|

N° d'ordre: 000

THÈSE

présentée

devant l'Université de Nice-Sophia Antipolis

pour obtenir

le grade de :
DOCTEUR DE L'UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS
Mention INFORMATIQUE

par

Marjorie RUSSO

Équipe d'accueil : OASIS
École Doctorale : STIC
Composante universitaire : INRIA SOPHIA ANTIPOLIS

Titre de la thèse :

Java et ses Aspects Concurrents : Sémantique Formelle, Visualisation et Propriétés

À soutenir le X devant la commission d'examen

M. :	Michel	RIVEILL	Président
MM. :	Egon	BÖRGER	Rapporteurs
	Traian	MUNTEAN	
MM. :	Jean Marc	JEZEQUEL	Examinateurs
MM. :	Isabelle	ATTALI	Directeurs
	Denis	CAROMEL	

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Le problème

La croissance de l'utilisation de l'informatique dans l'industrie est allée de pair avec une augmentation de la complexité des systèmes logiciels. Le fait par exemple de travailler en équipe sur un même programme, ou de devoir reprendre un programme écrit par un tiers, a été une des causes du développement d'un processus de qualité du logiciel prenant en compte la documentation du code, la facilité de maintenance, sa robustesse, etc).
L'approche objet, introduite en particulier avec le langage Simula [36, 110], semble constituer une bonne voie vers la qualité du logiciel avec ses caractéristiques principales :

- organisation en modules
- réutilisation du code, et plus précisément des différents modules
- protection des données par encapsulation
- possibilité de définir des modèles abstraits

Grâce à ce type de langages, on peut définir des objets logiciels définissant des objets réels. Par extension, les composants logiciels sont alors appelés des objets. Cependant, les notions de la programmation à objets sont difficiles à acquérir pour un programmeur découvrant ce domaine. Il faut adapter sa façon de penser et de considérer les problèmes avant de les exprimer sous forme de programmes

Avec la complexité des systèmes logiciels et la pression permanente pour obtenir des résultats toujours plus rapidement, sont apparus des problèmes d'efficacité, de performances. Deux réponses peuvent être apportées à ce problème : utiliser des machines plus puissantes, ou développer des techniques de programmation plus efficaces.
La programmation concurrente est une des techniques, qui, entre autres choses,

apporte cette possibilité. En effet, elle permet de concevoir des systèmes où plusieurs activités se produisent simultanément, réduisant ainsi la durée de l'exécution. Chaque activité est appelée selon les langages processus légers ou threads.
Il est apparu assez rapidement naturel de vouloir considérer les activités comme exécutées par des objets. Cela a créé une nouvelle famille de langages : les langages objet concurrents. Parmi les multiples langages de cette famille, on peut citer Smalltalk [81, 82], Eiffel// [26], et Java [61].

La programmation concurrente introduit de nombreuses difficultés. En effet, au niveau de la modélisation, il faut tout d'abord distinguer les différentes activités. De plus, si l'on utilise la programmation concurrente, il n'est plus possible de suivre l'exécution de façon linéaire à partir du source du programme. La mise au point des programmes devient donc très ardue, ce qui est encore amplifié par l'apparition de non-déterminisme au niveau de l'exécution. En effet, il devient extrêmement difficile de corriger un programme si l'on ne réussit pas à reproduire les erreurs survenues.
De plus, au sein d'un programme concurrent, il est fréquent que les différentes activités ne soient pas totalement indépendantes, mais aient besoin de communiquer, ou d'accéder à des données communes. Cela introduit de nouvelles sources d'erreurs. Tout d'abord, lorsque différentes activités accèdent à une donnée commune (partagée) à une même étape de l'exécution, il peut y avoir des problèmes d'homogénéité : quelle sera la valeur finale de cette donnée ? D'autre part, si une activité a besoin pour continuer son exécution d'un résultat déterminé par l'exécution d'une autre activité, il peut se créer un deadlock si ce résultat n'est jamais déterminé : l'activité restera en attente et ne pourra jamais terminer son exécution. Les deadlocks constituent une erreur courante des programmes concurrents.

Si les langages à objets concurrents répondent bel et bien à des besoins de réutilisabilité et d'amélioration des performances, leur utilisation n'est pas aisée. En effet, les aspects objets ne sont déjà pas faciles à assimiler, mais la concurrence introduit un nouveau degré de complexité difficile à appréhender.

1.2 Notre contribution, orientations et objectifs

Un domaine de recherche associé s'est donc développé dans le but d'apporter des solutions aux problèmes posés.
On peut citer deux niveaux :

- aide à la compréhension du langage lui-même, en fournissant des spécifications formelles du langage
- aide à la mise au point de programmes, en fournissant des environnements

de développement et d'exécution.

Le but de notre travail consiste à répondre à ces deux attentes dans le cadre du langage Java.

Le langage Java [61] nous a paru particulièrement pertinent, car il s'agit d'un langage très utilisé, mettant en jeu de très nombreuses notions liées à l'orientation objet et à la concurrence. Java est en effet un des premiers langages à intégrer directement une notion de concurrence à objets. De plus, ce langage ne disposait pas encore d'une définition formelle, mais uniquement d'une description informelle.

1.2.1 Une spécification formelle

Quand un langage est créé, il est souvent accompagné d'une description informelle de sa sémantique. Cependant, ce type de description n'est souvent pas très clair, ni très précis, et surtout souvent incomplet.
D'autre part, avec l'écriture de systèmes critiques et de logiciels embarqués s'est développé le besoin d'établir des preuves formelles (par opposition aux tests qui nécessitent l'exécution du programme) des propriétés de ces programmes avant de les exécuter, que ce soit pour des questions de coût ou de mise-en-œuvre.
Dans les deux cas, il devient nécessaire de disposer d'une sémantique formelle du langage, qui est forcément précise, et qui est décrite dans un formalisme mathématique facilitant ainsi les preuves.

Notre travail porte donc sur la définition d'une sémantique formelle du langage Java.
Nous avons décidé de spécifier directement le source Java. En effet, cela nous permet de ne pas nous limiter à une machine virtuelle particulière (standard, propriétaire ou modifiée). D'autre part, dans le cadre de notre environnement d'exécution, cela aide le programmeur en visualisant l'activité des threads sous la forme d'instructions Java et non de byte-code.
Une originalité de cette spécification réside dans le fait qu'elle soit exécutable au sein du système Centaur [19].
Le système Centaur est en quelque sorte un méta-environnement permettant de décrire la syntaxe concrète et abstraite d'un langage, de spécifier alors sa sémantique, puis d'en dériver un interprète. Centaur propose de plus des outils permettant de développer un environnement d'exécution basé sur la spécification formelle d'un langage.

1.2.2 Un environnement graphique d'exécution

La plupart des programmeurs n'ont toutefois ni les connaissances, ni surtout le temps d'étudier une sémantique formelle d'un langage pour en comprendre tous ses aspects.
Ces programmeurs cherchent avant tout à mettre au point rapidement leurs programmes, en ayant si possible de l'aide pour comprendre l'exécution et pour trouver où se trouvent leurs erreurs éventuelles. C'est pour répondre à ce besoin qu'ont été créés des environnements de développement et d'exécution attachés à un langage donné.
Dans un premier temps, ces environnements proposaient uniquement de visualiser des informations sous forme textuelle. Cependant, dans le cadre des langages objets concurrents deux raisons principales rendent cette visualisation insuffisante :

- Une vue textuelle rend très difficile le suivi des références existant entre les différents objets du programme, surtout dans le cas de cycles de références ou de références croisées.
- L'exécution de programmes concurrents ne peut pas être suivie de façon linéaire à partir du source du programme. En effet, plusieurs threads peuvent être en train d'exécuter une même méthode en même temps tout en étant à des instructions différentes de cette méthode.

Ces éléments ont amené au développement d'outils graphique, permettant de visualiser les objets et les références entre ceux-ci.

Nous avons voulu développer un environnement permettant la mise au point et la validation de programmes Java par la visualisation sous forme graphique, animée de l'exécution des programmes. Notre environnement comporte des aspects innovants, comme la visualisation animée sous forme graphique de l'exécution du programme et surtout le mécanisme d'abstractions. Les abstractions permettent de visualiser uniquement un sous-graphe donné ce qui est très utile en présence d'un grand nombre d'objets.
Cet environnement est d'autre part original par sa base formelle qui permet d'éviter de douter de la validité du compilateur ou de la machine virtuelle Java utilisé par les autres environnements.

La contribution essentielle de cette thèse consiste en la définition d'une sémantique formelle pour un langage réel à objet et concurrent, avec d'autre part la génération et le développement d'un environnement interactif et la possibilité d'effectuer des preuves de propriétés.

1.3 Plan du manuscrit

Le chapitre 2 de cette thèse étudie les modèles à objets et en particulier les modèles concurrents à objet. Nous présentons Java et ses principales caractéristiques en mettant l'accent sur les difficultés inhérentes à sa double-orientation : objet et concurrence. Nous décrivons ensuite le très large sous-ensemble de Java défini au sein de notre spécification en explicitant les choix effectués. Le cadre formel de notre spécification est alors exposé.

Le chapitre 3 décrit en détail notre spécification d'une sémantique formelle de Java à travers une présentation des bases de ce travail et de l'organisation générale de notre spécification. Dans un deuxième temps, la formalisation de la sémantique des différents aspects de Java est étudiée et la spécification de la concurrence et des mécanismes de synchronisation est détaillée.

Le chapitre 4 traite quant à lui de l'environnement de visualisation, appelé Jitan, développé à partir de notre spécification sémantique de Java. Nous en présentons une vue générale avant de revenir plus en détail sur chacun de ses aspects. Une troisième partie permet de mettre en évidence la visualisation des mécanismes de la concurrence et de la synchronisation entre threads, qui est alors présentée sur un exemple d'animation lors de l'interprétation d'un programme concurrent. Une cinquième partie met en avant un apport important de notre travail : la possibilité de visualiser un sous-graphe, ce qui s'avère très utile dans le cas d'un programme mettant en jeu de multiples objets et threads, rendant leur visualisation difficile.

Le chapitre 5 présente une des utilisations de notre spécification formelle : servir de base à des preuves de propriétés de langage ou de programmes. Nous nous sommes concentrés sur l'étude des propriétés de Java liées à la concurrence et aux mécanismes de synchronisation. Nous montrons dans un premier temps qu'une grande partie des propriétés étudiées sont induites directement par notre modélisation, que ce soit au niveau de la syntaxe abstraite des structures sémantiques que nous avons définies, ou au niveau de nos règles sémantiques. Quand la prise en compte n'est pas directe, notre spécification permet d'exprimer ces propriétés dans le même formalisme, intégrant ainsi la vérification des propriétés à l'interprétation des règles sémantiques.

Nous présentons en fin de manuscrit (chapitre 6) un bilan de nos travaux, et des perspectives de travaux futurs se basant sur notre travail.

Chapitre 2

Modèles et Techniques

Java est un langage à objets concurrent. Ces deux facettes introduisent de nombreuses notions, qui, déjà complexes prises isolément, deviennent très difficiles à appréhender quand elles sont réunies. Dans un tel cadre, il devient pratiquement obligatoire de disposer d'une sémantique formelle de Java (ou d'un sous-ensemble du langage) afin de pouvoir définir de façon formelle chaque notion et possibilité offerte par le langage étudié. Cette spécification devient alors une référence à laquelle implémenteurs et programmeurs peuvent.
La première section présente donc les modèles à objets et les modèles à objets concurrents. Elle décrit ensuite les principales caractéristiques du langage Java et les difficultés introduites. Enfin, elle développe le sous-ensemble du langage Java que nous avons décidé de traiter en détaillant les choix effectués. Une seconde section décrit enfin le cadre formel de notre travail, c'est-à-dire les différentes formes de sémantique existantes ainsi que Typol, le formalisme que nous avons choisi d'utiliser pour notre spécification.

2.1 Modèles à objets

Java est un langage de programmation à la fois à objets et concurrent. Nous replaçons Java dans le contexte des langages à objets et des langages à objets concurrents. Nous étudions ensuite les caractéristiques principales de Java qui nous ont amenés à faire des choix dans la formation du sous-ensemble défini formellement par la suite. D'autre part, les aspects concurrents de Java nous ont permis d'établir des critères de choix portant sur la sémantique utilisée pour spécifier formellement le langage.

2.1.1 Langages à objets

La programmation à objets [1] est un style de programmation (ayant conduit à la création de nombreux langages dont Java) permettant de développer des applications autour d'entités appelées *objets*, possédant un état modélisé par des *attributs*, et des *méthodes*, c'est-à-dire des procédures d'initialisation et de modification de cet état.

Les différents langages à objets mettant en oeuvre ce principe peuvent être classés au sein de deux grandes catégories : les *langages de prototype* et les *langages de classe* (le lecteur intéressé peut se reporter à [18, 140] pour mieux comprendre les bases de cette différentiation).
Les langages de prototypes [14] (comme Pargon [129], Self [28], et Omega [13], etc) sont généralement issus de Lisp, et définissent l'objet comme le clone d'un modèle appelé prototype.
La classe définit le type qui va permettre de programmer les objets. Elle constitue une sorte de "moule" pour créer les objets qui auront tous les mêmes méthodes et attributs, même si la valeur de ces derniers dépend de l'objet et non de la classe. La classe facilite la réutilisation et fournit un mécanisme permettant de lier des parties de code portant sur des objets identiques (voir [56] pour une étude plus approfondie de l'intérêt et des lacunes des langages de classe). Les langages de classe sont les plus courants des langages à objets ; on peut citer en particulier Smalltalk [81, 82], C++ [134], Eiffel [97] et naturellement Java [61].

2.1.2 Langages à objets concurrents

Un autre aspect essentiel est celui de la concurrence et plus précisément du *multithreading*. Le grand intérêt de cette technique est de pouvoir traiter plus vite des problèmes plus complexes, et de dépasser certaines limites des systèmes séquentiels. On parle alors de *parallélisme de situation*. Mais il existe aussi des problèmes intrinsèquement concurrents, on parle alors de *parallélisme de résolution*.
Dans tous les cas, la concurrence nécessite la définition d'*activités autonomes*. Une activité est une entité qui est chargée d'exécuter une certaine portion de code. Un programme séquentiel, comporte une seule activité, même si celle-ci passe d'un objet à l'autre au gré des appels de méthodes. Un programme concurrent possède quant à lui au moins deux activités qui s'exécutent en même temps.
Si les avantages de telles techniques semblent faire l'unanimité, leurs mises en pratique a donné lieu à de multiples études (le lecteur intéressé pourra notamment se référer à [130, 6, 58] et plus particulièrement à [7] en ce qui concerne l'étude des différents concepts et notations présents au sein des langages concurrents).

En considérant les langages à objets, il est apparu naturel d'associer les activités concurrentes à des objets, notamment afin de profiter des possibilités offertes par la réutilisation de code. Les langages concurrents à objets ont donné lieu à la définition de multiples modèles (voir par exemple [145, 152, 24, 104, 150], [2, 4, 3] pour le modèle d'acteurs, et [103, 105] pour un modèle d'objets actifs). Le plus célèbre modèle est le modèle d'acteurs. Dans celui-ci, tous les objets sont dits actifs, communication et synchronisation se fondent sur les envois de messages et d'objets. Certains langages à objets concurrent sont inspirés du modèle d'acteurs, comme par exemple le langage Eiffel// [8].
Certains langages se distinguent toutefois de ce modèle en proposant des objets actifs, et des objets non actifs. Eiffel// et Java en sont deux exemples.

2.1.3 Java : un langage à objets concurrent

Java est un langage fortement typé, ce qui implique que toute variable est déclarée comme étant d'un type donné. On peut distinguer deux catégories de type : les *types simples* (comme un entier ou un booléen, etc) et les *types référence* qui indiquent que la variable ainsi déclarée est une référence vers un objet ou un tableau de ce type. Dans le cadre de Java, et contrairement à d'autres langages objets comme Eiffel, la notion de type se distingue de celle de classe par la présence d'interfaces : l'interface spécifie un modèle général du type d'un objet, alors que la classe en est l'implantation. Java supporte l'héritage simple au niveau des classes, mais aussi l'héritage multiple au niveau des interfaces.

Dans la terminologie Java, on peut remarquer le terme de *membres* d'une classe qui regroupe à la fois les *attributs*, et les *méthodes* permettant d'agir sur ces derniers. Le terme de *variable* est quant à lui uniquement utilisé pour désigner les variables locales à un bloc d'instructions.

Cette section étudie ces différents concepts en s'appuyant sur l'exemple du programme Java présenté en Figure 2.1.

2.1.3.1 Classes et objets

Une classe constitue en quelque sorte un "moule" qui permet de créer des objets. De tels objets sont alors appelés des *instances* de cette classe. Plus formellement, une déclaration de classe introduit un nouveau type référence et décrit son implantation.

SequentialEx.java

File Display Edit Selections Editing-Tools java

```java
class SequentialEx {
    public static void main(String[] args) {
        /*Creation des objets et appels des constructeurs*/
        Rectangle rect = new Rectangle();
        Square square = new Square();
        /*Appels des methodes pour fixer la longueur des cotes*/
        rect.setSides(5, 8);
        square.setSides(6);
    }
}

class Rectangle {
    int length;
    int width;
    Rectangle() {
    }
    void setWidth(int w) {
        width = w;
    }
    void setLength(int l) {
        length = l;
    }
    void setSides(int w, int l) {
        width = w;
        length = l;
    }
}

class Square extends Rectangle {
    Square() {
    }
    void setWidth(int s) {
        /*Masque la methode de meme nom de la classe Rectangle*/
        width = s;
        length = s;
    }
    void setLength(int s) {
        /*Masque la methode de meme nom de la classe Rectangle*/
        width = s;
        length = s;
    }
    void setSides(int s) {
        /*Surcharge la methode de meme nom de la classe Rectangle*/
        setWidth(s);
    }
}
```

FIG. 2.1 – *Exemple de programme séquentiel.*

Les classes

Cette déclaration est composée principalement des éléments suivants : nom de la classe, ancêtre de la classe s'il existe (voir le paragraphe suivant sur l'héritage), corps de la classe. Le corps de la classe est constitué de ses membres (champs et méthodes), ainsi que de constructeurs. Les champs d'une classe peuvent être soit des variables de classe (variables créées une seule fois et qui sont liées à la classe et communes à toutes les instances de cette classe), soit des variables d'instance ou attributs. Un *constructeur* est une méthode particulière qui est appelée de façon implicite (non demandée) à la création de l'objet, et qui permet par exemple d'initialiser les attributs de façon différente selon l'objet. Pour cela il suffit d'écrire plusieurs constructeurs ayant un nombre de paramètres différent, ou de types différents, et lors de la création de l'objet de donner les arguments correspondant au constructeur désiré.
Si l'on se réfère à l'exemple de la Figure 2.1, ce programme déclare trois *classes* : `SequentialEx`, `Rectangle` et `Square`. Prenons la classe `Rectangle` : tous les objets instances de cette classe possèderont deux attributs `length` et `width` de type entier initialisé à la valeur 0 (par défaut). Cette classe définit par ailleurs un *constructeur* `Rectangle`, ainsi que trois *méthodes* : `setWidth`, `setLength` et `setSides`.

Les objets

L'instruction Java de création d'un objet est la suivante :

new ClassName (ArgumentList)

et son utilisation la plus courante est l'affectation de la référence de l'objet créé à une variable par l'instruction :

variable = new ClassName (ArgumentList);

Le résultat de cette affectation peut se représenter graphiquement, comme cela est visualisé dans la Figure 2.2. L'élément *variable* (variable simple, attribut ou variable de classe) est relié par une flèche à l'objet de type *ClassName* dont il possède une référence. Cette référence est la valeur de l'élément *variable*.
Mais cette instruction de création d'objet peut aussi se rencontrer en dehors d'une affectation, en tant qu'argument lors d'un appel de méthode ou comme valeur de retour lors de l'exécution d'une méthode.

Dans l'exemple de la Figure 2.1, des instructions de création d'objets sont présentes au sein de la méthode `main` de la classe `SequentialEx` : par exemple l'instruction, `square = new Square ()` qui crée un objet de type `Square` et dont la référence est stockée dans la variable `square`.

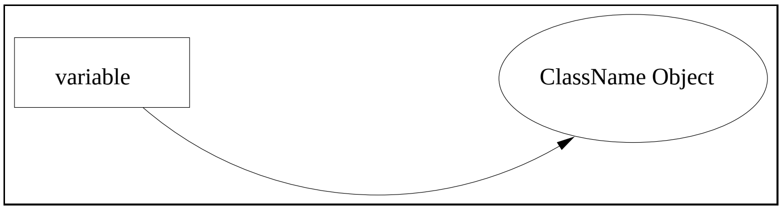


FIG. 2.2 – *Visualisation d'une référence vers un objet.*

Héritage d'interface et de classe

L'*héritage* est une notion importante au sein des langages à objets : il constitue un des intérêts de ces langages en permettant notamment la réutilisation de code. En effet, l'héritage de classe permet de définir successivement des classes à partir d'autres classes, on dit que les premières héritent des secondes. Il existe aussi une notion d'héritage (ou plus précisément d'implémentation) entre une classe et des interfaces que nous abordons dans le paragraphe suivant.

Une déclaration d'interface introduit un nouveau type référence, dont les membres sont des constantes et des méthodes abstraites (c'est-à-dire que l'on ne donne pas le corps de ces méthodes, uniquement leur nom et paramètres). Le type ainsi défini n'a pas d'implémentation. Des classes peuvent en hériter en fournissant des implémentations pour ses méthodes abstraites. Cet héritage est déclarée par la clause *implements InterfaceList* qui suit celle d'héritage des classes (ce point est développé ci-dessous).
Une interface peut-elle même être déclarée comme étant une extension directe d'autres interfaces grâce à la clause *extends InterfaceList* utilisée juste après son nom au sein de sa déclaration. Cet héritage multiple au niveau des interfaces permet aux objets de supporter plusieurs comportements communs sans pour autant partager des implémentations.

Le mécanisme d'héritage de classe, qui est le plus couramment utilisé (par rapport à l'héritage d'interfaces), est le suivant : une classe A héritant d'une classe B possédera tous les membres de la classe B (attributs, méthodes et constructeurs), et pourra d'autre part définir ses propres attributs et méthodes. A est appelée *descendant* de B qui en est l'*ancêtre*.
En Java, une classe A (appelée *sous-classe*) hérite d'une classe B (appelée *super-classe*) par l'utilisation de la clause *extends* `B`. Il est important de noter qu'en Java une classe ne peut avoir qu'une seule super-classe directe, contrairement à d'autres langages comme Eiffel qui permettent l'héritage multiple. Par contre, toutes les classes Java héritent implicitement de la classe *Object* (classe définie

dans la bibliothèque *java.lang*). En résumé, les membres de la classe A seront :
- les membres hérités de la classe B
- et les membres déclarés au sein de la classe A.

Revenons à l'exemple du programme présenté en Figure 2.1. Dans ce programme la classe `Square` hérite de la classe `Rectangle` et donc des attributs `length` et `width`.

En ce qui concerne l'héritage des méthodes, différents mécanismes peuvent se produire.
Soit MName le nom d'une méthode de la classe B (super-classe), cette méthode étant héritée par la classe A, on peut avoir :
- Héritage de la méthode MName par la classe A.
- *Masquage* de la méthode MName de la classe B par une méthode de A si ces deux méthodes ont la même *signature*.
- *Surcharge* de la méthode MName de la classe B par une méthode de la classe A si cette dernière porte le même nom mais possède une signature différente.

La *signature* d'une méthode est constituée de son nom, ainsi que du nombre et du type de ses paramètres formels.
Dans l'exemple de la Figure 2.1, on peut à la fois observer le masquage des méthodes `setWidth` et `setLength` de la classe `Rectangle` par les méthodes de la classe `Square` qui possèdent la même signature, et la surcharge de la méthode `setSides` de la classe `Rectangle` par celle de la classe `Square` car la méthode `setSides` de la classe `Square` ne possède pas la même signature.
Ces différents mécanismes entrent en jeu lors de l'appel d'une méthode sur un objet, afin de déterminer quelle méthode appeler en fonction du type de l'objet sur lequel on l'appelle.

Le concept d'héritage est très important, car il permet, par la construction de hiérarchie d'héritage, la définition de bibliothèques de classes, celles-ci pouvant être réutilisées par d'autres programmeurs. La version de base de Java propose déjà un ensemble de classes utilisables au sein de bibliothèques appelées *API*.
Il permet d'autre part de modéliser un problème différemment au cours de sa résolution en définissant plusieurs niveaux d'abstraction. En se plaçant au niveau le plus abstrait lors du début de la conception, on introduit chaque nouvel aspect ou précision en héritant de la classe précédemment définie.

2.1.3.2 Threads et concurrence

La concurrence en Java (voir [87, 86] pour une étude détaillée des différents mécanismes mis en jeu) s'exprime au moyen de *threads* (le lecteur intéressé peut se

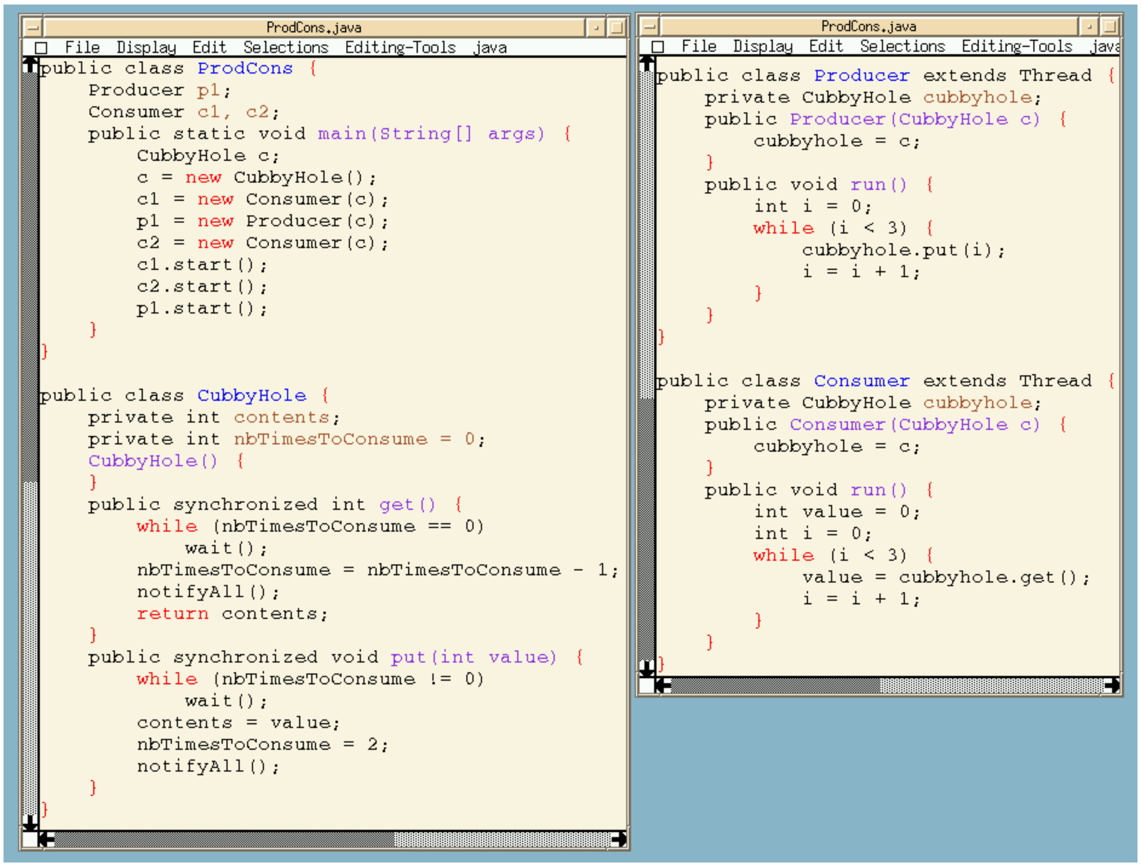


FIG. 2.3 – *Programme Producteur-Consommateur.*

reporter à [111, 70, 45]), qui sont des objets particuliers au sein d'un programme Java car disposant d'une activité qui leur est propre. Ces threads, qui par ailleurs sont considérés comme des objets "simples" (qui ne sont pas des threads), possèdent des activités indépendantes, mais peuvent toutefois interagir, en appelant par exemple des méthodes sur un même objet commun à plusieurs threads. Dans ce cas, il peut être nécessaire d'utiliser un mécanisme permettant d'éviter des accès multiples à une même donnée en même temps, ceci afin d'éviter des problèmes de cohérence. En Java, ce mécanisme est appelé synchronisation (voir [95] pour plus de détails).

Cette section étudie les différents aspects liés au multithreading, en prenant comme illustration un programme Java inspiré du modèle du producteur-consommateur, présenté en Figure 2.3.

Un exemple de programme concurrent

Le programme présenté en Figure 2.3 met en jeu un producteur et deux consommateurs d'une donnée (ici un entier). Le producteur doit placer successivement dans un buffer disposant d'une seule case des données qui sont alors lues par les deux consommateurs. Les consommateurs ne peuvent naturellement rien lire avant qu'une première donnée n'ait été placée dans le buffer par le producteur. De la même façon, le producteur ne peut pas placer une nouvelle donnée dans le buffer avant que la précédente n'ait été lue par les deux consommateurs. Cet exemple introduit de nombreux concepts Java : création d'objets et de threads, exécution concurrente, appels de méthodes, synchronisation, notifications.

Ce programme est constitué de quatre classes :

`ProdCons, CubbyHole, Consumer et Producer`.

La classe `ProdCons` est la classe principale du programme. L'exécution du programme débute avec l'exécution de son thread principal (le thread créé implicitement pour exécuter la méthode *main*). Ce thread crée successivement un objet de type `CubbyHole`, un thread `Consumer`, un thread `Producer`, et finalement un autre thread `Consumer`. Les trois appels successifs de la méthode *start* sur les trois threads créés démarrent leurs exécutions respectives. Ces trois threads vont alors s'exécuter de manière concurrente (avec entrelacement sur une machine mono-processeur) : chaque thread exécute sa méthode *run* qui est définie dans la classe correspondant à son type. Les deux consommateurs essaient de lire la valeur contenue dans le buffer, et le producteur essaie d'y placer une donnée. Afin d'éviter les conflits lors de l'accès au buffer, les méthodes `put` et `get` sont déclarées comme *synchronized*, c'est-à-dire comme des sections critiques.

Ces différentes notions vont être détaillées dans les paragraphes suivants.

Les threads : généralités

En Java, un thread est considéré comme un objet. Il est donc créé par la même instruction de création d'objets *new ClassName ()*. Les threads sont classés par groupes, mais par défaut tous les threads créés au cours de l'exécution d'un programme appartiennent au même groupe (c'est le cas le plus courant). Afin de définir sa particularité, son activité qui fait de lui un thread et non pas un objet simple, il est nécessaire que la classe dont il est une instance hérite (directement ou non) de la classe *Thread* définie dans la bibliothèque *java.lang*.
Si l'on considère le programme de la Figure 2.3, les classes `Producer` et `Consumer`, définissent des threads : on reconnaît la clause *extends Thread*.

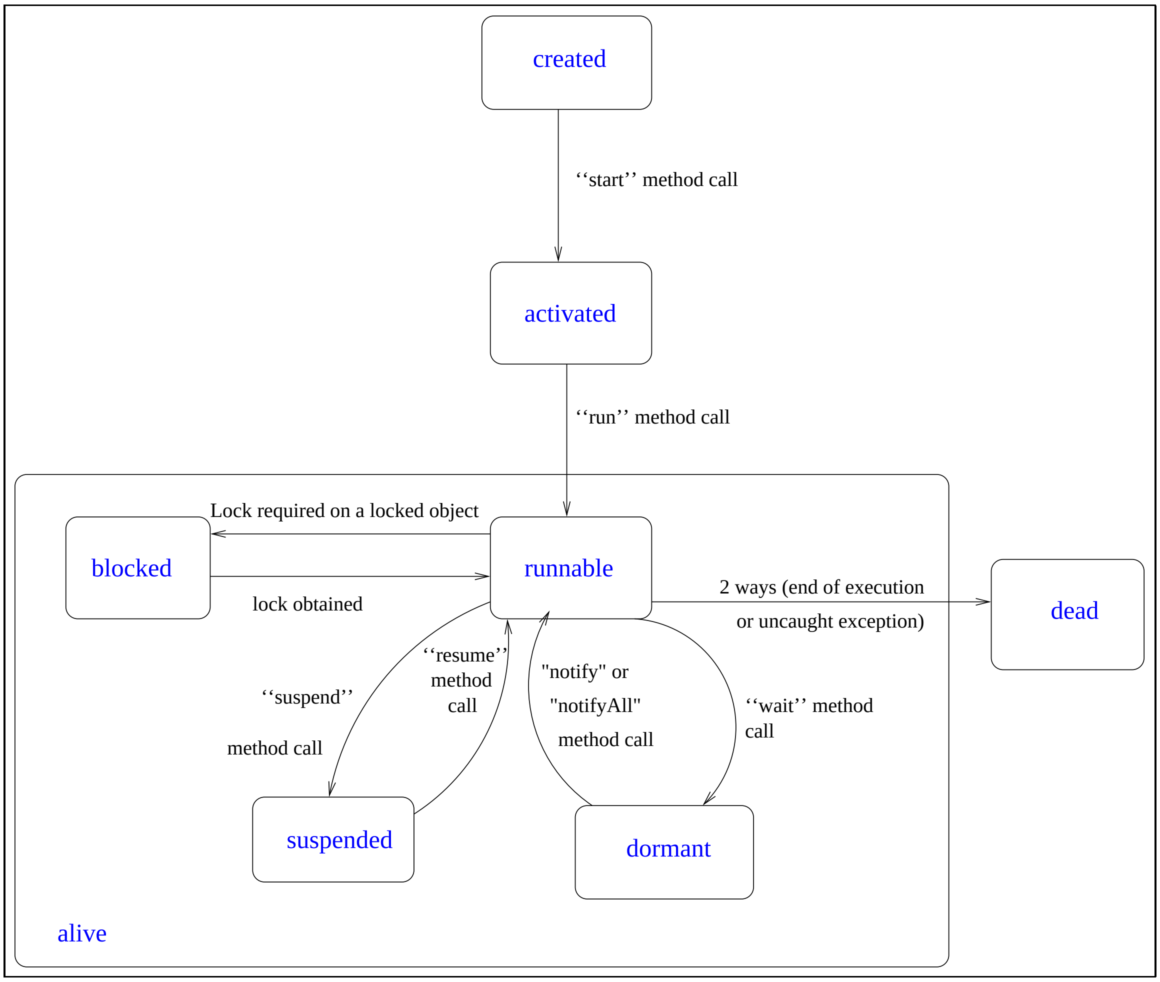


FIG. 2.4 – *Les différents états d'un thread.*

Contrairement aux autres classes qui possèdent des méthodes variées, les classes définissant des threads doivent définir une méthode *run* qui va représenter l'activité du thread dès qu'il aura été lancé. En effet, à sa création, un thread `T` se trouve dans l'état *created*. Cela signifie qu'il n'est pas encore actif. Ce thread `T` passe de l'état *created* à l'état *activated* lorsqu'un autre thread `T'` appelle la méthode *start* sur le thread `T`. Cela pousse `T` à invoquer sa méthode *run*, le thread `T` passe alors à l'état *alive*. Le thread `T` reste alors *alive* jusqu'à ce qu'un des cinq évènement suivants se produise :

1. La méthode *run* se termine normalement
2. La méthode *run* se termine brutalement par la levée d'une exception
3. Le thread `T` invoque la méthode *stop*
4. Un autre thread invoque la méthode *stop* sur le thread `T`

5. Un thread invoque la méthode *exit* de la classe *Runtime* sur l'objet courant de type RunTime. Cet appel stoppe la machine virtuelle Java courante.

Ces cinq évènements provoquent le passage du thread de l'état courant à l'état *dead*. Un thread dans l'état *alive* est dit *runnable* s'il est en train de s'exécuter normalement, ou s'il est mis en attente du fait de l'exécution du système par l'ordonnanceur. Il est dit *executing* lorsqu'il est en cours d'exécution.
Ces différents états d'un thread, ainsi que les états liés à la synchronisation qui seront explicités dans le paragraphe 2.1.3.2, sont présentés dans la figure 2.4.

L'ordonnancement

Du fait de la concurrence, plusieurs threads sont amenés à s'exécuter en même temps. Cependant, cette simultanéité n'est possible que sur des machines multi-processeurs. Il peut cependant arriver que les programmes soient exécutés sur des machines possédant un seul processeur. Pour simuler la concurrence, les différents threads s'exécutent alors chacun leur tour avec entrelacement. L'ordonnanceur est chargé de décider quel thread va s'exécuter, le thread choisi prend alors la main et exécute une instruction atomique et rend ensuite la main à l'ordonnanceur qui sélectionne alors un autre thread.
En Java, l'ordonnancement réellement suivi n'est pas spécifié, et dépend le plus souvent des différentes machines virtuelles proposées sur le marché. Dans notre spécification de Java, l'ordonnancement se fait de façon déterministe dans l'ordre de création des threads, mais il peut être envisagé de passer à un ordonnancement non déterministe, comme cela a été fait dans le cadre de la spécification d'une sémantique d'Eiffel// (voir [8]).

Le mécanisme de synchronisation

La synchronisation permet d'obtenir un comportement déterministe au niveau des différents threads d'un programme. Elle est obtenue en utilisant des moniteurs, mécanismes de haut niveau qui permettent qu'une partie de code ne soit exécutée que par un seul thread à la fois. Ils fonctionnent à travers un mécanisme de verrous. En Java, un verrou est associé à chaque objet. Seuls les threads peuvent demander le verrouillage ou le déverrouillage d'un objet. Les accès aux verrous sont réglés par des files d'attente.
La synchronisation permet l'exclusion en assurant dynamiquement qu'un seul thread à la fois puisse accéder à l'objet considéré. Elle est implémentée par l'accès exclusif au verrou interne (appelé parfois un "mutex") qui est associé à chaque objet Java. Chaque verrou se comporte comme un compteur.
Dans l'exemple de la Figure 2.3, les deux méthodes de la classe `CubbyHole` sont déclarées comme étant *synchronized* ce qui permet d'assurer la cohérence de la

donnée lors de son écriture ou de sa lecture par les différents threads.

La concurrence est vraiment un des points-clés de Java, en offrant la possibilité d'exécuter des tâches en parallèle, permettant ainsi un gain de temps, par exemple en calcul numérique sur des matrices. Mais elle introduit aussi une nouvelle complexité dans l'écriture et la mise au point des programmes dont l'exécution devient indéterministe : par exemple l'ordonnancement dépend de la machine virtuelle utilisée.

2.1.3.3 Les exceptions

Un autre aspect important innovant de Java est la notion d'*exception*.

Une exception est un signal d'erreur lancé en cours d'exécution de programme. Un tel évènement se produit soit quand un programme viole une des contraintes sémantiques de Java (par exemple en voulant accéder à une case d'indice négatif d'un tableau), soit à la demande du programmeur (par exemple pour réagir à une valeur inattendue d'un argument donné lors du lancement de l'exécution d'un programme). On dit alors qu'une exception est *levée*.
La levée d'une exception stoppe le flot d'exécution du thread courant et transfère le contrôle de ce point à un point qui a pu être spécifié par le programmeur ou qui est défini par défaut au sein d'une bibliothèque Java. Ainsi il est possible d'effectuer un traitement d'une erreur si l'on avait prévu qu'elle puisse se produire. Cela permet de dissocier le traitement d'erreurs du code principal, améliorant ainsi la lisibilité d'un programme.

Chaque exception est représentée par une instance de la classe *Throwable* ou d'une de ses sous-classes. Durant le processus de lancement d'une exception, chaque expression, instruction, chaque invocation de constructeur ou de méthode, dont l'exécution a été commencée par le thread courant mais qui n'est pas encore achevée, est terminée de façon abrupte. Ce processus se poursuit jusqu'à ce qu'un bloc écrit pour la gestion d'une exception de ce type soit rencontré au sein de la pile des appels.
Un tel bloc d'instructions, appelé *handler*, est introduit par la clause *catch* (voir 3.4 pour une étude détaillée de la sémantique des exceptions). Si cette clause n'est pas trouvée, la méthode *uncaughtException* est invoquée. Cette méthode stoppe alors l'exécution du thread courant, ainsi que celles de tous les threads de son groupe.

