et `false` dans le modèle à états finis du programme.
4. La formule temporelle `P`, à vérifier sur le modèle engendré, est modifiée de deux manières. D'abord, le profil `P` dans la portée du quantificateur, est étendu en utilisant le nom `var` comme paramètre des prédicats référencés ; on appelle cette formule étendue `P_var`. Ensuite, la formule étendue est intégrée au contexte contrôlant la séquence d'états sur

laquelle `P_var` est évaluée. Dans le cas particulier de LTL, la formule étendue est de la forme :

```
(!var_selected U (var_selected && P_var)) || []!var_selected
```

La vérification de la formule modifiée sur le modèle modifié exploite l'aspect exhaustif des vérificateurs de modèles. Pour chaque trace du modèle non modifié qui serait générée par Bandera, le modèle modifié crée une trace dans laquelle chaque instance de QuantifiedClass est liée à var. À l'état où la liaison est établie, la formule temporelle modifiée provoque la vérification de `P_var`.

Il est à noter que quand on utilise une quantification imbriquée, la condition `var_selected` est définie de telle façon qu'elle est vraie seulement quand toutes les variables liées dans les quantificateurs on reçu une valeur différente de `null`.

Aspect implémentation. Il y a plusieurs avantages à implémenter la prise en compte de la quantification tôt dans le processus d'extraction des modèles à états finis à partir de programmes Java : indépendance vis à vis du vérificateur et optimisation.

La technique décrite ci-dessus est appliquée à la représentation interne du code Java avant la génération du modèle d'entrée du vérificateur, par exemple le code Promela pour SPIN. Les spécifications temporelles quantifiées peuvent donc être vérifiées à l'aide d'un quelconque des vérificateurs pris en compte. De plus, puisqu'il est possible de générer du Java à partir de notre représentation interne, on peut utiliser un vérificateur tel que Java Pathfinder 2 du Laboratoire Ames de la NASA pour vérifier des spécifications temporelles quantifiées, pour peu qu'il fasse correspondre les appels à la méthode `Bandera.choose` à un choix non déterministe dans le modèle sous-jacent.

La portée, dans les profils de spécification, définit les points limite des régions dans lesquelles le profil doit être vérifié. Si ces points n'apparaissent pas, alors la spécification est trivialement vraie. Pour cette raison, en effectuant des analyses de flot d'objets [8], on peut déterminer les ensembles d'objets influant potentiellement sur la satisfaction des prédicats déterminant la portée. Cette information peut alors être utilisée pour restreindre l'ensemble d'objets sur lequel le quantificateur doit s'appliquer. Considérons la spécification :

```
forall[s:SuperType].
  {Pred(s)} is absent after {init.Return(s)}
```

Dans ce cas, on n'a besoin de calculer que l'ensemble d'instances de `SuperType` pour lesquelles la méthode `init` est appelée. Une approximation par excès de cet ensemble peut être calculée en termes de sous-types de `SuperType` pouvant apparaître comme objet récepteur d'une invocation d'`init`. Le code résultant de l'étape 3 ci-dessus ne doit être rajouté qu'à ces sous-types ; ceci peut réduire de façon significative le coût de la vérification de la propriété quantifiée en réduisant le nombre de valeurs qui sont affectées à la variable liée `s`.

Chapitre 4

Réduction de modèle par utilisation du slicing

4.1 Introduction

Le *slicing* (ou découpage) des programmes constitue une technique de réduction des programmes qui a déjà été abondamment utilisée et appliquée [Tip95] dans le domaine du génie logiciel, de l'analyses statique ou des applications de débogage. L'objectif poursuivi dans le cadre de ce sous-projet est de construire des modèles d'états finis associés à un programme logiciel Java. On s'appuie pour cela sur un noyau d'outils proposés dans la suite Bandera [CDH+00]. Les outils de model checking existants peuvent alors effectuer la vérification de ces modèles au regard de spécifications exprimées dans des logiques temporelles. Les techniques de slicing, en permettant d'éliminer les composantes du programme qui s'avèrent irrelevantes au regard de la propriété à vérifier, permettent ainsi de réduire la taille du modèle produit et de diminuer le temps requis par la vérification.

Le langage des programmes analysés est le langage Java. Le langage interprété par la machine virutelle Java est un langage de byte code, de type SSA. Les outils Bandera offrent un traducteur de ce langage en un langage *Jimple* qui est essentiellement un langage d'expression du graphe de flot de contrôle dans lequel les instructions apparaissent en forme SSA, où les manipulations explicites de pile effectuées dans le byte code ont été éliminées en introduisant des variables temporaires. Par ailleurs, les constructions Java telles que les invocations de méthodes et les instructions de synchronisation sont représentées par leur équivalent dans la machine virtuelle (`virtualinvoke`, `monitorenter` ou `monitorexit` par exemple).

Les opérations de slicing que ce travail a cherché à étendre et corriger dans les outils de Bandera ont été réalisées sur les texte *Jimple*. On s'appuie dans ce rapport sur la définition d'un langage Jimple simplifié qui capture l'essence des graphes de contrôle de *Jimple* et propose des instructions proches de celles de la machine virtuelle Java pour la synchronisation des threads telles que `wait`, `notify` ou `notifyall`.

Ce rapport présente l'ensemble des notions qui concourrent à établir quels sont les différents types de dépendances qui sont pris en compte pour construire un programme résiduel. En se fondant sur un critère de slicing donné et en construisant l'ensemble des dépendances qui affectent les éléments constitutifs du critère, il est possible de construire le programme résiduel en ne conservant du programme initial que les composantss dont dépendent les éléments du

critère de slicing.

Les programmes de découpage opérent en se fondant sur l'analyse du flot de contrôle d'un programme original. On explicite donc les techniques utilisées pour décrire et construire une représentation de ce flot de contrôle. On définit ensuite l'ensemble des dépendances auxquelles peuvent être soumis les éléments qui composent un critère de découpage. On montre enfin comment, en tenant compte de ces dépendances, on produit un programme résiduel.

4.2 Flot de contrôle pour un langage

4.2.1 Présentation d'un langage simple et concurrent : LSC

On se donne ici la définition d'un langage simple et concurrent (LSC). Les domaines syntaxiques du langage sont donnés par le tableau suivant :

p	$\in$	Programs	x	$\in$	Variables
d	$\in$	Threads	k	$\in$	Locks
b	$\in$	Blocks	t	$\in$	Thread-Indentifiers
l	$\in$	Block-Labels	e	$\in$	Expressions
a	$\in$	Assignments	c	$\in$	Constants
s	$\in$	Syncs	j	$\in$	Jumps
o	$\in$	Operationss			

Alors la grammaire du langage peut être décrite par :

$$
\begin{array}{lcl}
p & ::= & (x^*)(k^*)(d^+) \\
d & ::= & \texttt{begin-thread}\ t\ (l)\ b^+\ \texttt{end-thread;} \\
b & ::= & l\colon a^*\ j \mid l\colon s\ \texttt{goto}\ l; \\
a & ::= & x := e; \\
s & ::= & \texttt{enter-monitor}\ k; \mid \texttt{exit-monitor}\ k; \mid \texttt{wait}\ k; \mid \texttt{notify}\ k; \mid \texttt{notifyall}\ k; \\
e & ::= & c \mid x \mid o(e^*) \\
j & ::= & \texttt{goto}\ l; \mid \texttt{return} \mid \texttt{if}\ e\ \texttt{then}\ l_1\ \texttt{else}\ l_2;
\end{array}
$$

Un programme commence aisni par une déclaration de variable x^* ; chaque variable est implicitement typée comme un entier naturel. Pour représenter les verrous implicitement associés aux objets Java dans les accès synchronisés, on déclare explicitement une liste d'identifieurs de verrous k^* qui peuvent être utilisés dans les primitives de synchronisation. Le corps d'un programme est une série de déclarations d^* de un ou plusieurs threads. Une déclaration de thread comporte un identifieur de thread t et l'étiquette l du bloc initial du thread, suivie d'une liste b^+ de blocs de base. Il y a deux types de blocs de base :

- les blocs comportant une liste d'affectations suivie d'un saut ;
- les blocs de synchronisation contenant une instruction unique de synchronisation et une instruction de saut inconditionnel.

Les instructions d'affectation opèrent sur des variables partagées par tous les threads. Dans les expressions conditionnelles, toute valeur non nulle représente une valeur booléenne *true* et une valeur nulle représente la valeur booléenne *false*. Chaque construction de synchronisation agit en référence à un verrou particulier k. On impose les conditions suivantes :

- les identifieurs de threads et les étiquettes de bloc sont uniques ;

– toute étiquette l parraissant comme cible d'une instruction de saut dans un thread t doit être déclarée dans t (il n'y a pas de saut inter-thread) ;

– les verrous apparaissant dans les instructions de synchronisation doivent appraitre dans les déclarations de verrou en tête d'un programme.

Dans la suite, on va s'attacher à raisonner sur les noeuds d'un graphe de flot de contrôle de niveau instruction. Il s'agit d'un graphe qui, pour chaque thread t, associe un noeud n à chaque instruction d'affectation, de synchronisation ou de saut.

4.2.2 Exemple de programme

On donne ici un exemple de texte LSC qui dénote un programme de producteur et consommateur avec une taille de buffer de 2.

```
(compteur total prod cons) /* variables partagees */
(buffer compte) /*identifieurs de verrous */
```

```
/*code du producteur */                    /* code du consommateur */

begin-thread producteur                    begin-thread consommateur
(prod_obt_verrou)                          (cons_obt_verrour)
prod_obt_verrou :                          cons_obt_verrou :
enter-montor buffer;          [1]          enter-monitor buffer ;        [1]
goto prod_verif ;             [2]          goto cons_verif;              [2]

prod_verif:                                cons_verif:
if =(compteur 2)              [1]          if =(compteur 0)              [1]
then prod_wait                             then cons_wait
else prod_put;                             else cons_get;

prod_wait :                                cons_wait:
wait buffer ;                 [1]          wait buffer                   [1]
goto prod_verif;              [2]          goto cons_verif;              [2]

prod_put:                                  cons_get
compteur := +(compteur 1);    [1]          compteur := -(compteur 1);    [1]
if =(compteur 1)              [2]          if =(compteur 1)              [2]
then prod_reveil                           then cons_reveil
else prod_relache_verrou;                  else cons_relache_verrou;

prod_reveil:                               cons_reveil:
notify buffer;                [1]          notify buffer;                [1]
goto prod_relache_verrou;     [2]          goto cons_relache_verrou;     [2]

prod_relache_verrou :                      cons_relache_verrou:
exit-monitor buffer;          [1]          exit-monitor buffer           [1]
goto prod_entrer_compte;      [2]          goto cons_entrer_compte;      [2]

prod_entrer_compte:                        cons_entrer_compte:
enter-monitor compte;         [1]          enter-monitor compte;         [1]
goto prod_maj_compte;         [2]          goto cons_maj_compte;         [2]

prod_maj_compte:                           cons_maj_compte:
total := +(total 1);          [1]          total := +(total 1);          [1]
goto prod_sortir_compte;      [2]          goto cons_sortir_compte;      [2]

prod_sortir_compte:                        cons_sortir_compte:
exit-monitor compte;          [1]          exit-monitor compte;          [1]
goto prod_boucle;             [2]          goto cons_boucle;             [2]

prod_boucle:                               cons_boucle:
prod := +(prod 1)             [1]          cons := +(cons 1);            [1]
if 1                          [2]          if 1                          [2]
then prod_obt_verrou                       then cons_obt_verrou
else sortie_prod;                          else sortie_cons;

sortie_prod: return;          [1]          sortie_cons: return;          [1]
end-thread;                                end-thread;
```

4.2.3 Graphe de flot de contrôle

On suppose que chaque instruction est indicée par un indice i dans chaque bloc et que chaque étiquette de bloc est unique dans tout le programme. Aussi un noeud n (une instruction) est complètement identifié par le couple $[l.i]$ où l est l'étiquette du bloc et i l'indice de l'instruction dans le bloc désigné. Dans le programme LSC présenté en 4.2.2, les indices sont notés entre crochets `[.]`. Par exemple la première instruction d'affectation du bloc `cons_get` a comme identifieur unique et numéro de noeud : [`cons_get`.1]. On introduit alors les définitions suivantes.

Définition 1 *On se donne les définitions suivantes :*

- *Un* graphe de flot de contrôle *G=(N,E,s,e) est constitué d'un ensemble N de noeuds d'instructions, un ensemble E d'arcs de flot de contrôle dirigés, un noeud racine unique s et un noeud terminal e également unique. Tous les noeuds de N sont atteignables depuis s et e est atteignable depuis tout noeud de N.*
- *Le noeud n* domine *le noeud m dans G (on écrit dom(n,m)) si tout chemin du noeud racine s vers m passe à travers n.*
- *Le noeud n* postdomine *le noeud m dans G (on écrit post-dom(n,m)) si tout chemin du noeud m vers le noeud terminal e passe à travers n.*
- *Un* arc de retour *du flot de contrôle est un arc dont la cible domine la source dans G.*
- *Etant donné un arc de retour $m \rightarrow n$ dans G, la* boucle *naturelle de $m \rightarrow n$ est le sous-graphe composé de l'ensemble des noeuds qui contient n et tous les noeuds à partir desquels m peut être atteint dans le graphe sans passer par n et de l'ensemble des arcs reliant tous les noeuds du sous-graphe. Le noeuds n est le noeud d'entrée de la boucle.*
- *Un graphe de flot de contrôle $G = (N,E,s,e)$ est* réductible *(ou* bien structuré*) si E peut être partitionné en deux ensembles d'arcs disjoints E_f (ensemble des arcs* avant*) et E_b (ensemble des arcs* arrière*) tels que (N,E_f) forme un graphe dirigé sans cycle dans lequel chaque noeud peut être atteint depuis le noeud racine et tels que les arcs de E_b sont des arcs de retour dans G. Ainsi, dans un graphe de flot réductible, les boucles sont des boucles naturelles caractérisées par leur arc de retour. Il n'y a pas de sauts au milieu des boucles : on ne peut pénétrer dans une boucle que par le noeud d'entrée de la boucle.*

On impose aussi un certain nombre de contraintes sur la structure du flot de contrôle du langage simplifié défini de manière à calquer des contraintes ou des propriétés héritées des codes sources Java.

- Puisque Java autorise des constructions bien structurées uniquement (sans instructions `goto`), la structure du contrôle de chaque thread, dans le langage simplifié, devra former un graphe de contrôle réductible. On maintient cette restriction même si Java n'impose pas un point de sortie unique du contrôle dans ses programmes.
- Comme les constructions `enter-monitor` k et `exit-monitor` k sont héritées des blocs Java `synchronized`, on suppose que ces constructions sont bien parenthésées. De plus elles constituent le point d'entrée unique et le point de sortie unique d'un sous-graphe de flot du graphe de flot du thread contenant. Ainsi, à partir de chaque délimiteur moniteur on peut extraire un graphe de flot de contrôle correspondant à la section critique ainsi délimitée. On peut alors définir la fontion CR qui associe tout noeud n à la région critique la plus interne au sein de laquelle ce noeud est placé. Si $CR(n) = (m_1,m_2)$ alors

m_1 est une instruction `enter-monitor` k dont le `exit-monitor` k correspondant est m_2 ; ces deux noeuds délimitent la région critique la plus interne dans laquelle n apparaît.