Les aspects objets et concurrence de Java introduisent de nombreuses notions, qui déjà complexes prises isolément, deviennent très difficiles à appréhender quand elles sont réunies. Dans un tel cadre, il devient pratiquement obligatoire de disposer d'une sémantique formelle de Java afin de pouvoir définir de façon formelle (et donc précise) chaque caractéristique du langage. Cette spécification devient alors une référence pour l'implémenteur de compilateurs ou le programmeur.

2.1.4 Notre sous-ensemble de Java

Notre but est de définir une spécification formelle de Java, pouvant par la suite servir de base au développement d'un environnement d'exécution, et à la vérification de propriétés du langage. Afin de pouvoir nous concentrer sur les notions les plus importantes, nous avons été amenés à déterminer un sous-ensemble du langage, le plus représentatif possible, et permettant d'écrire des programmes réalistes, mais sans nous disperser dans la multitude d'instructions, d'expressions et d'opérateurs du langage.
Cette section présente le sous-ensemble que nous avons sélectionné et justifie les choix effectués.

Le sous-ensemble que nous avons sélectionné est composé des éléments suivants :

- types simples : entiers et booléens
- instructions de base : évaluation arithmétique, test *if ... then ... else ...*, boucles *while ... do ...*et *for ... do ...*, tests multiples *switch ...*,
- classes et interfaces,
- types complexes : références vers des objets,
- création d'objets,
- héritage,
- liaison dynamique,
- appels de méthodes (d'instance et de classe),
- création de threads,
- multithreading et synchronisation,
- exceptions,
- tableaux,
- API (classes *Object*, *Thread*, *Throwable*).

Les aspects non traités sont la séparation des classes dans différents fichiers et packages, le chargement dynamique des classes, certains aspects du modèle de gestion de la mémoire en présence de multi-threading, ainsi que les aspects statiques liés au typage.

2.1.4.1 Répartition des classes dans différents fichiers

Il est classique lors de l'écriture de programmes Java de répartir les différentes classes au sein de différents fichiers (une classe par fichier).
Cependant, le système Centaur au sein duquel nous avons écrit notre spécification n'est pas adapté à l'analyse d'un code source réparti dans plusieurs fichiers. Nous pensons qu'il aurait été possible de résoudre ce problème, mais nous avons préféré nous concentrer sur la spécification sémantique. Nous avons donc pris comme base le modèle d'un programme Java dont toutes les classes sont présentes au sein du même fichier.

2.1.4.2 Chargement dynamique des classes

On appelle chargement dynamique des classes le fait qu'une classe d'un programme Java ne soit chargée, c'est-à-dire que ses variables statiques soient initialisées, qu'à la création du premier objet instance de cette classe.
Avec la sémantique que nous avons définie, ce chargement est effectué au début de l'interprétation du programme, et pour toutes les classes présentes dans le fichier source.
Dans le cadre de la plupart des programmes Java, cela n'a pas d'impact sur l'exécution, et notre spécification reste correcte. Le chargement dynamique des classes peut par contre avoir une importance dans le cas où les classes sont distantes, elles peuvent alors par exemple être modifiées entre le début de l'exécution et leur chargement. Ce cas n'est alors pas traité dans notre sémantique, ne serait-ce que parce nous considérons un modèle où toutes les classes Java d'un programme sont présentes au sein d'un unique fichier.

2.1.4.3 Aspects statiques liés au typage

Nous avons fait le choix de définir une sémantique dynamique, et prenons comme hypothèse lors de l'interprétation des programmes que ceux-ci sont correctement typés. Cela nous a permis de nous concentrer sur la spécification des aspects objets et surtout de la concurrence et des mécanismes de synchronisation, en nous évitant d'avoir à définir tout d'abord une sémantique statique.
De même, nous avons décidé de ne pas spécifier dans un premier temps le mécanisme de masquage des attributs qui repose sur la différence pouvant exister entre le type statique (de déclaration) d'un attribut ou d'une variable, et son type dynamique (durant l'exécution du programme). Une étude de ce mécanisme et de ses conséquences sur notre spécification et notre environnement est proposée en Sections 3.6 et 4.6.

2.1.4.4 Modèle de la concurrence

Le manuel de référence de Sun ([61]) décrit un modèle de mémoire pour la mémoire partagée entre différents threads. Ce modèle comprend une mémoire principale et des mémoires de travail, et des règles décrivant comment et quand un thread peut (ou doit) écrire dans la mémoire principale à partir de sa mémoire de travail.

Ces mécanismes de copie et de lecture au sein de différentes mémoires forment surtout la base du modèle de la machine virtuelle Java, et font appel à des instructions de bas-niveau. Notre travail se situe à un plus haut niveau, au niveau du code source Java. Nous avons donc choisi de considérer une mémoire unique qui contient tous les objets.
De plus, nous n'avons pas spécifié les notions de groupes et de priorités de threads, qui sont des mécanismes assez peu utilisés en pratique.

2.1.4.5 Autres API

Il existe un grand nombre d'API Java, même si l'on se restreint aux API de base, décrites dans la spécification informelle donnée par Sun [61]. Nous nous sommes efforcés dans notre spécification de définir les méthodes principales de ces API, c'est-à-dire celles liées à la concurrence et aux mécanismes de synchronisation (méthodes *start*, *wait*, *notify*, etc), mais nous avons choisi de ne pas spécifier l'intégralité de ces API, ce qui aurait été un travail assez répétitif et ne présentant pas un intérêt théorique important. Nous avons par contre défini une sémantique modulaire permettant ainsi d'ajouter aisément la définition d'autres API.

Les différents aspects que nous avons inclus dans notre spécification nous ont paru pertinents car ils permettent d'écrire des programmes évolués, et d'étudier des notions complexes comme la concurrence et la synchronisation entre threads, les mécanismes des exceptions, etc. Les aspects que nous avons omis sont soit d'anciens aspects systèmes (comme la répartition des classes au sein de différents fichiers) par opposition aux notions purement langage, soit des aspects peu connus et très peu utilisés (comme le masquage des attributs sur lequel nous revenons dans la Section 3.6).

Dans la suite de ce document et à fin de simplification, nous utiliserons le terme *Java* pour désigner le sous-ensemble traité du langage.

2.2 Techniques formelles

Les langages à objets s'appuient sur des notions complexes, et ce phénomène ne fait que s'aggraver avec les langages à objets concurrents. Disposer d'une spécification formelle d'un tel langage apparaît donc comme une nécessité, pour d'une part comprendre parfaitement le langage et ses constructeurs syntaxiques (sans ambiguïté), et d'autre part comme base pour des preuves formelles que ce soit des preuves de propriétés du langage ou de programmes écrits dans ce langage. Il existe différents types de sémantiques, certains étant plus adaptés à un but ou à une famille de langages que d'autres. On distingue notamment la sémantique statique (pour définir le typage), la sémantique dynamique (pour étudier le comportement et l'exécution des programmes) ou la transformation (ou traduction).

Cette section décrit les différentes techniques de spécification d'une sémantique formelle d'un langage de programmation et présente le cadre formel de notre spécification d'une sémantique de Java, c'est-à-dire les différents formalismes utilisés et leurs caractéristiques.

2.2.1 Panorama

Ce paragraphe présente les méthodes formelles de spécification sémantiques: la sémantique algébrique, les grammaires attribuées, la sémantique dénotationnelle, la sémantique naturelle et la sémantique opérationnelle.

2.2.1.1 Sémantique algébrique

La sémantique algébrique (voir [65, 146, 147] pour une étude plus poussée) repose sur la notion de types abstraits algébriques, c'est-à-dire sur un ensemble de définitions de types munis d'opérations vérifiant des axiomes donnés. La description obtenue est très formalisée et mathématique.
Cette sémantique est utilisée principalement pour valider des compilateurs (voir par exemple [40] pour un compilateur du langage Pascal, [29] pour un compilateur Lisp, etc.), ou pour prouver des propriétés de programmes (comme dans [127] pour faire des preuves de terminaison de programmes). Le lecteur intéressé pourra se reporter à [64] pour avoir des détails sur l'utilisation de la sémantique algébrique.

2.2.1.2 Sémantique des grammaires attribuées

Les grammaires attribuées (voir [39, 112, 144]) représentent un essai de généralisation des grammaires non-contextuelles à la spécification de la sémantique des langages de programmation.

Le principe est d'attacher aux noeuds d'un arbre de syntaxe des symboles appelés *attributs*, pouvant avoir une valeur. On distingue les *attributs synthétisés*, calculés à partir des feuilles de l'arbre de syntaxe, des *attributs hérités*, obtenus à partir des noeuds parents. Une (ou des) règle sémantique est définie pour chaque règle de production de la grammaire du langage ; ces règles doivent préciser comment sont obtenus les attributs concernant cette règle de production. On obtient alors un arbre abstrait pour une phrase du langage. Un évaluateur calcule ensuite les attributs nécessaires à l'obtention de tous les attributs synthétisés de la racine. Le résultat final est la valeur sémantique du programme. Celle-ci peut être du code machine, un type, de la syntaxe abstraite plus riche (contenant plus d'information) ou *arbre abstrait décoré*, ou une autre représentation de la sémantique comme par exemple une λ-expression.
Les grammaires attribuées sont très souvent employées pour définir des compilateurs (par exemple [148] pour le langage Pascal, [50] pour ADA, etc.).

2.2.1.3 Sémantique dénotationnelle

La sémantique dénotationnelle (voir [99, 149, 121] pour une étude approfondie) repose sur le λ-calcul qui permet de représenter toutes les fonctions *calculables*, c'est-à-dire que l'on peut calculer avec des algorithmes informatiques. Elle se définit par la traduction des constructions du langage à spécifier dans le λ-calcul. La sémantique dénotationnelle a été souvent utilisée pour spécifier la sémantique se systèmes concurrents ([125]), de langages parallèles (comme OCCAM [126]) ou de langages de spécification de systèmes concurrents ou distribués (comme LOTOS [23]). Ce type de sémantique a aussi été employé afin de spécifier des langages logiques (comme dans [42] pour définir un langage de programmation logique par contraintes). Watt [149] a présenté un travail sur une sémantique dénotationnelle qui soit exécutable.
Le principal inconvénient de cette technique est la difficulté de manipulation des termes obtenus dès que l'on traite des langages de taille réelle. De plus, les aspects statiques du langage sont difficiles à représenter dans cette sémantique.

2.2.1.4 Sémantique opérationnelle

La sémantique opérationnelle (voir [5] pour plus de détails) est le type de sémantique préféré des implémenteurs car elle permet de décrire le comportement d'un programme par son exécution sur une machine abstraite. Elle repose sur la description des constructeurs d'un langage par un ensemble de règles de réécriture. Ces règles définissent ainsi une machine abstraite qui spécifie le langage. Cette technique permet de spécifier la méthode d'exécution ou d'évaluation choisie pour le langage à décrire. Parmi les différents modèles de sémantiques opérationnelles, on peut citer la Sémantique Opérationnelle Structurelle (ou SOS) de

Gordon Plotkin [117] (aussi appelée "small-step") et la Sémantique Naturelle de Gilles Kahn [76] (aussi appelée "big-step).
La sémantique opérationnelle a été utilisée pour spécifier formellement des langages à objets (par exemple TOOPLE [22]), des langages à commandes gardées (voir [93]), etc.
Son principal avantage est d'être assez intuitive, car proche de la vision du programmeur.

2.2.1.5 Sémantique naturelle

La sémantique naturelle a été conçue à l'INRIA par Gilles Kahn et constitue le prolongement de l'approche de Gordon Plotkin avec la Sémantique Opérationnelle Structurelle. Elle est inspirée de la déduction naturelle [122], dans le sens où les règles sont écrites sous leur forme la plus simple, sans prendre en compte la forme des hypothèses.
La sémantique naturelle consiste en l'écriture d'une spécification à l'aide d'axiomes et de règles d'inférence. Ces règles peuvent aussi bien spécifier le typage, que la traduction d'un langage dans un autre langage (pour écrire un compilateur), ou la description de transitions d'une machine abstraite.
La sémantique naturelle s été employée pour spécifier une logique modale [94], en intelligence artificielle [25], pour spécifier l'évaluation paresseuse [85], etc.

Le choix d'une méthode dépend principalement de l'usage que l'on souhaite en faire : on ne fera pas le même choix pour étudier le typage, ou définir un compilateur, mais aussi de la culture de l'utilisateur (un théoricien aura sans doute plus de facilité à aborder des techniques mathématiques qu'un programmeur ou implémenteur). Le lecteur intéressé pourra approfondir cette étude en consultant notamment [151]. La sémantique naturelle et la sémantique opérationnelle sont quant à elles étudiées de manière plus approfondie dans la prochaine section.

2.2.2 Cadre formel de notre spécification de Java

Cette section établit le cadre formel de notre spécification de Java à travers une présentation rapide du système et des formalismes que nous avons utilisés.

Cette spécification s'est effectuée au sein du système Centaur [19], qui offre un cadre général pour la description formelle des langages. A partir des spécifications décrivant un langage de programmation, le système Centaur permet de générer un environnement de programmation pour ce langage. Les spécifications fournies pour cela regroupent des aspects syntaxiques (structures des programmes), des aspects de présentation des programmes (formattage de l'affichage ou *pretty-*

printing), ainsi que naturellement des aspects sémantiques. A partir de ces spécifications, on peut dériver des éditeurs structurés, des vérificateurs de type, des interprètes ou des compilateurs.

Le système Centaur fournit les formalismes suivants :

- Metal [77], utilisé pour les spécifications syntaxiques ; il permet de décrire les syntaxes concrète et abstraite du langage de programmation considéré, ainsi que la transformation de la représentation concrète (le texte du programme) en une représentation abstraite (un arbre de syntaxe abstraite).
- Ppml [98], pour les spécifications d'affichage ; il permet la transformation d'une représentation abstraite du programme en une représentation concrète.
- Typol [41], pour les spécifications sémantiques ; il permet de décrire des spécifications formelles en Sémantique Naturelle.

Les deux prochains paragraphes vont détailler la spécification des langages de programmation grâce à ces différents formalismes.

2.2.2.1 Syntaxe

Ce paragraphe décrit la spécification des syntaxes concrète et abstraite d'un langage de programmation grâce à Metal, pour ensuite revenir au formattage de l'affichage avec Ppml.

Syntaxe concrète

Le langage Metal permet la description de la syntaxe sous forme de grammaire non contextuelle.
Par exemple, l'affectation en Java est décrite en partie par les productions suivantes :

```
<Assignment> ::= <Name> <AssignmentOperator> <AssignmentExpression> ; (1)
  binaryAssign (<Name>, <AssignmentOperator>, <AssignmentExpression>) (2)
<AssignmentOperator> ::= "=" ; (3)
  assignment () (4)
.......
```

où le symbole *::=* sépare la tête de la production de sa définition.
Ces règles indiquent que lors d'une affectation (instruction considérée comme une expression en Java), on doit avoir successivement un terme `Name`, le symbole = et un terme `AssignmentExpression`. Les éléments `Name` et `AssignmentExpression` sont des non-terminaux qui restent à définir, alors que le symbole = est un terminal : son rôle est de signaler la présence d'une instruction d'affectation. L'ensemble des règles de ce type (règles (1) et (3)) forment la syntaxe concrète du langage.

Syntaxe abstraite

Le langage Metal donne aussi la possibilité d'indiquer l'arbre abstrait que l'on veut associer à une production de la grammaire non contextuelle décrivant le langage.
Considérons par exemple les règles (2) et (4) ci-dessus. La règle (2) indique que l'on veut construire un arbre nommé `binaryAssignment`, ayant pour fils un sous-arbre abstrait représenté par le terme `Name`, l'opérateur abstrait d'affectation représenté par le terme `AssignmentOperator` qui indique seulement le type d'opérateur binaire concerné par ces règles et un sous-arbre abstrait représenté par le terme `AssignmentExpression`.

La définition de la syntaxe abstraite est donnée par un ensemble de règles décrivant des schémas d'arbres. Voici la syntaxe abstraite associée à l'expression Java définissant un opérateur binaire restreint au cas de l'affectation :

```
Expression ::= binaryAssignment ... ;
binaryAssignment -> Name BinaryOperator AssignmentExpression ;
BinaryOperator  ::= assignment ... ;
assignment  -> implemented as SINGLETON ;
```

Les règles de la syntaxe abstraite sont composées de phyla, comme `Expression` et `BinaryOperator`, et d'opérateurs comme `binaryAssignment` et `assignment`. Les phyla représentent les catégories syntaxiques (les types du langage) par exemple les instructions et les expressions, alors que les opérateurs représentent des schémas d'arbres primitifs. Il y a des opérateurs d'arité fixe ou variable.
Dans la spécification de la syntaxe abstraite de Java, les termes commençant par une majuscule sont des phyla, alors que ceux commençant par une minuscule sont des opérateurs.

A partir de ces spécifications syntaxiques de Java, Centaur génère un analyseur syntaxique qui prend en entrée un programme Java sous forme textuelle et produit en sortie un arbre de syntaxe abstraite.

2.2.2.2 Sémantique

La conception de la sémantique naturelle a donné lieu à la définition du formalisme Typol, qui permet de spécifier une sémantique (statique ou dynamique) d'un langage de programmation.
L'implantation actuelle de Typol au sein de Centaur comprend un vérificateur de types, et un traducteur utilisant Prolog comme langage cible. L'approche suivie par Gilles Kahn et Thierry Despeyroux permet donc d'obtenir des spécifications exécutables. Au sein du système Centaur, la compilation des règles Typol fournit directement des fichiers Prolog exécutables.

Le formalisme Typol

Une spécification Typol est représentée par un ensemble de règles d'inférence, ces règles pouvant être structurées au sein de *sets* qui traitent d'un même objet. Nous présentons ici comme exemple le set `evaluateExpression` qui spécifie une sémantique de l'évaluation des expressions d'un langage très simple :

`set evaluateExpression is`

judgement Environment $\vdash$ Expression $\rightarrow$ Environment, Value;

`Assignment_Rule:`

$$\frac{Env \vdash Expr \rightarrow Env', Val \ \& \quad \text{assignValue}(Variable, Val \vdash Env' \rightarrow Env'')}{Env \vdash \texttt{assignment(Variable, Expr)} \rightarrow Env'', Val;}$$

provided getUserAction();

do updateWindow(Env");

`Binary_Expression_Rule:`

$$\frac{Env \vdash Expr1 \rightarrow Env', Val1 \ \& \quad Env' \vdash Expr2 \rightarrow Env'', Val2 \ \& \quad \text{applyBinaryOperator}(Val1, Val2 \vdash Op \rightarrow Val3)}{Env \vdash \texttt{binary(Expr1, Expr2, Op)} \rightarrow Env'', Val3;}$$

...

`end evaluateExpression;`

Chaque règle d'inférence est composée d'un ensemble fini de prémisses (qui est vide pour un axiome), et d'une conclusion. Les *prémisses* (au dessus de la ligne), et la *conclusion* d'une règle (en dessous de la ligne), sont des relations représentées par des séquents dans le style de la déduction naturelle de Gentzen. Les règles Typol indiquent comment un séquent peut être déduit d'autres séquents. Les objets du langage sont manipulés à travers leur arbre de syntaxe abstraite.

Un *séquent* exprime le fait que des hypothèses (la liste de termes qui est du côté gauche du symbole séquent) sont nécessaires pour prouver une propriété particulière, au sujet d'un terme de syntaxe abstraite appelé le *sujet* du séquent.
Dans l'exemple ci-dessus, le sujet de la règle appelée `Assignment_Rule` est le terme de syntaxe abstraite `assignment(Variable, Expr)` correspondant au terme de syntaxe concrète habituel `variable:=Expr`. Les séquents sont typés, en fonction de la nature syntaxique de leur sujet; ce type est défini dans un *jugement*, qui est introduit par le mot réservé *judgement*. Dans l'exemple ci-dessus, le jugement

était :

`judgement Environment ⊢ Expression → Environment, Value;`

Il exprime que les règles du set Typol que l'on est en train de définir prennent une hypothèse de type `Environment`, ont un sujet du type `Expression` et ont une comme conclusion deux termes de types respectifs `Environment` et `Value`.
Au sein d'un set, un séquent se trouvant en prémisse d'une règle se réfère au même set, à moins qu'un autre set ne soit explicitement indiqué par un séquent nommé (comme dans l'exemple avec la prémisse `assignValue`).

La clause *provided* définit une condition qui doit être vérifiée avant d'essayer d'appliquer la règle correspondante. Par exemple, dans le set `evaluateExpression` ci-dessus, la clause *provided* appelée `getUserAction` est appliquée en premier pour vérifier d'abord si l'utilisateur veut stopper (ou non) l'interprétation du programme. La clause *do* indique une action qui doit être exécutée si la règle correspondante est prouvée. Nous utilisons cette clause en particulier pour envoyer des notifications à l'environnement (par exemple `updateWindow`).

Sémantique naturelle et sémantique opérationnelle

La sémantique naturelle est une sémantique de type *big-step*, par opposition à une sémantique SOS qui est de type *small-step*, dans le sens où elle ne permet pas de capturer certains détails de l'exécution d'un programme, comme par exemple les états intermédiaires des calculs.

En effet, la sémantique naturelle adopte une vision *logique* des langages, au sein de laquelle un programme est assimilé à un théorème, son exécution à une preuve de ce théorème, et sa sémantique à la définition d'une logique. Par contre, une sémantique SOS adopte une vision plus *calculatoire* (plus traditionnelle pour les programmeurs) : une sémantique se rapproche alors plus de la description des transitions entre états d'un automate. Prenons comme exemple l'exécution d'une instruction `I` dans un état `S`. En sémantique big-step, cette exécution sera décrite par une règle de la forme :

$$\mathrm{S} \vdash \mathrm{I} \rightarrow \mathrm{S'}$$

Ce séquent exprime le fait que l'exécution de l'instruction `I`, dans l'état `S`, se termine en une seule étape et retourne l'état `S'`.

Dans le cadre d'un langage concurrent comme Java, nous ne recherchons pas une sémantique donnant l'état final de l'exécution, mais plutôt une sémantique décrivant les étapes élémentaires de cette exécution. La sémantique big-step n'est alors plus adaptée à cette partie de notre étude. L'exécution d'une instruction en small-step est décrite par une règle du type :

$$<\mathrm{S},\, \mathrm{I}> \rightarrow\, <\mathrm{S'},\, \mathrm{I'}>$$

Cette règle se traduit par :
Après une étape d'exécution de l'instruction I dans l'état S, on obtient l'état S' et il reste à exécuter l'instruction I'.
Cette instruction `I'` est alors appelée *continuation*.
En sémantique small-step, il est nécessaire de définir le niveau de granularité, c'est-à-dire de définir les "étapes élémentaires" d'exécution. Par exemple, on on peut considérer que l'opération `2 + 5` est élémentaire contrairement à l'opération `2 + (3 * 4)` dont l'évaluation nécessitera d'abord celle de `3 * 4`. C'est à ce niveau qu'apparaît le terme de *continuation*. Une continuation est une instruction qui représente ce qui reste à exécuter après une étape élémentaire de calcul. Il se peut que les instructions du langage spécifié ne suffisent pas à coder toutes les continuations désirées. Dans ce cas, de nouveaux opérateurs permettant de coder les continuations nécessaires sont introduits et leur sémantique est spécifiée. Ce mécanisme a été utilisé pour spécifier notre sémantique formelle de Java et a constitué une de ses difficultés.

Comme nous venons de le voir, la sémantique naturelle, si elle est parfaitement adaptée à la spécification des aspects objets de Java, n'est pas utilisable pour la spécification des aspects concurrents, et plus particulièrement pour la description des pas élémentaires d'exécution. Par contre, une sémantique small-step n'est pas nécessaire pour la description des aspects objets de Java.
Cela a été une des raisons de notre choix du formalisme Typol pour l'écriture de notre spécification formelle de Java. En effet Typol offre la possibilité de faire cohabiter la sémantique naturelle et une sémantique small-step au sein d'une même spécification.

Une sémantique exécutable

Notre sémantique formelle du langage Java est exprimée dans le formalisme Typol. Cette spécification se présente alors sous forme de fichiers Typol que l'on peut écrire, corriger et exécuter au sein du système Centaur. En effet, le formalisme Typol est spécifié au sein de Centaur comme nous spécifions Java. Tout fichier Typol correct du point de vue typage (Centaur propose un *type-checker* afin de s'en assurer) peut être compilé afin de devenir exécutable. La compilation traduit chaque fichier Typol en un fichier Prolog qui est alors exécutable par un interpréteur Prolog.
Cette traduction directe de Typol vers Prolog nous permet d'obtenir directement un interpréteur à partir de notre spécification formelle du langage Java. Cependant, cet interpréteur donne un résultat minimal : `yes` en cas de réussite dans l'interprétation du programme ou `fail` en cas d'échec. C'est pour cela que nous avons préféré développer un environnement d'interprétation dédié au langage

Java (présenté dans le Chapitre 4) proposant de nombreuses autres fonctionnalités et utilisant les différentes ressources offertes par Centaur afin de visualiser l'exécution des programmes. Cet environnement constitue un développement de l'environnement minimal d'interprétation proposé par Centaur.

Preuves en Coq

Un autre aspect important des spécifications formelles écrites en Typol au sein du système Centaur est la volonté d'utiliser cette base formelle pour le développement de preuves de propriétés de langages ou de programme. Delphine Terrasse [142, 143] a étudié une possibilité de traduction de spécifications en Sémantique Naturelle dans le système de preuves Coq [44].

2.3 Conclusion

Ce chapitre a présenté le langage Java, en insistant sur les principales difficultés que rencontre un programmeur débutant (découverte des objets et de leur création, de la concurrence), ou confirmé (mise au point de programmes concurrents). Le sous-ensemble de Java dont nous avons choisi de spécifier une sémantique comprend ces notions capitales, et prend notamment en compte la concurrence et les mécanismes liés à la synchronisation. Cette sémantique, mêlant sémantique naturelle (pour les aspects objets) et sémantique opérationnelle (pour les aspects concurrents), est directement exécutable au sein du système Centaur, et nous a permis ainsi de développer un environnement de visualisation de l'exécution de programmes Java.

Chapitre 3

Une Sémantique Formelle de Java

Le chapitre précédent a décrit les aspects principaux de Java, ainsi que les difficultés liées à la concurrence, à la synchronisation et aux objets partagés (comme le `CubbyHole` de l'exemple décrit en Section 2.1.3.2). Cette section s'attache à décrire et expliciter notre spécification sémantique de Java. Celle-ci s'appuie sur la description informelle de Java donnée par Sun dans le livre `The Java Language Specification` [61]. Il est important de noter que les règles Typol présentées dans ce chapitre afin d'expliciter certains aspects de notre spécification sont exactement celles utilisées, sans aucune modification ou simplification.

La sémantique dynamique permet de décrire l'exécution d'un programme Java, contrairement à une sémantique statique qui donne des informations au niveau du typage. L'exécution d'un programme Java construit et fait évoluer un ensemble d'objets ; nous avons donc considéré que le résultat de cette exécution était l'état final de cet ensemble d'objets. Notre spécification suppose d'autre part que le programme Java donné en entrée sous la forme d'un arbre de syntaxe abstraite est syntaxiquement correct (ce qui est assuré par le parseur), et correctement typé. Ce dernier point pourrait être vérifié par un vérificateur de typage (ou *type-checker*), basé sur la spécification d'une sémantique statique du langage Java, et qui pourrait être couplé à notre sémantique dynamique. Un tel vérificateur permettrait d'autre part de disposer d'informations sur les types statiques des variables, capitales au moment de la recherche de la méthode à appliquer, comme le type statique de la variable représentant l'objet sur lequel la méthode est appelée, ou le type statique des arguments.
Au lieu de construire des structures intermédiaires (tables de symboles ou mémoires d'un compilateur traditionnel), nous avons choisi d'utiliser directement la syntaxe abstraite du programme pour y chercher les informations nécessaires à son exécution. Nous ne construisons donc pas de tables de symboles pour mettre en mémoire le type des variables, ou le source d'une méthode. Cela évite un par-

cours du source du programme préalable à toute interprétation. A la place, nous parcourons directement l'arbre de syntaxe abstraite du source du programme en recherchant au sein de cet arbre les informations désirées.
Cette absence de table des symboles possède toutefois un inconvénient. En effet, cela nous oblige à parcourir le source du programme lors de chaque recherche, même si elle concerne le même attribut ou la même méthode plusieurs fois de suite, comme dans le cas d'un appel récursif de méthode, où l'on a besoin du source de la même méthode lors de chaque appel. Une table des symboles permettrait dans ce cas précis de n'effectuer cette recherche qu'une seule fois et de garder ensuite son résultat en mémoire.

Une première section décrit une syntaxe abstraite de Java ainsi que les structures sémantiques introduites en détaillant les différents choix effectués. Une deuxième section présente une vue d'ensemble de notre spécification sémantique de Java. La spécification des aspects objets de Java est étudiée au sein de la troisième section. Une quatrième section porte sur les différents mécanismes des exceptions alors que les aspects concurrents sont traités dans la cinquième section. Un sixième section présente alors une piste pour l'intégration du typage statique à travers une étude du masquage des attributs. Enfin, la dernière section permet de situer notre spécification par rapport aux différents travaux connexes.

3.1 Syntaxe abstraite et structures sémantiques

Cette section présente les bases de notre spécification sémantique, c'est-à-dire la définition d'une syntaxe abstraite pour les différents constructeurs du langage Java, ainsi que celle des différentes structures sémantiques introduites.

3.1.1 Spécifications syntaxiques

La spécification des aspects syntaxiques de Java inclut une syntaxe concrète et une syntaxe abstraite. Nous ne nous attachons ici qu'à la description de la syntaxe abstraite qui sert de base à notre spécification sémantique.

La syntaxe abstraite que nous avons définie est constituée de 128 opérateurs et 56 types. Elle couvre la totalité du langage Java dans sa version 1.2, sans se limiter aux aspects inclus au sein de notre sous-ensemble de Java. Nous avons choisi de coller aux unités syntaxiques décrites dans [61] afin de rendre notre spécification lisible pour le plus grand nombre. De plus, la description syntaxique fournie dans [61] était particulièrement adaptée au langage Metal dans lequel les spécifications syntaxiques sont tournées vers les outils d'édition structurée du

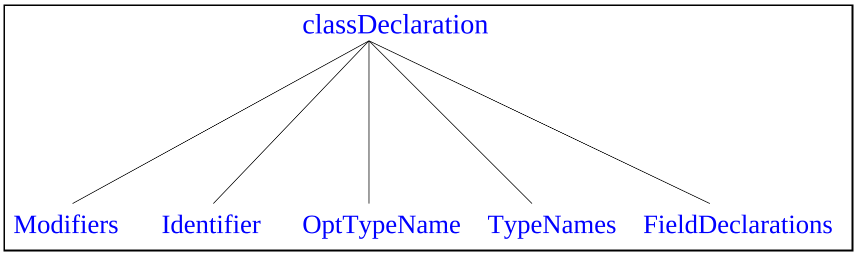


FIG. 3.1 – *Arbre de syntaxe abstraite d'une déclaration de classe.*

système Centaur.
Ce paragraphe présente la syntaxe abstraite des principaux composants d'un programme Java, afin de permettre une meilleure compréhension des règles sémantiques décrites dans les prochaines sections. C'est sous cette forme que les programmes Java sont conservés en mémoire au sein de Centaur, et c'est elle qui est la base de notre sémantique formelle.
Dans cette section, les noms de type (ou phylum) commenceront par des majuscules, alors que les noms des opérateurs commencent par une minuscule.

L'unité racine de la syntaxe abstraite d'un programme Java est l'opérateur `compilationUnit` dont le principal fils est composé d'une liste de déclarations de types (classes ou interfaces). Java étant un langage à objets, la classe est le composant essentiel de tout programme Java. La Figure 3.1 présente l'arbre de syntaxe abstraite associé à une déclaration de classe. Les éléments les plus importants sont : `Identifier` qui est le nom de la classe, `OptTypeName` qui représente le nom de la super-classe si elle existe, et `FieldDeclarations` qui est une liste de déclarations de membres de la classe.

Comme nous l'avons vu dans la Section 2.1.1, les membres d'une classe peuvent être :
- des attributs,
- des constructeurs,
- des méthodes.

L'arbre de syntaxe abstraite de la déclaration des attributs est représenté dans la Figure 3.2. Prenons l'exemple de la déclaration d'attribut suivante (sous forme de syntaxe concrète) :

```
public int longueur = 0;
```

La syntaxe abstraite correspondante est :

```
fieldVariableDeclaration(modifiers[public()], int(),
variableDeclarators[ intializedVariable(identifier "longueur", integer 0))]])
```

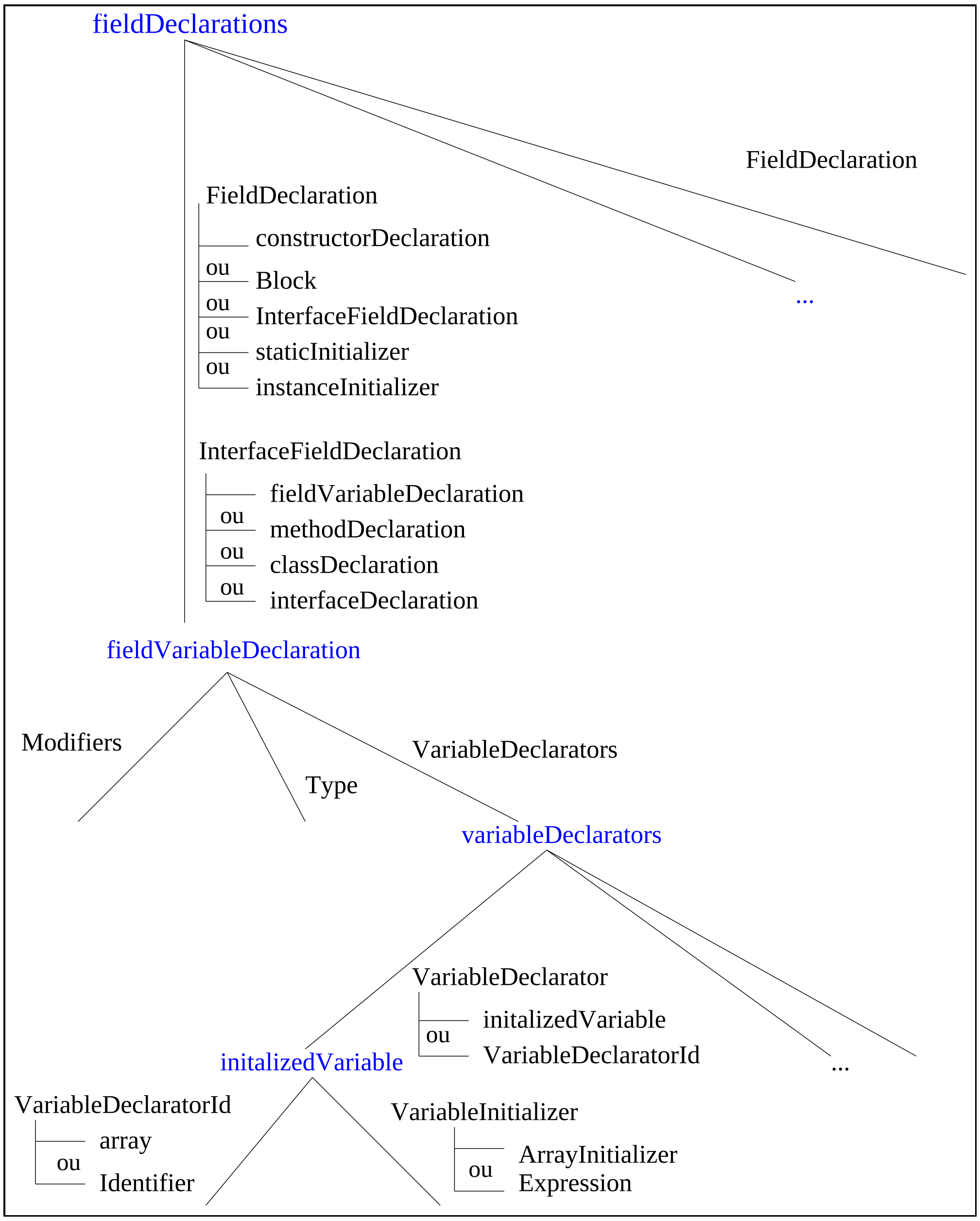


FIG. 3.2 – *Arbre de syntaxe abstraite d'une déclaration d'attribut.*

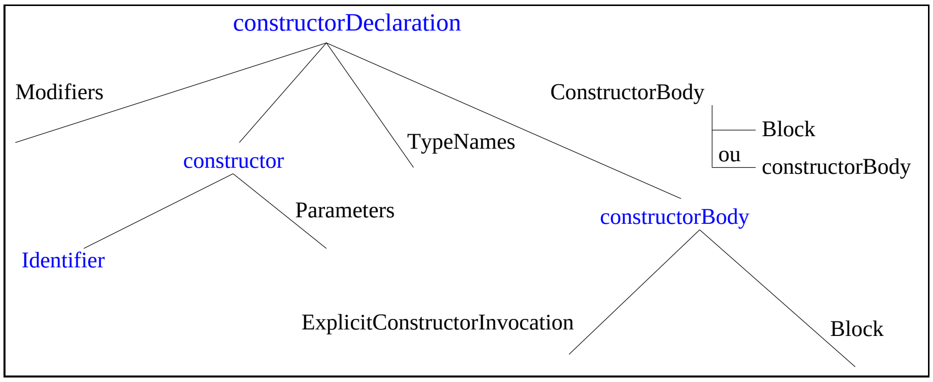


FIG. 3.3 – *Arbre de syntaxe abstraite d'une déclaration de constructeur.*

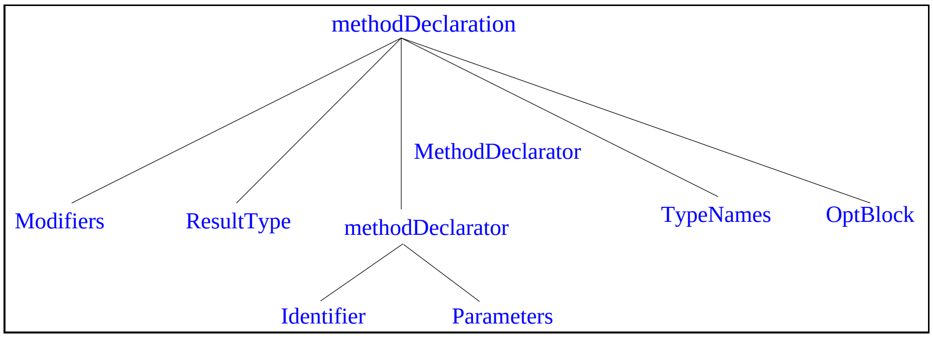


FIG. 3.4 – *Arbre de syntaxe abstraite d'une déclaration de méthode.*

L'arbre de syntaxe abstraite présenté en Figure 3.2 est complexe, bien plus que ne l'est la syntaxe concrète d'une déclaration d'attribut. Cela est dû à la présence de sous-typage représenté par les différentes alternatives proposées pour un type donné. Par exemple, le type *Interface Declaration* est un sous-type du type *InterfaceFieldDeclaration*, qui est lui-même un sous-type du type *FieldDeclaration*.

L'arbre de syntaxe abstraite d'un constructeur est présenté dans la Figure 3.3. Au sein de cet arbre on remarque notamment le nom du constructeur (`Identifier`) et ses paramètres (`Parameters`) tous deux fils de l'opérateur `constructor`, ainsi que la liste d'instructions du constructeur dans `ContructorBody`. Les deux possibilités d'arbres du type `ConstructorBody` tiennent compte de la possibilité de la présence d'un appel explicite d'un autre constructeur (de la même classe ou de la classe ancêtre) en tête de la liste des instructions.

La Figure 3.4 présente l'arbre de syntaxe abstraite d'une déclaration de méthode. Nous pouvons remarquer que cet arbre de syntaxe abstraite est assez proche de celui d'un constructeur, si ce n'est la présence d'un type de la valeur de retour `ResultType`

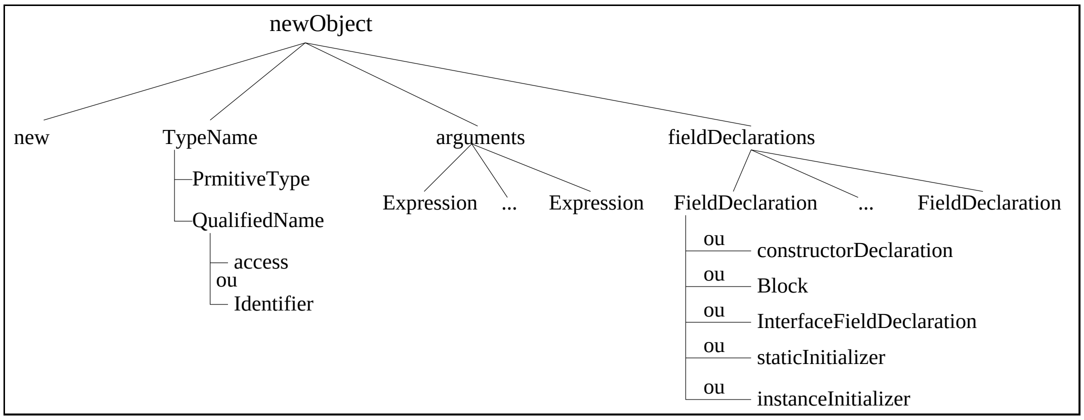


FIG. 3.5 – *Arbre de syntaxe abstraite d'une création d'objet.*

(qui est soit un type du programme soit le mot-clé *void*).

Chaque méthode comprend ainsi une liste d'instructions Java dont nous présentons ici un exemple de syntaxe abstraite.
Considérons la création d'un nouvel objet, dont la Figure 3.5 représente l'arbre de syntaxe abstraite la définissant.
Voici un exemple (sous forme concrète) d'expression de création d'objet :

```
new Producer(cubbyhole);
```

La syntaxe abstraite de cette expression est :

```
newObject(new(), identifier ‘‘Producer’’, arguments[identifier
        ‘‘cubbyhole’’], fieldDeclarations[])
```

L'intégralité de cette syntaxe abstraite est disponible en Annexe 1 (Section 6.2.2.3). Cette syntaxe abstraite des constructeurs du langage Java doit être complétée avec la définition des structures sémantiques et de leur syntaxe abstraite.

3.1.2 Structures sémantiques

Les structures sémantiques sont des arbres de syntaxe abstraite qui ne sont pas issus directement de l'analyse syntaxique des constructions du langage étudié. Ces structures (ou arbres sémantiques) sont introduites pour les besoins spécifiques de la définition sémantique du langage, ici pour les continuations dans le système de réécriture.

Ce paragraphe explicite la définition des différentes structures sémantiques que nous avons été amenés à introduire, ainsi qu'à la spécification de leur syntaxe abstraite. Ces structures sémantiques portent à la fois sur les objets et les threads, et sur les continuations nécessaires à la simulation de la concurrence sous forme d'entrelacement.

3.1.2.1 Objets et threads

Java étant un langage à objets, la première structure sémantique importante doit être un objet. Nous pouvons cependant nous demander quels sont les éléments définissant un objet, c'est-à-dire de quelles branches sera formé l'arbre abstrait représentant un objet.
Les informations devant figurer dans ce sous-arbre abstrait sont au minimum :

- Un identificateur permettant de le reconnaître au sein des autres objets.
- Un identificateur de la classe dont il constitue une instance afin de connaître son type.
- L'ensemble de ses attributs.

L'identificateur d'un objet est un entier (solution la plus simple) qui sert de référence lorsque l'on souhaite utiliser l'objet ou simplement le désigner. L'identificateur de la classe est son nom.
Cette liste de caractéristiques n'est pas spécifique au langage Java mais pourrait être reprise pour un autre langage à objets. Il faut de plus introduire des éléments propres au langage Java :

- Un attribut *locked* qui indique, dans le cas où l'objet est verrouillé, par quel thread et combien de fois il l'est.
- Un ensemble *wait* de threads qui vont vouloir le verrouiller à une certaine étape future de l'exécution.

Pour représenter les différents objets créés au cours d'un programme, la structure sémantique la plus simple est un ensemble d'objets (qui est implanté sous forme de liste). En effet, à chaque étape de l'exécution du programme, l'état de l'exécution est parfaitement décrit par l'ensemble courant des objets.
Nous devons choisir une structure pour cet ensemble d'objets qui réponde aux critères suivants :

- Conformité par rapport à l'optique de SUN selon laquelle les threads sont avant tout des objets comme les autres,
- Lisibilité de la sémantique, facilitant ainsi sa mise au point et sa compréhension par un lecteur non averti.
- Facilité de visualisation : la structure de liste choisie pour représenter l'ensemble des objets permet d'afficher directement cet ensemble sous forme textuelle à l'aide d'un formatage direct, sans aucune transformation de la structure par l'environnement.
- Efficacité : dans le cadre de notre sémantique qui est directement exécutable au sein du système Centaur, il est nécessaire pour que notre sémantique soit utilisable qu'elle soit efficace, permettant ainsi d'interpréter un programme Java en un temps raisonnable.
- Contrôle de l'entrelacement : au sein de notre sémantique, nous n'avons pas introduit de possibilité de contrôle sur l'entrelacement par l'utilisateur, mais cela serait facilement réalisable. Cela a d'ailleurs déjà été fait dans le cadre de la spécification d'Eiffel// [8].

Nous devons également trouver une structure sémantique pour représenterles threads. Plusieurs possibilités existent. Notre première alternative a été : faut-il considérer des structures séparées pour les threads et les objets "simples" ou doit-on les réunir au sein d'une même structure ?
Dans l'optique des fondateurs de Java, une chose est très importante : *un thread est avant tout un objet*. Notre structure doit donc mettre ce fait en évidence. C'est pour cela que nous avons décidé de représenter les threads et les objets "simples" dans la même structure. Nous différencierons simplement ces deux types d'objets par leur identificateur. Celui-ci sera un nœud ayant deux fils : un fils sera un booléen prenant la valeur *true* dans le cas d'un thread et *false* sinon, le second fils étant le véritable identificateur de l'objet au sein de la liste. Sur le plan visuel, lorsque nous affichons la liste des objets à l'écran, nous faisons seulement apparaître l'identificateur, en introduisant une différence textuelle : un thread a un identificateur du type *#Tnum*, alors que l'identificateur d'un objet "simple" est du type *#num*.

D'autre part, comment représenter les activités des threads au sein de cette structure sémantique ? Une nouvelle alternative se présente :

1. On peut représenter la structure comme un arbre sémantique ayant deux fils : un premier fils représentant les activités des threads sous forme d'une liste de paires identificateur-activité et un second fils représentant la liste de tous les objets en considérant les threads comme des objets "simples".
2. On peut aussi garder la structure sémantique décrite dans le paragraphe précédent en ajoutant au niveau des objets un élément définissant les différentes activités.

Ces deux propositions ont l'avantage de montrer de manière claire que les threads sont avant tout des objets. Cependant, la première proposition possède un important défaut : la structure obtenue peut être totalement déséquilibrée. En effet considérons le cas d'un programme Java n'utilisant qu'un thread mais créant successivement un grand nombre d'objets simples ; l'arbre sémantique obtenu aura alors une première branche menant à un seul élément alors que la seconde branche mènera à une liste d'un grand nombre d'objets.
Par contre, la deuxième structure a l'avantage d'être équilibrée. De plus si l'on considère la solution évidente concernant les activités, qui est de rajouter à chaque objet thread un constituant représentant son activité, la lecture de la structure sémantique sera simplifiée : elle ne nécessitera pas de se reporter à des références pour connaître les autres éléments définissant le thread considéré.

Il reste à définir précisément ce qu'est l'activité d'un thread. Cette activité doit rassembler d'une part l'état courant du thread qui sera utile au moment de l'ordonnancement et de l'entrelacement (un thread n'étant pas dans l'état *runnable* ne peut s'exécuter) et d'autre part les instructions à exécuter ainsi que l'environnement dans lequel elles doivent être exécutées. Ce dernier point (les instructions et leur environnement d'exécution) constitue une closure qui définit l'activité future du thread considéré. Chaque thread possède donc une closure représentant son activité future. A ce niveau

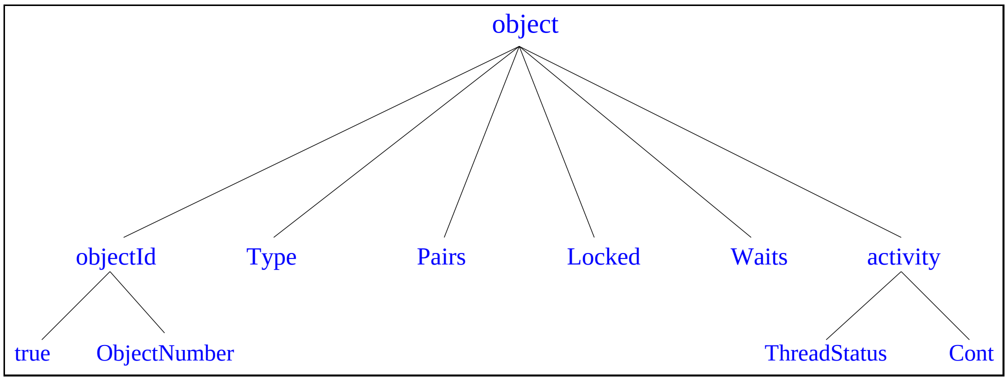


FIG. 3.6 – *Structure sémantique d'un thread.*

se pose encore une question : doit-on avoir une liste de closures (une pour chaque appel de méthode) ou doit-on considérer que la closure constituant un appel de méthode est en fait une instruction comme une autre pour le thread considéré ? Nous avons choisi cette dernière solution qui semble être plus en accord avec l'optique selon laquelle un appel de méthode est une instruction simple du langage Java. Cela évite d'autre part d'avoir à ajouter des instructions pour la communication d'une closure à la suivante.
Par contre, il est nécessaire de garder une trace de l'objet sur lequel la méthode a été appelée. Pour cela, nous avons défini une closure comme étant rattachée à un objet : l'identificateur de l'objet sur lequel s'applique la closure sera placé comme premier argument de la closure.

Voici la structure sémantique que nous avons retenue après cette étude : il s'agit d'un ensemble (implanté sous forme de liste) dans lequel seront rassemblés les threads et les objets simples.
La Figure 3.6 présente la structure sémantique correspondant à un thread. Dans le cas d'un objet simple, la seule différence par rapport à la structure sémantique d'un thread est que le champ *activity* prend la valeur *empty*. L'activité d'un thread est composée de son statut (*executing* ou *blocked* par exemple) et d'une continuation (ce que le thread doit exécuter). Cette continuation est constituée :

- de l'identifiant du thread
- du nom de la méthode courante
- d'un environnement d'exécution (paramètres et variables locales) sous forme de deux listes de paires (nomVariable, valeurVariable)
- d'une liste d'instructions représentant ce que le thread doit exécuter lors des prochains pas d'exécution. Cette liste contient à la fois des instructions du langage Java et des closures correspondantes aux appels de méthodes.

Comme on l'a vu plus haut, durant son exécution, un programme Java crée, utilise et met à jour des objets et des threads. Mais il faut aussi tenir compte des classes

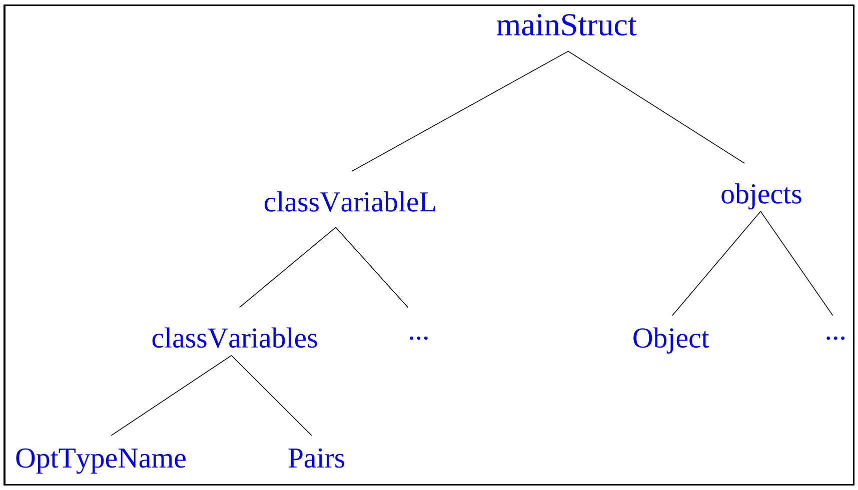


FIG. 3.7 – *Arbre de syntaxe abstraite de la structure sémantique principale.*

possédant des variables de classe ; ces dernières ne sont pas modélisées par la structure sémantique d'un objet, il faut donc les représenter par une nouvelle structure. La méthode la plus simple est de construire un ensemble de classes munies chacune de leur liste de variables de classe, en ne faisant figurer dans cette liste que les classes possédant des variables de classe (pour des raisons évidentes d'efficacité).
Afin de simplifier les règles sémantiques, nous avons décidé de réunir cette nouvelle structure avec celle des objets sous la forme d'un arbre dont les deux fils à la racine sont l'ensemble des classes possédant des variables de classes, et l'ensemble des objets. Cela nous permet ainsi de n'introduire qu'un seul paramètre dans les règles Typol ne manipulant pas ces deux structures, et de ne transmettre qu'un argument à l'environnement. La Figure 3.7 présente l'arbre de syntaxe abstraite obtenu. On y retrouve d'une part une liste de classes possédant des variables de classe sous forme de paires (nom de la classe-liste des variables de classe), et d'autre part la liste des objets.

3.1.2.2 Continuations

Des continuations, comme l'explique la Section 2.2.2.2, sont des constructeurs de syntaxe abstraite que nous sommes amenés à introduire afin de pouvoir simuler l'entrelacement entre threads lors de l'exécution de programmes Java concurrents. En effet, si les constructeurs propres au langage Java suffisent en sémantique big-step, ils sont largement insuffisants en sémantique small-step, où chaque instruction du langage Java est divisée en sous-instructions atomiques. Ce paragraphe présente un exemple de continuations introduites, afin d'expliciter notre propos.

Considérons l'exemple d'une affectation :

```
Variable = ExpressionGauche + ExpressionDroite
```

dont la syntaxe abstraite est

```
binaryAssign( Variable, assignment(), binary( ExpressionGauche, plus(),
                        ExpressionDroite))
```

La Figure 3.8 résume les différentes étapes de l'évaluation de cette affectation. D'après les spécifications informelles de Java, la première étape de cette affectation est l'évaluation de :

`ExpressionGauche + ExpressionDroite`

(notée 3 sur la Figure 3.8), suivie de l'affectation proprement dite (notée 4). Cela représente déjà deux étapes distinctes lors de l'évaluation de cette expression. De plus, l'évaluation de la somme :

`ExpressionGauche + ExpressionDroite`

se décompose elle aussi en trois étapes : calcul de `ExpressionGauche` (étape 1), calcul de `ExpressionDroite` (étape 2), suivi de l'évaluation de la somme proprement dite (étape 3). D'autre part, `ExpressionGauche` et `ExpressionDroite` peuvent elles aussi être des expressions complexes nécessitant plusieurs étapes d'évaluation, par exemple si ce sont des appels de méthode.

D'après notre politique d'ordonnancement (définie à la section 2.1.3.2), il y a interruption du calcul après chaque étape élémentaire afin d'introduire de l'entrelacement. Il faut donc à chaque reprise de l'exécution de cette affectation savoir exactement à quelle étape du calcul nous nous trouvons. De plus, suite à la décomposition du calcul, on peut se retrouver à une étape donnée avec une valeur définissant une référence (par exemple `#T4`), qui n'est pas prévue dans les constructeurs du langage Java. C'est pour cela que nous avons introduit des continuations, et donc que nous avons spécifié leur syntaxe abstraite.
Revenons à l'exemple ci-dessus de l'affectation. Pour pouvoir évaluer cette expression, nous avons défini principalement un opérateur *semAssign* et un type *SemExpression*. L'opérateur *semAssign* permet de traiter par exemple le cas de l'expression `Variable = #4` qui affecte une valeur référence à une variable, ou encore le cas de `#3.frere = #6`, qui affecte la valeur référence `#6` à l'attribut `frere` de l'objet `#3`.
Le type *SemExpression* est un type englobant à la fois le type *Expression* qui appartient au langage Java, et le type *Ref* (entre autres). Cela permet de représenter les valeurs références comme `#T3` par exemple.
Explicitons cette transformation sur l'exemple suivant :

$$\text{Simon.frere} = \text{Pascal}$$

La Figure 3.9 présente l'évaluation de cette expression à la fois sous forme de syntaxe concrète (pour être clair), et de syntaxe abstraite (pour observer l'introduction des continuations). Les continuations ont été introduites en particulier pour gérer les nouveaux éléments des structures sémantiques étudiées dans le paragraphe précédent, par exemple l'opérateur `objectId` qui représente la référence d'un objet (e.g. `#3`), le booléen indiquant s'il s'agit d'un thread (valeur *true()*) ou non.

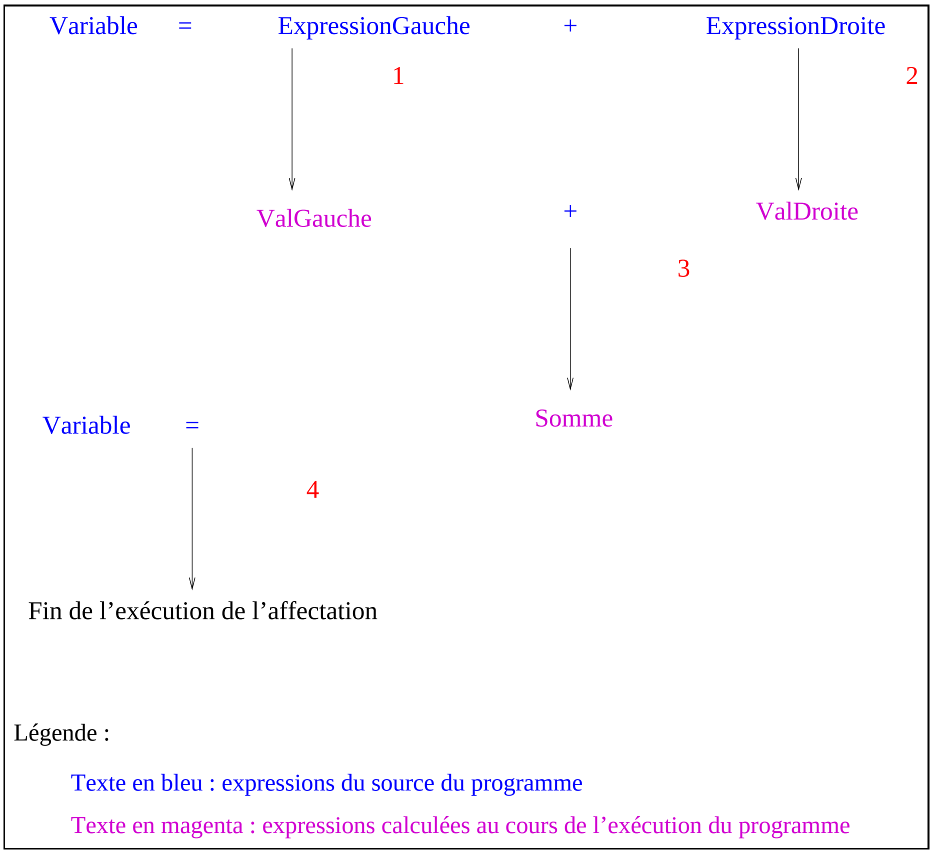


FIG. 3.8 – *Etapes de l'évaluation d'une affectation.*

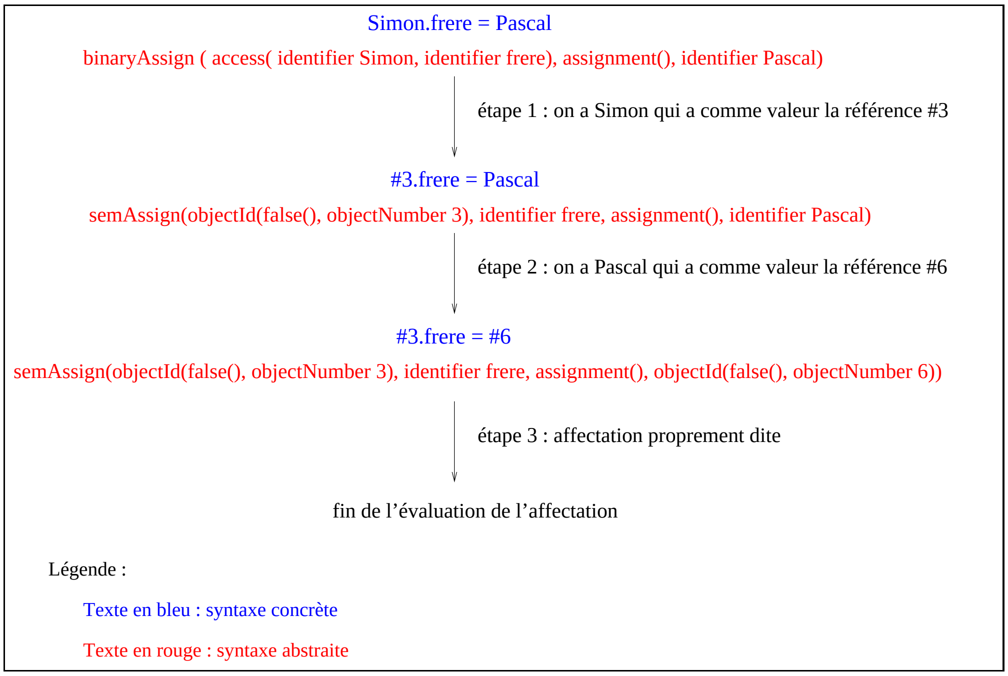


FIG. 3.9 – *Etapes de l'évaluation d'une affectation en syntaxe concrète et abstraite.*

L'intégralité de la syntaxe abstraite des structures sémantiques et des continuations introduites pour spécifier une sémantique de Java est disponible en Annexe 2.

3.2 Organisation générale

L'objectif de cette section est de donner une vue d'ensemble de notre spécification d'une sémantique formelle dynamique de Java. Les prochaines sections présentent de façon approfondie les principaux aspects de cette définition.

Cette section présente les différents modules dont est composée notre sémantique, et le système de transitions que nous avons développé.

3.2.1 Une spécification modulaire

Ce paragraphe présente les différents modules constituant notre spécification d'une sémantique formelle dynamique de Java.
Certains modules sont écrits en Sémantique Naturelle, principalement en ce qui concerne les aspects objets, par exemple la création d'un nouvel objet dont les différentes phases

(création, recherche des attributs, de l'activité s'il s'agit d'un thread) sont considérées comme une étape atomique au sein de notre sémantique. Les autres modules sont écrits dans un style opérationnel utilisant la Sémantique Naturelle (possibilité offerte par Typol), particulièrement en ce qui concerne les aspects liés à la concurrence comme l'exécution des instructions ou l'évaluation des expressions. Le style opérationnel nous permet ainsi d'introduire de l'entrelacement au niveau de l'exécution des threads et donc de simuler la concurrence.

Nous avons opté pour une spécification modulaire, nous rapprochant en cela de l'approche suivie par E. Börger et W. Schulte dans [17]. Cette approche nous a paru naturelle. En effet, Java est un langage qui permet de dissocier clairement des aspects forts :

- Les objets
- L'héritage
- La gestion des activités concurrentes
- L'exécution des instructions au niveau de chaque thread
- L'évaluation des expressions
- La présence de bibliothèques (ou API).

Il est important que la sémantique que nous définissons conserve cette modularité. En effet, cela facilite la mise au point de cette spécification, et introduit une plus grande lisibilité de notre sémantique. De plus, cela nous permet de plus de la développer de manière incrémentale, c'est-à-dire d'ajouter la spécification de nouveaux aspects du langage (comme par exemple des API), sans pour autant modifier les modules déjà définis (si ce n'est l'ajout d'une règle introduisant le nouveau module).
Elle doit de plus intégrer de nouveaux aspects :

- Le lancement de l'exécution
- La gestion de la structure sémantique introduite : la liste des objets
- La gestion de la communication avec l'environnement (voir le chapitre 4).

Ces différents aspects forment autant de modules au sein de notre définition d'une sémantique de Java.

Notre spécification se compose d'environ 800 règles Typol organisées en modules présentés dans la Figure 3.10. La partie gauche de la Figure présente les modules principaux, alors que la partie de droite regroupe les modules qui sont utilisés par les modules principaux.
Ces modules ont tous une fonction bien particulière :

- Modules "principaux" :
  - *java_program_interpretation.ty* : crée le thread principal (celui qui exécute la méthode *main* et qui est déclaré de façon implicite), lance l'interprétation du programme en modélisant le système de transitions (voir paragraphe suivant pour plus de détails), et achève l'interprétation.

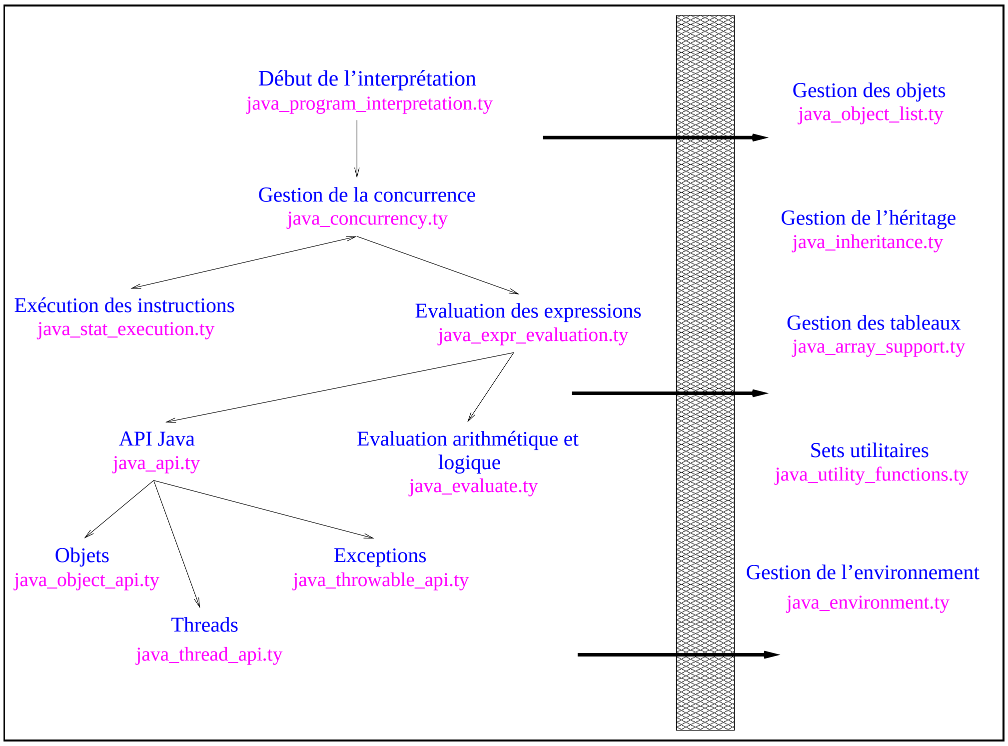


FIG. 3.10 – *Modules de la spécification sémantique de Java.*

  - *java_concurrency.ty* : détermine quel thread va s'exécuter au prochain pas d'exécution, et lance l'exécution d'une instruction atomique de ce thread.
  - *java_api.ty* : se charge de détecter quelle API est appelée et fait appel au module correspondant. Nous détaillons au sein de ce chapitre la spécification des méthodes des API liées à la concurrence et à la synchronisation.
- Modules "utilitaires" :
  - *java_object_list.ty* : crée les différents objets et threads, et gère les différents accès à la liste des objets que ce soit pour lecture ou pour modification.
  - *java_inheritance.ty* : traite les différents aspects liés à l'héritage, comme par exemple la détermination des attributs d'un objet, ou de la méthode appelée.
  - *java_environment.ty* : gère la transmission des différentes informations à l'environnement, par exemple pour mettre les différentes vues à jour après une modification de la liste des objets.

La spécification des principaux modules présentés ici est décrite dans les prochaines sections de ce chapitre.

3.2.2 Des commentaires structurés pour les règles Typol

Dans la suite de ce chapitre, nous employons au sein des sets et des règles Typol des commentaires suivant une syntaxe précise, dont voici la légende :

- `-SC` (*Set Comment*) représente un commentaire qui concerne le set Typol dans son ensemble, son utilisation et son action.
- `-JC` (*Judgment Comment*) représente un commentaire sur le jugement principal du set.
- `-iJC` (*internal Judgment Comment*) représente un commentaire sur un jugement interne au set, c'est-à-dire qui ne sera pas utilisé à l'extérieur du set.
- `-RC` (*Rule Comment*) représente un commentaire sur la règle figurant juste après le commentaire.
- `-CS` (*Concrete Syntax*) donne la syntaxe abstraite correspondant au sujet de la règle où il figure.
- `-PC` (*Premisse Comment*) indique un commentaire sur une prémisse d'une règle.
- `-c` (*comment*) représente un commentaire sur un aspect précis de la règle où il figure.
- `-BSC` (*Begin Structured Comment*) pour indiquer un commentaire portant sur un ensemble de règles (commentaire structuré), par exemple l'ensemble des règles définissant l'affectation. Cela permet d'écrire un commentaire global sur l'exécution de l'instruction concernée, et de pouvoir par la suite commenter chaque phase de son exécution en commentant chaque règle.
- `-ESC` (*End Structured Comment*) pour indiquer la fin d'un commentaire structuré.

Cette syntaxe permet de reconnaître au premier coup d'œil le style de commentaire que l'on est en train de lire et à quel élément (set, jugement, règle ou prémisse) il se rapporte.

D'autre part, on peut imaginer comme perspective de permettre la génération automatique de documentation à partir de la spécification grâce à ces commentaires.

3.2.3 Système de transitions

La Figure 3.11 présente un set Typol qui spécifie notre système de transitions. Les transitions entre les configurations (ou états) sont spécifiées avec des règles qui décrivent un pas d'exécution d'un thread donné. Ces transitions sont de la forme :

$$< ObjL0, ClVarL0 > \rightarrow < ObjL1, ClVarL1 >$$

où chaque configuration est composée d'un ensemble d'objets (*ObjL*) et d'un ensemble de classes avec leurs variables de classes (*ClVarL*).
Cette transition s'interprète comme suit :
Un système dans une configuration initiale $< ObjL0, ClVarL0 >$ *effectue un pas d'exécution et change sa configuration en* $< ObjL1, ClVarL1 >$.
L'exécution est alors modélisée par une séquence de transitions.

La configuration initiale est composée :

- D'un singleton : le thread principal, qui va exécuter la méthode *main*
- D'un singleton formé du nom de la classe contenant la méthode *main* (classe principale) et de ses variables de classe, ou d'un ensemble vide si la classe principale ne possède pas de variable de classe.

La configuration finale (lorsque l'interprétation du programme sera achevée) sera composée :

- D'un ensemble d'objets : c'est-à-dire de tous les objets créés au cours de l'exécution. Dans le cas d'une exécution "normale", sans levée non récupérée d'exception, tous les threads de cet ensemble seront dans le statut *dead*.
- D'un ensemble de noms de classes et des variables de classe correspondantes.

Ces éléments définissant les configurations (l'ensemble des objets, et l'ensemble des classes avec leurs variables de classes) sont donc présents au sein de la plupart de nos règles, en hypothèse pour la configuration initiale, et en conclusion pour la configuration finale de la règle.

Nous montrons dans la Figure 3.11 comment ces transitions sont exprimées en Typol.
La première règle (règle appelée *InitialisationOfTheInterpreter*) du set *interpretProgram* initialise l'exécution du programme : elle crée le thread principal (celui qui va exécuter la méthode *main*) qui devient *executing* et appelle la règle récursive (la règle appelée *RecursiveRule_InterpretOneStep*). Il est important de noter que le sujet de cette règle est constitué de l'opérateur `compilationUnit` qui est en fait l'arbre de syntaxe abstraite correspondant au texte du programme à interpréter. Ce programme est alors stocké dans une variable globale, ce qui permettra de le réutiliser au cours de l'interprétation, par exemple quand on recherchera le source d'une méthode, sans pour autant le faire figurer au niveau des hypothèses au sein de chaque séquent.

```
set interpretProgram is
--SC: Loads the classes (retrieves the class variables), creates the "main"
--SC: thread (the thread which executes the "main" method) and launches the
--SC: program interpretation.
--
--JC: |- Compilation Unit (i.e. the program abstract syntax tree)
--JC: -> Final list that contains all the objects created during this
--JC: interpretation, Final class variable list.
judgement |- CompilationUnit -> Objects, ClassVariablesL;
--iJC: Objects, Class variables
--iJC: -> updated object list, updated class variable list
judgement (Objects, ClassVariablesL -> Objects, ClassVariablesL);

initialisationOfTheInterpreter:
    loadClasses(TDecL -> ClVarL1)  &  getMainThread(|- TDecL -> Thr)  &
    setFirstExecutingThread(objectId(true(), objectNumber 0))  &
    (objects[Thr], ClVarL1 -> ObjL, ClVarL1_1)
    ----------------
    |- compilationUnit(PackName, ImportL, TDecL) -> ObjL, ClVarL1_1 ;
            provided
                ENVgetUserAction()  &  initialize()  &
                setCompilationUnit(compilationUnit(PackName, ImportL, TDecL))
                &
                setNewCompilationUnit(compilationUnit(
                                        PackName, ImportL, TDecL));

RecursiveRule_InterpretOneStep:
    getTheNewExecutingThread(ClVarL1 |- ObjL1 -> ObjL1_1, OThId)  &
    executeThread(ObjL1_1, ClVarL1, OThId -> ObjL1_2, ClVarL1_1)  &
    (ObjL1_2, ClVarL1_1 -> ObjL1_3, ClVarL1_2)
    ----------------
    (ObjL1, ClVarL1 -> ObjL1_3, ClVarL1_2) ;
            provided ENVgetUserAction();

DeadLock:
    --RC : No thread can execute itself but there is at least one alive thread.
    --RC : Deadlock.
    areAllThreadsDead(ObjL -> false())
    ----------------
    (ObjL, ClVarL -> ObjL, ClVarL) ;
            do deadlock()  &  ENVfreeEnvironment();

InterpretationEnd:
    --RC : All threads are dead => end of the program interpretation.
    areAllThreadsDead(ObjL -> true())
    ----------------
    (ObjL, ClVarL -> ObjL, ClVarL) ;
            do ENVfreeEnvironment();
end interpretProgram;
```

FIG. 3.11 – *Notre système de transitions en Typol.*

Dans cette deuxième règle, le set *getTheNewExecutingThread* détermine de manière déterministe un thread à exécuter, l'appel du set *executeThread* effectue alors un pas d'exécution du thread précédemment choisi, et enfin le set courant (dont le nom est dans ce cas implicite) est appelé récursivement.
Les deux dernières règles correspondent à la fin de l'exécution, soit parce qu'il y a *deadlock*, c'est-à-dire qu'aucun thread ne peut plus s'exécuter, mais qu'au moins un des threads n'a pas fini son exécution, soit parce qu'il n'y a plus de thread à exécuter, la configuration considérée est alors la configuration finale (elle est renvoyée sans être modifiée).

La modularité de notre spécification permet comme on l'a vu dans cette section de préserver les différents concepts de Java, et favorise par ailleurs une bonne compréhension de notre sémantique. La présentation du système de transitions a posé les bases de notre spécification sémantique dont les principaux aspects sont développés dans les sections suivantes.

3.3 Objets, héritage et appels de méthodes

Cette section s'intéresse plus particulièrement à la spécification sémantique des aspects reliés à l'orientation objet. Les aspects concurrents seront traités dans la section suivante.
Même si cette section ne concerne pas la concurrence, il est important de garder à l'esprit certains impératifs dus à cette dernière. En effet, même si notre spécification est modulaire du point de vue de l'ajout de la spécification de nouvelles notions, cette modularité ne permet pas de passer d'un style sémantique (comme la Sémantique Naturelle) à un autre (comme la Sémantique Opérationnelle). Ce passage demande une modification complète des règles : pour passer d'une sémantique *big-step* à une sémantique *small-step*, il est nécessaire de fractionner chaque exécution d'instruction. De même, l'interprétation d'instructions "simples" du langage Java peut donner lieu à la création de continuations afin de permettre une simulation de la concurrence par un entrelacement des threads. Il faut donc prévoir de spécifier le langage directement en sémantique *small-step*, même si dans un premier temps, les aspects totalement concurrents ne sont pas définis.

Les caractéristiques principales des langages à objets sont la présence d'objets, la notion d'héritage entre classes et les appels de méthodes. Ces différents aspects sont successivement étudiés au sein de cette section.

3.3.1 Héritage

Comme cela est présenté dans la Section 3.2.1, un module de notre spécification est consacré à la gestion de l'héritage. Plus précisément, nous nous intéressons au parcours du graphe d'héritage afin de déterminer principalement :

- Les attributs d'un objet,

– La méthode à appeler et son code source.

La recherche de la hiérarchie d'héritage entre les différentes classes du programme se fait directement à partir du source de celui-ci (sous forme d'un arbre de syntaxe abstraite). En effet, nous avons choisi de ne pas stocker de structures intermédiaires (par exemple une table des symboles) afin d'éviter d'avoir à effectuer une passe "préparatoire" sur le programme avant de pouvoir lancer véritablement son interprétation. Nous ne construisons donc pas ce graphe mais le reconstituons en parcourant le code du programme. Cependant, cela présente un inconvénient, cela nous oblige à refaire parfois plusieurs fois la même recherche, surtout lors d'un appel récursif de méthode. Dans ce dernier cas, notre sémantique effectue cette recherche à chaque appel, étant donnée l'absence de structure mise en mémoire.
La condition d'arrêt lors de toute remontée dans le graphe d'héritage est soit l'obtention de la structure recherchée (par exemple une classe, ou une méthode), soit l'arrivée au niveau de la classe *Object* qui constitue la racine de tout graphe d'héritage. Dans ce cas, la recherche continue au sein des différentes API. Si elle échoue encore, cela signifie qu'il y a une erreur au niveau du programme car l'attribut ou la méthode recherché n'a pas été trouvé. Ce cas n'est pas sensé se produire car nous prenons comme hypothèse que les programmes à interpréter sont corrects du point de vue de la sémantique statique.
Cette recherche est liée au typage statique : selon le type statique de la variable référençant un objet, cela peut modifier la recherche d'un attribut de cet objet. Notre sémantique ne prend pas en compte cet aspect statique : nous avons en effet choisi de nous concentrer sur les aspects purement dynamiques du langage, en particulier sur la concurrence. Certains mécanismes liés au typage statique des variables sont d'ailleurs assez peu utilisés comme le masquage des attributs (ce dernier point est cependant étudié dans la Section 3.6). Par contre nous avons spécifié la surcharge des attributs et des méthodes.

La Figure 3.12 présente le principal set (appelé `getFieldDeclaration`) permettant de retrouver un membre donné d'une classe à partir de la liste des objets (*ObjL*), du nom de la classe (*ClName*) dans laquelle on va le chercher, d'une liste éventuelle d'arguments (*ArgL*) (dans le cas d'une recherche de méthode), et bien entendu du nom du membre (*FName*) à retrouver. Ce set comporte deux règles, qui représentent en fait les deux cas pouvant se produire : soit le membre cherché se trouve dans la classe courante (première règle), soit il faut le chercher dans l'ancêtre de cette classe (deuxième règle). Ces deux règles présentent des points communs. En effet, les deux premières prémisses sont quasiment identiques. Cela est du à une caractéristique de Typol : on ne doit pas tenir compte de l'ordre des règles dans leur écriture. C'est à nous d'assurer l'exclusivité entre les règles Typol d'un même set. Cela possède un inconvénient majeur : la deuxième règle doit vérifier qu'elle s'applique bien en exécutant les deux prémisses de la première règle (en attendant un échec) alors qu'elles viennent de conduire à un échec lors de la tentative d'application de la première règle.
Cela complique l'écriture des règles si l'on souhaite éviter cet écueil (qui se traduit par un allongement de la durée d'interprétation d'un programme). Cela est dû à l'absence

```
set getFieldDeclaration is
--SC: Retrieves the field declaration for the given field. Returns false if
--SC: it is not found.
--
--JC: Objects, Current class name where the fields is sought, Field arguments
--JC: (useful to find the right method or constructor declaration),
--JC: True (for a constructor declaration)/ False (for an attribute or method
--JC: declaration)
--JC: |- Field name -> Field declaration, Name of the class that owns this field (can be
--JC: a different name from the given one. In this case, it is
--JC: an ancestor), True if found/False otherwise.
judgement
    Objects, Identifier, Vals, Bool |- Identifier -> FieldDeclaration, Identifier, Bool;

FieldInTheGivenClass:
    getClassDeclaration(|- ClName -> ClDec)  &
    getFieldDeclarationInClass(ObjL, FName, ArgL, TF |- ClDec -> FDec, true())
    ----------------
    ObjL, ClName, ArgL, TF |- FName -> FDec, ClName, true() ;

FieldInAnAncestorNotInTheGivenClass:
    getClassDeclaration(|- ClName1 -> ClDec)  &
    getFieldDeclarationInClass(ObjL, FName, ArgL, TF |- ClDec -> _, false())  &
    getDirectAncestor(|- ClName1 -> Ancestor)  &
    getFieldDeclarationInAncestors(ObjL, Ancestor, ArgL, TF
                                       |- FName -> FDec, ClName2, true())
    ----------------
    ObjL, ClName1, ArgL, TF |- FName -> FDec, ClName2, true() ;
end getFieldDeclaration;
```

FIG. 3.12 – *Gestion de l'héritage lors de la recherche d'un membre d'une classe.*

du phénomène de coupure qui interdirait de tenter d'appliquer la seconde règle si la première s'appliquait.