- Les instructions conditionnelles peuvent être héritées d'instructions conditionnelles pures de Java ou peuvent traduire une condition de sortie de sortie de boucle Java (`while` ou `for`). Il est impossible de savoir si une boucle Java termine ; aussi les instructions conditionnelles de fin de boucle constitueront ce que l'on appellera des *point de pre-divergence.*

Certains threads peuvent ne pas satisfaire la contrainte de point de sortie unique du flot de contrôle telle que l'exige la définition du graphe de flot de contrôle. Le thread peut ne pas avoir d'instruction `return` ou peut, au contraire, en avoir plusieurs.

Le premier cas peut se produire si le thread contient une ou plusieurs boucles sans conditionnelle de sortie, ce qui garantit des boucles infinies. Puisque tous les graphes de flot doivent être réductibles, de telles boucles sont naturelles et caractérisées uniquement par leur arc de retour. Le noeud source de l'arc de retour correspond à une instruction `goto` l que l'on remplace alors par une instruction `if 1 then` l `else return`. Cela ne change pas la sémantique du programme, mais permet d'introduire le noeud de sortie de flot du thread.

Dans le second cas (plusieurs noeuds `return`), lorsque l'on construit le graphe de flot de contrôle associé au thread t, on insère un noeud additionnel étiquetté `halt` sans successeur et dont les prédécesseurs sont les noeuds associés aux instructions `return` dans t.

Pour accéder au code à des points particuliers du programme dans un programme p, on utilise aussi les fonctions suivantes.

- Une fonction *code* associe à un noeud n le code de l'instruction que ce noeud étiquette. Par exemple, dans le programme présenté en 4.2.2, *code*([`cons_get`.1]) retourne l'instruction `compteur ::=-(compteur 1);`.
- La fonction *first* associe une étiquette l de p le premier noeud du graphe de flot de contrôle du bloc étiquetté l. Par exemple *first*(`cons_get`) = [`cons_get`.1].
- La fonction *def* associe à chaque noeud l'ensemble des variables définies dans ce noeud (ensemble des variables qui se voient assignées une valeur) ; il s'agit toujours d'un singleton ou de l'ensemble vide.
- La fonction *ref* associe à chaque noeud l'ensemble des variables référencées dans ce noeud.
- La fonction θ fait correspondre à un noeud n l'identifieur du thread auquel n appartient. Par exemple θ([`prod_reveil`.1]) = `producteur`.

La synchronisation, dans les programmes Java est assurée par utilisation de primitives `wait` et `notify` sur des moniteurs [LY97]. Le programme présenté en 4.2.2 montre l'usage de tels moniteurs pour assurer la synchronisation entre un producteur et un consommateur. Le code présenté ne contrôle que l'accès au buffer sans réellement décrire l'usage fait de ce buffer. Le verrou qui, implicitement en Java, est associé à l'objet buffer est le verrou `buffer` du programme.

Le processus (thread) producteur du programme commence par tenter d'acquérir le verrou `buffer` dans le bloc `prod_obt_verrou`. Jusqu'à ce qu'il obtienne ce verrou (et puisse entrer dans la section critique protégée par le moniteur), il est inséré dans l'ensemble des processus bloqués derrière ce verrou. Une fois le verrou acquis et posé, le programme vérifie l'état du buffer et s'assure que celui-ci n'est pas déja plein (auquel cas le `compteur` des éléments qu'il contient vaut 2). Si le buffer est plein, alors le producteur libère le verrou et se met en attente

derrière lui (instruction `wait`). Sinon, il incrémente le `compteur` d'objets placés dans le buffer. Si, ce faisant, le buffer sort de l'état vide, alors le producteur notifie le consommateur qui peut être en attente du producteur après avoir constaté que le buffer était vide. Lorsqu'un processus effectue un `notify` k, il conserve encore le verrou, mais les processus placés en attente sont alors déplacés vers l'ensemble des processus bloqués derrière le verrou. Immédiatement après avoir notifié les processus en attente, le producteur libère le verrou en exécutant l'instruction `exit-monitor`. La libération de ce verrou peut permettre au consommateur d'obtenir ce verrou et d'achever son instruction `enter-monitor buffer`. Le proucteur tente ensuite d'obtenir un autre verrou (`compte`) pour exécuter une autre section critique : la mise à jour de la variable partagée `total`. Lorsque le verrou est acquis, le producteur incrémente la variable et libère le verrou. Il incrémente ensuite son propre compte d'actions de production opérées et recommence un nouveau cycle de production. Le comportement du consommateur est symétrique.

On peut donner une vision plus formelle de ces comportements en notant que le programme p dénote des transitions sur des configurations de la forme $(pc,\sigma,\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L})$. Les définitions de ces éléments sont données par :

v	$\in$ Valeurs	=	{0,1,2,...}
l	$\in$ Labels	=	Block-Labels $\cup$ {`halt`}
σ	$\in$ Memoires	=	Variables $\rightharpoonup$ Valeurs
m	$\in$ Lock-Counts	=	{0,1,2,...}
pc	$\in$ PCs	=	Thread-identifiers $\rightharpoonup$ Noeuds
$\mathcal{R}$	$\in$ Runnable	=	Thread-identifiers $\rightharpoonup$ $\{true,false\}$
$\mathcal{B}$	$\in$ Blocked-Sets	=	Locks $\times$ Thread-identifiers $\rightharpoonup$ Locks-Counts
$\mathcal{W}$	$\in$ Wait-Sets	=	Locks $\times$ Thread-identifiers $\rightharpoonup$ Locks-Counts
$\mathcal{L}$	$\in$ Locks-Sets	=	Locks $\rightharpoonup$(Thread-identifiers $\times$ {1,2,3,...} $\cup$ $\{false\}$)

Le compteur ordinal (ou compteur de programme) pc assigne à chaque thread t le noeud de t qui doit être exécuté. La mémoire σ donne pour chaque variable sa valeur. La table d'exécutabilité $\mathcal{R}$ associe à tout thread t la valeur $true$ (indiquant que t est dans un état exécutable et que l'instruction pointée par pc peut être immédiatement exécutée) ou la valeur $false$ (indiquant que le thread est bloqué parce qu'il est en cours d'acquisition d'un verrou, ou parce qu'il s'est mis en attente par un `wait`, ou parce qu'il a terminé son exécution après un `return`). La table des processus bloqués $\mathcal{B}$ associe à chaque couple $\langle k,t\rangle$ le nombre de fois que le thread t aura besoin d'acquérir le verrou k une fois débloqué. Chaque thread peut acquérir un verrou un nombre multiple de fois. $\mathcal{B}\langle k,t\rangle = 0$ indique que t n'est pas bloqué sur k. La table des processus en attente $\mathcal{W}$ associe à chaque couple $\langle k,t\rangle$ le nombre de fois que le thread t aura besoin d'acquérir le verrou k une fois réveillé. $\mathcal{W}\langle k,t\rangle = 0$ indique que t n'est pas en attente du verrou k. La table de verrous $\mathcal{L}$ associe à chaque verrou k un couple $\langle t,n\rangle$ où t est le thread qui a le verrou k et n est le nombre de fois que t a acquis le verrou (n est la différence entre le nombre des précédentes instructions `enter-monitor` k `exit-monitor` k qu'il a effectuées). Si le verrou k n'est posé par personne, alors $\mathcal{L}(k) = false$.

On se donne une configuration d'exception `bad-monitor` qui correspond à l'exception levée par la JVM en cas d'erreur sur l'état des moniteurs.

L'exécution d'un programme démarre avec une configuration initiale $(pc,\sigma,\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L})$ dans laquelle on a :

- $\forall t\ pc(t) = n_{init}$ où n_{init} est le noeud d'entrée du graphe de flot de contrôle du thread t ;
- $\forall x\ \sigma(x) = 0$,
- $\forall t\ \mathcal{R}(t) = true$,

- $\forall k,t\ \mathcal{B}\langle k,t\rangle = 0$,
- $\forall k,t\ \mathcal{W}\langle k,t\rangle = 0$,
- $\forall k\ \ mathcalL(k) = false$.