La deuxième règle est la plus intéressante du point de vue de l'héritage. C'est donc elle que nous allons analyser ici. Tout d'abord, la première prémisse appelle un set (`getClassDeclaration`) permettant d'extraire la déclaration de la classe (au sein de laquelle le membre va être recherché) de l'arbre de syntaxe abstraite. Puis on utilise le set `getFieldDeclarationInClass`, qui effectue la recherche proprement dite.
Le set `getFieldDeclarationInClass` analyse la déclaration de la classe donnée en parcourant la liste de ses membres et de ses constructeurs afin de retrouver le membre cherché. La condition d'arrêt de cette recherche est soit son succès, soit l'arrivée à la fin de la liste des membres et des constructeurs. Le *false()* que l'on trouve en conclusion de cette seconde prémisse permet d'indiquer qu'en fait le membre recherché n'a pas été trouvé : si la conclusion renvoyée par l'appel de ce set vaut *true()*, la règle conduira directement à un échec. On remonte alors dans le graphe d'héritage pour trouver l'ancêtre de la classe courante (à l'aide du set `getDirectAncestor`), et on poursuit la recherche dans la classe trouvée ainsi que dans ces ancêtres si cela est nécessaire (set `getFieldDeclarationInAncestors`).

Il est intéressant de signaler que, contrairement à d'autres langages comme Eiffel, la notion d'héritage en Java est simplifiée par l'absence d'héritage multiple. Cela permet d'éviter de parcourir de multiples branches du graphe d'héritage lorsque l'on recherche successivement tous les ancêtres d'une classe donnée par exemple. Cependant, les règles détaillées dans cette section seraient très facilement utilisables dans le cas d'un langage avec héritage multiple : il suffirait d'ajouter des règles permettant de parcourir les différentes branches de l'arbre d'héritage, et d'utiliser au sein de chacune les sets déjà définis.
En ce qui concerne les interfaces, dans le cadre de notre sémantique dynamique nous ne nous intéressons qu'aux déclarations concrètes d'éléments (attributs ou méthodes) afin d'obtenir, surtout dans le cas des méthodes, le code correspondant, et non aux déclarations abstraites. Les recherches de ces éléments ne se font donc qu'au sein du graphe d'héritage des classes.

3.3.2 Création des objets

La création d'un objet est un élément essentiel dans le cadre des langages à objets. Avant même le début de l'exécution des instructions du programme, elle intervient dans le cadre de la création du thread principal, qui est créé de manière implicite pour exécuter la méthode *main*.
En Java, l'instruction de création d'un objet s'écrit *new ClassName()*, et on la retrouve le plus souvent dans la forme *newObject = new ClassName()* qui permet d'associer une variable (*newObject*) à l'objet ainsi créé. Il s'agit d'une expression complexe qui s'exécute en fait en deux étapes successives (chacune pouvant prendre plusieurs pas d'exécution

```
NewObjectCreation:
    --RC: Creates an instance of the given class and adds this instance to the
    --RC: object list that contains all the objects created by the current
    --RC: interpreted program. The next execution step of the current
    --RC: executing thread will be the constructor call.
    --
    --CS: new Thread();
    generateANumberForTheNewObject(|- ObjL1 -> i)  &
    createObject(OThId |- i, TVClName -> Obj, ObjId)  &
    appendtree(ObjL1, objects[Obj], ObjL1_1)
    ----------------
    ObjL1, ClVarL, Env, OThId, _, _
        |- newObject(new(), TVClName, ArgL, fieldDeclarations[]) ->
            ObjL1_1, ClVarL, Env, ObjId,
            insts[constructorCall1(TVClName, ArgL, ObjId)], runnable() ;
```

FIG. 3.13 – *Création d'un objet (règle générale).*

du thread courant) :

1. Création de l'objet (expression *new ClassName()*) : elle s'effectue en trois sous-étapes
   (a) Détermination de la référence interne de l'objet à créer : c'est le nombre d'objets présents dans la liste d'objets avant sa création.
   (b) Création proprement dite de l'objet
   (c) Appel ascendant des constructeurs.
2. Affectation de la référence de l'objet créé (qui pour l'instant n'est pas accessible) à la variable `newObject` (expression *newObject = ...*).

La première étape (1) est spécifiée par la règle Typol présentée en Figure 3.13. Les deux premières sous-étapes (a et b) sont représentées par les deux premières prémisses de la règle (appels des sets `generateANumberForTheNewObject` et `createObject`), la troisième prémisse est l'appel d'une primitive Prolog (*appendtree*) permettant d'ajouter le nouvel objet à la liste des objets déjà créés. La troisième sous-étape (c) est indiquée sous forme de continuation par la partie grisée de la règle. Cet élément de la conclusion de la règle représente la liste des instructions à ajouter en tête de l'activité du thread courant. Elle indique que lorsque le thread courant s'exécutera à nouveau, sa première action sera l'appel `constructorCall1(TVClName, ArgL, ObjId)` du constructeur correspondant à la classe `TVClName` dont l'objet créé `ObjId` est une instance. On parle d'appel ascendant, car cet appel va remonter dans le graphe d'héritage de cette classe afin de commencer par appeler le constructeur de la racine du graphe d'héritage pour redescendre successivement jusqu'à la classe considérée.

La deuxième étape (2) est exécutée après les appels successifs aux constructeurs. Il s'agit d'une simple affectation.

```
SimpleObjectCreation_GetAttributeList:
   makeAttributeList(|- ClName -> AttrL)
   ----------------
   (ObjId, ClName -> object(ObjId, ClName, AttrL, unlocked(), waits[], empty())) ;
```

FIG. 3.14 – *Création d'un objet simple.*

Revenons à la création proprement dite de l'objet (étape 1, sous-étape b). Celle-ci s'exprime en sémantique naturelle par la règle Typol présentée en Figure 3.14, ce qui signifie que cette création s'effectue de façon atomique, en un seul pas d'exécution.
Nous avons été amenés à considérer trois catégories d'objets :

- Les objets simples : qui ne sont pas des threads.
- Les threads : qui sont des objets ayant une activité propre décrite dans la méthode *run* de la classe dont ils sont une instance.
- Le thread principal : qui est créé de façon implicite afin d'exécuter la méthode *main*.

La règle présentée ne porte que sur la création d'un objet simple. Le cas des threads sera traité en Section 3.5.

Une phase de la création est commune à ces trois catégories d'objets, il s'agit de la détermination de l'ensemble des attributs de l'objet. Cette phase (qui est l'unique phase lors de la création d'un objet simple comme on le voit sur le Figure 3.14), présentée en Figure 3.15, fait appel à un parcours du graphe d'héritage afin de déterminer l'ensemble complet des attributs (hérités et non hérités). Le set `makeAttributeList` contient deux règles, la première traite les objets instances d'une classe du programme, la deuxième quant à elle gère le cas des objets instances d'une classe d'une bibliothèque. Considérons la première règle. Le sujet de la règle est le nom de la classe dont l'objet est une instance. La première étape consiste à récupérer la déclaration de celle-ci (appel du set `getClassDeclaration` qui parcourt l'arbre de syntaxe abstraite du programme à la recherche de la bonne classe). L'ensemble des attributs locaux à cette classe est alors déterminé lors de l'appel du set `getLocalAttributeList`. Le set `getDirectAncestor` permet d'obtenir l'ancêtre direct (ou super-classe) de la classe courante par remontée du graphe d'héritage. Le set `makeInheritedAttributeList` se charge alors de déterminer l'ensemble total des attributs de l'objet à créer en parcourant le graphe d'héritage. L'ensemble des attributs locaux à une classe est obtenu en analysant tous les composants (membres et constructeurs) de la classe. Plusieurs cas peuvent se présenter :

- déclaration de méthode ou de constructeur : passage au composant suivant
- déclaration d'une variable de classe (mot-clé *static*) : passage au composant suivant
- déclaration d'un attribut : création de la structure sémantique correspondante avec comme valeur initiale :
  - attribut sans valeur d'initialisation : la connaissance de son type permet

```
set makeAttributeList is
--SC: Determines the attribute list correponding to a given type (useful for
--SC: an object creation).
--
--JC: |- Class name of the new instance
--JC: -> Initialized Attribute list
judgement |- OptTypeName -> Pairs;

DeclarationObject_InProgramClasses:
   getClassDeclaration(ClName1 -> ClDec)  &
   getLocalAttributeList(|- ClDec -> LocAttrL)  &
   getDirectAncestor(|- ClName1 -> ClName2)  &
   makeInheritedAttributeList(LocAttrL |- ClName2 -> AttrL)
   ----------------
   |- ClName1 -> AttrL ;

DeclarationObject_inAPI:
   getAPIObjectAttributes(|- ClName -> AttrL)
   ----------------
   |- ClName -> AttrL ;
end makeAttributeList;
```

FIG. 3.15 – *Obtention de l'ensemble des attributs d'un objet.*

d'attribuer une valeur par défaut

- attribut avec valeur d'initialisation : celle-ci prend la place de la valeur par défaut.

Cette phase d'obtention de l'ensemble des attributs est commune à tous les objets.

Un tel mécanisme de spécification de la création des objets, même s'il est spécifique au langage Java, est facilement adaptable à d'autres langages à objets en ce qui concerne l'obtention des attributs de l'objet, et concurrents en ce qui concerne la création des threads. Si l'on considère le langage Eiffel// [8], celui-ci possède aussi un objet actif principal exécutant la routine *create* et des objets actifs autres, ainsi que des objets inactifs. On reconnaît là les trois types d'objets que nous distinguons dans notre spécification du langage Java.

3.3.3 Appels de méthodes

Java étant un langage à objets, les appels de méthodes constituent un point clé des programmes après celui de la création proprement dite des objets.
L'exécution d'un appel de méthode (qui est considéré comme une expression en Java) peut être fractionnée en quatre étapes (chacune pouvant représenter plusieurs pas d'exécution du thread courant) :

1. Détermination de l'objet sur lequel la méthode est appelée

2. Détermination de la valeur des arguments (ils sont évalués les uns après les autres de gauche à droite)
3. Détermination de la méthode à appeler et construction de la closure correspondante
4. Exécution proprement dite de la méthode.

Cette section va étudier successivement la spécification de ces quatre étapes (nous nous restreignons ici aux méthodes d'instance, qui porte sur un objet par opposition aux méthodes de classe).

3.3.3.1 Détermination de l'objet

La détermination de l'objet sur lequel la méthode est appelée peut paraître simple si l'on considère les appels de base comme `obj.MethodName(ArgL);` où `obj` est une variable référençant un objet. Par contre, cela peut s'avérer complexe dans le cas d'un appel de méthode construit sur le modèle :

```
obj.MethodName1(ArgL1).MethodName2(ArgL2)...MethodNameN(ArgLN);
```

Nous avons donc été amenés à écrire des règles générales permettant de traiter ces différents cas. Le mécanisme principal en est le suivant : on remplace l'expression :

```
obj.MethodName1(ArgL1).MethodName2(ArgL2)...MethodNameN(ArgLN);
```

par la liste d'expressions :

```
obj.MethodName1(ArgL1);
7RETURN.MethodName2(ArgL2)...MethodNameN(ArgLN);
```

Dans cette liste d'expressions, `7RETURN` représente une variable interne à notre sémantique (une variable Java ne pouvant commencer par un chiffre, nous sommes sûrs de ne pas créer de conflits) utilisée pour représenter la valeur de retour d'une méthode. La première expression :

```
obj.MethodName1(ArgL1);
```

réalise l'appel de la méthode `MethodName1` sur l'objet `obj`. La valeur de retour de cette méthode sera affectée de façon interne à notre sémantique à la variable `7RETURN`. Cette variable sera à son tour évaluée lors de l'évaluation de l'expression :

```
7RETURN.MethodName2(ArgL2)...MethodNameN(ArgLN);
```

afin de déterminer sur quel objet doit être appelée la méthode `MethodName2`.
L'expression :

```
obj.MethodName1(ArgL1).MethodName2(ArgL2)...MethodNameN(ArgLN);
```

et la liste d'expressions :

```
obj.MethodName1(ArgL1);
7RETURN.MethodName2(ArgL2)...MethodNameN(ArgLN);
```

sont donc équivalentes du point de vue de l'exécution.

Ce mécanisme est mis en oeuvre par la règle Typol présentée en Figure 3.16.
Cette règle s'applique lorsque le sujet est de la forme :

```
call(access(call(Expr1, ArgL1), MethodName), ArgL2)
```

```
MethodCall_TargetEvaluationWithAnotherMethodCall:
    --RC: Before executing the given method, the target object represented
    --RC: here by a method call must be retrieved. Therefore this method call
    --RC: is evaluated. Moreover, a special variable called 7RETURN is added to
    --RC: the local variable list in order to handle the value returned by
    --RC: this method.
    --
    --CS: myObject.bar().setAttrValue(var2 - var1);
    evaluateExpression(ObjL1, ClVarL1, Env, OThId, ObjId, Mode
                            |- call(Expr1, ArgL1) ->
                                    ObjL1_1, ClVarL1_1, env(ParamL1, LocVarL1), _,
                                    InstL1, ThStatus)   &  diff(InstL1, insts[])   &
    appendtree(InstL1, insts[call(access(identifier "7RETURN", MethName), ArgL2)],
                InstL1_1)   &
    addReturnVariableInLocalVariableList(LocVarL1 -> LocVarL1_1)
    ----------------
    ObjL1, ClVarL1, Env, OThId, ObjId, Mode
        |- call(access(call(Expr1, ArgL1), MethName), ArgL2) ->
            ObjL1_1, ClVarL1_1, env(ParamL1, LocVarL1_1), identifier "7RETURN",
            InstL1_1, ThStatus ;
```

FIG. 3.16 – *Règle générale des appels de méthodes.*

c'est-à-dire possède une syntaxe concrète de la forme :

`Expr(ArgL1).MethodName(ArgL2).`

La première prémisse de la règle figurant dans la Figure 3.16 permet d'effectuer un pas d'évaluation de l'expression `call(Expr1, ArgL1)`, la troisième de concaténer la liste d'instructions restant à exécuter obtenue lors de cette évaluation et l'instruction :

`call(access(identifier “7RETURN”, MethodName), ArgL2)`

et la dernière de vérifier que l'on a bien ajouté la variable `7RETURN` à la liste des variables locales. Le prochain pas d'exécution du thread courant permettra d'effectuer un pas d'évaluation de l'expression `call(Expr1, ArgL1)`, et cela continuera jusqu'à l'obtention d'une valeur qui sera affectée à la variable `7RETURN`. Le prochain pas d'interprétation sera alors l'exécution de l'expression :

`call(access(objectId(Bool, objectNumber Nb), MethodName), ArgL2)`

où `objectId(Bool, objectNumber Nb)` représente la référence affectée à la variable `7RETURN` (le booléen indique si l'objet est un thread, valeur *true()* ou non, Nb indique le nom interne de l'objet).

Notre spécification comporte plusieurs autres règles permettant de traiter les autres cas de l'appel de méthode, c'est-à-dire :

- Les appels de méthodes sans indication d'objet (par exemple `MName(...);`). Dans ce cas, la méthode est appelée sur l'objet courant (celui sur lequel a été appelée la méthode que l'on est en train d'interpréter).
- Les appels de méthodes de classes.
- Les appels de la forme `Expr.MName(...)` où `Expr` n'est pas un autre appel de méthode.

```
NonStaticMethodCallWithArguments_FirstArgumentEvaluation:
   --RC: One evaluation step of the first argument is made.
   --
   --CS: #4.myMethod(Obj2.Attr2, 3+6*9, Obj1.getVal());
   evaluateArgument(ObjL1, ClVarL1, Env1, OThId, ObjId1, Mode1
                         |- ArgL1 ->
                                ObjL1_1, ClVarL1_1, Env1_1, ArgL1_1, InstL1)  &
   diff(ArgL1, ArgL1_1)  &
   appendtree(InstL1, insts[methodCall2(Mode2, TVObjId, MethName, ArgL1_1)],
              InstL1_1)
   ----------------
   ObjL1, ClVarL1, Env1, OThId, ObjId1, Mode1
      |- methodCall1(Mode2, TVObjId, MethName, ArgL1) ->
            ObjL1_1, ClVarL1_1, Env1_1, empty(), InstL1_1, runnable() ;
            provided diff(ArgL1, arguments[]);
```

FIG. 3.17 – *Evaluation du premier argument lors d'un appel de méthode.*

– Les appels de méthode conduisant à la levée d'une exception (voir section 3.4), par exemple quand une méthode est appelée sur un objet n'ayant pas été créé.

Pour des raisons évidentes de concision, nous ne les détaillons pas toutes ici. Le lecteur peut se référer à l'Annexe 3 pour étudier l'intégralité des règles spécifiant une sémantique des appels de méthode.

3.3.3.2 Evaluation des arguments

Après avoir déterminé sur quel objet la méthode allait être appelée, l'expression à interpréter est de la forme : `#4.MethName(ArgL1);` (en considérant la syntaxe concrète pour plus de clarté). La syntaxe abstraite de cette expression est :

`methodcall1(Mode2, TVObjectID, MethName, ArgL1)`

où la variable `TVObjectId` représente la référence de l'objet sur lequel la méthode est appelée, le préfixe `TV` indiquant qu'il s'agit d'une variable globale dont le type est fixe (ici, il s'agit d'un type référence). Ce mécanisme de variable globale typée est une des caractéristiques du formalisme Typol. Cette variable typée permet à cette règle de ne s'appliquer que lorsque l'unification entre le type de la valeur et celui de la variable est possible du point de vue du typage. Il est intéressant de remarquer que là aussi, une continuation a été introduite, il s'agit de l'opérateur *methodcall1* afin de pouvoir y intégrer une valeur référence.

La deuxième étape de l'interprétation de l'appel de méthode consiste alors à évaluer les arguments, les uns après les autres, de gauche à droite. La première étape de cette évaluation est spécifiée par la règle représentée par la Figure 3.17. Le sujet de cette règle, qui apparaît en grisé, est exactement l'arbre de syntaxe abstraite décrit précédemment. Cette règle permet d'effectuer un pas d'évaluation du premier argument, et d'introduire une nouvelle continuation, l'opérateur `methodcall2` qui représente les ap-

```
set getMethodClosure is
--SC: Gets the closure corresponding to a method call.
--
--JC: Objects, Class variables
--JC: |- Method name for which the closure is wanted, Object (on which the
--JC: method is called) reference, method parameter values
--JC: -> Closure for the given method call.
judgement Objects |- Identifier, Ref, Vals -> Clr;

getObjectType(ObjId |- ObjL -> ClName)  &
getFieldDeclaration(ObjL, ClName, ValueL, false()
                        |- MethName -> MethDec, _, true())  &
makeClosure(ClName, ValueL, ObjId |- MethDec -> Clr)
----------------
ObjL |- MethName, ObjId, ValueL -> Clr ;
end getMethodClosure;
```

FIG. 3.18 – *Obtention de la closure correspondante à un appel de méthode.*

pels de méthode dont certains arguments ont été évalués et peuvent donc comprendre des valeurs références.

3.3.3.3 Recherche de la méthode à appeler

Une fois que tous les arguments auront été évalués, l'on passe à la troisième étape de la spécification des appels de méthode : la recherche de la méthode à appeler, et la construction de la closure correspondante.

Le set le plus important à considérer est alors le set `getMethodClosure` présenté en Figure 3.18. Ce set permet tout d'abord de rechercher le type de l'objet sur lequel est appelée la méthode, pour ensuite récupérer le source de cette méthode, à l'aide de ce type et de la liste des arguments donnée lors de l'appel, grâce au set `getFieldDeclaration`. La liste des arguments est nécessaire afin d'établir la signature de la méthode appelée. Cette signature sera comparée à celle des méthodes des différentes classes lors de la remontée dans le graphe d'héritage à partir de la classe dont l'objet sur lequel la méthode est appelée est une instance. La troisième prémisse de l'unique règle de ce set permet de construire la closure correspondante au source de la méthode obtenu par la prémisse précédente.

C'est le set `makeClosure` qui a la charge de la construction de cette closure. La Figure 3.19 présente la principale règle de ce set. Cette règle s'applique dans le cas des méthodes non *synchronized* (voir section 3.5.3 pour avoir plus de détails sur la synchronisation), comme le montre la première prémisse de cette règle. Le sujet de cette règle est l'arbre de syntaxe abstraite représentant la déclaration de la méthode à partir

```
NonSynchronizedMethod:
   --RC: Gets the closure for this non synchronized method
   isThisModifierInTheList(synchronized() |- ModL -> false())  &
   getMethodNameAndParameters(TVMethodDecl -> MethName, ParamL)  &
   makeBindedParamList(ParamValueL |- ParamL -> BindParamL)  &
   getModeAndTargetClass(ClName |- ModL -> _, TargetClName)  &
   getInstructionsAndLocalVariables(insts[], locs[pairs[]]
                                    |- MBlock => InstL, LocVarL)
   ----------------
   ClName, ParamValueL, ObjId
      |- methodDeclaration(ModL, _, TVMethodDecl, _, MBlock) ->
           clr(ObjId, TargetClName, MethName, env(BindParamL, LocVarL), InstL) ;
```

FIG. 3.19 – *Construction de la closure.*

Méthode setSides de la classe Square	Closure associée :

void setSides(int s){
setWidth(s);
}
	Closure on the object : X
	Method name : setSides
	Environment :
	Bound parameters : s = Y
	Local variables : none
	Instruction list :
	setWidth(s);

FIG. 3.20 – *Exemple de closure associée à une méthode.*

duquel va être définie la closure.
Cette règle permet tout d'abord de déterminer le nom de la méthode ainsi que ses paramètres (deuxième prémisse), puis effectue le liage entre ces paramètres et les valeurs des arguments (troisième prémisse). La dernière prémisse de cette règle permet d'analyser le bloc de la méthode afin de déterminer la liste des variables locales et les instructions à exécuter. Ces données vont former d'une part l'environnement d'exécution de la méthode (variables locales et paramètres), ainsi que l'activité proprement dite que le thread courant aura à exécuter.
La Figure 3.20 illustre ce mécanisme en montrant la closure obtenue après analyse de l'arbre de syntaxe abstraite associé à la méthode `setSides` de la classe `Square` du programme présenté en Figure 2.1. La partie gauche de cette figure permet de visualiser le source de la méthode, alors que la partie droite montre la closure construite à partir de ce source. Nous présentons ici les données sous forme de syntaxe concrète afin de faciliter la lecture de la figure. Nous utilisons les lettres majuscules `X` et `Y` dans la closure à la place des données qui sont fournies par l'appel de méthode proprement dit : référence de l'objet sur lequel est appelée la méthode (`X`), et valeur donnée en argument lors de l'appel (`Y`).
Lors du prochain pas d'exécution du thread courant, ce dernier va analyser cette closure afin de déterminer quelle instruction exécuter; dans l'exemple de la Figure 3.20, la

prochaine instruction exécutée sera l'appel de méthode `setWidth`.

Les appels de méthode étant un point crucial des langages à objets, leur spécification est complexe. Nous nous sommes efforcés dans cette section de développer la spécification du cas le plus général, qui est aussi le plus courant (par rapport aux appels de méthodes de classe), mais le lecteur intéressé pourra retrouver l'ensemble des règles concernant les appels de méthode au sein du set `evaluateExpression` du fichier *java_expr_evaluation.ty* en Annexe 3.

3.4 Les exceptions

Comme on l'a vu dans le chapitre précédent, les exceptions sont un mécanisme du langage Java permettant de transférer le contrôle de l'exécution du thread courant du point où une contrainte sémantique (comme un pointeur non initialisé ou une division par zéro) a été violée, ou lors d'une instruction directe du programmeur, à un point qui peut être spécifié par le programmeur. Ce point de récupération de l'exception dépend de son type. En effet, toute exception est un objet, instance de la classe Java *Throwable*, ou d'une de ses sous-classes.

Nous modélisons au sein de notre sémantique les principales exceptions de la classe *java.lang.RuntimeException* et de ses sous-classes :
- *NullPointerException* : utilisation d'une variable de type référence non initialisée.
- *ArithmeticException* : exceptions portant sur l'évaluation arithmétique (division par zéro).
- *IllegalThreadStateException* : appel de méthode alors que le thread n'est pas dans le bon statut.
- *IllegalMonitorStateException* : appel de méthode alors que le thread ne possède pas le lock de l'objet sur lequel la méthode est appelée.

Nous ne traitons pas des exceptions appelées *Error* qui sont liées à des problèmes physiques (mémoire ou autre) et non directement à la sémantique du langage Java.

Cette section présente tout d'abord des généralités sur les mécanismes des instructions et la structure sémantique qui leur est associée au sein de notre spécification du langage Java. Les deux autres paragraphes de cette section décrivent une sémantique de deux instructions permettant de manipuler les exceptions en Java, c'est-à-dire les instructions *try...catch...finally...* et *throw*.

3.4.1 Mécanisme général et structures sémantiques

Il existe deux catégories d'exceptions : les exceptions levées automatiquement et les exceptions levées explicitement par l'appel de l'instruction *throw*. Dans les deux cas, le mécanisme déclenché est identique. Ses principales étapes sont :

1. Récupérer le type de l'exception.

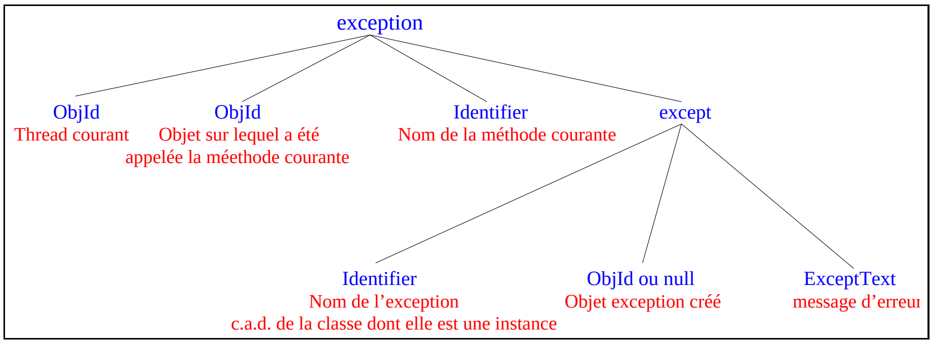


FIG. 3.21 – *Structure sémantique d'une instruction exception.*

2. Relâcher les verrous en cours s'il y en a (au sein du bloc courant).
3. Si on est dans un bloc *try*, rechercher un bloc *catch* convenable du point de vue du type de l'exception :
   - S'il y a un *catch* convenable, on exécute le bloc *catch* puis le bloc *finally*.
   - S'il n'y en a pas, on exécute le bloc *finally* et on propage l'exception.
4. Effacer la suite des instructions du bloc courant.

Afin de spécifier ce mécanisme, nous avons introduit dans notre spécification une structure sémantique modélisant une instruction de lancement d'une exception. L'arbre de syntaxe abstraite de cette structure sémantique est présenté en Figure 3.21. Cette structure sémantique permet de connaître le contexte d'exécution au moment où l'exception a été lancée : elle indique le thread courant, la méthode qui était en train d'être exécutée et sur quel objet elle l'était. D'autre part, cette structure sémantique comporte des informations sur l'exception elle-même (sous-arbre *except*) : son type, c'est-à-dire la classe dont l'objet exception créé est une instance, la référence de cet objet, ainsi qu'un message d'erreur qui pourra aider le programmeur.

Quand une exception est levée, la première étape est de créer l'objet exception qui lui est associé. Cette création est identique à celle d'un objet simple, que ce soit au niveau de la création proprement dite ou au niveau des appels ascendants des constructeurs associés. Prenons l'exemple d'une tentative d'affectation d'une expression à l'attribut d'un objet référencé par *null* (c'est-à-dire sans référence), comme cela est présenté en Figure 3.22.

Le fait de tenter d'accéder à l'objet représenté par *null* constitue une violation de la sémantique de Java, et déclenche par l'appel du set `createExceptionObject` la levée d'une exception . Ce set va non seulement créer l'objet représentant l'exception, mais aussi permettre d'obtenir la liste d'instructions à placer en tête des instructions à exécuter par le thread courant lors de son prochain pas d'exécution. Cette liste se compose d'une part de l'appel du constructeur associé à la classe dont l'objet exception est une instance, et d'autre part de l'instruction de levée de l'exception qui est présentée plus haut.

```
ComplexAssignExpressionViaNull:
    --RC: Attempt to assign via a null reference to an object.
    --RC: Throw an exception.
    --
    --CS: null().Attr = Var1 * Var2;
    createExceptionObject(ObjL1_1, OThId
                                |- identifier "NullPointerException",
                                   exceptText "Use of a pointer with a null value."
                                       ->
                                       _, ObjL1_2, InstL)
    ----------------
    ObjL1, ClVarL1, Env1, OThId, ObjId1, Mode
       |- semAssign(null(), _, _, _) ->
              ObjL1_2, ClVarL1_1, Env1_1, empty(), InstL, runnable() ;
```

FIG. 3.22 – *Exemple de levée d'une exception suite à une violation d'une contrainte sémantique.*

Lorsqu'une exception est levée, deux possibilités se présentent. Soit rien n'est prévu pour la récupérer, et l'exécution du thread courant est arrêtée après relâchement des locks qu'il possède, soit elle est récupérée par une clause *catch* comme nous allons l'étudier dans le paragraphe suivant.

3.4.2 L'instruction *try...catch...finally...*

Une instruction Java est directement associée aux exceptions. Il s'agit de l'instruction *try* ... *catch* ... *finally* ... qui se présente comme suit :

- Bloc *try* : bloc d'instructions contenant le code que l'on veut exécuter.
- Blocs *catch* : chaque bloc *catch* prend comme paramètre un type d'exception et une variable référence. Il contient d'autre part un bloc d'instructions.
- Bloc *finally* : bloc d'instructions qui sera exécuté qu'il y ait levée d'une exception ou non.

Sa sémantique informelle est la suivante. Lors de l'exécution du bloc *try*, deux cas se présentent :

- Aucune exception est levée. Dans ce cas, le bloc *finally* est exécuté directement à la sortie du bloc *try*.
- Une exception est levée au sein du bloc *try*. Dans ce cas, il faut chercher si le type de l'exception levée est compatible avec le type du paramètre de l'un des blocs *catch*. Là encore, deux possibilités :
  - Un tel bloc est trouvé. Ce bloc (le premier qui convient dans la liste des blocs *catch*) est exécuté, le paramètre du bloc se voyant affecter la référence de l'objet exception venant d'être créé. Le bloc *finally* est exécuté à la suite de ce bloc *catch*.

- Aucun bloc catch n'a un paramètre dont le type est compatible avec celui de l'exception levée. Dans ce cas, le bloc *finally* est exécuté. Deux possibilité peuvent apparaître à ce niveau :
  - Ce bloc s'exécute sans levée d'une autre exception et l'on remonte dans la hiérarchie d'inclusion des blocs jusqu'à ce qu'un bloc *catch* compatible avec le type de l'objet exception créé au sein du bloc *try* soit trouvé.
  - L'exécution du bloc *finally* lève une autre exception. Dans ce cas, l'on remonte dans la hiérarchie d'inclusion des blocs jusqu'à ce qu'un bloc *catch* avec un paramètre ayant un type compatible avec celui de l'objet exception créé au sein du bloc *finally* soit trouvé. L'exception levée au sein du bloc *try* est alors oubliée.

Dans ce qui précède, nous appelons remontée dans la hiérarchie d'inclusion des blocs le fait de passer du bloc courant au bloc dans lequel il est directement inclus. Par exemple si une exception est levée au sein d'une méthode sans être récupérée par une clause *catch* au sein de celle-ci, l'exécution se poursuit en propageant l'exception au sein du bloc dans lequel a eu lieu l'appel de la méthode courante. A chaque propagation, les locks acquis au sein du bloc dont on abandonne l'exécution sont relâchés avant cette propagation.
D'autre part, on dit que le type `TC` du bloc *catch* est compatible avec celui `TE` de l'objet exception lorsque la classe `TE` est soit la classe `TC`, soit une de ses sous-classes.

Comme nous pouvons le voir, la sémantique informelle de cette instruction est assez complexe. Cette complexité se retrouve au sein de notre sémantique. En effet, le traitement des différents cas a nécessité de nombreuses règles, notamment dans le cas de la levée d'une exception au sein du bloc *try*, et de la propagation de celle-ci au sein de la hiérarchie d'inclusion. Nous ne pouvons détailler ici les différentes règles concernées, mais elles sont toutes disponibles en Annexe 3, principalement dans le fichier `java_stat_execution.ty`.

3.4.3 L'instruction *throw*

Comme nous l'avons vu précédemment, une exception peut être levée soit à cause d'une erreur de programmation entraînant une violation de la sémantique du langage, soit de manière explicite par une instruction du programmeur. Cela est possible par l'utilisation de l'instruction *throw*. Les règles Typol spécifiant une sémantique de cette instruction sont présentées en Figure 3.23. Nous avons été amenés à utiliser trois règles afin de permettre l'entrelacement au niveau de l'exécution des différents threads. Les deux dernières règles permettent l'évaluation de l'expression donnée en argument de l'instruction *throw* dont la syntaxe abstraite est représentée par l'opérateur *throw*, puis par l'opérateur *semThrow*, ce dernier permettant de traiter la fin de l'évaluation, c'est-à-dire lorsque l'argument de l'instruction est la référence d'un objet exception. La première règle s'applique lorsque l'évaluation de l'argument est terminée, ce dernier étant alors la référence d'un objet exception.

```
ThrowExpression_EvaluatedExpression:
    --BSC: Throw Statement
    --
    --CS: throw #4
    getObjectType(TVObjId |- ObjL -> ClName)
    ----------------
    ObjL, ClVarL, OThId
        |- clr(ObjId, Mode, MethName, Env, insts[semThrow(TVObjId).InstL]) ->
               ObjL, ClVarL,
               clr(
                   ObjId, Mode, MethName, Env,
                   insts[except(ClName, TVObjId, exceptText "thrown exception").InstL]),
               runnable() ;

ThrowExpression_EvaluateExpression:
    --CS: throw Expr;
    evaluateExpression(ObjL1, ClVarL1, Env1, OThId, ObjId, Mode
                            |- Expr1 ->
                                   ObjL1_1, ClVarL1_1, Env1_1, Expr1_1, InstL2,
                                   ThStatus)  &
    appendtree(InstL2, insts[semThrow(Expr1_1).InstL1], InstL3)
    ----------------
    ObjL1, ClVarL1, OThId
        |- clr(ObjId, Mode, MethName, Env1, insts[semThrow(Expr1).InstL1]) ->
               ObjL1_1, ClVarL1_1, clr(ObjId, Mode, MethName, Env1_1, InstL3),
               ThStatus ;

ThrowExpression_EvaluateExpression:
    --CS: throw Expr;
    --
    --ESC: Throw Statement
    evaluateExpression(ObjL1, ClVarL1, Env1, OThId, ObjId, Mode
                            |- Expr1 ->
                                   ObjL1_1, ClVarL1_1, Env1_1, Expr1_1, InstL2,
                                   ThStatus)  &
    appendtree(InstL2, insts[semThrow(Expr1_1).InstL1], InstL3)
    ----------------
    ObjL1, ClVarL1, OThId
        |- clr(ObjId, Mode, MethName, Env1, insts[throw(Expr1).InstL1]) ->
               ObjL1_1, ClVarL1_1, clr(ObjId, Mode, MethName, Env1_1, InstL3),
               ThStatus ;
```

FIG. 3.23 – *Sémantique de l'instruction throw.*

```
NonCaughtExceptionInRunMethod:
    --RC: As the exception is uncaught, the thread execution
    --RC: is aborted (empty instruction list) and the exception
    --RC: is sent to the environment.
    ObjL, ClVarL, OThId
        |- clr(
                OThId, Mode, identifier "run", Env,
                insts[
                    exception(
                        OThId, ObjId1, MethName, except(ClName, ObjId2, ExceptText)).._]) ->
                ObjL, ClVarL, clr(OThId, Mode, identifier "run", Env, insts[]), runnable() ;
                do
                    ENVsendException(exception(
                                            OThId, ObjId1, MethName,
                                            except(ClName, ObjId2, ExceptText)));
```

FIG. 3.24 – *Fin de l'exécution d'un thread suite à une exception non récupérée.*

La Figure 3.24 présente la règle Typol qui termine l'exécution du thread courant lorsque son exécution a levé une exception qui n'a pas été récupérée. Cette règle s'applique dans tous les cas, quelle que soit la raison de la levée de l'exception. L'interprétation de l'exception qui est en tête de la liste des instructions de la méthode *run* du thread courant permet de comprendre qu'elle n'a pas été récupérée et ne peut plus l'être (car elle n'est pas incluse dans un bloc *try*). La liste des instructions restant à exécuter est donc rendue vide, l'exécution du thread courant va alors être stoppée. Il est intéressant de noter la notification vers l'environnement (appel du set `ENVsendException`), qui va permettre de renseigner le programmeur sur l'exception qui a été levée, comme nous allons le voir dans le prochain chapitre (chapitre 4). Naturellement, la même règle existe avec la méthode *main*, permettant ainsi de traiter le cas de la levée d'une exception au cours de l'exécution du thread principal.

Nous avons vu dans la section 3.3 que, même si cette dernière portait sur les éléments liés à l'orientation objets du langage Java, comme la création d'objets ou les appels de méthodes, elle se devait de prendre en compte les contraintes amenées par les aspects liés à la concurrence, notamment au niveau de la granularité fine de la sémantique permettant un entrelacement au niveau de l'exécution des threads et donc une simulation de la concurrence. Cette remarque est encore plus forte en ce qui concerne les exceptions. En effet, si dans un programme avec un unique thread la levée d'une exception entraîne l'arrêt de l'exécution du programme si l'exception n'est pas récupérée, dans le cas d'un programme créant plusieurs threads, cela entraîne non seulement la fin de l'exécution du thread ayant levé l'exception, et mais aussi l'arrêt de l'exécution du programme et donc des autres threads.

```
ThreadCreation:
   makeAttributeList(|- ClName1 -> AttrL)  &
   getFieldDeclaration(objects[], ClName1, vals[], false()
                        |- identifier "run" -> FieldDecl, ClName2, true())  &
   makeClosure(ClName2, vals[], OThId1 |- FieldDecl -> Clr)
   ----------------
   (OThId1, ClName1, OThId2
       ->
       object(
          OThId1, ClName1, AttrL, unlocked(), waits[],
          activity(created(OThId2), Clr))) ;
            provided diff(OThId1, objectId(true(), objectNumber 0));
```

FIG. 3.25 – *Création d'un thread.*

3.5 Multithreading et concurrence

Cette section décrit la spécification sémantique des différentes caractéristiques du langage Java liées à la concurrence. La concurrence, sur une architecture mono-processeur, est simulée grâce à l'introduction d'entrelacement au niveau de l'exécution des différents threads. Nous avons choisi de spécifier une sémantique avec une granularité fine afin d'introduire cet entrelacement de la manière la plus réaliste possible.
Cette section présente tout d'abord la spécification de la création des threads. Une deuxième partie discute les choix effectués au niveau de l'ordonnancement. Une troisième partie permet d'étudier la formalisation du mécanisme de synchronisation, alors qu'une dernière partie décrit une sémantique des méthodes manipulant les locks sur les objets.

3.5.1 Création des threads

Nous avons étudié dans les sections précédentes la structure sémantique d'un thread (Section 3.1.2.1), ainsi que les mécanismes de création d'un objet simple (Section 3.3.2). Comme nous l'avons vu, la principale différence entre un thread et un objet simple est la présence d'une activité au sein du premier.
La première phase de création d'un objet est l'obtention de la liste de ses attributs. Cependant, si l'objet est un thread, il faut de plus construire son activité, en recherchant sa méthode *run* (ou sa méthode *main* dans le cas du thread principal).

Considérons l'exemple d'un thread autre que le thread principal. Sa règle de création est présentée par la Figure 3.25. Cette règle possède comme sujet l'identificateur du thread à créer (`OThId1`), son type (`ClName1`), ainsi que l'identificateur du thread qui le crée (`OThId2`). Comme indiqué précédemment, la première action est de déterminer l'ensemble des attributs du thread à créer. La deuxième prémisse de la règle Typol présentée en Figure 3.25 permet de déterminer le corps de la méthode *run* du thread. La troisième prémisse construit la continuation du thread à partir du source de la

```
GetTheNewExecutingThread_EntryPoint:
   canAThreadBeInTheExecutingStatus(|- ObjL1 -> true(), LastThreadNb)  &
   getExecutingThread(OThId1)  &
   OThId1, LastThreadNb |- ObjL1 -> OThId2, ObjL1_1
   ----------------
   ClVarL |- ObjL1 -> ObjL1_1, OThId2 ;
            do
               ENVsendObjectAndClassVariableLists(ObjL1_1, ClVarL)  &
               ENVsendCurrentExecutingThread(ObjL1_1, OThId2)  &
               setNewExecutingThread(OThId2);
```

FIG. 3.26 – *Règle principale de l'ordonnanceur.*

méthode *run*. Ce mécanisme est détaillé dans la Section 3.3.3 qui traite de la spécification sémantique des appels de méthode.
La conclusion de la règle permet de créer l'arbre de syntaxe abstraite représentant le thread à partir des éléments obtenus dans les conclusions des prémisses. L'état du thread ainsi créé est alors *created(O2ThId)*, précisant par la référence *OThId2* quel thread l'a créé. Un tel thread, bien que pourvu d'une activité, n'est pas encore actif. Il ne pourra commencer son exécution propre qu'une fois que sa méthode *start* aura été appelée par un autre thread.

3.5.2 Ordonnancement

La simulation de la concurrence au niveau de l'exécution des threads nous a conduit à introduire de l'entrelacement : après chaque pas atomique d'exécution, le thread courant se stoppe afin de permettre à un autre thread de s'exécuter.
C'est à l'ordonnanceur que revient le choix du thread qui va s'exécuter à un instant donné. Au sein de notre sémantique, nous avons spécifié un ordonnancement déterministe s'assurant que chaque thread va avoir l'occasion de s'exécuter. La Figure 3.26 présente la règle principale du *set* permettant de spécifier l'ordonnanceur.
La sémantique informelle de cet ordonnanceur est la suivante. Tout d'abord, la liste des objets est parcourue à partir du dernier thread exécuté (qui a été dans le statut *executing*) à la recherche du prochain thread pouvant s'exécuter (voir Figure 3.27). Si aucun thread valable n'a été trouvé avant la fin de la liste, la recherche reprend au début de celle-ci (voir Figure 3.28).
Voici les différents cas possibles quand on analyse un objet donné :

- C'est un objet simple, on passe à l'objet suivant.
- C'est un thread, il faut analyser son statut. Différents cas se présentent alors.
   1. Son statut est *runnable*, il devient alors le prochain thread à exécuter, et son statut va devenir *executing*.
   2. Son statut est *blocked* sur un objet qui n'est plus verrouillé. Dans ce cas, il devient le prochain thread à s'exécuter et son statut va devenir *executing*.

```
LookForTheNextExecutingThread_StartAfterTheCurrentThread:
   inf(NbThr, NbLastThr)  &
   ObjL1 |- ObjL1, objectId(true(), objectNumber NbThr) => OThId, ObjL1_1
   ----------------
   objectId(true(), objectNumber NbThr), objectNumber NbLastThr
      |- ObjL1 -> OThId, ObjL1_1 ;
```

FIG. 3.27 – *Recherche à partir du dernier thread executing.*

```
LookForTheNextExecutingThread_StartByTheFirstObject:
   supeq(NbThr, NbLastThr)  &  ObjL1 |- ObjL1 => OThId, ObjL1_1
   ----------------
   objectId(true(), objectNumber NbThr), objectNumber NbLastThr
      |- ObjL1 -> OThId, ObjL1_1 ;
```

FIG. 3.28 – *Recherche à partir du début de la liste.*

3. Son statut ne correspond pas aux deux cas précédents. Dans ce cas, on passe à l'objet suivant au sein de la liste.

La Figure 3.29 présente la règle Typol spécifiant une sémantique du cas considéré en 2. La prémisse (appel du set `isTheObjectLocked`) de cette règle permet de vérifier que le thread dont le statut courant est *blocked* (sinon la règle ne s'applique pas) est en attente d'un lock qui est libre. Le nouveau statut du thread devient alors *executing*.
Il est bon de noter les deux clauses (précédées du mot clé *do*) permettant de mettre à jour l'environnement. La première est utilisée pour effacer le noeud correspondant au lock sur l'objet dans la vue graphique de notre environnement (voir Section 4.2.2) si plus aucun thread n'est dans le statut *blocked* ou *dormant* en attente d'un lock sur l'objet considéré dans cette règle. La seconde met à jour notre liste classant les threads et les objets en fonction de leur statut (voir Section 4.2.3.1).

```
HasFoundTheNextExecutingThread_Blocked:
   isTheObjectLocked(ObjId |- ObjL1 -> false())
   ----------------
   ObjL1 |- objects[object(OThId, Type, AttrL, Lock, WaitSet, activity(blocked(ObjId), Clr)).ObjL2] =>
               OThId,
               objects[object(OThId, Type, AttrL, Lock, WaitSet, activity(executing(), Clr)).ObjL2] ;
             do
               ENVupdateEnvIfNonDormantOrBlockedThreadOnObject(ObjId, ObjL1, 1)  &
               ENVupdateThreadStatusList(OThId, blocked(ObjId), executing());
```

FIG. 3.29 – *Règle permettant à un thread blocked de devenir executing.*

Si aucun thread ne peut devenir *executing*, tous ont fini leur exécution et dans ce cas, l'exécution du programme est terminée; sinon nous détectons et signalons un deadlock.

3.5.3 Formalisation de la synchronisation

La concurrence entre les threads a développé de nouveaux phénomènes comme les problèmes de cohérence au niveau des attributs des objets partagés. En effet, si deux threads exécutent, l'un une action de lecture et l'autre une action d'écriture sur un même attribut d'un objet sur lequel ils possèdent tout deux une référence, le résultat global dépend de l'exécution, et plus précisément de l'entrelacement et l'ordonnancement.
Le mécanisme de synchronisation a été introduit afin de lever cet indéterminisme en permettant de protéger l'accès aux objets partagés. A cette fin, un lock est associé à chaque objet, et à chaque étape de l'exécution, ce lock ne peut être possédé que par un unique thread. Au concept de lock est lié celui de synchronisation qui permet d'affirmer qu'une portion de code (par exemple une méthode manipulant un attribut d'un objet) ne pourra être exécutée que par un thread à la fois. Cela reprend le concept de *section critique* bien connu dans la littérature.

En Java, le seul moyen pour un thread d'acquérir un lock sur un objet, est soit d'entrer dans un bloc déclaré *synchronized*, soit d'appeler une méthode déclarée *synchronized* sur cet objet. Nous avons choisi (comme cela est fait dans la plupart des autres formalisations de Java présentées en Section 3.7.1) de considérer qu'une méthode *synchronized* est équivalente à une méthode "normale" dont le corps est inclus dans un bloc *synchronized*. L'exécution d'un bloc *synchronized* peut se résumer de la façon suivante :

1. Calcul de la référence de l'objet que l'on veut verrouiller (cette référence est donnée sous forme d'une expression en argument de la clause *synchronized*).
2. Tentative de verrouillage de l'objet : le statut du thread courant devient *blocked* si l'objet est déjà verrouillé par un autre thread.
3. Exécution du corps du bloc *synchronized* dès que le lock est acquis.
4. Libération du verrou à la sortie du bloc.

Notre sémantique reflète ce comportement : les Figures 3.30, 3.31, 3.33 et 3.34 montrent les règles Typol correspondant à chaque étape de la spécification informelle ci-dessus.
La Figure 3.30 présente deux règles Typol : la seconde concerne l'évaluation de l'expression donnée en entrée du bloc pour obtenir la référence de l'objet à verrouiller. Elle s'applique tant que l'expression à évaluer n'est pas une valeur référence. Lorsque l'expression à évaluer a été ramenée à une valeur référence de l'objet dont le thread désire acquérir le lock, la première s'applique. Elle permet la transition entre la première et la deuxième étape de l'exécution du bloc *synchronized.*
La Figure 3.31 présente la règle déterminant si le thread courant peut acquérir le lock sur l'objet dont la référence vient d'être obtenue. L'appel du set `getObjectLockInformation` permet d'obtenir les informations relatives au lock de l'objet : quel thread le possède, et combien de fois l'a-t-il acquis (zéro si l'objet n'est pas verrouillé, n locks, avec n

```
SynchronizedStatement_HasTheObjectReference:
    --CS: synchronized(#8) Statement
    ObjL1, ClVarL, OThId
        |- clr(ObjId, Mode, MethName, Env, insts[synchroLock(TVObjId, Statement).InstL]) ->
               ObjL1_1, ClVarL_1, Clr, ThStatus
    -----------------
    ObjL1, ClVarL, OThId
        |- clr(ObjId, Mode, MethName, Env, insts[synchroBegin(TVObjId, Statement).InstL]) ->
               ObjL1_1, ClVarL_1, Clr, ThStatus ;

SynchronizedStatement_EvaluateExpressionToGetTheObjectReference:
    --CS: synchronized(Obj.getObj2()) Statement
    evaluateExpression(ObjL1, ClVarL1, Env1, OThId, ObjId, Mode
                            |- Expr1 -> ObjL1_1, ClVarL1_1, Env1_1, Expr1_1, InstL2, ThStatus)  &
    appendtree(InstL2, insts[synchroBegin(Expr1_1, Statement).InstL1], InstL3)
    -----------------
    ObjL1, ClVarL1, OThId
        |- clr(ObjId, Mode, MethName, Env1, insts[synchroBegin(Expr1, Statement).InstL1]) ->
               ObjL1_1, ClVarL1_1, clr(ObjId, Mode, MethName, Env1_1, InstL3), ThStatus ;
```

FIG. 3.30 – *Détermination de la référence de l'objet.*

```
SynchronizedStatement_GetTheLockOrBeBlocked:
    --CS: synchronized(#8) Statement
    getObjectLockInformation(ObjId2 |- ObjL1 -> OThId2, nbLocks)  &
    getTheLockOrBeBlocked(OThId1, ObjId2, Statement, ObjL1, ClVarL
                               |- OThId2, nbLocks -> ObjL1_1, Inst, ThStatus)
    -----------------
    ObjL1, ClVarL, OThId1
        |- clr(ObjId1, Mode, MethName, Env, insts[synchroLock(ObjId2, Statement).InstL]) ->
               ObjL1_1, ClVarL, clr(ObjId1, Mode, MethName, Env, insts[Inst.InstL]), ThStatus ;
```

FIG. 3.31 – *Acquisition d'un verrou sur l'objet ou blocage du thread courant.*

```
set getTheLockOrBeBlocked is
--SC: Locks the given object if it is not already locked.
--SC: If the given object is already locked by
--SC: 1] by the current thread, the number of locks on this object is
--SC:    increased by one.
--SC: 2] by an other thread, the current existing thread is blocked to
--SC:    wait the unlock of the object
--
--JC: Current executing thread reference, Reference of the object to lock,
--JC: Statement, Objects, Class variables (only used for the environment)
--JC: |- Thread that demands the lock on the given object,
--JC: Number of existing locks on the given object
--JC: -> modified object list, Instruction,
--JC: New current executing thread status
judgement
   ObjectId, ObjectId, Statement, Objects, ClassVariablesL
      |- ObjectId, integer -> Objects, Inst, ThreadStatus;

UnlockedObject:
   setLock(ObjId, OThId, integer 1 |- ObjL1 -> ObjL1_1)
   ----------------
   OThId, ObjId, Statement, ObjL1, ClVarL
      |- _, 0 -> ObjL1_1, synchroExecution(ObjId, Statement), runnable() ;
            do ENVsendObjectAndClassVariableLists(ObjL1_1, ClVarL);

ObjectLockedByAnotherThread:
   OThId1, ObjId, Statement, ObjL1, _
      |- OThId2, nbLocks -> ObjL1, synchroLock(ObjId, Statement), blocked(ObjId) ;
            provided diff(nbLocks, 0)  &  diff(OThId1, OThId2);

ObjectLockedByTheCurrentThread:
   plus(nbLocks1, 1, nbLocks1_1)  &
   setLock(ObjId, OThId, integer nbLocks1_1 |- ObjL1 -> ObjL1_1)
   ----------------
   OThId, ObjId, Statement, ObjL1, ClVarL
      |- OThId, nbLocks1 -> ObjL1_1, synchroExecution(ObjId, Statement), runnable() ;
            provided diff(nbLocks, 0);
            do ENVsendObjectAndClassVariableLists(ObjL1_1, ClVarL);
end getTheLockOrBeBlocked;
```

FIG. 3.32 – *Set permettant au thread d'acquérir le lock ou d'être blocked.*

```
SynchronizedStatement_BeginBlockExecution:
   appendtree(insts[Statement, synchroEnd(ObjId2)], InstL1, InstL2)
   ----------------
   ObjL, ClVarL, _
      |- clr(ObjId1, Mode, MethName, Env, insts[synchroExecution(ObjId2, Statement).InstL1]) ->
           ObjL, ClVarL, clr(ObjId1, Mode, MethName, Env, InstL2), runnable() ;
```

FIG. 3.33 – *Début de l'exécution du bloc synchronized.*

supérieur à zéro, sinon).
Le set `getTheLockOrBeBlocked` représenté en Figure 3.32 analyse ces valeurs. Les trois règles de ce set spécifient les cas suivants :

- L'objet considéré n'est pas verrouillé (valeur 0). Dans ce cas, le thread courant acquiert le lock.
- L'objet considéré est verrouillé par un thread autre que le thread courant. Dans ce cas, le prochain statut du thread courant sera *blocked* en attente d'un lock sur l'objet considéré.
- L'objet considéré est déjà verrouillé par le thread courant. Dans ce cas, le nombre de locks possédés par le thread courant est augmenté de un. Cela peut se produire si un thread appelle successivement plusieurs méthodes *synchronized* sur cet objet, notamment dans le cas de l'appel récursif d'une méthode *synchronized*.

La règle présentée en Figure 3.33 s'applique lorsque le thread courant a pu obtenir un lock sur l'objet considéré. Elle exprime le début de l'exécution du bloc *synchronized*.
Les deux règles Typol de la Figure 3.34 portent sur la fin de l'exécution d'un bloc *synchronized*. La première traite le cas où l'objet est verrouillé une seule fois par le thread courant : dans ce cas, l'objet n'est plus verrouillé. La seconde traite le cas où l'objet est verrouillé plusieurs fois par le thread courant : dans ce cas, on décrémente juste le compteur de verrous, et l'objet reste verrouillé.
Ces règles permettent d'une part de libérer un lock sur l'objet considéré et de déterminer son nouveau statut (verrouillé ou non), et d'autre part de mettre à jour l'environnement en tenant compte des ces modifications (par les clauses *do* suivant la conclusion de ces règles).

3.5.4 Manipulation des verrous

La synchronisation, si elle permet un certain déterminisme lors de l'accès à un attribut d'un même objet par différents threads, introduit de nouveaux problèmes. En effet, un thread peut rester bloqué au sein d'un bloc *synchronized* dans l'attente d'un évènement qui ne pourra se produire que lorsqu'un autre thread aura verrouillé l'objet considéré. Ce phénomène peut conduire à un dead-lock.
C'est pour éviter cela qu'ont été introduites au sein des bibliothèques de “base” de Java certaines méthodes permettant de bloquer un thread au sein d'un bloc *synchronized* tout en relâchant le lock qu'il possédait (méthode *wait*), puis de l'autoriser à acquérir

```
EndSynchronizedStatementWithOneLock_RemoveLock:
   getObjectLockInformation(ObjId2 |- ObjL1 -> OThId, 1)   &
   releaseObjectLock(ObjId2 |- ObjL1 -> ObjL1_1)
   ----------------
   ObjL1, ClVarL, OThId
      |- clr(ObjId1, Mode, MethName, Env, insts[synchroEnd(ObjId2).InstL]) ->
            ObjL1_1, ClVarL, clr(ObjId1, Mode, MethName, Env, InstL), runnable() ;
            do
               ENVsendObjectAndClassVariableLists(ObjL1_1, ClVarL)   &
               ENVupdateEnvIfNonDormantOrBlockedThreadOnObject(ObjId2, ObjL1_1, 0)   &
               ENVchangeObjectStatusInStatusList(ObjId2);

EndSynchronizedStatementWithSeveralLocks_RemoveOneLock:
   --ESC: End Synchronized
   --
   getObjectLockInformation(ObjId2 |- ObjL1 -> OThId, nbLocks1)   &
   diff(integer nbLocks1, integer 1)   &   sub(nbLocks1, 1, nbLocks1_1)   &
   setLock(ObjId2, OThId, integer nbLocks1_1 |- ObjL1 -> ObjL1_1)
   ----------------
   ObjL1, ClVarL, OThId
      |- clr(ObjId1, Mode, MethName, Env, insts[synchroEnd(ObjId2).InstL]) ->
            ObjL1_1, ClVarL, clr(ObjId1, Mode, MethName, Env, InstL), runnable() ;
            do ENVsendObjectAndClassVariableLists(ObjL1_1, ClVarL);
```

FIG. 3.34 – *Suppression d'un verrou sur l'objet.*

```
Wait:
   --RC: Makes the current thread "dormant", adds it to the wait set of the current
   --RC: object and releases the locks on the current object. The current thread is
   --RC: disabled for thread scheduling until another thread call the notifyAll method
   --RC: on the current object (or the notify method).
   --
   --CS: ObjId.wait();
   --PC: Checks that the object 'ObjId' is locked by the current executing
   --PC: thread 'OThId' and determines how many times the object is locked
   --PC: (nb times).
   --PC: Adds the thread identifier 'OThId' to the wait set of the object 'ObjId'.
   --PC: Releases the nb locks on the object 'ObjId'.
   getObjectLockInformation(ObjId |- ObjL1 -> OThId, nb)  &  diff(nb, 0)  &
   addThreadToTheObjectWaitSet(OThId, ObjId |- ObjL1 -> ObjL1_1)  &
   releaseObjectLock(ObjId |- ObjL1_1 -> ObjL1_2)
   ----------------
   ObjL1, ClVarL, OThId, ObjId
      |- identifier "wait", vals[]
           ->
           ObjL1_2, ClVarL, insts[synchroWaitEndRelock(ObjId, integer nb)],
           dormant(ObjId, integer nb) ;
            do ENVchangeObjectStatusInStatusList(ObjId);
```

FIG. 3.35 – *Sémantique de la méthode wait.*

de nouveau ce lock lorsque par exemple un évènement se sera produit (méthode *notify*).

La Figure 3.35 présente une sémantique de la méthode *wait*. Cette méthode est appelée par un thread possédant un lock sur un objet et qui doit attendre qu'un certain évènement se produise avant de continuer l'exécution du bloc *synchronized*. Cela lui permet de passer dans le statut *dormant* tout en relâchant ce lock, et donc d'autoriser un autre thread à l'acquérir.

La première prémisse de la règle Typol appelle le set *getObjectLockInformation* afin de vérifier que le thread courant possède bien un lock sur l'objet considéré (celui sur lequel la méthode *wait* a été appelée), et de connaître le nombre de ces locks. Si cet objet est verrouillé par un autre thread ou s'il n'est pas verrouillé, ce qui démontre une erreur de programmation, une exception est levée ; ce cas fait l'objet d'une seconde règle Typol (voir Figure 3.36).

La deuxième prémisse de la règle de la Figure 3.35 (règle traitant du cas général) ajoute le thread courant à l'ensemble *wait* de l'objet. Cet ensemble réunit les threads ayant appelé la méthode *wait* sur cet objet et qui donc souhaiteront, quand ils quitteront le statut *dormant*, acquérir de nouveau un ou des locks sur cet objet (selon son statut initial déterminé par la première prémisse) comme nous allons le voir par la suite avec la méthode *notify*. La troisième prémisse relâche quant à elle le ou les locks que détenait le thread courant sur l'objet considéré.

Il est important de noter qu'une instruction spéciale est ajoutée en tête de la liste des

```
Wait_Exception:
   --RC: An exception is thrown because the current object is not locked by
   --RC: the current thread.
   --
   --CS : ObjId1.wait();
   getObjectLockInformation(ObjId1 |- ObjL1 -> _, 0)   &
   createExceptionObject(ObjL1, OThId1
                            |- identifier "IllegalMonitorStateException",
                               exceptText
                                  "Method 'wait' called on a non locked object."
                                  ->
                                  _, ObjL1_1, InstL)
   -----------------
   ObjL1, ClVarL, OThId1, ObjId1
      |- identifier "wait", vals[] -> ObjL1_1, ClVarL, InstL, runnable() ;
```

FIG. 3.36 – *Sémantique de la méthode wait lors de la levée d'une exception.*

instructions que le thread aura à exécuter quand il quittera le statut *dormant*. Il s'agit de l'instruction *synchroWaitEndRelock(ObjId, integer nb)*, qui indique que la première action du thread lorsqu'il s'exécutera sera de tenter d'acquérir `nb` locks sur l'objet considéré dont la référence est `ObjId`.

La méthode *notify*, dont la spécification en Typol est décrite par la Figure 3.37, constitue en quelque sorte le pendant de la méthode *wait*. En effet, cette méthode permet à un thread détenant un lock sur un objet de libérer un (méthode *notify*) ou tous les threads (méthode *notifyAll*) appartenant à l'ensemble *wait* de cet objet. Les threads ainsi libérés deviennent alors *blocked* et vont entrer en compétition afin d'acquérir à nouveau le ou les locks qu'ils possédaient sur l'objet considéré avant d'appeler la méthode *wait*.
Nous ne présentons ici que la règle concernant la méthode *notify*, la règle portant sur la méthode *notifyAll* étant similaire à cette dernière si ce n'est l'appel au set *makeBlockedAllDormantThread* à la place du set *makeBlockedTheOldestDormantThread*. De même, il existe aussi des règles traitant, comme dans le cas de la règle présentée en Figure 3.36, des exceptions levées dans le cas où le thread courant ne possède pas de lock sur l'objet considéré.

Il est important de noter que dans le cas de la méthode *notify*, nous avons choisi de sélectionner de façon déterministe le thread à retirer de l'ensemble *wait* de l'objet considéré en choisissant celui qui y avait été ajouté en premier. Ce choix peut varier selon les sémantiques formelles de Java, sa référence informelle ne spécifiant pas la politique à suivre. On pourrait par exemple envisager de retirer de l'ensemble *wait* le dernier thread à y avoir été ajouté, ou un thread déterminé au hasard.

```
Notify:
   --RC: Wakes up a single thread that is waiting on this object monitor.
   --
   --CS: ObjId.notify();
   --PC: A check is done to be sure that the given object is locked by the
   --PC: current executing thread The oldest dormant thread of the wait set
   --PC: is taken, its status becomes blocked on 'ObjId' instead of
   --PC: dormant=> waiting for nb locks on 'ObjId'.
   getObjectLockInformation(ObjId |- ObjL1 -> OThId, nb)  &  diff(nb, 0)  &
   makeBlockedTheOldestDormantThread(ObjId |- ObjL1 -> ObjL1_1)
   ----------------
   ObjL1, ClVarL, OThId, ObjId
      |- identifier "notify", vals[] -> ObjL1_1, ClVarL, insts[], runnable() ;
```

FIG. 3.37 – *Sémantique de la méthode notify.*

Comme nous l'avons vu dans cette section, les aspects liés à la concurrence forment une part très importante de notre spécification en définissant non seulement l'exécution des programmes, mais aussi ses éléments principaux, les threads. Ces aspects touchent aussi la majorité des règles de notre spécification suite à l'introduction de l'entrelacement qui a induit une division de l'exécution en instructions atomiques.
C'est aussi à ce niveau qu'apparaissent les principales failles de la description informelle de Java de Sun. En effet, certains points comme l'ordonnancement, ou le choix du thread à retirer de l'ensemble *wait* d'un objet ne sont pas détaillés.

3.6 Typage statique : masquage des attributs, une piste

Comme nous l'avons vu dans la Section 2.1.4, notre spécification formelle de Java n'intègre pas le mécanisme de masquage des attributs. Cependant nous avons étudié les problèmes posés par ce dernier, et ses conséquences au niveau de notre spécification formelle et de notre représentation graphique.

3.6.1 Présentation

Prenons le programme présenté en Figure 3.38. Ce programme comporte trois classes : `SequentialEx`, `Shape` et `Square`, cette dernière héritant de la classe `Shape`.
Dans cet exemple, l'instruction `(3)` est une affectation portant sur des variables (respectivement `shape` et `square`) référençant des objets de types différents (respectivement de type `Shape` et `Square`). Après cette instruction, la variable `shape`, déclarée comme étant de type `Shape`, référence un objet de type `Square`. On dit alors que le *type statique*

```java
class SequentialEx {
    public static void main(String[] args) {
        int i;
        boolean b;
        Square square = new Square();
        Shape shape = new Shape();
        /*recuperation de la valeur de nbVisibleSides --> false (1)
        */
        b = square.nbVisibleSides;
        /*recuperation de la valeur de nbVisibleSides --> 0 (2)
        */
        i = shape.nbVisibleSides;
        /*
        affectation polymorphe (3)*/
        shape = square;
        /*recuperation de la valeur de nbVisibleSides --> false (4)
        */
        b = square.nbVisibleSides;
        /*recuperation de la valeur de nbVisibleSides --> 0 (5)
        */
        i = shape.nbVisibleSides;
    }
}

class Shape {
    int nbVisibleSides = 0;
    Shape() {
    }
}

class Square extends Shape {
    boolean nbVisibleSides = false;
    int nbSides = 2;
    Square() {
    }
}
```

FIG. 3.38 – *Programme avec masquage des attributs d'une classe.*

de la variable `shape` est `Shape`, alors que son *type dynamique* est `Square`. Contrairement au type statique qui est connu à la compilation, le type dynamique d'une variable n'est connu que lors de son exécution.
On peut donc se demander quel sera le résultat de l'instruction `(5)`, et plus précisément quelle sera la valeur retournée par l'expression `shape.nbVisibleSides`. Deux possibilités sémantiques seraient envisageables :

1. la valeur de retour est `0` , c'est-à-dire la valeur de l'attribut au sein de la classe `Shape`, si l'on considère une liaison statique
2. la valeur de retour est `false`, c'est-à-dire la valeur de l'attribut au sein de la classe `Square`, si l'on considère une liaison dynamique.

En Java (contrairement à d'autres langages comme Eiffel [1]), la règle est la suivante : *la liaison au niveau des attributs se fait de façon statique* afin d'éviter des problèmes de typage. La valeur retournée sera donc `0`.
Cela illustre le fait que l'instance de la classe `Square` contient en fait l'attribut entier `nbVisibleSides` déclaré dans la classe `Shape` même si celui-ci est masqué par l'attribut booléen de même nom déclaré dans la classe `Square`. La variable `shape` ayant comme type statique `Shape`, elle accède directement à l'attribut déclaré dans la classe `Shape`, alors que toute variable de type statique `Square` accédera à l'attribut booléen déclaré dans la classe `Square`.

3.6.2 Conséquences sur la sémantique formelle

Si notre sémantique veut prendre en compte cet aspect, la structure sémantique d'un objet, et plus précisément de la liste des attributs d'un objet doit être améliorée. Soient `Obj` un objet instance de la classe `C`, et `C'` l'ensemble des classes ancêtres de la classe `C`. La liste des attributs d'un objet serait alors représentée par la structure suivante (visible sous forme graphique dans la Figure 3.39):

- Liste des attributs propres au type `C` sous forme de paires noms-valeurs
- Liste des classes appartenant à l'ensemble `C'` avec pour chacune d'entre elles :
    - la liste des attributs spécifiques à cette classe
    - la liste des attributs communs à cette classe et ses ancêtres.

Il faut de plus modifier la structure sémantique des variables : la nouvelle structure doit permettre de conserver non seulement le nom de la variable, mais aussi son type statique, son type dynamique étant donné par le type de la valeur de cette variable. La Figure 3.40 présente cette nouvelle structure sémantique.

3.7 Travaux connexes et discussion

Cette section permet, en étudiant tout d'abord les principaux travaux portant sur la définition d'une sémantique formelle du langage Java ou plus précisément d'un de ses

1. En Eiffel, la sémantique est dynamique.

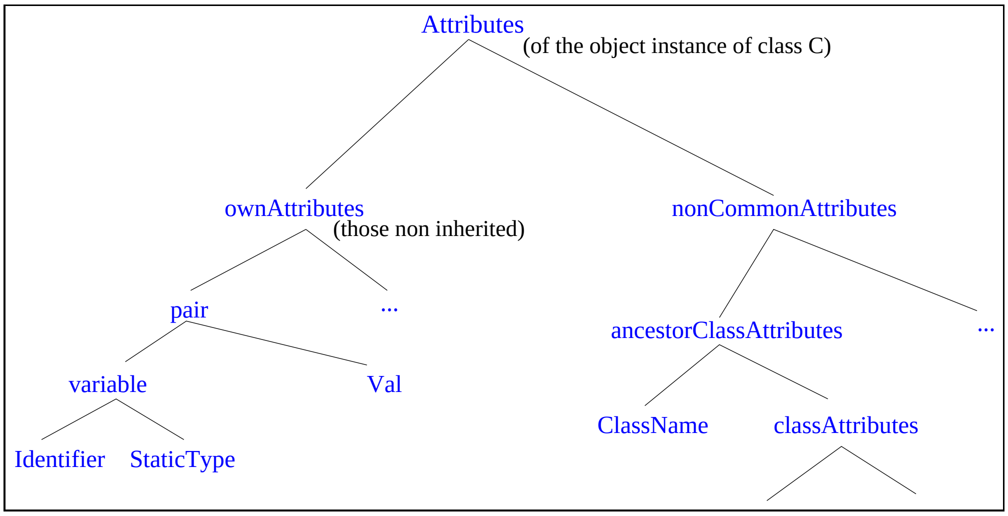


FIG. 3.39 – *Possibilité de structure sémantique pour la liste d'attributs d'un objet instance de la classe* `C`*.*

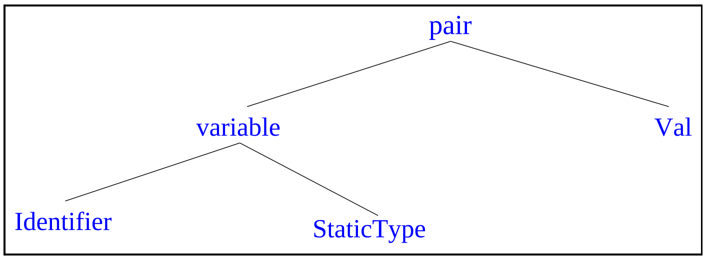


FIG. 3.40 – *Possibilité de structure sémantique pour une paire (variable; valeur).*

sous-ensembles, et en les comparant à notre spécification, de dégager des similarités au niveaux des techniques et modèles employés, mais aussi certaines différences qui font l'originalité de notre travail de spécification.

3.7.1 Sémantiques formelles de Java

La spécification d'une sémantique formelle de Java est un domaine actif de recherche, sur lequel de nombreuses équipes travaillent. Comme le montre cette section, si les méthodes et les sous-ensembles de Java traités diffèrent, le but principal de ces travaux est d'effectuer des preuves de propriétés.

Sémantiques de la machine virtuelle Java (JVM)

Une première approche dans la spécification d'une sémantique formelle de Java est de s'intéresser au bytecode de la machine virtuelle.

Cette approche a été notamment suivie par Qian [123] qui a spécifié une sémantique d'un sous-ensemble des instructions de la JVM portant sur les objets, les méthodes et les instructions de base du langage. Il décrit les comportements à l'exécution dans l'espace mémoire par des transitions entre états, et les contraintes structurelles sur les instructions à l'aide d'un système de typage statique.
Börger et Schulte [16] définissent de façon modulaire une sémantique du bytecode afin d'effectuer des preuves de correction sur la compilation Java.
Jensen, Le Metayer et Thorn [74] ont formalisé les mécanismes de chargement dynamique des classes dans la JVM afin d'étudier des propriétés de sécurité de Java.
Cette approche a l'avantage d'être au plus proche de l'interpréteur Java, et de porter sur un ensemble d'instructions de base moins large que les instructions du code source, mais si l'on se place au niveau du programmeur, le bytecode est trop bas niveau. C'est pourquoi, comme nous allons le voir dans la section suivante, l'approche la plus suivie est la spécification d'une sémantique du code source Java.

Sémantiques du source Java

La principale approche suivie par les équipes de recherche (et que nous avons adoptée) consiste en la spécification d'une sémantique formelle du source Java, ou plus précisément d'un sous-ensemble de Java.

Drossopoulou et Eisenbach (voir [46, 47, 48, 49]), Syme ([138]), et Nipkow et Oheimb ([107]) spécifient une sémantique de différents sous-ensembles de Java afin de prouver la sûreté des types au sein de ces sous-ensembles. Drossopoulou et Eisenbach travaillent dans les quatre articles cités sur un large sous-ensemble séquentiel de Java (incluant les types primitifs, les classes, l'héritage, les variables et méthodes d'instance, les interfaces, le masquage des variables d'instance, la liaison dynamique, les tableaux et les exceptions) en définissant une sémantique opérationnelle de ce sous-ensemble, et prouvent que

l'exécution des programmes préserve le typage au moyen du *Subject Reduction Theorem*. Nipkow et Oheimb [107] définissent et prouvent des propriétés sur leur sous-ensemble de Java appelé "Java Light" à l'aide du prouveur de théorèmes Isabelle/HOL.
Dans [57], Flatt, Krishnamurthi et Felleisen spécifient un sous-ensemble de Java appelé "ClassicJava" à travers la définition de sa syntaxe et de règles de typage et d'évaluation. Ils définissent alors une sémantique opérationnelle à travers un système de réécriture contextuelle, et vérifient la sûreté du typage.
Dans [118], Müller et Poetzsch-Heffter présentent une logique proche du style de Hoare ([83]) pour un noyau séquentiel de Java. Ils spécifient alors une sémantique opérationnelle de ce sous-ensemble, et prouvent la sûreté de cette sémantique en la formalisant avec des règles de typage dans une logique d'ordre supérieur.
Börger et al [17, 15] cherchent quant à eux à spécifier une sémantique formelle de l'ensemble du langage Java en incluant les exceptions et surtout la concurrence. Pour cela, ils ont choisi d'utiliser une modélisation dirigée par l'exécution, en utilisant des machines à états abstraits (ou ASM [128]). Les auteurs construisent successivement cinq modèles, chacun étant une extension du précédent : ils distinguent les modèles impératif, procédural, à objets, avec exceptions et enfin concurrent. Un de leurs buts est de formaliser la sémantique de Java afin de pouvoir utiliser cette formalisation pour effectuer des preuves de propriétés du langage et de programmes Java.

3.7.2 Discussion

La plupart des spécifications formelles de Java ne portent que sur un sous-ensemble séquentiel de ce langage, et s'intéressent principalement au typage. Notre travail porte quant à lui sur un sous-ensemble plus large, prenant en compte la concurrence et les aspects liés à la synchronisation qui apportent une complexité supplémentaire à la spécification formelle du langage.

Parmi les travaux portant sur la définition d'une sémantique dynamique du source du langage, les spécifications de Börger et al [15] paraissent les plus proches des notres. Ils souhaitent en effet définir une sémantique formelle de l'ensemble du langage, en essayant de refléter le plus directement possible les spécifications informelles du livre The Java Language Specification ([61]). Ces auteurs poursuivent principalement les mêmes buts que nous, c'est-à-dire définir une base permettant une meilleure compréhension du langage, la standardisation et la clarification des composants critiques de Java, et surtout utiliser cette définition afin de réaliser des preuves de propriétés du langage et des programmes Java (voir chapitre 5).
Contrairement à notre approche qui est dirigée par la syntaxe (la sémantique du langage est définie à partir de la spécification de la syntaxe abstraite des constructeurs de Java), ils ont choisi une approche dirigée par l'exécution utilisant des machines à états abstraits. Le lecteur intéressé pourra par contre retrouver dans les travaux de Börger et Schulte des simplifications que nous avons aussi introduites dans notre spécification. Par exemple, lors des déclarations des variables locales, une déclaration dans laquelle est incluse une initialisation sera décomposée d'une part en une déclaration simple, et

d'autre part en une instruction d'affectation permettant d'affecter à la variable considérée l'expression donnée pour son initialisation.

3.8 Conclusion

Les spécifications sémantiques décrites dans ce chapitre ont été exprimées directement en Typol. L'un des intérêts de ce langage est de permettre d'écrire des spécifications modulaires, rendant ainsi la sémantique plus lisible. De plus, nous avons dû non seulement utiliser Typol de façon classique en écrivant des règles en Sémantique Naturelle, mais aussi écrire des spécifications en Sémantique Opérationnelle Structurelle afin de décrire les différents aspects liés à la concurrence.
Cette utilisation conjointe des deux sémantiques est réalisable simplement au sein du langage Typol, mais est en opposition avec la volonté de fournir une sémantique modulaire. En effet, il n'est pas possible de décrire d'une part une sémantique des instructions "simples" du langage Java, de la notion de création d'objets ou d'appels de méthodes en ignorant la présence de multithreading, puis d'ajouter un module traitant des aspects concurrents sans pour autant modifier les autres modules. Cela est dû à la nécessité de simulation de la concurrence par l'introduction d'entrelacement au niveau de l'exécution des différents threads. Les règles qui pourraient être écrites en Sémantique Naturelle comme celles de l'évaluation des expressions si l'on considérait uniquement un langage séquentiel doivent être transformées afin de suivre la granularité fine imposée par la simulation de l'entrelacement. Cela a entraîné une augmentation assez conséquente du nombre de règles de notre spécification.

D'un autre côté, Typol et le système Centaur permettent de dériver un interpréteur du langage en compilant directement les fichiers Typol de spécifications formelles en des fichiers Prolog interprétables. Cela nous a conduit à développer Jitan, un environnement d'exécution de Java basé sur notre spécification sémantique.
Cet environnement fait l'objet du prochain chapitre.

Chapitre 4

Jitan, un Environnement de Visualisation

L'apprentissage d'un langage de programmation est toujours délicat pour les programmeurs néophytes. Dans ce cadre-là, un environnement de programmation permettant de visualiser l'exécution des programmes peut apporter un plus réel au niveau de la compréhension. De nombreux environnements ont donc tout d'abord été développés dans un but pédagogique. D'autre part, lorsque des programmeurs ont à reprendre des programmes et ne disposent pas d'une bonne documentation, un tel environnement peut aussi apporter une aide précieuse en facilitant la compréhension du programme, ainsi que sa mise au point.
Notre environnement ne souhaite en aucun cas se placer en concurrence avec les environnements commerciaux existant pour le langage Java comme Visual J++ de Microsoft [63] ou Kawa [141], mais plutôt introduire des aspects innovants marquant une avancée par rapport à ces derniers.
Java, avec sa double orientation mêlant concepts objets et concurrence, introduit un nouveau degré de difficulté au niveau de la programmation, principalement par la complexité des mécanismes liés à la concurrence. Ces nouveaux aspects augmentent le besoin d'un environnement de programmation, que ce soit pour aider à l'écriture des programmes ou leur mise au point.

La mise au point de programmes concurrents (avec la présence de synchronisation, de blocage entre les threads) nécessite une bonne connaissance de la topologie des objets à chaque pas d'exécution. Une vue textuelle oblige le programmeur à imaginer cette topologie à travers une succession de références, en naviguant dans la liste textuelle des objets. Une vue graphique permet quant à elle de visualiser cette topologie d'un seul regard. Elle fournit ainsi une vue d'ensemble des relations entre les différents objets et threads. L'intérêt d'une vue graphique visualisant cette topologie est alors évident.

De nombreux environnements proposent une vue statique du programme par exemple à l'aide d'un browser de classes. Si cela permet de comprendre la structure du programme, cela n'apporte que peu d'informations sur son exécution, surtout dans le cadre de programmes concurrents. Il est alors très important de disposer d'une vue animée, permettant ainsi de suivre de façon dynamique l'exécution du programme, à travers les créations et les mises à jour des objets.