Une transition se produit lorsqu'un thread au moins est exécutable comme l'indique $\mathcal{R}$. Un thread exécutable t est alors arbitrairement choisi et le compteur de programme pc ainsi que la relation *code* sont consultés pour déterminer la prochaine instruction à évaluer. Il existe une règle pour décrire la transition opérée par chaque classe de commande. Chaque règle se fonde sur un jugement qui définit la sémantique associée à cette classe de commande.

Soit la règle

$$\frac{\sigma \vdash^t_{assign} a \Rightarrow \sigma'}{\langle pc,\sigma,\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L}\rangle \longmapsto \langle pc[t \mapsto n'],\sigma',\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L}\rangle}$$

si $\mathcal{R} = true$ et $pc(t) = n$ et $code(n) = a$ et $n' = succ(n)$. Alors $\sigma \vdash^t_{assign} a \Rightarrow \sigma'$ indique que dans l'état mémoire σ l'instruction d'affectation a exécutée par le thread t conduit à l'état mémoire σ'. Et si tel est le cas, alors la configuration est modifiée de la façon suivante : le compteur d'instruction du thread t passe à l'instruction n' suivante et l'état mémoire est celui consécutif à l'exécution de l'affectation a.

De même

$$\frac{\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L} \vdash^t_{sync} s \Rightarrow \mathcal{R}',\mathcal{B}',\mathcal{W}',\mathcal{L}'}{\langle pc,\sigma,\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L}\rangle \longmapsto \langle pc[t \mapsto n'],\sigma,\mathcal{R}',\mathcal{B}',\mathcal{W}',\mathcal{L}'\rangle}$$

si $\mathcal{R} = true$ et $pc(t) = n$ et $code(n) = s$ et $n' = succ(n)$. Alors dans ce cas, $\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L} \vdash^t_{sync} \mathcal{R}',\mathcal{B}',\mathcal{W}',\mathcal{L}'$ signifie que l'instruction s de synchronisation peut modifier l'état des ensembles $\mathcal{R},\mathcal{B},\mathcal{W}$ et $\mathcal{L}$. Si tel est le cas, après exécution de s la configuration résultante est celle constituée par les ensembles modifiés et le compteur de programme incrémenté à l'instruction suivante. On peut aussi avoir, sur une instruction de synchronisation mettant en oeuvre un moniteur une exception, ce que l'on traduit par :

$$\frac{\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L} \vdash^t_{sync} s \Rightarrow \text{bad-monitor}}{\langle pc,\sigma,\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L}\rangle \longmapsto \text{bad-monitor}}$$

avec $\mathcal{R} = true$ et $pc(t) = n$ et $code(n) = s$. Concernant les sauts on peut établir la règle de transition suivante entre configurations :

$$\frac{\sigma \vdash^t_{jump} j \Rightarrow l}{\langle pc,\sigma,\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L}\rangle \longmapsto \langle pc[t \mapsto n'],\sigma',\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L}\rangle}$$

si $\mathcal{R} = true$ et $pc(t) = n$ et $code(n) = j$ et $n' = first(l)$. $\sigma \vdash^t_{jump} j \Rightarrow l$ indique ici que avec la mémoire σ, l'instruction de saut j conduit au bloc étiquetté par l. Mais on peut également avoir un arrêt :

$$\frac{\sigma \vdash^t_{jump} j \Rightarrow \text{halt}}{\langle pc,\sigma,\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L}\rangle \longmapsto \langle pc[t \mapsto n'],\sigma',\mathcal{R}[t \mapsto false],\mathcal{B},\mathcal{W},\mathcal{L}\rangle}$$

si $\mathcal{R} = true$ et $pc(t) = n$ et $code(n) = j$ avec $n' = \text{halt}$.

Du point de vue des instructions de synchronisation, on dispose d'autres règles qui indiquent comment sont modifiés les relations $\mathcal{R},\mathcal{B},\mathcal{W}$ et $\mathcal{L}$.

S'agissant des instructions `enter-monitor` k, si k n'est pas déjà verrouillé, la table des verrous pour k est modifiée pour montrer que le thread t a acquis le verrou une première fois.

Si le verrou est déjà posé par t n fois, alors le compteur d'acquisition du verrou est incrémenté de 1. Si un autre thread t' détient le verrou, alors t est placé dans l'ensemble des processus bloqués, et son état n'est plus exécutable.

$$\frac{\mathcal{L}(k) = false}{\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L} \vdash^t_{sync} \text{enter-monitor } k \Rightarrow \mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L}[k \mapsto \langle t,1\rangle]}$$

$$\frac{\mathcal{L}(k) = \langle t,n\rangle}{\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L} \vdash^t_{sync} \text{enter-monitor } k \Rightarrow \mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L}[k \mapsto \langle t,n+1\rangle]}$$

$$\frac{\mathcal{L}(k) = \langle t',n\rangle \text{ avec } t \neq t'}{\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L} \vdash^t_{sync} \text{enter-monitor } k \Rightarrow \mathcal{R}[t \mapsto false],\mathcal{B}[\langle k,t\rangle \mapsto 1],\mathcal{W},\mathcal{L}}$$

Pour les instructions `exit-monitor` k, si le compteur de verrou de t est plus grand que 1, ce compteur est simplement décrémenté sans que le verrou soit libéré. Lorsque ce compteur vaut 1, alors le verrou est libéré. Si l'ensemble des threads bloqués derrière le verrou n'est pas vide, un autre thread t' est choisi arbitrairement dans cet ensemble et se voit attribuer le verrou.

$$\frac{\mathcal{L}(k) = \langle t,n\rangle \text{ avec } n > 1}{\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L} \vdash^t_{sync} \text{exit-monitor } k \Rightarrow \mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L}[k \mapsto \langle t,n-1\rangle]}$$

$$\frac{\mathcal{L}(k) = \langle t,1\rangle}{\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L} \vdash^t_{sync} \text{exit-monitor } k \Rightarrow \mathcal{R}',\mathcal{B}',\mathcal{W},\mathcal{L}'}$$

$$\text{avec } \mathcal{R}',\mathcal{B}',\mathcal{L}' = \left\{ \begin{array}{l} \mathcal{R},\mathcal{B},\mathcal{L}[k \mapsto false] \\ \mathcal{R}[t' \mapsto true], \\ \mathcal{B}[\langle k,t'\rangle \mapsto 0], \\ \mathcal{L}[k \mapsto \langle t',n\rangle] \end{array} \right\} \begin{array}{l} \text{si } \forall t\ \mathcal{B}\langle k,t\rangle = 0 \\ \\ \text{si } \exists t'\ \mathcal{B}\langle k,t'\rangle = n > 0 \end{array}$$

Pour les instructions `wait` k, si le thread t possède le verrou alors il libère ce dernier et t est placé dans l'ensemble des threads en attente derrière le verrou. Un autre thread t' est choisi dans l'ensemble des threads bloqués derrière le verrou k et se voit attribuer le verrou en devenant ainsi exécutable.

$$\frac{\mathcal{L}(k) = \langle t,n\rangle}{\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L} \vdash^t_{sync} \text{wait } k \Rightarrow \mathcal{R}',\mathcal{B}',\mathcal{W}[\langle k,t\rangle \mapsto n],\mathcal{L}'}$$

$$\text{avec } \mathcal{R}',\mathcal{B}',\mathcal{L}' = \left\{ \begin{array}{l} \mathcal{R}[t \mapsto false],\mathcal{B},\mathcal{L}[k \mapsto false] \\ \mathcal{R}[t \mapsto false,t' \mapsto true], \\ \mathcal{B}[\langle k,t'\rangle \mapsto 0], \\ \mathcal{L}[k \mapsto \langle t',m\rangle] \end{array} \right\} \begin{array}{l} \text{si } \forall t\ \mathcal{B}\langle k,t\rangle = 0 \\ \\ \text{si } \exists t'\ \mathcal{B}\langle k,t'\rangle = m > 0 \end{array}$$

Pour les instructions `notify` k, si le thread t détient le verrou k et si l'ensemble des threads en attente derrière ce verrou est vide, il ne se passe rien. Si l'ensemble des threads en attente derrière le verrou n'est pas vide, un thread (avec son compteur de verrou) est extrait de cet ensemble et ajouté à l'ensemble des threads bloqués derrière ce verrou (concourant pour l'acquérir).

$$\frac{\mathcal{L}(k) = \langle t,n\rangle}{\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L} \vdash^t_{sync} \text{notify } k \Rightarrow \mathcal{R},\mathcal{B}',\mathcal{W}',\mathcal{L}}$$

avec $\mathcal{B}',\mathcal{W}' = \begin{cases} \mathcal{B},\mathcal{W} & \text{si } \forall t\ \mathcal{W}\langle k,t\rangle = 0 \\ \mathcal{B}[\langle k,t'\rangle \mapsto n],\ \mathcal{W}[\langle k,t'\rangle \mapsto 0] & \text{si } \exists t'\ \mathcal{W}\langle k,t'\rangle = n > 0 \end{cases}$

Concernant les instructions `notify-all` k, si le thread t détient le verrou k et si l'ensemble des threads en attente derrière ce verrou est vide, il ne se passe rien. Si l'ensemble des threads en attente derrière le verrou n'est pas vide, tous les threads de cet ensemble sont extraits cet ensemble et ajoutés à l'ensemble des threads bloqués derrière ce verrou (concourant pour l'acquérir).

$$\frac{\mathcal{L}(k) = \langle t,n\rangle}{\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L} \vdash^{t}_{sync} \text{notify-all } k \Rightarrow \mathcal{R},\mathcal{B}',\mathcal{W}',\mathcal{L}}$$

avec $\forall k',t'\ \mathcal{B}'\langle k',t'\rangle, \mathcal{W}'\langle k',t'\rangle = \begin{cases} \mathcal{B}'\langle k',t'\rangle + \mathcal{W}'\langle k',t'\rangle, 0 & \text{si } k = k' \\ \mathcal{B}'\langle k',t'\rangle, \mathcal{W}'\langle k',t'\rangle & \text{si } k \neq k' \end{cases}$

Enfin, lorsqu'un thread exécute une instruction de synchronisation sur le verrou k et qu'il ne le détient pas, alors une exception est levée, ce que l'on exprime par :

$$\frac{(\mathcal{L}(k) = \langle t',n\rangle \text{ avec } t \neq t') \vee (\mathcal{L}(k) = false)}{\mathcal{R},\mathcal{B},\mathcal{W},\mathcal{L} \vdash^{t}_{sync} \text{s} \Rightarrow \text{bad-monitor}}$$

avec $s \in \{$`exit-monitor` k, `wait` k, `notify` k, `notify-all` $k\}$.

Si l'on revient à la définition des transitions, la configuration de terminaison normale est celle où tous les threads ne sont plus dans l'état *exécutable* et où tous les compteurs de programmes pointent sur l'instruction `halt`. Toute transition $c_1 \longmapsto c_2$ exécute une et une seule instruction dans un noeud n. On note $c_1 \overset{n}{\longmapsto} c_2$ le fait que la transition $c_1 \longmapsto c_2$ soit réalisée par exécution du noeud n, $c_1 \overset{N}{\longmapsto} c_2$ le fait que la transition $c_1 \longmapsto c_2$ soit réalisée par exécution de l' instruction d'un noeud $n \in N$ et $c_1 \overset{\neg N}{\longmapsto} c_2$ le fait que la transition $c_1 \longmapsto c_2$ soit réalisée par exécution de l' instruction d'un noeud $n \notin N$.

4.3 Slicing de programmes

Les opérations de slicing sur un programme p ont pour objectif d'extraire de ce dernier les seules instructions qui sont susceptibles d'affecter les valeurs de variables constitutives de points d'intérêt [Tip95] de ce programme. Ces points d'intérêt sont décrits par les critères de slicing.

4.3.1 Critère de slicing

Définition 2 *Un critère de slicing C d'un programme p est un ensemble non vide de noeuds $\{n_1,...,n_k\}$ dans lequel chaque n_i est un noeud du graphe de flot de contrôle de p.*

Sachant que l'on se place dans le contexte de la vérification de programmes à l'aide des techniques de model checking, on dispose du programme p est d'une spécification exprimée en logique temporelle ϕ. Il y a ainsi des noeuds ou des instructions du programme dont l'action influent sur l'évaluation de ϕ et qui sont ainsi observables et d'autres qui ne le sont pas au regard de ϕ.

La définition qui suit décrit une situation où le programme peut effectuer zero ou plus instructions non observables, ensuite une instruction observable, suivie de zero ou plus instructions non observables. De plus, la définition oblige que les variables référencées dans le

noeud observable correpondent à des valeurs contenues dans une mémoire σ. Ceci permet de maintenir une correspondance entre la mémoire manipulée par p et le programme découpé p_s.

Définition 3 *Soit C un critère de slicing de p, et soit $n \in C$ et σ une mémoire telle que $domaine(\sigma) = ref(n)$. On définit la relation $c_1 \overset{n,\sigma}{\Longrightarrow}_C c_2$ qui tient si il existe des configurations c'_1,c'_2 telles que $c_1 \overset{\neg C}{\longmapsto}^* c'_1 \overset{n}{\longmapsto} c'_2 \overset{\neg C}{\longmapsto}^* c_2$ et pour tout $x \in ref(n)$, $\sigma(x) = \sigma'_1(x)$ où σ'_1 est la mémoire de la configuration c'_1.*

Cette notion permet de alors de définir comment deux programmes peuvent se simuler l'un l'autre au regard d'actions observables décrites dans le critère de slicing.

Définition 4 *Etant donnés deux programmes p_1 et p_2, soit C un critère de slicing pour p_1 et $C \subseteq N_2$ où $N2$ est l'ensemble des noeuds du graphe de flot de contrôle de p_2. Une relation binaire $S \subseteq Configurations[p_1] \times Configurations[p_2]$ est une C-bisimulation si, avec $(c_1,c_2) \in S$ alors :*

1. *pour tout $n \in C$ et toute mémoire σ, si $c_1 \overset{n,\sigma}{\Longrightarrow}_C c'_1$ dans p_1 alors il existe une configuration c'_2 de p_2 telle que $c_2 \overset{n,\sigma}{\Longrightarrow}_C c'_2$ dans p_2 et $(c'_1,c'_2) \in S$, et*
2. *pour tout $n \in C$ et toute mémoire σ, si $c_2 \overset{n,\sigma}{\Longrightarrow}_C c'_2$ dans p_2 alors il existe une configuration c'_1 de p_1 telle que $c_1 \overset{n,\sigma}{\Longrightarrow}_C c'_1$ dans p_1 et $(c'_1,c'_2) \in S$*

Deux configurations sont alors bisimilaires si elles sont reliées par une relation de C-bisimulation.

Définition 5 *Soit C un critère de slicing pour p_1 et p_2 et soient c_1 et c_2 deux configurations respectivement de p_1 et p_2. Alors c_1 et c_2 sont dites C-bisimilaires (et l'on note ce fait par $c_1 \approx c_2$) si il existe une C-bisimulation S et $(c_1,c_2) \in S$. En d'autres termes , $\approx_C = \bigcup\{S | S$ est une C-bisimulation$\}$.*

En se fondant sur ces définitions, alors on peut exprimer ce qu'est un découpage correct d'un programme. Schématiquement, un programme slicé p_s est un découpage correct d'un programme p si leurs configurations initiales sont C-bisimilaires.