Notre environnement, appelé Jitan, est obtenu directement et gratuitement à partir de notre spécification sémantique de Java. En effet, la sémantique opérationnelle que nous avons définie est exécutable (après une simple compilation) au sein du système Centaur. L'interpréteur associé est ainsi obtenu gratuitement sans aucune manipulation de l'utilisateur. Notre sémantique, de part son aspect formel, introduit une confiance dans la spécification. Cette confiance se retrouve alors au niveau de l'interpréteur. Le programmeur peut en effet être certain du respect de la définition informelle de Sun [61]. Nous avons ensuite enrichi cet interpréteur de vues textuelles et graphiques dynamiques pour obtenir un véritable environnement de programmation et d'exécution. Notre interprète est basé sur une sémantique du source du langage (contrairement aux interpréteurs travaillant à partir du byte-code) et est totalement indépendant par rapport aux machines virtuelles Java existantes.
Jitan constitue une réponse aux attentes des programmeurs, qu'ils soient débutants dans la programmation objet et la concurrence, ou confirmés et souhaitant mettre au point des programmes concurrents. En effet, il fournit une visualisation textuelle et surtout graphique des objets (création, mise à jour, etc.) au cours de l'exécution d'un programme Java et offre des vues synthétiques ou d'abstraction très utiles lorsque le programme crée un grand nombre d'objets.

Ce chapitre présente les différents problèmes liés à la visualisation de l'exécution d'une application concurrente et à objets, ainsi que les solutions apportées par notre environnement. Après avoir présenté les aspects généraux de notre environnement ainsi que son obtention à partir des spécifications sémantiques, nous détaillons les mécanismes de visualisation de la synchronisation et des verrous. Nous décrivons ensuite comment des abstractions du graphe d'objets peuvent être utilisées pour la visualisation d'applications plus complexes. Une section est alors consacrée à la description de pistes sur la visualisation des objets dans le cadre de la prise en compte du masquage des attributs. Le dernier paragraphe permet de comparer notre environnement avec les différents outils existants.

4.1 D'une spécification formelle à un environnement de visualisation

Cette section est dédiée aux techniques utilisées pour obtenir un environnement de visualisation à partir des spécifications sémantiques.
Nous allons plus particulièrement nous intéresser à l'obtention des différentes vues (textuelles et graphique) et à l'animation de l'environnement durant l'exécution des pro-

grammes. A partir des spécifications sémantiques, Centaur génère automatiquement un simulateur. Celui-ci prend en entrée l'arbre de syntaxe abstraite correspondant au programme dont on veut simuler l'exécution, et produit en sortie une structure sémantique qui dénote le comportement (la signification) du programme. Dans notre cas, la structure sémantique choisie est, comme on l'a vu dans la Section 3.1.2, une liste de classes possédant des variables de classe, ainsi qu'une liste d'objets et de threads.

Notre environnement montre la liste des objets grâce à deux moteurs de visualisation, tous deux étant basés sur les structures sémantiques permettant de modéliser la liste d'objets comme cela est présenté dans la Section 3.1.2. Les vues textuelles sont des formatages directs de la liste des objets et des threads avec leurs attributs (noms et valeur) pour la première vue, et les activités des threads dans la seconde, ainsi que les statuts des différents objets et threads et leurs interactions au sein de deux vues de résumé. La vue graphique, qui présente la topologie du graphe d'objets, est visualisée avec le serveur de graphes de Centaur [72].
Les noeuds et les arêtes du graphe sont construits en utilisant un parcours de l'arbre de syntaxe abstraite représentant la liste des objets : pour chaque objet, un noeud le modélisant est créé, et pour chaque attribut ou variable locale dont la valeur est une référence, une arête est créée entre les deux objets impliqués. Nous ne contrôlons pas directement le dessin du graphe. C'est la responsabilité du serveur de graphes d'utiliser une heuristique particulière pour le placement des noeuds et des arêtes. Mais le programmeur peut interactivement modifier le placement des noeuds et des arêtes sur le graphe à l'aide de la souris. Les formes des noeuds et des arêtes peuvent quant à elles être spécifiées au sein d'un fichier de ressources.

Pour afficher un objet sur le graphe, la structure graphique correspondant à cet objet est envoyée au serveur de graphes. De même, des instructions sont envoyées au serveur afin de créer des arêtes et des flèches entre les différents noeuds du graphe. Nous avons défini des structures au niveau du serveur de graphes correspondant à chaque entité du graphe (noeud ou arête). Par exemple, la structure graphique d'un objet est constituée de :

- son nom
- son type
- son statut
- sa liste d'attributs
- sa liste de variables locales
- un booléen qui indique si l'objet est zoomé ou non (afin de lire les valeurs des attributs et des variables locales).

Ces données sont obtenues directement à partir de la structure sémantique de l'objet (voir Section 3.1.2) déterminée durant l'interprétation du programme et sont ensuite transmises au serveur de graphes.

Afin de fournir une animation de la visualisation *pendant* l'exécution des programmes,

l'interpréteur sémantique notifie les changements importants (changements d'état) aux moteurs de visualisation textuelle et graphique. Une notification est un appel à des primitives avec l'entité concernée (là où un changement s'est produit) comme paramètre. La sémantique est équipée de telles notifications : quand la règle Typol appropriée est exécutée avec succès (qu'elle est prouvée), les notifications (si elles existent) sont déclenchées et les moteurs de visualisation sont alors informés des modifications des structures sémantiques. De telles notifications informent les moteurs de visualisation du changement de la valeur d'un attribut, du statut d'un thread, du verrouillage ou du déverrouillage d'un objet.
Dans l'ensemble de la sémantique, seules une dizaine de règles Typol ont besoin d'être équipées de telles notifications. Il convient de noter que les notifications ne sont pas mêlées au reste des spécifications, mais sont en fait concentrées au sein de deux modules : celui traitant de la liste des objets, et celui qui gère la concurrence (voir Figure 3.10).
Les principales règles d'où des notifications doivent être envoyées sont les suivantes :

1. création des objets et des threads,
2. changement du statut d'un thread (par exemple quand un thread devient exécutable),
3. changement du statut d'un objet (verrouillé ou non),
4. mise à jour des variables, attributs ou continuations.

La création d'un objet (ou d'un thread) est naturellement une information essentielle pour le programmeur, notamment au niveau de la topologie qui est modifiée par ce nouvel objet. Cette création apparaît très clairement au sein des différentes vues à travers l'apparition d'un nouvel élément pour les vues textuelles, ou d'un nouveau noeud et de nouvelles arêtes dans la vue graphique.

Le changement du statut d'un thread ou d'un objet, qui peut être une information critique dans le cadre de la recherche d'un deadlock par exemple apparaît principalement au sein de la vue graphique par le changement de couleur du noeud représentant l'objet. Cette information est aussi disponible dans les vues textuelles, mais de manière nettement moins lisible.
La mise à jour des attributs peut influer de façon notable sur la topologie des objets si la valeur modifiée est une valeur de type référence. C'est pour cela qu'il est important de notifier toute modification de la valeur d'une variable ou d'un attribut à l'environnement. Toute modification d'une variable ou d'un attribut de type référence est visualisable immédiatement sur la vue graphique de par la modification de la topologie (création ou déplacement d'une arête du graphe). La modification des autres attributs est visible en zoomant sur l'objet dans la vue graphique (cela est aussi valable pour les variables locales), et sur une vue textuelle.

Les continuations des threads, ou plus précisément leurs activités, procurent une information capitale au programmeur en lui apprenant ce qu'a exécuté chaque thread et ce qu'il lui reste à exécuter. Dans le cadre de programmes concurrents, cette information n'est pas disponible directement à partir du source du programme, il est donc

```
MainRule_SendMessagesToTheEnvironment:
   (ClVarL1 -> ClVarL1_1)  &
   getObjectsForGraphic(ObjL1, ClVarL1_1 -> ObjL1_1, ClVarL1_2)
   ----------------
   (ObjL1, ClVarL1) ;
            do
               emit_tree("displayMainStructure", mainStruct(ClVarL1_2, ObjL1_1))  &
               emit_tree("objl", ObjL1)  &  emit_tree("t_objl", ObjL1);
```

FIG. 4.1 – *Règle Typol communiquant les structures sémantiques à l'environnement.*

important que notre environnement la fournisse clairement aux programmeurs. Une vue textuelle est donc réservée à la présentation des activités des différents threads. C'est sur cette vue qu'apparaît toute mise à jour des activités suite à l'exécution d'une instruction.

Afin d'éviter la dispersion des messages d'émission au sein de notre sémantique, et de permettre un pré-traitement avant leur envoi, les règles sont équipées de clauses faisant appel à un module Typol spécialisé, qui se chargera de l'envoi proprement dit des structures sémantiques. Le mécanisme de communication avec l'environnement (qui permet de mettre à jour les différentes vues) est spécifié au sein du module Typol appelé :

`java_environment.ty`

Ce module permet de communiquer les structures abstraites à visualiser à l'environnement, et plus généralement de transmettre des informations entre la spécification sémantique et ce dernier. Le set Typol `ENVsendObjectAndClassVariableLists`, dont la règle importante du point de vue de l'envoi des structures sémantiques est présentée en Figure 4.1, permet de communiquer la liste des objets à représenter aux différentes vues de notre environnement. L'envoi proprement dit des structures sémantiques est effectué par les trois messages *emit_tree* de la clause *do* :

– *emit_tree("displayMainStructure", mainStruct(ClVarL1_2, ObjL1_1))* communique la structure sémantique principale à la vue graphique.
– *emit_tree("objl", ObjL1)* communique la liste des objets à la vue textuelle représentant les activités des threads
– *emit_tree("t_objl", ObjL1)* communique la liste des objets à la vue textuelle représentant la liste de tous les objets avec leurs types et leurs attributs.

Les prémisses de la règle de la Figure 4.1 permettent d'effectuer, si nécessaire, un pré-traitement des structures sémantiques, comme dans le cadre des abstractions 4.5.

Un autre aspect important pour le confort de l'utilisateur est l'incrémentalité : afin d'obtenir des visualisations efficaces et de bonne qualité (en évitant les flashes), les changements sont déterminés et affichés de manière incrémentale dans les différentes vues.

4.2 Vue générale de l'environnement

La Figure 4.2 présente les différentes vues de notre environnement d'interprétation au cours de l'exécution du programme producteur-consommateur présenté dans la Section 2.1.3.2 (Figure 2.3).

Tout d'abord, un panneau de contrôle (Figure 4.2(a)) fournit des outils de mise au point : choix de l'animation ou d'une vue unique (textuelle ou graphique), mode d'exécution pas-à-pas ou pause de l'exécution et redémarrage, réglage de la vitesse d'exécution, et enfin manipulation des abstractions disponibles du graphe d'objets.

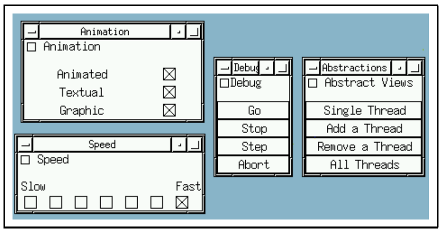
La vue 4.2(b) permet d'étudier les activités des différents threads. La liste des objets (incluant les threads qui sont avant tout des objets) est présentée sous forme textuelle (Figure 4.2(c)) avec pour chaque objet son type et ses attributs. La vue graphique (Figure 4.2(d)) présente la topologie de la liste des objets, les statuts des threads (de gauche à droite : *dead, dormant, blocked et executing*), ainsi qu'une visualisation des verrous (par exemple, l'objet #1 est verrouillé par le thread #T4).
Enfin, deux vues d'informations globales sur l'application sont fournies : elles présentent les statuts des objets et des threads (Figure 4.2(e)), et les interactions existant entre les objets et les threads (Figure 4.2(f)).

4.2.1 Visualisation textuelle de la liste d'objets

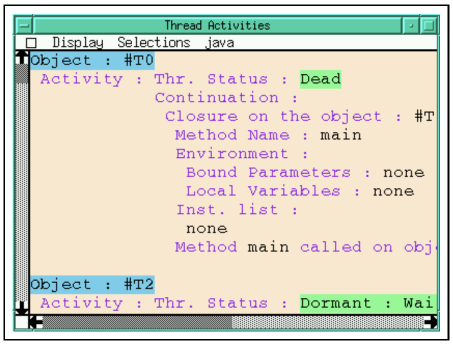
Tout d'abord, il est souvent difficile de vérifier à un point donné de l'exécution d'un programme la valeur d'un champ donné, ou si un objet est actuellement verrouillé. Pour aider le programmeur, notre environnement fournit deux vues (Figures 4.2(b) et 4.2(c)) qui donnent une *présentation textuelle* de la liste d'objets courante au cours de l'exécution.

Quand un programmeur rencontre des problèmes durant l'exécution d'un programme, sa première idée est de vérifier quels objets ont déjà été créés et les valeurs de leurs attributs ; ces données sont présentées sans aucune instrumentation par notre environnement grâce à la vue textuelle (Figure 4.2(c)).
Cette vue permet de visualiser tous les objets déjà créés au cours de l'exécution du programme avec leurs principales propriétés. Les programmeurs peuvent visualiser pour chaque objet son type (par exemple `CubbyHole` pour l'objet #1), la liste de ses attributs (deux pour l'objet #1 : `contents`, qui a pour valeur 0, et `nbTimesToConsume`), si l'objet est verrouillé ou non et dans le premier cas par quel thread (par exemple, l'objet #1 est verrouillé une fois par le thread #T4), et le *wait set* de l'objet s'il n'est pas vide. Un programmeur peut aisément différencier les threads des objets : un thread est nommé `#Ti` alors qu'un objet est nommé `#i`.

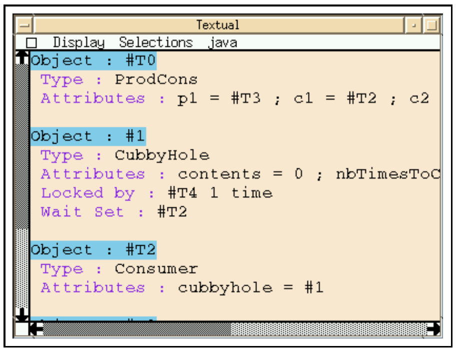
Une façon pour les programmeurs de traquer un problème dans un programme est de savoir ce qui a déjà été exécuté et ce qui ne l'a pas encore été, et ce pour chaque thread.



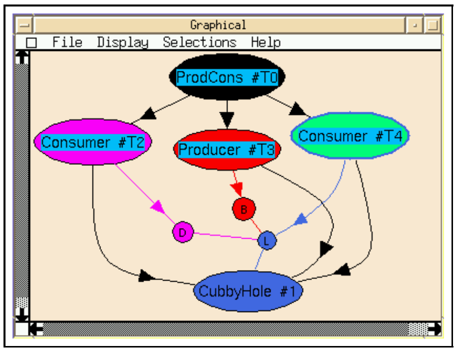
(a) Vues de contrôle.



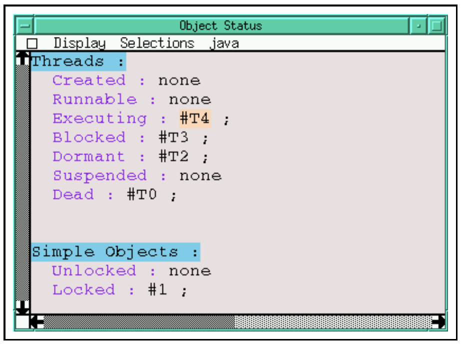
(b) Activités des threads.



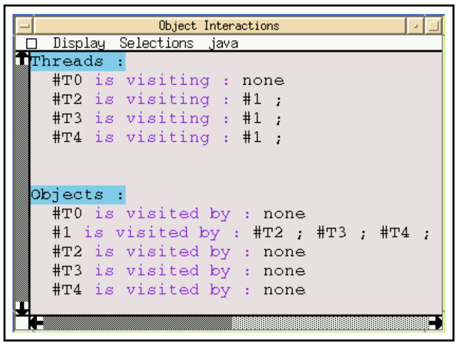
(c) Vue textuelle des objets.



(d) Vue graphique des objets.



(e) Statuts des objets (runnable, blocked, etc).



(f) Interactions entre les objets et les threads.

FIG. 4.2 – *Vues de notre environnement de visualisation.*

En effet, dans les applications séquentielles, il est assez simple de suivre l'exécution sur le source du programme. Mais cela devient impossible dans le cas d'une application concurrente où plusieurs threads exécutent diverses méthodes en même temps. En effet plusieurs threads peuvent en être à des instructions différents d'une même méthode à une étape donnée de l'exécution, et rien dans le source du programme ne permet de distinguer à quel stade de la méthode se trouvent les threads.
C'est l'avantage principal de la vue de la Figure 4.2(b) qui présente les activités des différents threads. Grâce à cette vue, le programmeur peut visualiser le statut de chaque thread (e.g. *runnable, blocked*, etc) et sa continuation courante. La continuation d'un thread est constituée d'un nom de méthode, d'un environnement d'exécution (paramètres et variables locales) et d'une liste d'instructions (voir la Section 3.1.2 pour plus de détails). Cette liste d'instructions est mise à jour à chaque pas d'exécution : seules les instructions restant à exécuter sont affichées, pas celles qui l'ont déjà été. Cela permet au programmeur de se concentrer sur la prochaine instruction qui va être exécutée. Et, ce qui est extrêmement important dans le cadre d'applications concurrentes, de connaître cela pour chaque thread.

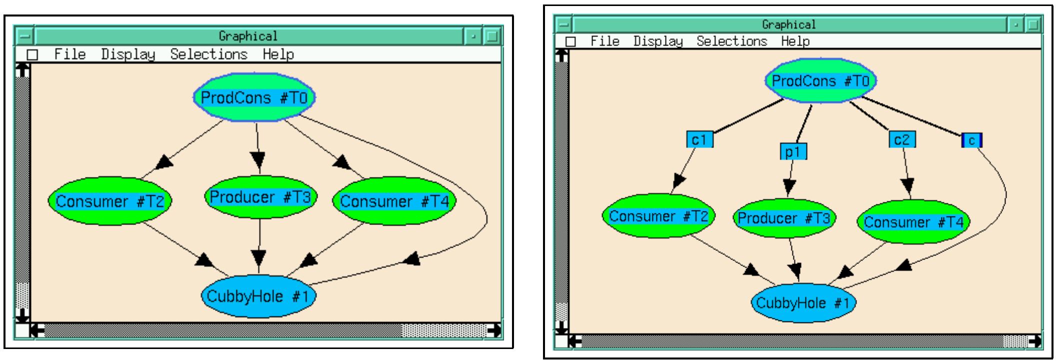
4.2.2 Présentation graphique des objets, threads, classes et variables statiques

Un autre problème important rencontré dans le contexte de la programmation à objets est l'évolution dynamique de la topologie des objets. Avec une représentation textuelle, il peut être très difficile de découvrir quels objets sont connectés, directement ou non, par une référence. Il est donc très utile de disposer d'une représentation graphique pour mettre en évidence les références entre les objets.

Notre environnement propose une représentation graphique de la liste d'objets, alors que la plupart des autres outils offrent seulement une représentation sous forme d'arbres qui oblige le programmeur à comparer les valeurs des champs pour découvrir et suivre les connections entre objets. Avec notre environnement, la visualisation de ces connections, et donc la compréhension de la topologie du graphe, est immédiate.

4.2.2.1 Représentation des objets

La Figure 4.2(d) présente notre vue graphique. Elle représente la liste d'objets au cours de l'interprétation du programme producteur-consommateur, en utilisant un graphe où les noeuds elliptiques modélisent les objets et les threads et où les flèches noires modélisent les références entre les objets. Les threads se distinguent des objets par la présence d'un rectangle autour de leur nom. Cette représentation met en avant les références entre objets (flèches) et le type des objets (dans l'étiquette de l'objet).
Les statuts des threads sont visualisées par la couleur du cadre des noeuds des threads. Ce code de couleurs permet de constater immédiatement le statut du thread considéré (par exemple un cadre rouge indique un thread dans le statut *blocked*). Ces codes sont présentés dans la section 4.3.



(a) Avant le zoom. (b) Après le zoom.

FIG. 4.3 – *Visualisation des attributs et des variables locales.*

4.2.2.2 Représentation des variables locales et des attributs

A la demande, en cliquant sur l'objet ProdCons #T0 par exemple (voir Figure 4.3), un procédé de zoom permet d'examiner les valeurs des différents attributs (par exemple c1, p1, et c2) et des variables locales (e.g. c) d'un objet. Dans ce cas, les noms de variables (attributs ou variables locales) apparaissent dans un noeud rectangulaire attaché aux flèches correspondantes. Les variables locales sont différenciées des attributs par une fonte plus petite et un cadre bleu foncé.

Afin de mettre au point un programme ou de comprendre son comportement, il est très important de pouvoir repérer les éventuelles arêtes constituant un DAG (Directed Acyclic Graph) et les cycles de références entre objets. Un DAG peut indiquer un accès partagé à un objet donné et donc des *race conditions* en présence de threads. Grâce à notre vue graphique, il est aisé de détecter de tels cas. Si deux threads atteignent par plusieurs références successives un même objet, ce dernier est alors partagé par ces deux threads, et cela peut potentiellement créer un problème de protection des données partagées. Par exemple, si un thread appelle une méthode qui modifie un attribut de l'objet pendant la lecture de cet attribut par un autre thread, il peut y avoir un problème de cohérence. Dans ce cas, il est nécessaire de protéger cet objet par un mécanisme de synchronisation. Dans l'exemple de la Figure 4.2(d), on peut voir que les threads #T2, #T3 et #T4 peuvent tous les trois accéder à l'objet #1. Cela indique immédiatement que cet objet doit pouvoir être verrouillé afin de protéger ses attributs d'un accès concurrent en écriture (le problème se pose uniquement lors des mises à jour et non pour les lectures).

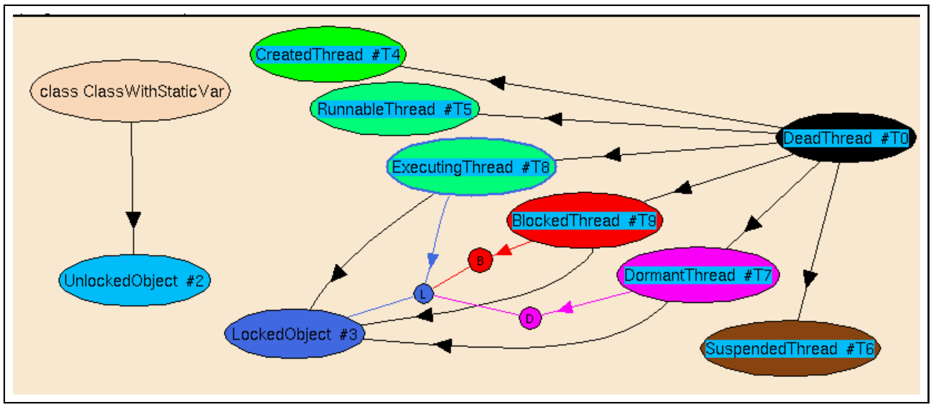


FIG. 4.4 – *Objets, classes, verrous, et statuts des threads.*

4.2.2.3 Représentation des variables statiques

La visualisation des variables statiques requiert un traitement spécial. En effet, les variables statiques sont attachées à leur classe, et sont donc communes à tous les objets instances de cette classe, alors que les variables d'instance (ou attributs) sont attachées à l'objet instance. Les variables statiques doivent être représentées sur la vue graphique, mais la représentation choisie doit permettre aux programmeurs de différencier aisément les variables statiques des variables d'instance, le tout sans rendre la représentation graphique trop complexe.

Nous utilisons la solution suivante : seules figurent sur notre représentation graphique les classes ayant des variables statiques, comme par exemple la classe figurant en partie gauche de la Figure 4.4. Les objets référencés par une telle classe (ou les valeurs des variables statiques) sont accessibles à tous les objets instances de cette classe.

Nous ne montrons pas les liens entre chaque objet d'un type donné et les variables statiques correspondantes pour simplifier la topologie, mais un programmeur peut facilement comprendre où se situent de telles références : en considérant un objet, le programmeur connaît directement son type (inclus dans le label de l'objet), et si la classe correspondante est représentée sur la vue graphique, cet objet possède des variables statiques qui peuvent être visualisées par un simple zoom sur la classe. Par exemple, dans la Figure 4.4, la classe `ClassWithStaticVar` possède une référence sur l'objet #2.

Pour ne pas rendre le graphique illisible à cause d'une surcharge d'informations, les autres éléments, comme par exemple les continuations des threads, ne sont pas disponibles sur cette vue graphique (on les retrouve sur les autres vues).

4.2.3 Vues synthétiques

Les véritables systèmes concurrents comportent de nombreux threads, objets et aussi de nombreuses interactions entre ces différentes entités. Une solution pour s'adapter à cette complexité repose sur l'utilisation de vues récapitulatives des objets et des threads. De telles vues facilitent la compréhension des comportements concurrents complexes (présence de deadlocks ou livelocks, besoins de synchronisation), surtout quand il y a trop de noeuds sur la représentation graphique.

4.2.3.1 Récapitulation des statuts des threads et des objets

La possibilité pour un programmeur de visualiser rapidement, par exemple, qu'aucun thread ne peut plus s'exécuter alors qu'il y a au moins un thread qui n'a pas le statut *dead* (et qui n'a donc pas fini son exécution), peut être très utile afin de détecter des deadlocks (comme dans l'exemple cité) ou des livelocks. De plus, un thread qui est créé mais ne devient jamais exécutable (*runnable*) peut indiquer un problème de programmation.
La Figure 4.2(e) montre une vue qui présente les différents objets et threads créés lors de l'interprétation d'un programme classés suivant leurs statuts. Sur l'exemple présenté, on peut directement noter que le thread #T4 est *executing* (en train de s'exécuter), le thread #T3 est bloqué, le thread #T2 est *dormant* et que le thread #T0 est *dead* (il a fini son exécution). Cette vue indique aussi quels objets sont verrouillés : dans l'exemple considéré, l'objet #1 est verrouillé.

4.2.3.2 Interactions entre threads

Les erreurs lors des appels récursifs de méthodes sont courantes en programmation. Il peut être très utile pour un programmeur de détecter un thread qui appelle successivement des méthodes sur un seul objet ou sur un cycle d'objets. La Figure 4.2(f) montre une vue qui présente les piles d'appels des threads et les activations des objets. Dans la partie haute de la vue, le programmeur peut voir sur quel objet est en train de travailler chaque thread, et la chaîne d'objets qui ont mené l'exécution à ce point. Par exemple, le thread #T2 est en train d'exécuter une méthode sur l'objet #1.

Un problème crucial avec les applications concurrentes est de déterminer où placer les synchronisations. Afin d'éviter le non-déterminisme, les accès à une donnée partagée (un attribut par exemple) doivent être inclus dans une section critique (un bloc *synchronized*). Il est donc très utile de pouvoir mettre en évidence les accès concurrents à des objets. Notre environnement indique tous les threads qui sont en train d'accéder à un objet donné, et ce pour chaque objet créé. Par exemple, on peut lire dans la partie basse de la vue de la Figure 4.2(f) que l'objet #1 est actuellement partagé entre les threads #T2, #T3 et #T4. Cela indique un besoin potentiel de synchronisation au niveau de toutes les méthodes permettant d'accéder à cet objet partagé.

Cette section a présenté les aspects généraux de notre environnement d'interprétation. Outre les vues textuelles qui sont assez classiques parmi les environnements d'exécution de programmes, Jitan comporte une vue graphique permettant de mettre en avant la topologie des objets ainsi que les statuts des threads. Comme nous allons le voir par la suite, cette vue représente un point clé de notre environnement lors de l'interprétation de programmes concurrents en mettant en évidence les phénomènes liés au multithreading et à la synchronisation.

4.3 Visualisation de la concurrence et des synchronisations

Chaque programmeur Java qui a développé des applications multi-threadées est d'accord sur le fait que la mise au point de tels programmes est difficile. Afin de comprendre, à une étape donnée de l'exécution, la cause d'un deadlock ou si une section critique est requise, il est nécessaire de déterminer quels objets sont verrouillés (*locked*), ou quels threads sont bloqués sur un lock (*blocked*).
Les informations textuelles compactes que nous avons présentées dans la section précédente sont une première réponse à ce problème. Cependant, ce n'est pas suffisant, c'est pourquoi nous fournissons aussi une solution permettant de visualiser graphiquement les locks et les mécanismes de synchronisation. Cela donne au programmeur une vue unifiée de la topologie des objets, des statuts des threads et des verrous.

Tout d'abord, nous visualisons les statuts des threads par des codes de couleur (rouge pour les threads bloqués sur un lock, noir pour les threads ayant fini leur exécution, etc). Pendant l'exécution, nous montrons aussi les objets verrouillés par des couleurs : bleu foncé pour les objets verrouillés, bleu clair sinon. Les informations permettant d'étudier et de comprendre la synchronisation peuvent être trouvées dans les diverses vues textuelles, mais nous avons choisi de les mettre en relief dans la présentation graphique, et de manière animée.

Ces éléments (locks, synchronisations, etc) nous ont amenés à introduire de nouveaux types de flèches entre les objets :

- Les `flèches-L` sont des flèches munies d'un label bleu `L` (L pour *locked*) ; cette flèche met en évidence un lock (obtenu par le thread quand il appelle un bloc ou une méthode *synchronized* sur l'objet considéré) entre le thread d'origine et l'objet destination (objet verrouillé).
- Les `flèches-B` sont des flèches munies d'un label rouge `B` (B pour *blocked*) ; cette flèche relie un thread et un lock sur un objet. Elle établit que le thread d'origine est bloqué, en attente d'un lock (le noeud lock destination) sur un objet qui est déjà verrouillé par un autre thread.
- Les `flèches-D` sont des flèches munies d'un label pourpre `D` (D pour *dormant*) ; cette flèche relie un thread *dormant* et un lock sur un objet. Elle indique que le

thread *dormant* (qui a appelé sa méthode *wait* sur l'objet auquel se rapporte le lock) va rentrer en compétition pour acquérir ce lock aussitôt qu'il sera réveillé (par le thread qui détient le lock considéré et qui appellera la méthode *notify* ou *notifyAll* sur l'objet considéré), et choisi par l'ordonnanceur.

La Figure 4.4 présente tous les codes graphiques et de couleurs correspondant aux différents statuts des threads et des objets.

Au sujet de la visualisation des locks, une première possibilité était de représenter le lock directement sur la flèche modélisant la référence entre le thread et l'objet. Mais, un thread qui ne dispose pas de référence directe sur un objet peut tout de même y accéder et par là même le verrouiller, indirectement, à travers d'autres objets dont l'un au moins dispose d'une référence directe sur l'objet. Les objets pouvant être verrouillés par un thread donné représentent la clôture transitive de l'ensemble des objets accessibles à partir des attributs, des variables statiques et des variables locales de sa méthode *run* (ou *main* pour le thread principal).

La Figure 4.5 montre un exemple de programme qui met en évidence le fait qu'un thread puisse verrouiller un objet sur lequel il n'a pas de référence directe. Ce programme crée une chaîne de communication dont la topologie est présentée par la Figure 4.6. Dans cette Figure, le thread #T0 possède un lock sur l'objet #3, mais en fait ce lock est obtenu au travers de trois appels successifs de méthode (pour les objets #1, #2, et #3). Cela implique que l'information concernant le lock ne peut pas être liée à une référence, mais doit avoir une existence propre. La Figure 4.7 présente un modèle plus fréquent de chaîne de communication où tous les objets formant cette chaîne sont verrouillés par le thread émetteur afin de protéger la donnée transmise. Cela peut être obtenu juste en ajoutant le mot-clé *synchronized* aux méthodes `transmit` des classes `Access_Way` et `Final_Step` de la Figure 4.5.

Cette section nous a permis de mettre en évidence un des principaux points forts de Jitan, la vue graphique et sa représentation de la topologie des objets et des phénomènes liés à la concurrence et à la synchronisation. La suite du chapitre détaille les apports de cette visualisation en s'appuyant sur différents exemples.

4.4 Exemples d'animation

Cette section présente des exemples concrets de figures obtenues lors de l'interprétation de programmes Java, ainsi que leur utilité à travers la détection de deadlocks. Nous nous sommes concentrés dans cette section sur la fenêtre graphique qui représente un point clé de notre environnement et une de ses originalités.

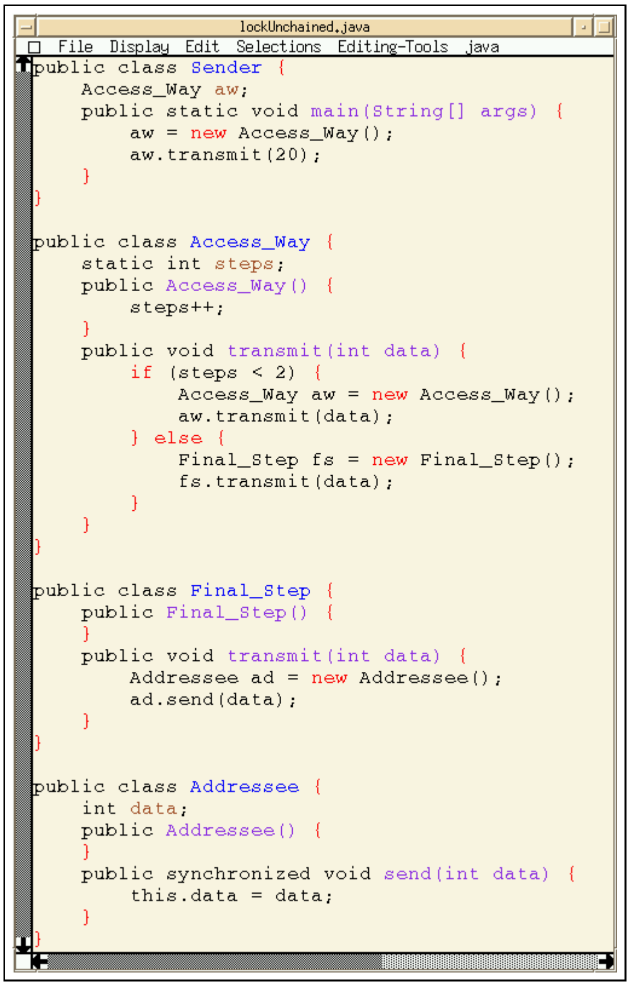


FIG. 4.5 – *Une chaîne de communication.*

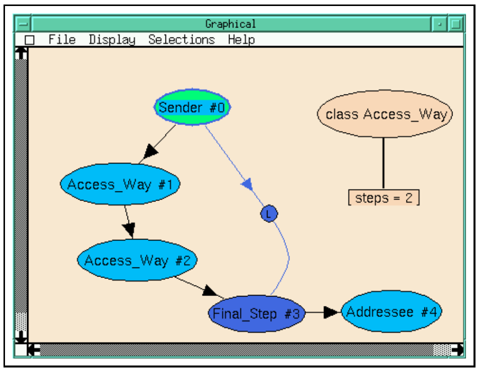


FIG. 4.6 – *Un exemple d'objet verrouillé indirectement.*

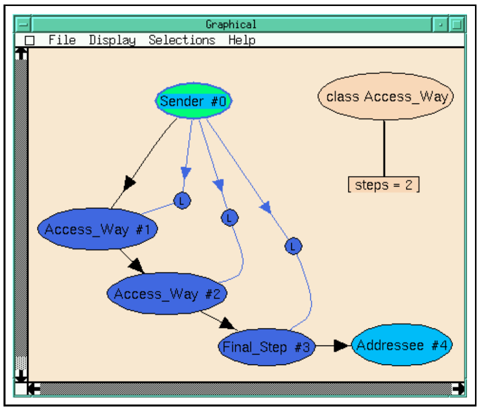


FIG. 4.7 – *Une chaîne d'objets verrouillés.*

4.4.1 Animation graphique de l'interprétation d'un programme

Nous avons choisi de présenter une exemple d'animation de la vue graphique obtenue lors de l'interprétation du programme producteur-consommateur (présenté à la Section 2.1.3.2, Figure 2.3) par l'interpréteur généré à partir de notre sémantique.
Les Figures 4.8, 4.9, 4.10 et 4.11 montrent comment la synchronisation et les statuts des objets et des threads sont visualisés au cours de cette interprétation grâce à l'animation de notre vue graphique.
Elles ont été obtenues en diminuant la vitesse d'exécution ce qui nous a permis de stopper celle-ci aux moments-clés. Nous avons donc utilisé deux fonctionnalités de Jitan : la modification de la vitesse d'exécution et celle de stopper et de reprendre l'exécution à volonté.

Au début de l'exécution, la liste d'objets est constituée uniquement du thread principal `#T0` (qui va exécuter la méthode *main*). Tous ses attributs (`c1`, `c2` et `p1`) ainsi que sa variable locale (`c`) sont initialisés à la valeur par défaut des variables références, comme nous pouvons le voir grâce au zoom effectué sur le noeud du thread `#T0`.

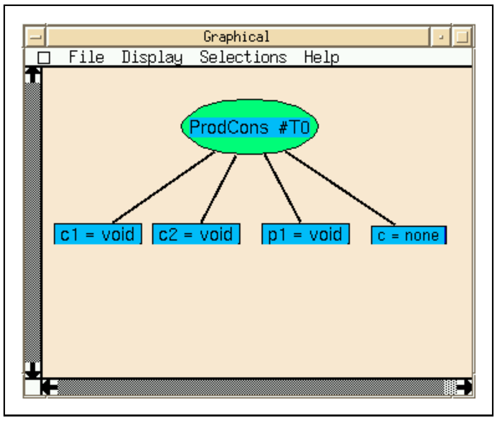


FIG. 4.8 – *Animation de la vue graphique (1).*

La première action du thread `#T0` est de créer l'objet CubbyHole `#1`. Ici, l'objet est en cours de création, sa référence n'a donc pas encore été affectée à la variable locale `c`.

La référence de l'objet `#1` a été affectée à la variable locale `c`. Une flèche a donc été ajoutée au graphe afin de représenter cette référence. Nous retrouvons dans cette Figure ainsi que dans la précédente les étapes de la création d'un objet : création proprement dite, puis, après l'appel des constructeurs, affectation de la référence de l'objet à une variable ou un attribut (étapes spécifiées à la Section 3.3.2).

A ce stade de l'exécution, le thread `#T0` a créé successivement l'objet `CubbyHole #1`, ainsi que les threads `Consumer #T2`, `Producer #T3` et `Consumer #T4`. Ces trois threads sont encore dans le statut *created*, ce que l'on peut remarquer à la couleur verte qui est plus vive que celle du thread `#T0`.

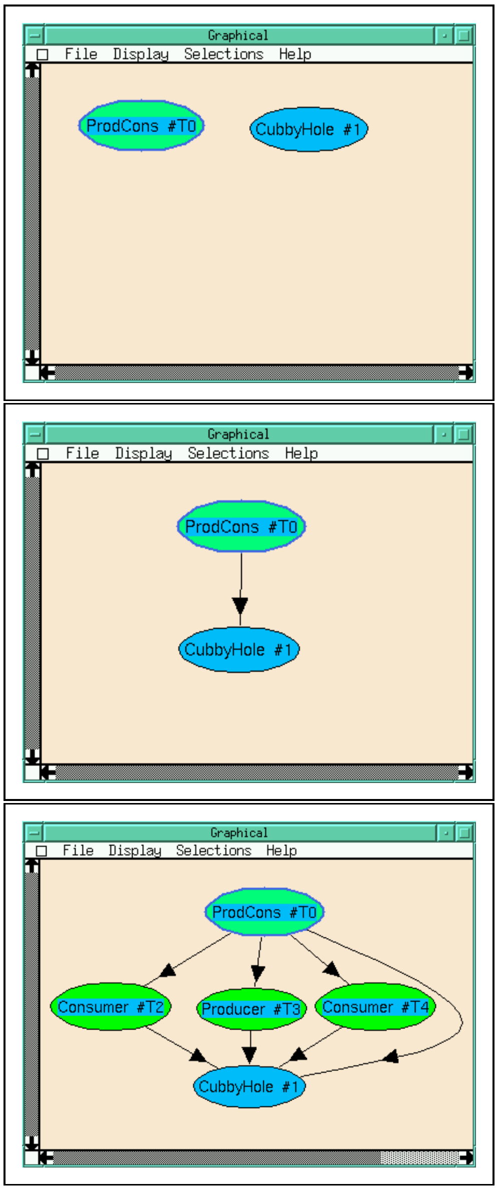


FIG. 4.9 – *Animation de la vue graphique (2).*

Plus tard durant l'exécution, les trois threads ont été lancés et s'exécutent donc de manière concurrente. Les threads `Consumer #T2` et `Producer #T3` sont *runnable*, alors que le thread `Consumer #T4` est *executing*. L'objet CubbyHole #1 est verrouillé par le thread #T2 (présence d'une flèche-L et couleur bleu foncé pour l'objet #1).

Quelques pas d'exécution plus tard, le thread principal (`#T0`) est *dead* (noeud en noir). `Consumer #T4` a essayé d'accéder au buffer (appel de la méthode *synchronized* `get`) et est maintenant *blocked* (flèche-B et thread en rouge), en attente du lock sur le buffer qui est pour l'instant détenu par `Consumer #T2`. `Consumer #T2` a verrouillé le `CubbyHole #1` avant que le `Producer #T3` y ait placé une valeur et n'a donc pas de donnée à y lire.

Le `Consumer #T2` a appelé sa méthode *wait* afin de permettre au `Producer #T3` (peut-être après plusieurs pas d'exécution) d'obtenir finalement un lock sur le `CubbyHole #1` et de mettre une donnée dans ce buffer. Le lock est relâché (`CubbyHole #1` et son lock en bleu clair) par le `Consumer #T2`. `Consumer #T2` devient *dormant* (couleur pourpre et flèche-D vers le lock attaché au buffer).

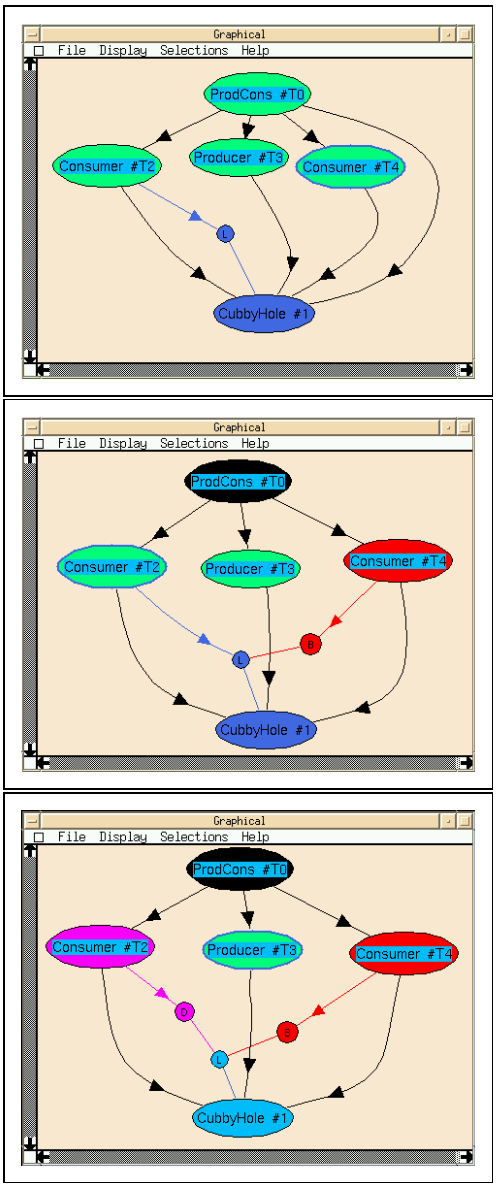


FIG. 4.10 – *Animation de la vue graphique (3).*

Le thread `Consumer #T4` est débloqué et prend possession du verrou attaché au `CubbyHole #1`. Le `Producer #T3` passe alors dans le statut *blocked*, en attente de ce lock.

Pour la même raison que précédemment, le thread `Consumer #T4` devient alors *dormant* et libère le lock sur le buffer. Le thread `Producer #T3` est alors débloqué et obtient un lock sur `CubbyHole #1`. Par la suite, `Producer #T3` va relâcher le lock (après avoir mis une valeur dans le buffer). `Consumer #T2 et #T4` vont être réveillés et vont entrer en compétition afin d'obtenir le lock et de lire la donnée placée dans le buffer.

A la fin de l'exécution, tous les threads sont dans le statut *dead*, l'exécution s'est donc achevée normalement (sans levée d'exception non rattrapée).

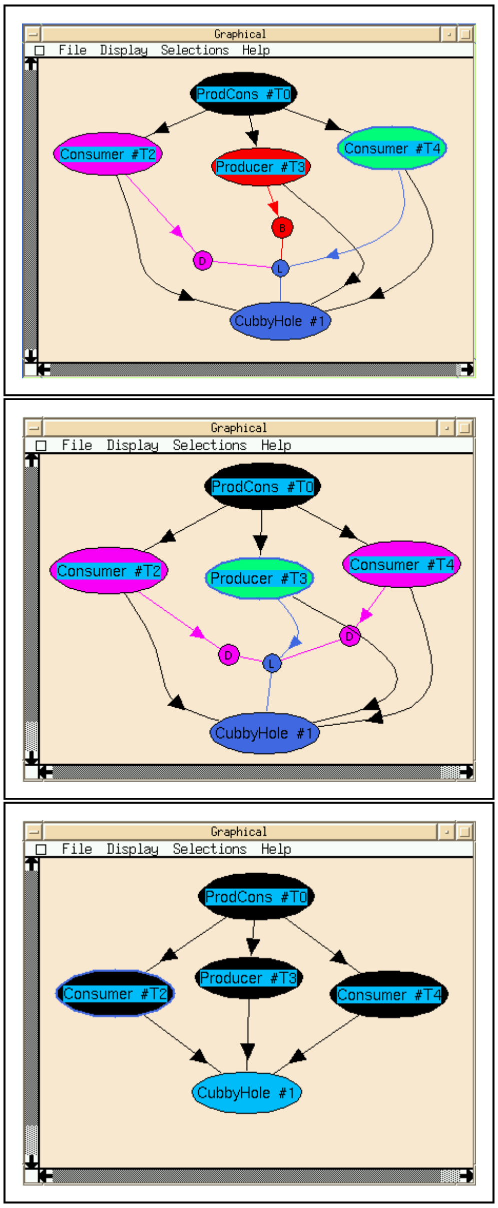


FIG. 4.11 – *Animation de la vue graphique (4).*

Quand une exception est levée, elle apparaît en rouge au sein de la liste des instructions du thread concerné. Si elle est récupérée, l'exécution se poursuit. Par contre, si une exception est levée et non rattrapée, une fenêtre indiquant l'exception est créée (sur le modèle de la fenêtre de la Figure 4.12), et l'exécution est stoppée.

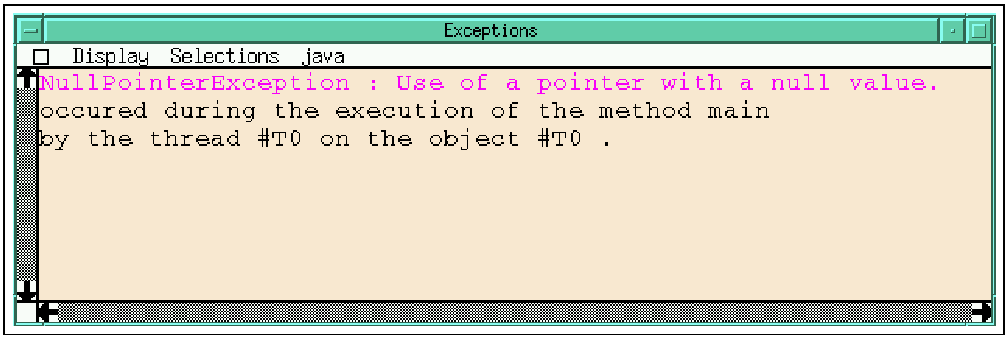


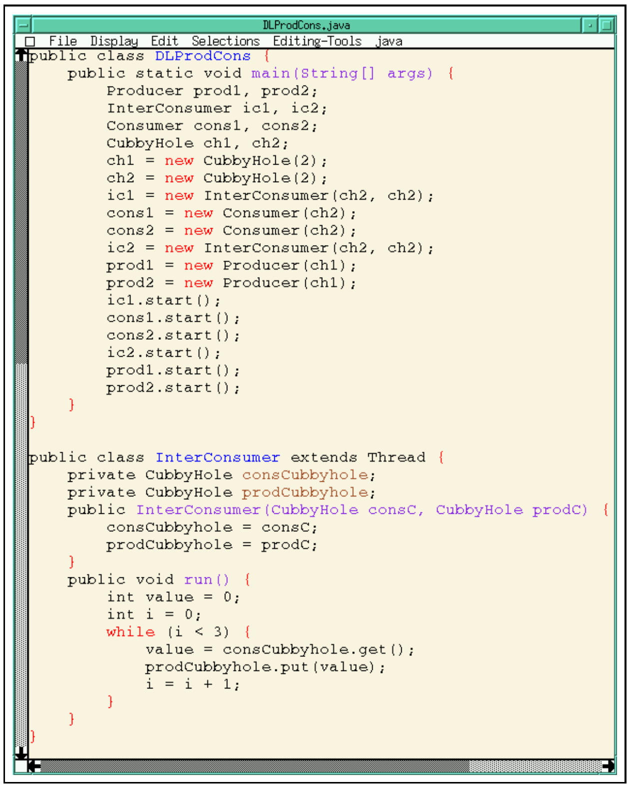
FIG. 4.12 – *Exemple d'exception non rattrapée.*

4.4.2 Détection et correction d'un deadlock

Cette section présente un exemple de détection de deadlock et l'importance de la visualisation graphique pour comprendre le problème et le résoudre.
La section précédente a montré l'exécution normale (c'est-à-dire sans deadlock, ni levée d'exception) d'un programme producteur-consommateur. Afin de décrire le mécanisme de détection d'un deadlock, nous avons transformé le programme présenté dans la section 2.1.3.2 : le nouveau programme crée maintenant deux producteurs qui placent des données dans le cubbyhole. Chaque donnée est lue par deux consommateurs intermédiaires (de type `InterConsumers`) qui la transmettent à un autre cubbyhole. Finalement, deux consommateurs finaux (de type `Consumers`) la lisent. Nous avons transformé la classe principale du programme qui s'appelle maintenant `DLProdCons`.
Les deux nouvelles classes `DLProdCons` et `InterConsumer` sont présentées dans la Figure 4.13.

Détection

Quand on interprète le nouveau programme, on obtient en cours d'exécution la vue graphique présentée dans la Figure 4.14 qui informe qu'un deadlock s'est produit. L'exécution est donc stoppée et cette vue devient la vue finale.
En effet, tous les threads sont soit *dead* (`DLProdCons #T0`) ou *dormant* (autres threads). Aucun thread ne peut donc s'exécuter. Si l'on regarde attentivement la Figure 4.14, on peut observer qu'un groupe de threads *dormant* et d'objets est déconnecté du reste du graphe des objets. En effet, il n'y a pas de référence entre `CubbyHole #1` et les consommateurs intermédiaires (`Interconsumers #T3 et #T6`). Afin de lire la donnée placée dans le premier cubbyhole, et de réveiller les threads *dormant*, les deux consommateurs intermédiaires doivent posséder, au moins de façon indirecte, une référence sur ce cubbyhole. On peut donc en déduire que le problème se situe principalement au niveau des consommateurs intermédiaires. Zoomer sur le thread `InterConsumer #T6` (comme cela est présenté sur la Figure 4.15) nous permet de comprendre que ses deux attributs `consCubbyHole` et `prodCubbyHole` ont la même valeur : la référence de l'objet `CubbyHole #2`, car les deux flèches pointent vers le même objet. L'erreur est donc




```java
public class DLProdCons {
    public static void main(String[] args) {
        Producer prod1, prod2;
        InterConsumer ic1, ic2;
        Consumer cons1, cons2;
        CubbyHole ch1, ch2;
        ch1 = new CubbyHole(2);
        ch2 = new CubbyHole(2);
        ic1 = new InterConsumer(ch2, ch2);
        cons1 = new Consumer(ch2);
        cons2 = new Consumer(ch2);
        ic2 = new InterConsumer(ch2, ch2);
        prod1 = new Producer(ch1);
        prod2 = new Producer(ch1);
        ic1.start();
        cons1.start();
        cons2.start();
        ic2.start();
        prod1.start();
        prod2.start();
    }
}

public class InterConsumer extends Thread {
    private CubbyHole consCubbyhole;
    private CubbyHole prodCubbyhole;
    public InterConsumer(CubbyHole consC, CubbyHole prodC) {
        consCubbyhole = consC;
        prodCubbyhole = prodC;
    }
    public void run() {
        int value = 0;
        int i = 0;
        while (i < 3) {
            value = consCubbyhole.get();
            prodCubbyhole.put(value);
            i = i + 1;
        }
    }
}
```

FIG. 4.13 – *Exemple de deadlock avec un pattern producteurs-consommateurs.*

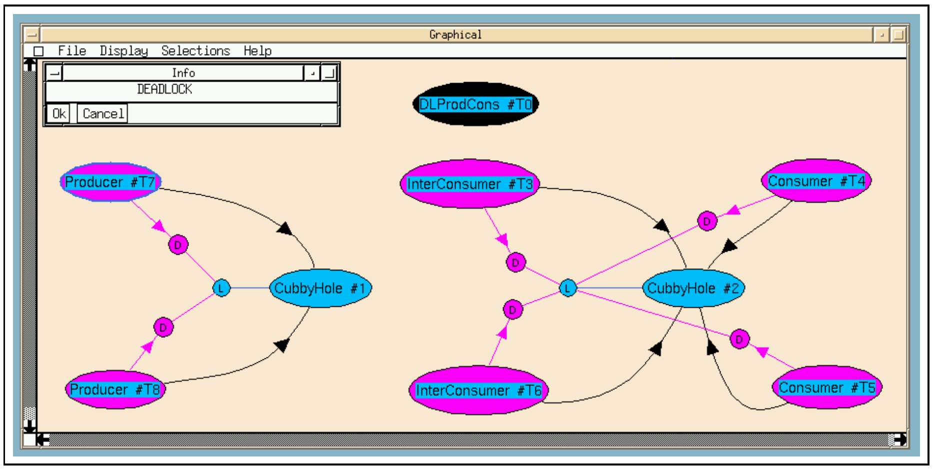


FIG. 4.14 – *Détection du deadlock.*

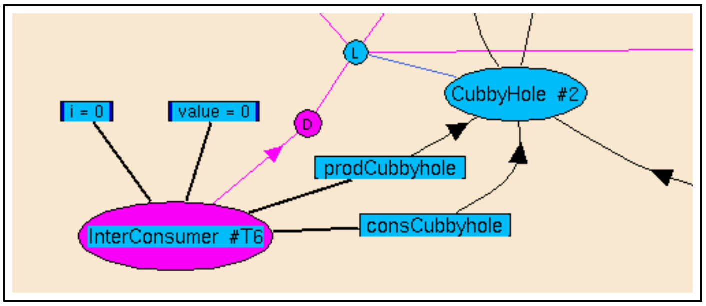


FIG. 4.15 – *Zoom sur la cause du deadlock.*

liée à la valeur de ces deux attributs. On peut voir que l'attribut `prodCubbyHole` a la bonne valeur : celle de la référence de l'objet `CubbyHole #2`, car les consommateurs intermédiaires sont des producteurs pour le second cubbyhole. Par contre, la valeur de l'attribut `consCubbyHole` doit être celle de la référence de l'objet `CubbyHole #1`, car les consommateurs intermédiaires doivent lire la donnée placée dans ce cubbyhole afin de la transmettre au second cubbyhole. Le deadlock est donc causé par une erreur lors de l'initialisation des attributs des consommateurs intermédiaires. Par cette erreur, tous les threads (excepté le thread principal qui est *dead*) sont en attente pour continuer leur exécution : les deux consommateurs finaux attendent qu'une donnée soit placée dans le second cubbyhole pour la lire, les deux consommateurs intermédiaires attendent la même chose (au lieu d'attendre de pouvoir lire la donnée dans le *premier* cubbyhole), et les deux producteurs attendent que la donnée que l'un d'entre eux a placé dans le premier cubbyhole soit lue pour en placer une seconde.

Correction

Maintenant que la cause du deadlock est trouvée, il ne reste plus qu'à corriger l'erreur commise, c'est-à-dire trouver dans le programme où sont créés et initialisés les consommateurs intermédiaires. Les deux consommateurs intermédiaires sont créés par le thread principal, dans sa méthode *main* par les instructions :

```
... = new InterConsumer (ch2, ch2)
```

C'est à ce niveau que se situe l'erreur : la même référence vers un objet de type `CubbyHole`, la référence `ch2`, est transmise deux fois ; l'instruction correcte serait :

```
... = new InterConsumer (ch1, ch2)
```

La Figure 4.16 présente la topologie du graphe d'objets obtenue après correction.

Cet exemple montre donc comment notre environnement de visualisation peut aider les programmeurs à détecter un deadlock, à comprendre le problème et à le résoudre.

4.5 Passage à l'échelle

Nous avons présenté précédemment notre vue graphique et son utilité dans la compréhension et la mise au point de programmes. Cependant, quand on manipule des

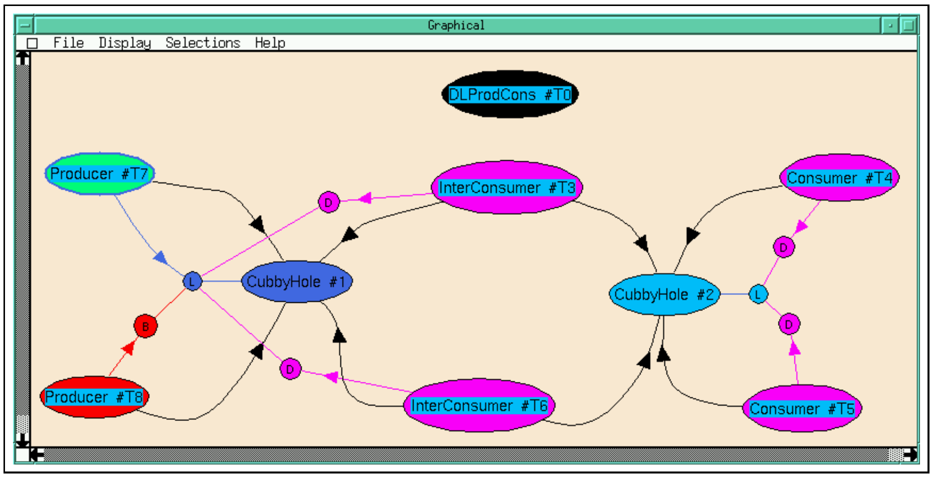


FIG. 4.16 – *Topologie obtenue avec la version corrigée.*

programmes Java, on doit rapidement faire face à un grand nombre d'objets. La vue graphique représente alors un graphe d'objets très complexe, ce qui entraîne des problèmes au niveau de la lisibilité (tout le graphe ne pouvant apparaître dans la vue graphique). De plus un tel graphe est aussi plus délicat à interpréter.
Nous avons donc étudié différentes possibilités de visualisation graphique de l'exécution des programmes Java mettant en jeu un grand nombre d'objets et de threads. Pour cela, il fallait prendre en compte plusieurs critères : la facilité d'emploi du mécanisme utilisé, son efficacité au niveau de la compréhension de l'exécution du programme, la prise en compte des mécanismes liés à la concurrence.
Afin d'être plus explicite, nous nous appuyons sur l'exemple du graphe présenté en Figure 4.17, obtenu en cours d'exécution d'un programme Java au sein de notre environnement. Au cours de l'exécution de ce programme, le thread principal `#T0` (le thread qui exécute la méthode *main*) crée deux modèles connus : un arbre binaire et un système producteur-consommateurs. Chaque objet de type `BinTree` est construit grâce à une méthode dite d'*object factory*, ce qui explique les références entre le thread principal et tous les objets `BinTree`. Le graphe d'objets obtenu, visualisé dans la Figure 4.17 sans instrumentation de la part de l'utilisateur, est assez complexe. Il est très difficile avec une telle vue, même en utilisant les "ascenseurs", de détecter les différents modèles ou les sous-graphes indépendants.

4.5.1 Etude des différentes possibilités

Une première idée, très simple, est de permettre au programmeur de se concentrer sur une fraction particulière du graphe d'objet (à déterminer par ailleurs), par exemple en grisant le reste du graphe. Cette technique, si elle peut être utile pour mettre en valeur certains objets comme par exemple tous les objets d'un type donné par l'utilisateur, possède un inconvénient majeur : elle n'agit en rien sur le placement des objets,

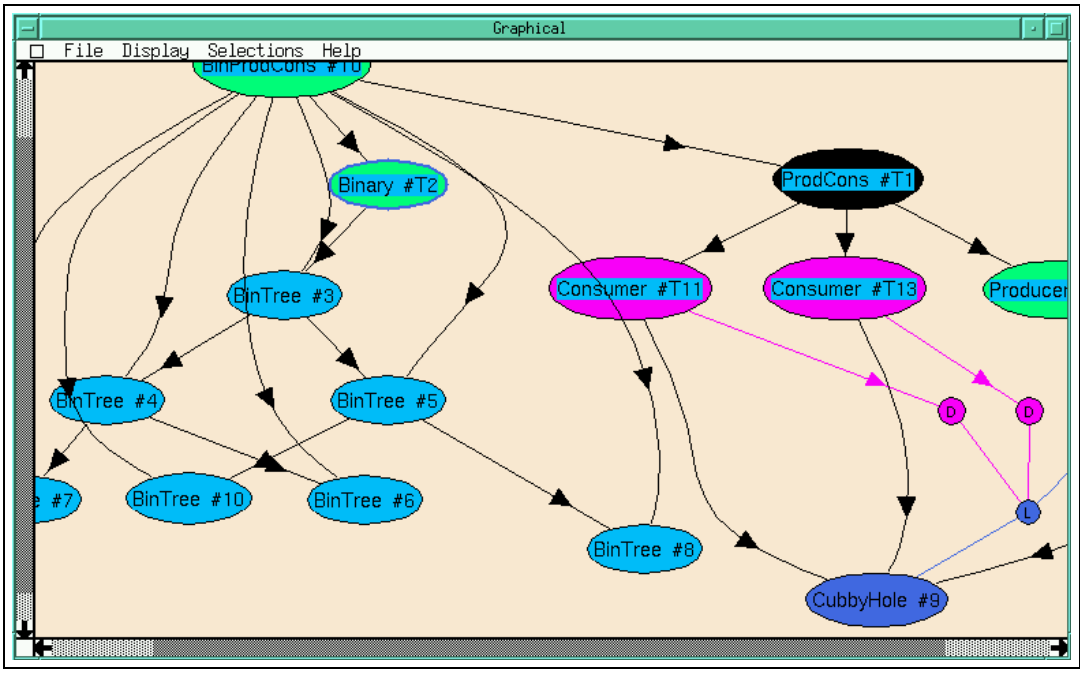


Fig. 4.17 – *Vue globale du graphe d'objets.*

ou sur l'échelle du graphe. Le problème reste donc entier si l'on considère un graphe comportant un grand nombre d'objets.

Une seconde idée, paraissant régler le problème de lisibilité d'un graphe important, consiste à proposer un zoom de la vue graphique. Cette possibilité est offerte par notre environnement, à l'aide du menu déroulant *Display* proposant les options *Zoom In* et *Zoom Out*. Cependant, ce zoom ne manipule aucunement le graphe, mais uniquement la vue donnée, c'est-à-dire que lorsque l'utilisateur demande un zoom (*Zoom In*), les noeuds apparaissent plus gros (voir Figure 4.18), et par conséquent des objets qui étaient visibles ne le sont plus. De même, l'action *Zoom Out* diminue la taille des noeuds (voir Figure 4.19), ce qui fait que plus d'objets sont visibles dans la fenêtre, mais il devient très difficile de discerner le nom des objets et la topologie précise du graphe d'objets.

Quand un graphe comporte un grand nombre d'objets, une possibilité intéressante est de pouvoir discerner des sous-ensembles au sein de ce graphe. Dans le cadre de la programmation objet, cela peut aider le programmeur à reconnaître des sous-graphes classiques (*patterns*). D'autre part, disposer du graphe d'accessibilité d'un thread (graphe formé par les objets pouvant être atteints directement ou non par le thread considéré) est d'une aide précieuse pour la mise au point des programmes, par exemple pour détecter une absence non désirée de référence (voir Section 4.4.2).
Dans le cadre de programmes concurrents il est très souvent nécessaire d'analyser pré-

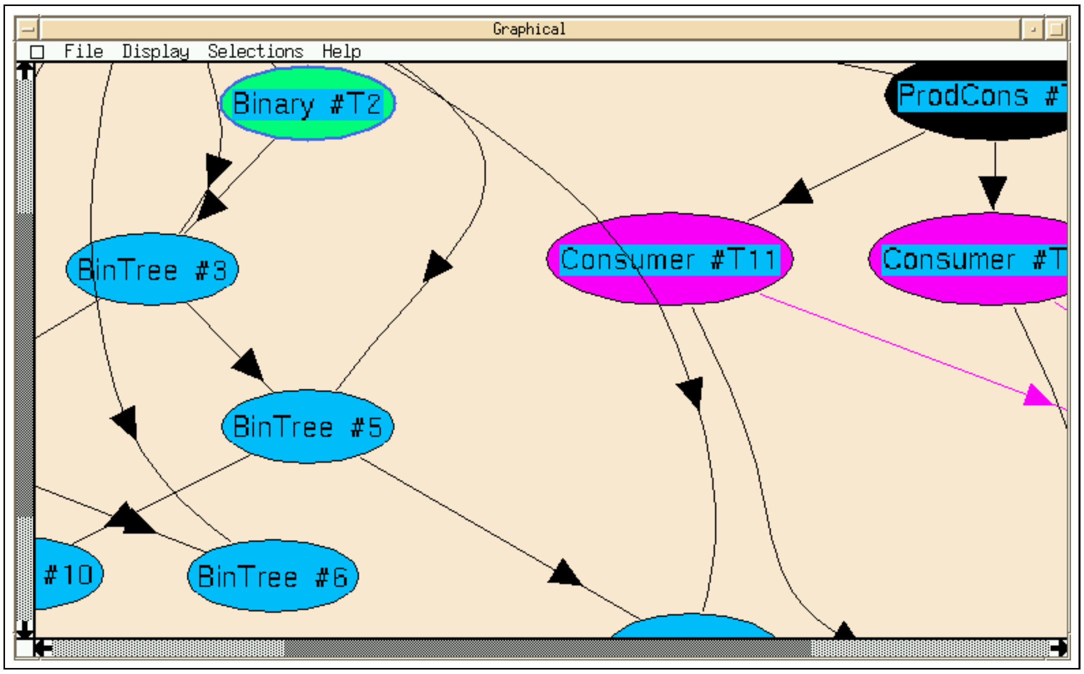


FIG. 4.18 – *Vue du graphe d'objets après l'action Zoom In.*

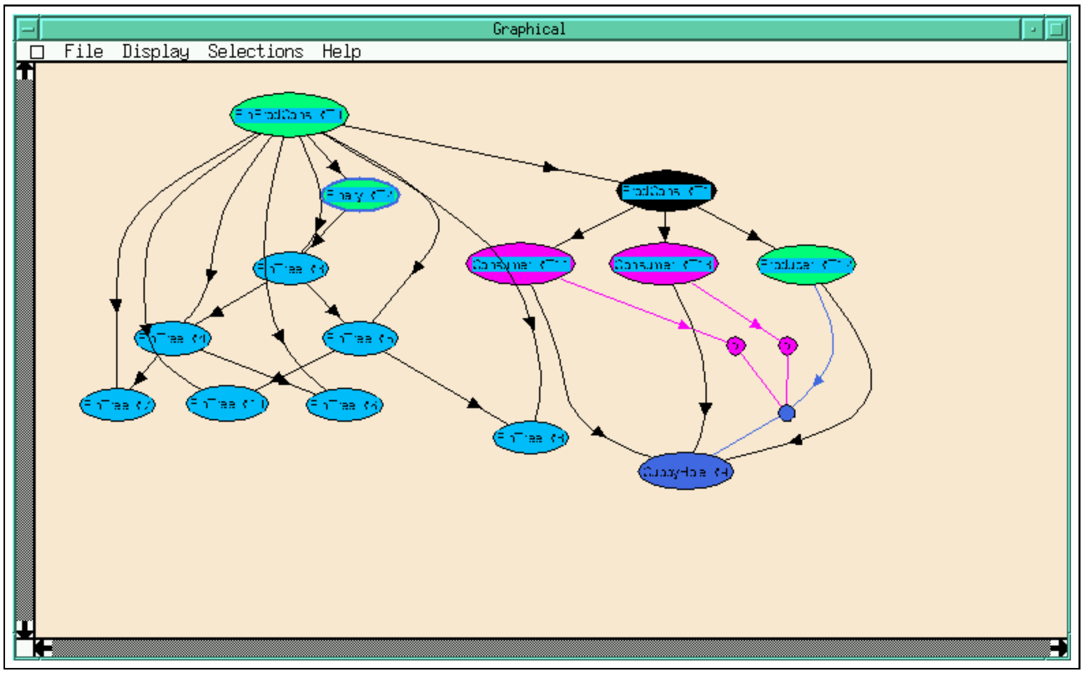


FIG. 4.19 – *Vue du graphe d'objets après l'action Zoom Out.*

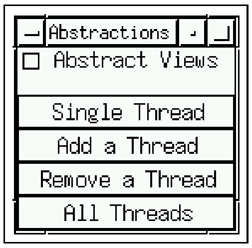


FIG. 4.20 – *Menu des abstractions.*

cisément le comportement d'un nouveau thread au sein du graphe d'exécution sans voir les autres threads lorsqu'on sait qu'ils ne posent pas de problème. Par contre il est important de savoir si les objets de ce sous-graphe sont partagés avec d'autres threads ou non afin de détecter un besoin de synchronisation.

4.5.2 Application à Jitan : les abstractions

Nous avons donc étudié cette possibilité afin de l'inclure à notre environnement. La suite de cette section présente notre solution au problème de la visualisation des graphes d'exécution comportant un grand nombre d'objets : les *abstractions*, et donne une idée de la spécification de ce mécanisme.
Les abstractions permettent à l'utilisateur d'indiquer un thread dont il désire visualiser le sous-graphe (tous les objets accessibles par ce thread directement ou non). Ce sous-graphe apparaît alors dans la fenêtre graphique à la place du graphe global, à la même échelle, ce qui permet de bien discerner les objets et leurs types. A chaque instant, l'utilisateur peut aussi choisir de revenir à la vue du graphe global.

4.5.2.1 Fonctionnalités offertes

Au sein de notre environnement, le programmeur peut utiliser le mécanisme d'abstractions grâce à la fenêtre du même nom présentée en Figure 4.20.
Cette fenêtre propose quatre options :

- *Single Thread*
- *Add a Thread*
- *Remove a Thread*
- *All Threads*

Cette section présente successivement ces quatre possibilités.

Nouvelle abstraction
Quand l'utilisateur clique sur l'option *Single Thread* de la fenêtre de la Figure 4.20, une fenêtre (présentée en Figure 4.21) apparaît et permet à l'utilisateur de rentrer le nom du thread dont il veut visualiser le sous-graphe.
Prenons comme exemple le graphe de la Figure 4.17. Si l'on choisit de visualiser le

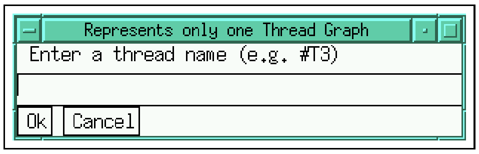


FIG. 4.21 – *Fenêtre de choix du thread dont on veut le sous-graphe.*

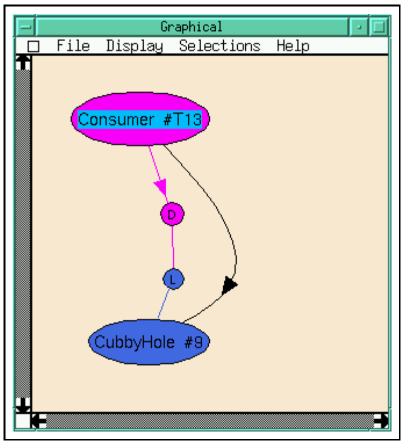


FIG. 4.22 – *Vue du sous-graphe du thread* `#T13`.

sous-graphe du thread `#T13`, on obtient la vue de la Figure 4.22. Cette vue permet de comprendre que le thread `#T13` est dans le statut *dormant* (couleur pourpre), et que lorsqu'il quittera ce statut, il voudra acquérir un lock (ou plus) sur l'objet `CubbyHole #9`. De plus la couleur du noeud `L` nous apprend que le lock sur l'objet `CubbyHole #9` est possédé par un autre thread (sur lequel le thread `#T13` ne possède pas de référence).

Ajout d'un sous-graphe

Notre environnement permet d'ajouter des sous-graphes à la vue graphique lorsqu'elle visualise déjà un sous-graphe associé à un thread.

Lorsque l'utilisateur clique sur l'option *Add a Thread* de la fenêtre de la Figure 4.20, une fenêtre du même type que celle présentée en Figure 4.21 apparaît. Elle permet à l'utilisateur de choisir le thread dont il veut visualiser le sous-graphe. Ce dernier est alors visualisé dans la même vue graphique.

Si l'on reprend l'exemple ci-dessus, et que l'on demande l'ajout du sous-graphe associé au thread `#T2`, la vue graphique est modifiée et visualise les deux sous-graphes comme cela est présenté dans la Figure 4.23.

Cette vue permet de réaliser que les deux threads `#T2` et `#T13` sont indépendants : aucun ne possède une référence, directe ou non, sur l'autre. On peut d'autre part remarquer de façon directe le pattern d'arbre binaire présenté par le sous-graphe du thread `#T2`, ce qui n'était pas évident sur la vue globale de la Figure 4.17.

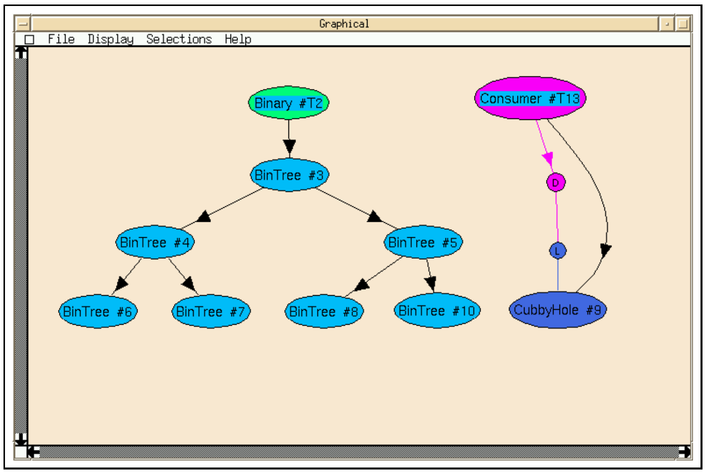


FIG. 4.23 – *Ajout du sous-graphe du thread* `#T2`.

Suppression d'un sous-graphe

De même qu'il est possible d'ajouter un sous-graphe, l'utilisateur peut aussi en supprimer un grâce à l'option *Remove a Thread* du menu de la Figure 4.20.
Si dans l'exemple précédent l'utilisateur est surtout intéressé par le déroulement de l'exécution du thread `#T2` (l'arbre binaire), il peut demander la suppression du sous-graphe associé au thread `#T13`. Il obtient ainsi la vue graphique de la Figure 4.24.

Retour à la vue globale

L'utilisateur peut, à tout moment, redemander la vue graphique globale grâce à l'option *All Threads*. Il obtient alors la vue de la Figure 4.25.

Naturellement, comme on peut le voir avec ces exemples, la visualisation des statuts et des synchronisations n'est pas supprimée ni modifiée par le mécanisme d'abstraction. De même, toutes les autres spécificités de la vue graphique sont conservées : animation, exécution pas-à-pas, détection de deadlock. De plus, les vues textuelles continuent à présenter les caractéristiques de la totalité des objets.

4.5.2.2 Spécification des abstractions

Le principe est de transmettre un sous-ensemble des objets à la vue graphique pour la visualisation. Ce sous-ensemble est déterminé par le set :

`getObjectsForGraphic`

appelé par une prémisse de la règle `MainRule_SendMessagesToTheEnvironment` du set `ENVsendObjectAndClassVariableLists` présenté en Figure 4.1.
Il est important de noter que seul le premier message de la clause *do* de la règle `MainRule_SendMessagesToTheEnvironment` utilise la liste des objets *ObjL1_1* obtenue

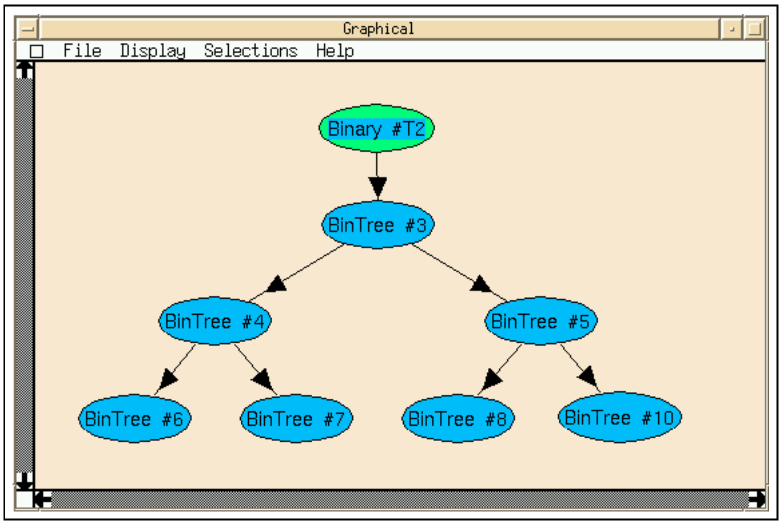


FIG. 4.24 – *Suppresion du sous-graphe du thread* `#T13`.