Définition 6 *Soit C un critère de slicing pour le programme p. Le programme p_s est un découpage de p au regard de C si $c \approx_C c_s$ où c et c_s sont les configurations initiales de p et p_s respectivement.*

4.3.2 Dépendances dans les programmes threadés

Etant donné un critère de slicing $C = \{n_1, \ldots, n_k\}$, calculer un découpage du programme p nécessite de trouver les noeuds de p dont dépendent les instructions des noeuds n_i de C. Ces noeuds sont parfois désignés comme les noeuds d'intérêt. Les variables d'intérêt sont alors les variables du programme qui sont définies ou référencées aux noeuds d'intérêt. La construction de l'ensemble des noeuds d'intérêt s'opère souvent en deux phases. Une première phase consiste à construire un graphe de dépendance du programme qui capture différentes dépendances entre les noeuds du graphe de flot de contrôle de p. Dans une seconde passe, les noeuds dont dépendent les n_i sont trouvés en réalisant la fermeture transitive des dépendances concernant les n_i.

Plusieurs types de dépendances interviennent dans cette analyse. La dépendance par les données [Tip95] est reliée à la notion de définition atteignante : un noeud n dépend par les données de m si pour une variable v référencée à n une définition de v à m atteint n. Le noeud n dépend donc du noeud m car l'affectation réalisée à m peut déterminer la valeur calculée au noeud n.

Définition 7 *Un noeud n est* dépendant par les données *d'un noeud m (et on note $n \xrightarrow{dd} m$ si il existe une variable v telle que :*

1. *il existe un chemin p non trivial de m à n tel que pour tout noeud $m' \in p - \{m,n\}$, $v \notin def(m')$, et*
2. *$v \in def(m) \cap ref(n)$.*

Par exemple dans le programme proposé en section 4.2.2 on peut écrire que `[prod_put.2]` $\xrightarrow{dd}$ `[prod_put.1]`, `[prod_verif.1]` $\xrightarrow{dd}$ `[prod_put.1]`, `[prod_put.1]` $\xrightarrow{dd}$ `[prod_put.1]`, et `[prod_maj_compte.1]` $\xrightarrow{dd}$ `[prod_maj_compte.1]`,`[prod_boucle.1]` $\xrightarrow{dd}$ `[prod_boucle.1]`, et il en va de même avec le thread `consommateur`.

L'information de dépendance sur le contrôle permet d'identifier les expressions conditionnelles du programme qui peuvent affecter l'exécution d'un noeud du critère de slicing.

Définition 8 *Un noeud n est* dépendant par le contrôle *d'un noeud m (et on note $n \xrightarrow{cd} m$ si il existe une variable v telle que :*

1. *il existe un chemin p non trivial de m à n tel que tout noeud $m' \in p - \{m,n\}$ est post-dominé par n, et*
2. *m n'est pas post-dominé par n.*

Pour qu'un noeud n soit dépendant par le contrôle d'un noeud m, il faut que m ait au moins deux successeurs immédiats dans le graphe de flot de contrôle (m est donc une évaluation d'expression conditionnelle) et il doit y avoir deux chemins qui relient m avec le noeud de sortie tels que l'un contienne n et l'autre pas. Si on reprend le programme donné en 4.2.2 alors `[prod_reveil.1]` $\xrightarrow{cd}$ `[prod_put.2]`, mais pas `[prod_relache_verrou.1]` qui n'est pas post-dominé par `[prod_put.2]`.

Il est nécessaire de s'assurer que les programmes découpés préservent la sémantique de programmes qui ne terminent pas. Cela revient à dire que ces programmes doivent incorporer les points du programme initial pouvant causer la non terminaison de l'exécution et appartenant à des chemins menant à des noeuds d'intérêt.

Définition 9 *Un noeud n est* dépendant par divergence *d'un noeud m (et on note $n \xrightarrow{\Omega d} m$ si :*

1. *m est un point de pre-divergence, c'est à dire un point auquel sont décidées la sortie d'une boucle ou une nouvelle itération ;*
2. *il existe un chemin p non trivial de m à n tel que aucun noeud $m' \in p - \{m,n\}$ ne soit un point de pre-divergence.*
3. *m n'est pas post-dominé par n.*

Cette définition doit pouvoir permettre de ne pas conserver des boucles infinies qui sont pas susceptibles de retarder indéfiniment des noeuds d'intérêt. En reprenant le programme

donné en 4.2.2 alors les noeuds `[prod_verif.1]` et `[prod_reveil.2]` sont des noeuds de pre-divergence. Tous les noeuds du thread `producteur` sont atteignables à partir de ces deux noeuds, tous les noeuds du thread en sont donc dépendants par divergence.

Si l'on intègre les dépendances introduites par l'exécution concurrentes des threads d'un programme, on s'intéressera particulièrement à la manière dont les définitions des variables partagées par les différents threads atteignent les noeuds qui utilisent ces variables dans les threads considérés.

Définition 10 *Un noeud n est* dépendant par interférence *d'un noeud m (et on note $n \xrightarrow{id} m$) si $\theta(m) \neq \theta(n)$ et s'il existe une variable v telle que $v \in def(m)$ et $v \in ref(m)$.*

En reprenant toujours le même programme de la section 4.2.2 on peut dire que les références à la variables `compteur` dans les noeuds `[prod_verif.1]`, `[prod_put.1]` et `[prod_put.2]` sont toutes dépendantes par interférence de la définition qui est faite de cette variable au noeud `[cons_get.1]`.

Si une variable d'intérêt est définie au noeud n au sein d'une section critique, la pose du verrou sur cette section critique constitue également un objet d'intérêt et les noeuds comportant les commandes `enter-monitor` et `exit-monitor` correspondantes doivent être inclus dans le programme découpé. Omettre cette gestion du moniteur pourrait permettre d'introduire une interférence entre actions sur des variables partagées non présente en fait dans le programme original du fait de leur encadrement dans une section critique. Dans ce cas précis, on dit que n *dépend par synchronisation* des noeuds `enter-monitor` et `exit-monitor` qui le parenthèsent. Dans le cas d'imbrications de régions critiques, la dépendance sera établie par fermeture transitive sur cette relation.

Si dans le programme de la section 4.2.2 la variable `compteur` est une variable d'intérêt pour le critère de slicing C, alors les moniteurs `buffer` sont également déclarés d'intérêt pour éviter que des incrémentations et décrémentations concurrentes de cette variable `compteur` ne soient déclarées interférer l'une avec l'autre.

Définition 11 *Un noeud n est* dépendant par synchronisation *d'un noeud m (et on note $n \xrightarrow{sd} m$) si $CR(n) = (m_1,m_2)$ et $m \in (m_1,m_2)$.*

On cherche à capturer une autre forme de dépendance. Une instruction n dépend d'une autre instruction m si l'impossibilité pour m de se terminer (parce qu'elle n'est jamais atteinte ou parce que s'il s'agit d'un `wait` l'instruction `notify` correspondante n'est jamais exécutée) provoque un blocage du thread $\theta(n)$ de n avant que n soit atteinte ou terminée (provoquant un retard infini de n).

Définition 12 *Un noeud n est* dépendant sur disponibilité *d'un noeud m (et on note $n \xrightarrow{rd} m$) si :*

1. *$\theta(n) = \theta(m)$ et n est atteignable depuis m dans le graphe de flot de contrôle de $\theta(m)$ et $code(m) =$`enter-monitor k` ou*
2. *$\theta(n) \neq \theta(m)$ et $code(n) =$`enter-monitor k` et $code(m) =$`exit-monitor k` ou*
3. *$\theta(n) = \theta(m)$ et n est atteignable depuis m et $code(m) =$`wait k` ou*
4. *$\theta(n) \neq \theta(m)$ et $code(n) =$`wait k` et $code(m) \in$ {`notify k`, `notify-all k`}.*

Ainsi, informellement, n dépend de m lorsqu'ils sont dans le même thread si m est une instruction `enter-monitor k` qui est susceptible de ne pas terminer (parce que le verrou `k` n'est jamais libéré par exemple) alors que n est atteignable depuis m et peut donc être bloquée également. C'est le même schéma qui s'impose si m est une instruction `wait k` qui endort le thread sans qu'il ne soit jamais réveillé par une notification. La définition de la dépendance étend donc cette dernière pour prendre en compte ces défauts possibles d'interaction entre threads. Si n et m ne sont pas dans le même thread alors si n est une instruction `enter-monitor k` elle dépend de toute instruction `exit-monitor k` susceptible de bloquer le verrou `k` si elle n'est pas terminée. De même si n est une instruction `wait k` elle dépend de la terminaison d'instructions `notify k` dans un autre thread.

Si l'on se réfère toujours au programme de la section 4.2.2 on constate que tous les noeuds du thread `producteur` sont atteignables depuis l'instruction `wait` du noeud `prod_wait.1`. Donc tous les noeuds du thread en dépendent par disponibilité. Tous les noeuds du thread dépendent aussi de l'instruction `enter-monitor compte` du noeud `prod_entre_compte.1`. L'instruction `wait buffer` au noeud `prod_attendre.1` dépend de l'instruction `notify buffer` à `cons_reveil.1`.

On peut dès lors, étant donné un programme p, construire la relation de dépendance $\xrightarrow{d}$ relativement à p comme l'union des relations de dépendances définies précédamment (c'est à dire l'union de $\xrightarrow{dd}$, $\xrightarrow{cd}$, $\xrightarrow{\Omega d}$, $\xrightarrow{id}$, $\xrightarrow{sd}$ et $\xrightarrow{drd}$). Le *graphe de dépendances du programme p* est alors constitué des noeuds du graphe de flot de contrôle de p et des arcs formés à partir de la relation d.

Définition 13 (Ensemble de découpage) *Soit C un critère de slicing pour le programme p et P le graphe des dépendances du programme de p. Alors l'ensemble de découpage de p, noté S_C relativement à C est défini par $S_C = \{m \mid n \in C \text{ and } n \xrightarrow{d}{}^{*} m\}$.*

4.3.3 Calcul d'un programme résiduel

Le programme résiduel doit comporter les noeuds de l'ensemble S_C. Mais il doit également comporter d'autres noeuds (tels que des noeuds d'instructions `goto` ou des commandes de synchronisation `enter-monitor`/`exit-monitor` équilibrées) pour être bien formé. Etant donné C et S_C on s'intéresse à la manière de construire le programme résiduel, le programme découpé.

De manière intuitive, si une instruction d'affectation est dans S_C, alors elle doit aussi être dans le programme résiduel. De même si une commande `enter-monitor` ou son instruction `exit-monitor` correspondante est dans S_C, elle doit aussi être dans le programme résiduel. Tous les sauts engendrés par des instructions `goto` ou `return` doivent également apparaître dans le programme résiduel. Il faut noter toutefois que si une instruction `if` n'est pas dans S_C, aucun noeud de S_C n'est dépendant par contrôle de ce `if`. Ainsi, peut importe que l'exécution suive l'une ou l'autre branche de ce `if`. On peut susbstituer à sa condition un saut vers le point de convergence des deux branches.

Définition 14 (Programme résiduel) *Etant donné un programme p, un critère de découpage C et un ensemble de découpage résultant S_C, le programme résiduel p_s peut être construit ainsi :*

1. *Pour chaque variable x de p, x est une variable de p_s ssi $x \in ref(n) \cup def(n)$ pour un $n \in S_C$;*

2. *Pour chaque verrou k de p, k est un verrou dans p_s ssi il existe un noeud $n \in S_C$ et si $code(n)$ est une commande de synchronisation sur k.*
3. *Pour chaque thread t de p, on trouve un thread correspondant apparaissant dans p_s ssi il existe un noeud $n \in S_C$ et $\theta(n) = t$.*
4. *Pour chaque bloc d'affectations `l : a* j` de p, on forme un bloc résiduel b_s avec l'étiquette `l` dans p_s comme suit :*
   (a) *Pour chaque ligne d'affectation a dont l'identifieur est `[l.i]` si `[l.i]`$\in S_C$ alors l'affectation a apparait dans le bloc résiduel avec l'identifieur `[l.i]`, autrement le noeud n'est pas reproduit dans le programme résiduel.*
   (b) *Pour le saut j, si $j =$`goto l'` ou si $j =$`return` alors j est un saut dans le bloc résiduel. Sinon le saut en question est de la forme $j =$ `if` e `then` l_1 `else` l_2 avec un identifieur `[l.i]`. Si `[l.i]`$\in S_C$ alors j est un saut dans le bloc résiduel b_s, sinon le saut introduit dans b_s est `goto` l' avec l'identifieur `[l.i]` et l'étiquette l' est l'étiquette du plus proche bloc post-dominant à la fois l_1 et l_2.*
5. *Pour chaque bloc de synchronisation $l : s$ `goto` l' de p, on forme un bloc résiduel d'étiquette l dans p_s tel que :*
   (a) *Si s a l'identifieur de noeud `[l.i]` et `[l.i]`$\in S_C$, ou si s est un `enter-monitor` ou un `exit-monitor` et que son correspondant est dans S_C, alors le bloc résiduel est identique au bloc source. Sinon le bloc résiduel se réduit à l : `goto` l', un bloc d'affectation sans affectation.*

Le processus ainsi décrit peut conduire à produire un programme résiduel p_s qui contient des blocs triviaux, ne comportant aucune instruction d'affectation mais simplement une instruction de saut `goto`. Certains de ces blocs peuvent s'avérer inatteignables du fait de la transformation d'instructions `if` en instructions de saut `goto`.

Il est possible et souhaitable d'éliminer ces blocs triviaux dans une phase de traitement particulier. Tous les blocs inatteignables peuvent être éliminés directement : ils ne comportent pas de noeuds de S_C. Les autres blocs (atteignables) qui ne comportent pas de noeuds de S_C sont marqués comme *à éliminer*. Parmi ces blocs ainsi marqués, on prend tous ceux qui ne sont pas partagés (c'est à dire qui ne sont pas la cible de 2 instructions de saut ou plus) et on les synthétise avec leur prédécesseur ou successeur immédiat.

Théorème 1 (Correction) *Soit C un critère de découpage du programme p. Soit p_s le programme résiduel construit relativement à p et C trel que décrit précédemment. Alors le programme p_s est un découpage de p et les configurations initiales de p et p_s sont C-bisimilaires.*

4.3.4 Optimisations

Les dépendances présentées en 4.3.2 permettent de construire un programme résiduel correct. Pour un critère de découpage donné on peut construire bon nombre de programmes résiduels corrects. A la limite, le programme source original peut être considéré comme un découpage correct. On cherche cependant à produire des découpages corrects aussi réduits que possible. Les dépendances sur la disponibilité ont en général pour effet d'introduire dans le programme résiduel pratiquement toute la structure de synchronisation. Parce qu'il est très difficile d'établir, statiquement, si les conditions permettant à un thread mis en attente sur une instruction `wait` d'être notifié par une instruction `notify` se produiront lors de l'exécution.

Dès lors le découpage produit peut être volumineux. On peut appliquer des analyses statiques sur le graphe de flot de contrôle pour affiner les dépendances sur disponibilité et réduire alors la taille du programme résiduel.

Le problème vient de ce que chaque commande `enter-monitor k` pour un verrou `k` dans un thread t est dépendant sur disponibilité de commandes `exit-monitor k` dans les threads t' du programme, $t' \neq t$. Les dépendances ainsi construites vont provoquer l'insertion de toutes les commandes relatives à `k` dans le programme résiduel.

La dépendance sur disponibilité cherche à capturer le fait qu'une commande de synchronisation `enter-monitor k` peut provoquer le retard infini d'autres commandes de moniteur sur `k`. Or les verrous, dans la plupart des programmes threadés ne sont pas détenus indéfiniment. Aussi la présence de ces dépendances seront la plupart du temps inutiles.

On définit comme un *verrou sûr* un verrou qui ne peut pas être détenu infiniment. On dira qu'un verrou `k` est sûr si tous les chemins compris entre la commande `enter-monitor k` et `exit-monitor k` ne contiennent :

1. aucune boucle libre de `wait` ;
2. aucune commande `wait` sur d'autres verrous ;
3. aucune commande `enter-monitor` ou `exit-monitor` sur des verrous non sûrs.

Une boucle libre de `wait` est une boucle dont un chemin qui la traverse est exempt de commande `wait`. La sûreté des verrous peut être calculée par un parcours en première profondeur du graphe de flot de contrôle en démarrant l'exploration de ce graphe au noeud `enter-monitor k`. On peut ainsi construire une fonction booléenne $sur(k)$.

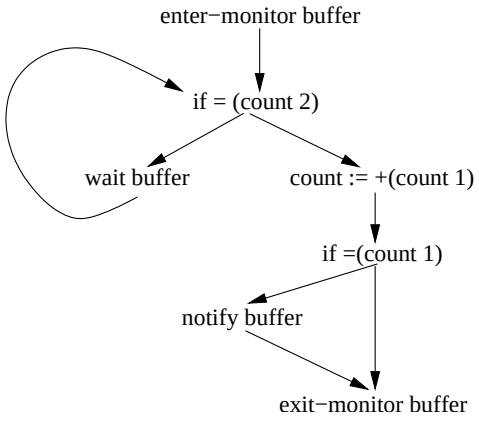

FIG. 4.1 – *Verrou sûr*

Si l'on reprend le programme de la section 4.2.2 et que l'on examine l'utilisation qui est faite du verrou `buffer`, on peut extraire le sous-graphe de contrôle du programme relatif à l'utilisation de ce verrou dans la figure 4.1. On y voit alors que les condition de sûreté définies précédemment sont réunies. Le verrou `buffer` sera libéré à la sortie du moniteur ou lors de l'appel à `wait` dans chaque itération de la boucle interne de son traitement. De la même manière, on peut constater que les moniteurs associés au verrou `compte` dans les deux threads `producteur` et `consommateur` sont garantis devoir nécessairement libérer leur verrou parce que chacun d'eux ne comporte qu'une simple instruction d'affectation.

On peut alors utiliser cette information de sûreté sur les verrous pour affiner la définition

12 en précisant que les deux premières conditions de dépendance de la définition n'introduisent effectivement une dépendance que si le verrou `k` n'est pas sûr ($\neg safe(k)$).

Ainsi, si on reprend l'exemple initial 4.2.2, tenter un découpage de ce programme sur le critère de découpage `prod_boucle.1` conduit à écarter du programme résiduel les moniteurs `compte` dans les deux threads. En effet `prod_boucle.1` ne dépend plus du moniteur `compte` car celui-ci est sûr et donc la première condition de dépendance qui s'appliquait initialement ne s'applique plus ; il ne dépend également d'aucune instruction de ce moniteur. En revanche le moniteur `buffer` est incorporé au programme résiduel car `prod_boucle.1` dépend de l'instruction `wait` du moniteur `buffer` associé : cela provoquera l'inclusion de l'ensemble de la structure de synchronisation des deux threads.

Si on reprend les définitions plus formelles établies pour construire la bismilarité, on peut dire que les actions des threads qui correspondent à se bloquer sur une commande `enter-monitor` ne sont pas observables. La seule manière dont un tel blocage peut influer sur le comportement observable d'un découpage est son caractère éventuellement indéfini. La dépendance sur disponibilité de verrous non sûrs préserve ce possible bloquage infini dans le programme résiduel. Pour des verrous sûrs, le blocage pour une durée finie est équivalent à une séquence finie d'actions non observables dans le thread bloqué qui est bisimilaire à un système dépourvu de ces actions non observables.

Chapitre 5

Conclusion

Ce rapport a présenté un ensemble de techniques et d'outils permettant de construire la formalisation d'un programme Java et des spécifications auxquelles ce programme sera confronté. Ces techniques et outils, implantés dans la suite Bandera, sont adaptés et étendus dans le cadre de ce projet, de manière à pouvoir traiter les modèles d'exécution Java d'applications interactives et mulitmodales également codées en Java.

Ce rapport a également présenté un ensemble de définitions et d'algorithmes permettant, en se donnant un critère de découpage donné, de produire un programme résiduel ne comportant que les composants susceptibles de déterminer les élements constitutifs du critère choisi et constituant une forme de bisimulation du programme original.

Le travail réalisé dans le cadre de ce projet consiste à étendre les algorithmes de calcul des différentes relations de dépendances dans la suite des outils Bandera pour prendre en compte de nouvelles structures syntaxiques des programmes Java. Le traitement des signatures de méthodes de classes est par exemple pris en compte dans le calcul des dépendances et dans le calcul du programme résiduel. Un certain nombre d'erreurs qu'il a fallu corriger dans les programmes Bandera conduisaient à des calculs de dépendances erronés. De la même façon les algorithmes de calcul du programme résiduel sont étendus pour prendre en compte le traitement des paramètres de méthodes. Les nouvelles constructions introduites pour la génération de scénarios de test ou de simulation sont également prises en compte dans l'élaboration des résidus.

Les codes de slicing produits permettent, dans leur nouvelle version, le traitement de programmes dont les structures syntaxiques intègrent pratiquement toutes les constructions du langage Java.

Bibliographie

[CDH+00] J. C. Corbett, M. D. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22th International Conference on Software Engineering*, June 2000.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[DAC98a] M. D. Dwyer, G. S. Avrunin, and J. C. Corbett. A System of Specification Patterns. **http://www.cis.ksu.edu/santos/spec-patterns**, 1998.

[DAC98b] M. D. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-State Verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15, March 1998.

[DAC99] M. D. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21th International Conference on Software Engineering*, pages 411–420, May 1999.

[DP98] M. D. Dwyer and C. S. Păsăreanu. Model-checking Generic Container Implementations. In *Proceedings of the 1st Symposium on Generic Programming*, volume 1766 of *LNCS*, pages 162–177, May 1998.

[DPC98] M. D. Dwyer, C. S. Păsăreanu, and J. C. Corbett. Translating Ada programs for model checking: A tutorial. Technical Report 98-12, Kansas State University, Department of Computing and Information Science, 1998.

[HCD+99] J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, and H. Zheng. A Formal Study of Slicing for Multi-Threaded Programs with JVM Concurrency Primitives. In *Proceedings of the 6th International Static Analysis Symposium (SAS'99)*, pages 1–18, September 1999.

[HDZ00] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing Software for Model Construction. *Higher-Order and Symbolic Computation*, 13(4), December 2000.

[HR04] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2° edition, June 2004.

[Kra98] R. Kramer. iContract – the Java Design by Contract tool. In *TOOLS, Proceedings of Technology of Object-Oriented Languages and Systems*, page 295. IEEE Press, 1998.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.

[PVE+00] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of Time Partitioning in the DEOS Scheduler Kernel. In *Proceedings of the 22th International Conference on Software Engineering*, June 2000.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[VRHS+99] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot – A Java Optimization Framework. In *Proceedings of CASCON'99*, November 1999.