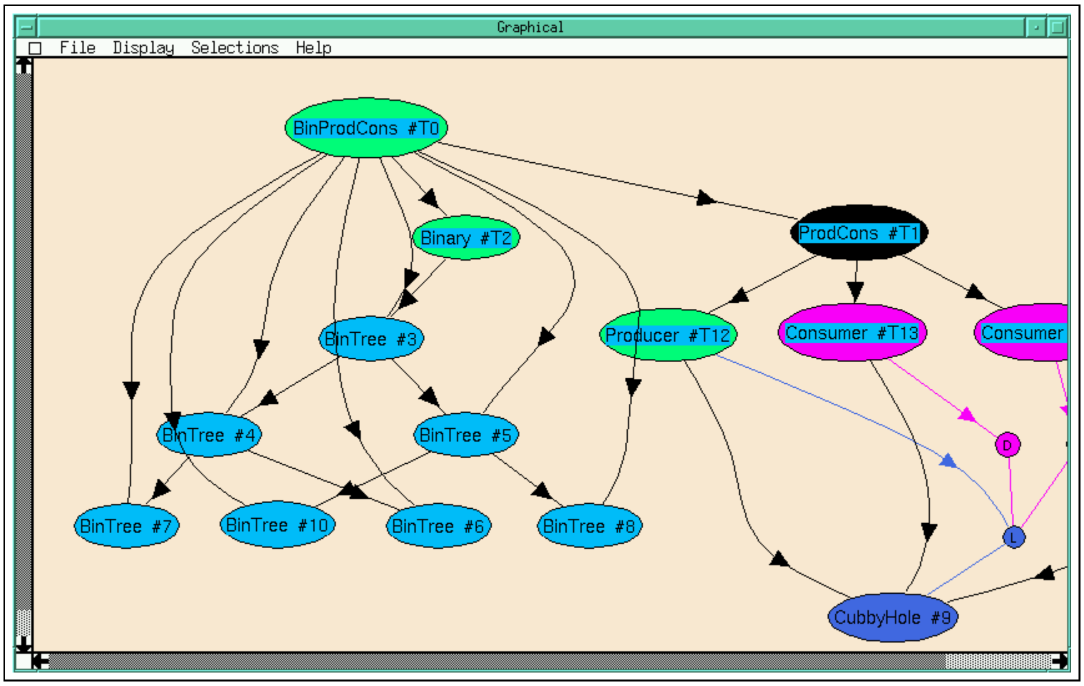


FIG. 4.25 – *Retour à la vue globale.*

```
NoAbstraction_ReturnsTheGivenLists:
   (ObjL, ClVarL -> ObjL, ClVarL) ;
            provided getAbstraction(objectNumber 0) ;
```

FIG. 4.26 – *Règle Typol traitant l'absence d'abstraction.*

```
NewAbstractionToDealWith_ReturnsTheObtainedLists:
   getAbstractionThread(OThNumber) --C: New abstraction on this thread.
                                               & getStackList(stacks(ThreadStacks, _))   &
   ObjL1, objects[], ThreadStacks |- abstractionThreads[OThNumber] -> ObjL2   &
   addObjectsCorrespondingToAttributesAndLocalVariables(ObjL1, ObjL2
                                                              |- ObjL2 -> ObjL2_1)   &
   getGoodObjectsOrder(ObjL2_1 -> ObjL2_2)   &
   checkGlobalView(ObjL1, ObjL2_2 -> ObjL3)
   ----------------
   (ObjL1, ClVarL1 -> ObjL3, ClVarL1) ;
            provided
                getAbstraction(objectNumber 1)
                    --C: There is an abstraction on a new thread
                    ;
            do setNewAbstractionThreadList(abstractionThreads[OThNumber]);
```

FIG. 4.27 – *Règle Typol traitant une nouvelle abstraction.*

après abstraction. En effet, les vues textuelles continuent de permettre la visualisation des informations sur tous les objets créés, même lorsque l'utilisateur a demandé une abstraction de la vue graphique. Grâce aux vues textuelles, l'utilisateur peut ainsi disposer d'une vue globale de l'exécution, même si tous les objets n'apparaissent pas sur le graphique.
Le set *getObjectsForGraphic* traite les différentes possibilités offertes par les abstractions :

- Le cas sans abstraction est spécifié par la règle présentée en Figure 4.26. La liste des objets n'est pas modifiée car l'utilisateur n'a pas demandé d'abstraction.
- Le cas, traité par la règle présentée en Figure 4.27, où l'utilisateur a demandé une abstraction (cette condition est assurée par la clause *provided*) sur le thread indiqué (thread `#T OThNumber`). Dans ce cas, les prémisses de cette règle permettent de déterminer l'ensemble des objets accessibles (directement ou non) par le thread, que ce soit au moyen d'attributs, ou de variables locales aux méthodes appelées par le thread.
- Le cas, spécifié par la règle de la Figure 4.28, où l'on considère juste les abstractions demandées précédemment, c'est-à-dire où il n'y a pas eu de nouvelle requête de l'utilisateur. On détermine l'ensemble des objets à visualiser de la même façon que pour une nouvelle abstraction (car de nouveaux objets ont pu s'ajouter à

```
AnOldAbstractionToDealWith:
   getAbstractionThreadList(OThNumbers)  &  getStackList(stacks(ThreadStacks, _))  &
   ObjL1, objects[], ThreadStacks |- OThNumbers -> ObjL2  &
   addObjectsCorrespondingToAttributesAndLocalVariables(ObjL1, ObjL2
                                                    |- ObjL2 -> ObjL2_1)  &
   getGoodObjectsOrder(ObjL2_1 -> ObjL2_2)  &
   checkGlobalView(ObjL1, ObjL2_2 -> ObjL3)
   ----------------
   (ObjL1, ClVarL1 -> ObjL3, ClVarL1) ;
           provided getAbstraction(objectNumber 5) ;
```

FIG. 4.28 – *Règle Typol traitant les précédentes abstractions.*

```
AddAThreadToAbstraction_ReturnsTheObtainedLists:
   getAbstractionThread(OThNumber) --C: New abstraction on this thread.
                                    &  getAbstractionThreadList(OThNumbers1)  &
   getStackList(stacks(ThreadStacks, _))  &
   getAbstractionList(OThNumber |- OThNumbers1 -> OThNumbers1_1)  &
   ObjL1, objects[], ThreadStacks |- OThNumbers1_1 -> ObjL2  &
   addObjectsCorrespondingToAttributesAndLocalVariables(ObjL1, ObjL2
                                                    |- ObjL2 -> ObjL2_1)  &
   getGoodObjectsOrder(ObjL2_1 -> ObjL2_2)  &
   checkGlobalView(ObjL1, ObjL2_2 -> ObjL3)
   ----------------
   (ObjL1, ClVarL1 -> ObjL3, ClVarL1) ;
           provided getAbstraction(objectNumber 2) ;
           do setNewAbstractionThreadList(OThNumbers1_1);
```

FIG. 4.29 – *Règle Typol traitant de l'ajout d'un sous-graphe.*

l'ensemble précédent, par exemple s'il y a eu création d'objet ou affectation d'une référence à une variable). Etant donné que notre spécification agit sur la structure sémantique globale, il est nécessaire de recalculer le sous-ensemble à représenter à chaque étape de l'exécution.

- Le cas, spécifié par la règle de la Figure 4.29, où l'utilisateur a demandé d'ajouter le sous-graphe du thread *#T OThNumber* à la vue graphique précédente qui tenait déjà compte d'autres abstractions. Dans ce cas, le nouveau thread est ajouté à la liste des threads dont on visualise les sous-graphes, et l'on détermine l'ensemble des objets à représenter dans la vue graphique, sans se soucier de l'appartenance des objets à tel ou tel sous-graphe : on calcule juste l'ensemble des objets accessibles par les threads dont on veut visualiser les sous-graphes. C'est le moteur de visualisation graphique qui se chargera de dessiner les sous-graphes en traduisant chaque référence en une arête du graphe.
- Le cas, spécifié par la règle de la Figure 4.30, où l'utilisateur a demandé de supprimer le sous-graphe du thread *#T OThNumber* à la vue graphique précédente

```
ThreadRemovedFromAbstraction_ReturnsTheObtainedLists:
   getAbstractionThread(OThNumber) --C: New abstraction on this thread.
                                         & getAbstractionThreadList(OThNumbers1) &
   getStackList(stacks(ThreadStacks, _)) &
   getAbstractionThreads(OThNumber |- OThNumbers1 -> OThNumbers1_1) &
   ObjL1, objects[], ThreadStacks |- OThNumbers1_1 -> ObjL2 &
   addObjectsCorrespondingToAttributesAndLocalVariables(ObjL1, ObjL2
                                                       |- ObjL2 -> ObjL2_1) &
   getGoodObjectsOrder(ObjL2_1 -> ObjL2_2) &
   checkGlobalView(ObjL1, ObjL2_2 -> ObjL3)
   ----------------
   (ObjL1, ClVarL1 -> ObjL3, ClVarL1) ;
           provided
              getAbstraction(objectNumber 3)
                 --C: The user wants to remove one thread from the abstraction
                 ;
           do setNewAbstractionThreadList(OThNumbers1_1);
```

FIG. 4.30 – *Règle Typol traitant de la suppression d'un sous-graphe.*

```
ReturnToGlobalView_ReturnsTheGivenLists:
   (ObjL, ClVarL -> ObjL, ClVarL) ;
            provided getAbstraction(objectNumber 4);
            --C: The user chooses to return to global view.
            do setNewAbstractionThreadList(abstractionThreads[]);
```

FIG. 4.31 – *Règle Typol traitant du retour à la vue globale.*

qui tenait déjà compte d'abstractions. Dans ce cas, le thread est supprimé de la liste des thread dont on veut visualiser les sous-graphes, et l'on détermine la liste des objets à représenter à partir de la nouvelle liste des threads.

– Le cas, spécifié par la règle de la Figure 4.31, où l'utilisateur a demandé de revenir à la vue globale du graphe d'objets. dans ce cas, la vue graphique doit représenter l'ensemble de tous les objets créés, et l'on efface la liste des threads dont on voulait précédemment visualiser les sous-graphes.

Les apports offerts par cette méthode d'abstraction sont très importants pour la compréhension d'une application Java. En effet, cela permet de mettre en évidence des modèles récurrents comme le montre l'exemple précédent, ou de se concentrer sur des sous-graphes indépendants à l'exécution.

Nous avons aussi étudié d'autres possibilités d'abstraction, comme par exemple, la visualisation de tous les objets d'un type donné, ou la visualisation de tous les threads pouvant accéder à un objet donné. Nous n'avons pas implémenté ces possibilités, mais le mécanisme de spécification serait assez proche du précédent : mise en mémoire du ou des objets considérés, et parcours de l'ensemble total des objets afin de déterminer le sous-ensemble à représenter dans la vue graphique.

4.6 Visualisation du masquage des attributs : des pistes

La prise en compte du masquage des attributs par notre sémantique entraîne aussi des transformations au niveau de la visualisation des objets, au niveau textuel mais aussi et surtout au niveau graphique.
Il est possible d'envisager différentes solutions que nous allons étudier à travers l'exemple donné à la Figure 4.32 . Cet exemple, plus complexe que celui présenté en Figure 3.38, nous permet de réaliser une étude plus poussée du mécanisme de masquage des attributs. Il serait très difficile de trouver un exemple réaliste mettant en jeu ce mécanisme, car le masquage des attributs est très peu utilisé en programmation.
Considérons trois classes A, B et C où la classe C hérite de la classe B et où la classe B hérite de la classe A. Prenons le cas des instructions suivantes :

```
A a;
B b;
C c = new C();
b = c;
a = c;
```

Les trois variables a, b, et c référencent alors le même objet (soit `C #5` cet objet) de type C, tout en possédant des types statiques différents. Elles permettent donc d'atteindre des attributs différents en fonction de leur type statique.
Au sein de la Figure 4.32, nous pouvons dinstinguer trois catégories d'attributs :

- L'attribut `int w` qui est déclaré dans la classe `A`, et qui est hérité normalement par les autres classes.
- Les attributs `x, y, et z` qui sont masqués au cours de l'héritage.
- Les attributs `int i, j et k` qui sont propres à la classe `C`.

La première solution permettant de représenter l'objet référencé par ces trois variables tout en tenant compte du masquage des attributs est donnée à la Figure 4.33. Il s'agit d'une solution textuelle, toujours réalisable. Cette solution bien que très simple du point de vue de la représentation, présente des inconvénients : elle manque de clarté quant au référencement de l'objet par les différentes variables et il est difficile de l'intégrer au sein de notre représentation graphique.

Deux autres solutions sont présentées dans les Figures 4.34 et 4.35 :

1. Figure 4.34. Principe : les attributs communs à toutes les classes, ici l'attribut `int w`, sont accessibles directement. On peut trouver les autres attributs (masqués ou propres à un type donné) grâce au type statique (A, B ou C) de la variable référençant l'objet.
2. Figure 4.35. Principe : la variable référençant l'objet permet de connaître directement la liste des attributs accessibles, qu'ils soient communs ou masqués.

La solution de la Figure 4.35 a l'avantage de connaître directement les attributs accessibles par une référence donnée par un simple parcours de la structure sémantique

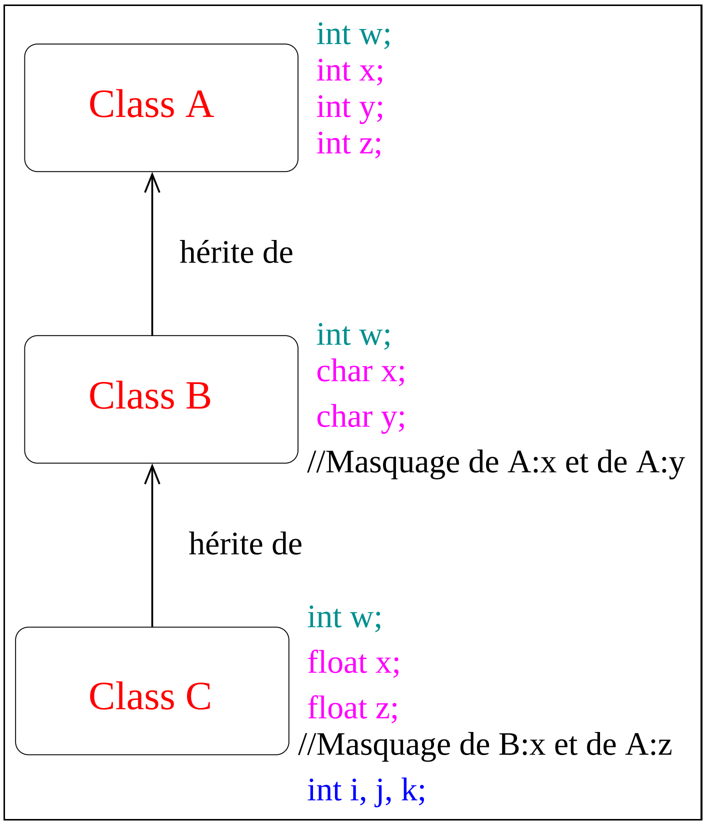


FIG. 4.32 – *Exemple de masquage des attributs.*

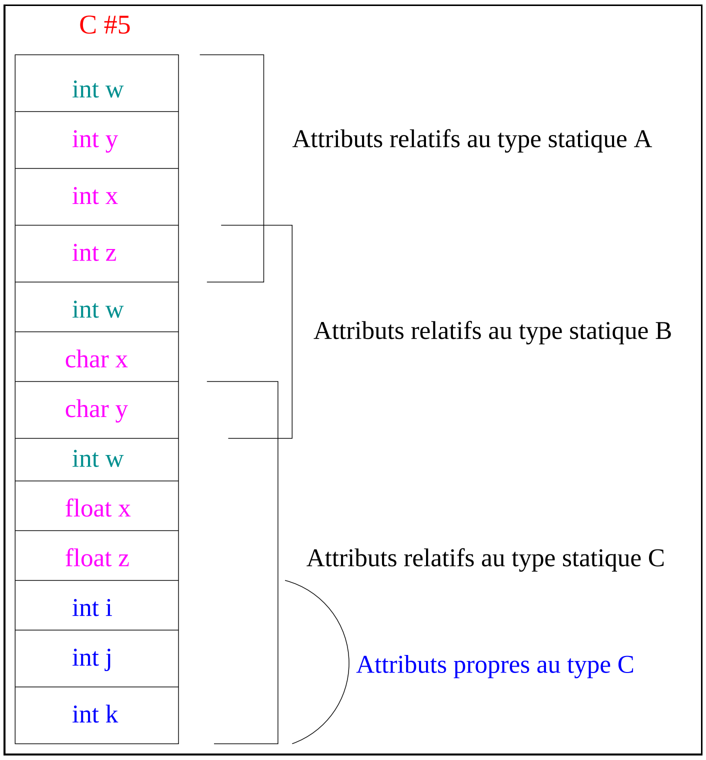


FIG. 4.33 – *Première solution pour la visualisation du masquage des attributs.*

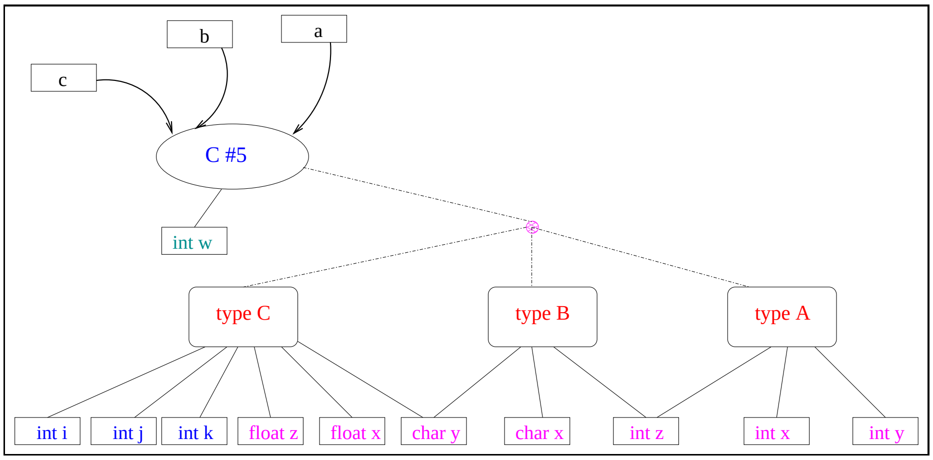


FIG. 4.34 – *Deuxième solution pour la visualisation du masquage des attributs.*

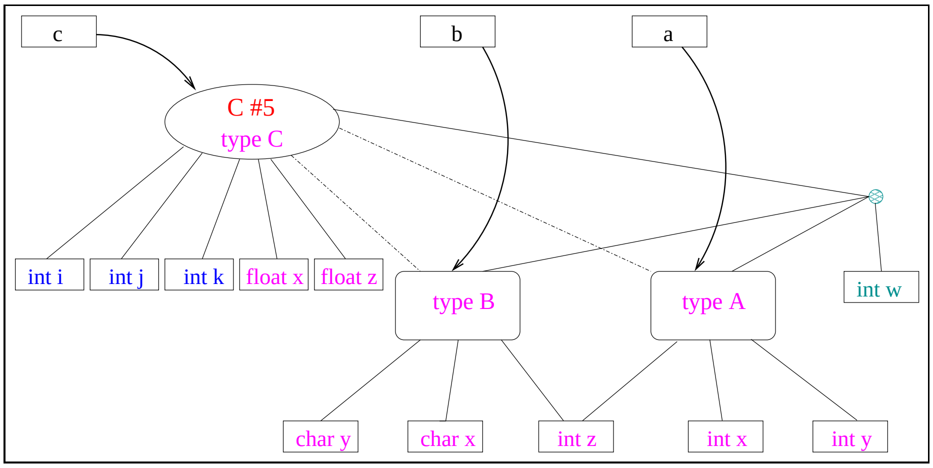


FIG. 4.35 – *Troisième solution pour la visualisation du masquage des attributs.*

référencée, contrairement à la deuxième solution qui nécessite de remonter aussi à la racine de l'objet pour connaître les attributs communs.
Par contre, elle impose de remonter à la racine de l'objet pour visualiser la sémantique de l'exécution du programme, par exemple lors de l'affectation :

```
C c2;
c2 = b;
```

Il apparaît donc que la solution de la Figure 4.34 est préférable. De plus, dans la cadre d'un programme créant un grand nombre d'objets, cette solution permet d'obtenir un graphique moins complexe.

4.7 Travaux connexes et discussion

De nombreux outils ont été développés afin d'accompagner le programmeur tout au long de l'écriture et de la mise au point de ses programmes. Nous présentons dans un premier temps les principaux outils disponibles sur le marché afin de pouvoir dans un second temps établir une comparaison entre ceux-ci et notre environnement.

4.7.1 Travaux connexes

Cette section présente différents outils qui ont été développés afin d'aider les programmeurs à écrire des programmes, ou à comprendre l'exécution de certains programmes, en particulier dans le langage Java, ou plus généralement des programmes orientés objets et/ou concurrents.

La visualisation de programmes est un secteur actif de recherche, utilisée dans dif-

férents domaines tels que la conception de logiciels, l'évaluation des performances, et l'apprentissage de la programmation.
On peut citer comme systèmes représentatifs Zeus [21] pour l'animation d'algorithmes, Incense [101] pour l'affichage des structures de données, Pastis [100] pour l'animation de programmes, Polka [131] pour l'animation de programmes sur des architectures parallèles, et Pavane [124] pour la visualisation de l'exécution de programmes concurrents (voir [79] pour une synthèse sur la visualisation des programmes parallèles).

4.7.1.1 Visualisation de la structure des programmes

La visualisation de la structure des programmes (par exemple de la hiérarchie des classes dans le cas du langage Java) représente une première catégorie parmi les systèmes directement liés à l'étude des langages de programmation et à leur utilisation. Il s'agit de l'affichage d'informations statiques sur le programme grâce à différents browsers.

CodeCenter [27] fournit des browsers graphiques affichant une trace des appels de fonctions et de l'utilisation des variables afin de détecter des erreurs statiques, et d'examiner et contrôler les différents modules et bibliothèques au sein d'un projet. CIA++ [62] bâtit une base de données d'informations sur des programmes C++, qui est ensuite utilisée pour afficher différentes vues de la structure des programmes. GraphLog [34], qui est un outil générique muni d'un langage de requêtes, permet de visualiser les différentes structures d'un logiciel et d'effectuer des requêtes sur celui-ci. Object Explorer [11] de Kent Beck est quant à lui dédié à la visualisation de programmes SmallTalk.

Felleisen et al., dans [55, 54], proposent un environnement graphique de développement de programmes Scheme nommé DrScheme. Cet environnement permet de visualiser le source du programme et fournit un outil interactif d'analyse statique appelé MrSpidey. Cet outil détecte des violations potentielles de sécurité avant l'exécution du programme, et explique le problème potentiel en affichant un graphe de flots de valeurs au sein du texte du programme.

4.7.1.2 Visualisation de l'exécution des programmes

Une autre catégorie importante parmi les outils de visualisation de programmes est celle des outils permettant de visualiser des informations dynamiques comme par exemple la création des objets au cours de l'exécution.

ObjView [59], qui est un système de conception de tableaux de bord électroniques basé sur un modèle écrit en C++, offre une interface avec à la fois une vue réaliste des objets du problème et des objets C++, avec une bonne visualisation des instances, des données membres et des fonctions. CodeCenter semble présenter les mêmes possibilités. Haarslev et Moeller, dans [66], proposent une structure représentant à la fois la hiérarchie des classes du programme, et les objets créés en utilisant l'architecture de niveau méta CLOS [113] ; une association explicite entre les objets visualisés et les objets de l'application est alors nécessaire. Object Visualizer [115, 116] et HotWire [80] sont quant à eux basés sur des mécanismes d'instrumentation de programmes et offrent les mêmes possibilités.

Plus récemment est apparu le besoin de traiter de grandes quantités d'objets, ce qui pose des problèmes, notamment au niveau graphique comme nous venons de le voir (tous les objets ne rentrent pas dans la vue graphique, ou alors à une trop petite échelle pour être utilisables). Kent a proposé une notation visuelle des invariants (à la fois au niveau des classes et au niveau des instances) pour des modèles à objets [78]. Lencevicius, Hölzle, et Singh développent le débuggage à base de requêtes qui permet d'après eux des manipulations efficaces d'une grande quantité d'objets [92]. Jerding, Stasko, et Ball se concentrent sur la compréhension des interactions entre classes et objets durant l'exécution des programmes [75].

4.7.1.3 Environnements de visualisation Java

En ce qui concerne le langage Java, plusieurs systèmes commerciaux d'aide au développement de programmes existent [1] : Visual J++ de Microsoft [63], Kawa [141], JavaWorkshop [91], Javelin [132], VisualAge pour Java d'IBM [73], Metrowerks CodeWarrior [96], Assure [52], Asymetrics SuperCede Java [137], et Visual Threads de Compaq [33] qui propose un graphique décrivant de façon continue l'état des threads du programme, et la fréquence des évènements liés aux thread. Le lecteur intéressé peut se reporter à [53] pour une présentation assez exhaustive des différents outils du commerce, et [43] pour une revue des différents outils de visualisation Java.

4.7.2 Discussion

Les composants avancés de notre environnement, qui lui permettent de se démarquer des outils du commerce et d'apporter une certaine contribution au domaine des environnements de développement, sont les suivants.

Le premier aspect à considérer est la visualisation graphique de la topologie du graphe d'objets. Bien entendu de tels environnements graphiques existent : Javelin et CodeWarrior sont, parmi les outils du commerce, ceux qui sont radicalement orientés vers les graphes avec par exemple des classes représentées comme des boites connectées par des lignes symbolisant les relations d'héritage. Les boites contiennent les attributs et les méthodes de la classe concernée et peut être éditée graphiquement. Mais ces informations sont statiques (par exemple la hiérarchie des classes), et nous croyons que la visualisation graphique d'informations dynamiques comme les références entre objets est un des points essentiels pouvant aider un programmeur. De plus, la visualisation des synchronisations entre threads est d'une importance majeure dans la programmation

1. JDB [135] qui n'est pas un environnement de développement à proprement parler, mais plutôt un débugger en ligne, et les profileurs tels JProbe [109] basés sur une étude "off-line" du graphe des appels et de l'utilisation de la mémoire sont omis dans cette liste.

concurrente.
Des informations dynamiques (sur les objets) sont données par des systèmes comme SuperCede Java, CodeWarrior, Kawa , Visual Threads ou NuMega, mais leur présentation graphique est sous forme d'arbres, et non de graphes. Metamata présente principalement des informations textuelles, comme le source du programme, un interpréteur de commandes en ligne, et la valeur de certaines variables sélectionnées : les références sont étendues vers les objets correspondants. Il n'y a pas de vue globale, le programmeur doit donc comparer les valeurs des variables pour détecter la présence d'objets partagés ou de cycles. Ces informations sont apparentes de façon immédiate avec notre outil, grâce à la visualisation orientée vers les graphes de la topologie des objets.

Le deuxième aspect est lié au débuggage interactif : aucune compilation du programme n'est nécessaire avec notre outil, à l'opposé de la re-compilation incrémentale nécessaire habituellement après toute modification ou addition d'un point d'arrêt. Avec Jitan, quand un programme est modifié (après arrêt de l'exécution), il suffit de relancer l'exécution. L'exception est SuperCede Java qui permet de modifier les programmes en cours d'exécution, sans pour autant recommencer celle-ci entièrement.
Cette possibilité offerte par Jitan provient du fait que notre environnement n'est pas basé sur l'exécution du byte-code instrumenté, ou n'utilise pas une machine virtuelle particulière (à la différence de la plupart des outils du commerce), mais utilise directement une sémantique opérationnelle, instrumentée avec des notifications (comme cela est expliqué dans le paragraphe 4.1). Cela ne demande donc aucune instrumentation de la part de l'utilisateur à l'inverse de l'outil JDI [136] qui est une bibliothèque Java qui peut seulement être utilisée comme base pour la construction d'un environnement de débuggage.

4.8 Conclusion

Ce chapitre nous a permis de présenter notre environnement de visualisation de l'exécution de programmes Java basé sur la sémantique formelle opérationnelle de Java que nous avons spécifiée. Cet environnement est développé en Lisp et représente environ 6000 lignes de code. Les arbres de syntaxe abstraite des structures sémantiques sont manipulés grâce au formalisme VTP.
Les différentes vues proposées sont : des vues textuelles (visualisation de la liste des objets avec leur type et leurs attributs, visualisation des activités des threads, vues récapitulatives des statuts des threads et des objets, et des interactions entre threads) et surtout une vue graphique mettant en évidence la topologie du graphe des objets, les statuts des objets et des threads, les différentes synchronisations et locks existant au niveau des objets. Cette visualisation est animée, et les vues sont mises à jour de façn incrémentale par soucis d'efficacité.
Ces différentes vues fournissent une aide précieuse à la mise au point de programme, en permettant par exemple de repérer la nécessité d'un synchronisation au niveau d'un objet lorsque deux threads peuvent y accéder, ou la présence d'un deadlock.

D'autre part, nous avons développé un mécanisme d'abstraction permettant de visualiser beaucoup plus clairement les graphes contenant beaucoup d'objets. L'utilisateur peut ainsi se concentrer sur l'exécution d'un thread donné ou sur l'exécution d'une partie indépendante du programme.
La présentation de notre vue graphique ainsi que le mécanisme d'abstraction constituent les points forts de l'environnement Jitan et apportent une contribution certaine au domaine de la visualisation de l'exécution de programmes Java.

Chapitre 5

Propriétés du Langage Java

Ce chapitre présente la suite logique de notre travail de spécification formelle du langage Java, c'est-à-dire la vérification de propriétés de la sémantique précédemment définie en Typol (Chapitre 3).
Considérons les propriétés de la synchronisation. Le fait de savoir que, par exemple, la propriété selon laquelle *deux threads ne peuvent détenir un lock sur un même objet à une même étape de l'exécution* est vérifiée, assure à un programmeur utilisant des méthodes *synchronized* que ses données seront protégées de tout accès simultané en écriture. Le respect de cette propriété par notre spécification introduit une confiance du programmeur en la sémantique. Il est donc primordial que notre sémantique vérifie ces propriétés définies au sein de la sémantique informelle du langage Java [61].
La plupart de ces propriétés sont incluses de fait dans notre spécification ce qui garantit une certaine correction de notre sémantique par rapport à ces propriétés.

Une première section décrit les techniques de base couramment utilisées à des fins de vérification ou de preuves de propriétés. Une deuxième section étudie alors des propriétés de notre sémantique liées aux aspects concurrents du langage Java. Nous montrons comment ces propriétés sont soit représentées directement par la syntaxe abstraite des structures sémantiques que nous avons définies, soit directement exprimées par notre spécification Typol. Dans le cas où ces propriétés ne sont pas directement exprimées dans notre modélisation, nous définissons des sets Typol dédiés à l'expression et à la vérification de ces propriétés. Nous présentons enfin une preuve d'une de ces propriétés.

5.1 Techniques de base

Différentes techniques sont utilisées pour prouver des propriétés de programmes ou de langages. Cependant on peut distinguer quatre techniques principales présentées dans cette section en s'appuyant sur des applications à Java et à la concurrence.

5.1.1 Analyse de flots de données

L'analyse des flots de données est utilisée pour vérifier statiquement des propriétés de programmes concurrents [51, 102]. Cette approche est basée sur la modélisation du programme Java considéré sous forme d'un graphe des flux.
Dans [51], Dwyer et Clarke décrivent un analyseur qui prend en entrée un programme Java et une propriété de correction et vérifie si l'exécution du programme satisfait cette propriété.

Dans [102], les auteurs décrivent la possibilité d'utiliser l'analyse de flots de données pour la vérification de propriétés de programmes Java concurrents. Plus précisément, ils montrent comment leur système d'analyse de flots de données, appelé Flavers, peut être adapté aux programmes Java concurrents. Il s'agit d'une vérification statique d'une propriété sur un programme modélisé comme un graphe de flots à l'aide d'un algorithme d'analyse de flots de données amélioré par la fourniture d'informations complémentaires représentées par automate à états finis.
L'article décrit d'autre part des modèles de communications entre threads pouvant conduire à des erreurs et montre comment Flavers permet de déceler la présence de telles communications. La sémantique des communications entre threads est spécifiée à l'aide de contraintes de réalisabilité. Les propriétés étudiées sont la présence de deadlocks ou de livelocks ainsi que des propriétés spécifiques à des programmes donnés. Les propriétés non-spécifiques sont d'assez bas niveau, comme par exemple le fait que la méthode *start* d'un thread ne soit pas appelée plusieurs fois.

5.1.2 Algèbres de processus

Le point clé des algèbres de processus (CCS [60] et CSP [84]) et des langages de programmation synchrones (comme Esterel [12] et Lustre [69]) est la notion de *comportement abstrait*, qui dans les systèmes concurrents consiste essentiellement en l'interaction du système avec son environnement. Les communications se font à travers des événements, qui sont des abstractions des canaux de communication.
Les programmes peuvent être liés par des combinateurs qui agissent sur les comportements et le résultat final de cette combinaison de programmes est un autre comportement. Ces combinateurs peuvent être notamment des combinateurs de composition parallèle permettant à deux programmes de s'exécuter de manière concurrente tout en communicant, ou des combinateurs de préemption permettant à des programmes d'être contrôlés par des événements.

Dans [133], Stiles défend l'idée que les problèmes de deadlocks et livelocks peuvent être éliminés en basant la conception des applications multithreadées sur les structures et les formalismes des Processus Séquentiels Communicant (CSP) de Hoare. D'autre part, cet article définit les deadlocks comme ayant lieu lorsqu'il existe un cycle de requêtes de communication entre processus non accordées. Les livelocks quant à eux se produisent quand il y a un cycle interne de communication entre processus sans interaction avec le monde extérieur.

Triveni

Triveni [30] est une algèbre de processus augmenté d'actions d'entrées-sorties et de combinateurs de préemption, en présence de *fairness*. Ce système a été implémenté sous forme d'une API Java (appelée *JavaTriveni*). Il est utilisé dans le cadre de la programmation concurrente avec des threads et des événements. Il forme une collection d'outils pour le langage Java et inclut un environnement de tests basé sur une spécification qui automatise les tests de propriétés de sûreté exprimées en logique temporelle en temps linéaire (PLTL).
Dans [32], les auteurs décrivent les sémantiques opérationnelle, dénotationnelle et logique de Triveni et prouvent des théorèmes de correspondance entre ces trois sémantiques.
Dans [31], les auteurs décrivent l'application de Triveni à un projet portant sur un composant utilisé dans les télécommunications, et vérifient des propriétés de sûreté spécifiques à cette application, comme par exemple le fait que le temps de réponse maximum (fixé auparavant) à une requête ne soit pas dépassé, ou que lorsqu'un processeur ne répond pas à une requête en moins de n unités de temps, cette dernière soit abandonnée, et le processeur suivant requis.

5.1.3 Model-checking

Une des techniques les plus courantes dans la littérature pour la vérification de propriétés de programmes Java concurrents est l'utilisation du model-checking afin de pouvoir étudier la vérification des propriétés considérées lors de toutes les exécutions possibles d'un programme Java.

Le model-checker SAL

Dans [114], les auteurs présentent les premiers résultats obtenus lors de l'utilisation du *model-checking* sur des programmes Java concurrents. Ces derniers sont traduits en un langage intermédiaire appelé SAL (*Symbolic Analysis Laboratory*). Le model-checker associé à ce langage analyse alors le programme afin de rechercher des deadlocks ou des échecs de certaines hypothèses émises par l'utilisateur. Le programme d'entrée peut être donné sous forme de source ou de bytecode.

Le model-checker SPIN

Spin [71] est un model-checker permettant la vérification formelle des systèmes distribués. Il utilise un langage de haut niveau appelé Promela afin de spécifier la description des systèmes logiciels. Spin permet de vérifier la cohérence logique d'une spécification, de détecter des deadlocks et des livelocks, des *race conditions*, des hypothèses non garanties au sujet des vitesses relatives des processus, etc. Spin est basé sur la logique temporelle en temps linéaire. Il est particulièrement adapté à la vérification des systèmes concurrents et surtout des protocoles de communication de données. Le système à vérifier est tout d'abord décrit en Promela, qui est le langage de modélisation associé à Spin. A partir de ce modèle Spin peut soit effectuer des simulations aléatoires de l'exécution du système, soit générer un programme C qui effectue une vérification en ligne efficace des propriétés de correction du système.

De nombreux outils utilisent le model-checker Spin ou permettent de traduire des programmes Java vers Spin. Cependant, Spin comporte actuellement certaines faiblesses, notamment en ce qui concerne l'absence de gestion dynamique de la mémoire, ou la difficulté d'expression des propriétés du programme considéré pour un programmeur n'étant pas familier avec la logique temporelle.

Le système Bandera [35] fournit des outils pour compiler un programme Java en un modèle à états finis qui peut être utilisé par différents model-checkers (comme Spin, etc.). Le programme Java est tout d'abord traduit en Jimple, puis en un modèle spécifié dans un langage de commandes gardées appelé BIR (Bandera Intermediate Representation). Ce modèle peut alors être traduit en un langage d'entrée d'un model-checker. Il existe des traducteurs BIR vers Promela, et BIR vers le langage d'entrée de SMV.
Bandera est écrit en Java, mais ne traite pas les aspects suivant de Java : méthodes récursives, exceptions, inner classes, méthodes natives et API. Les propriétés à vérifier sont exprimées dans le langage BSL (Bandera Specification Language). Ce langage est basé sur un système extensible de patterns de spécification temporelle qui permettent de faciliter l'expression des propriétés en un langage plus proche du langage naturel.

Java PathFinder [67, 68, 20] (aussi appelé JPF) est un outil de traduction d'un sous-ensemble du langage Java vers le langage Promela.
Ce traducteur permet de convertir un fichier Java en un programme Promela qui peut ensuite être soumis à Spin. Le programme Java peut être équipé d'assertions, elles-mêmes étant traduites en assertions équivalentes du langage Promela. Spin permet alors de détecter des deadlocks et des violations d'assertions booléennes au sein du programme Java initial.
Le sous-ensemble de Java pris en compte dans la première version de JPF comprend : la création dynamique d'objets, avec des attributs et des méthodes, l'héritage entre classes, les threads et leur primitives de synchronisation, les exceptions, les interruptions des threads, ainsi que la plupart des constructions "simples" du langage comme les affectations, les boucles, etc. Ne sont pas pris en compte principalement la surcharge et le masquage des méthodes, les packages, et la récursion.

L'article [67] est bâti autour de l'exemple d'un buffer borné auquel accèdent des producteurs et des consommateurs. L'article [68] propose une application de JPF à la recherche de problèmes de synchronisation au sein d'un serveur de jeu d'échec chinois. L'article [20] présente l'extension de JPF en un environnement de vérification, d'analyse et de tests pour Java.

L'article [38] présente comment un modèle abstrait formel est généré à partir du code source Java grâce à un traducteur Java vers Spin appelé Java2Spin et comment ce modèle est analysé par Spin afin de détecter des deadlocks au sein de programmes Java concurrents. Cette vérification est basée sur une analyse d'accessibilité et sur l'utilisation du model-checker Spin. Cet article décrit l'outil JCAT qui propose une interface graphique qui effectue de façon transparente toutes les opérations impliquées par les processus d'analyse et de débuggage. L'article fournit des résultats expérimentaux de l'utilisation de JCAT à travers deux exemples : celui des philosophes, et celui des lecteurs écrivains.
La version actuelle de ces outils ne propose que la détection de deadlocks dans le modèle. En effet, JCAT n'offre pas de possibilités de spécification de propriétés des programmes étudiés, même si cela est envisagé pour une prochaine version. De plus, JCAT et Java2Spin ne traitent ni les exceptions, ni le polymorphisme. Cela est dû aux limitations imposées par Spin et Promela concernant l'absence de création dynamique d'objets, de la gestion du polymorphisme au niveau des appels de méthodes, et de l'absence de mécanisme permettant de décrire le phénomène de *garbage collection*.

5.1.4 Systèmes de preuves

Les systèmes de preuve sont une autre technique de base, qui est plutôt utilisée pour prouver des propriétés des langages. Cette technique est particulièrement employée pour prouver la sûreté de Java au niveau des types.
La sûreté du typage de Java est une propriété importante étudié dans [46, 47, 48, 49, 107, 108, 106, 138, 139, 118], et qui peut se résumer en *l'exécution de programmes bien typés ne peut pas mal se passer*. Dans [106], Nipkow et al. présentent une définition plus technique de la sûreté du typage : *un langage de programmation est sûr au niveau typage si et seulement si son système de types prévient les disparités de type, c'est-à-dire des situations où les valeurs obtenues dynamiquement ne se conforment pas aux types correspondant obtenus statiquement.*
La preuve de la sûreté du typage repose sur la spécification d'une sémantique statique et dynamique. Dans [118], Müller et Poetzsch-Heffter présentent une logique de programmation dans un style proche de celui d'Hoare [83] pour un noyau séquentiel de Java. Ils spécifient aussi une sémantique formelle opérationnelle de ce sous-ensemble de Java et prouvent sa sûreté en formalisant cette sémantique et les règles de typage dans une logique d'ordre supérieur.

Drew Dean [37], de son côté, étudie les interactions entre typage statique et liaison dynamique. Il prend comme base une formalisation de Java et écrit une spécification PVS sur les aspects liés à la liaison dynamique pour obtenir des théorèmes sur la sûreté du typage des programmes.

Les travaux que nous avons cités jusqu'à présent avaient principalement pour but la vérification de propriétés du langage Java. En revanche, les travaux suivants portent

sur la vérification de propriétés de programmes.
L'article [89] présente brièvement le langage JML (Java Modeling Language) et le projet LOOP. JML [88, 90] est un langage à base d'assertions permettant de spécifier des propriétés des classes et interfaces Java. Il permet d'écrire des pré et post-conditions sur le comportement de méthodes, ainsi que des invariants de classes. L'outil développé par le projet LOOP permet de traduire les annotations JML en propriétés à prouver. Ces propriétés sont utilisées dans un prouveur de théorèmes interactif comme PVS ou Isabelle pour vérifier la correction de l'implémentation Java.

L'article [120] présente une application de l'utilisation du langage JML à la spécification des API JavaCard. JavaCard est étudié car il s'agit d'un langage "réel", mais plus simple que Java car ne comprenant pas le multithreading et le chargement dynamique des classes. Dans cet article, JML est utilisé afin de décrire des conditions de terminaisons normales et anormales, à l'aide de pré-conditions et d'invariants.
Les auteurs ont développé une sémantique formelle d'un sous-ensemble de Java incluant tous les aspects de JavaCard. L'outil Loop permet alors de traduire un programme Java dans une théorie logique décrivant sa sémantique. Le format obtenu peut servir d'entrée à un prouveur de théorèmes, afin de prouver des propriétés du programme initial. Pour l'instant, il existe une interface vers Isabelle et PVS. Un développement est en cours afin que l'outil Loop accepte les annotations JML au sein du programme Java. Cet article présente uniquement la spécification des propriétés sans aller jusqu'aux preuves.

L'article [119] étend le travail décrit dans l'article précédent à la vérification du code muni d'annotations à l'aide du prouveur de théorèmes PVS.

La plupart des travaux présentés ici reposent sur une traduction de programmes Java ou d'une sémantique du langage dans un autre formalisme permettant d'exprimer des propriétés. Notre idée a plutôt été de pouvoir exprimer les propriétés que l'on souhaite vérifier au sein même de notre spécification, dans le même formalisme, évitant en cela une phase de traduction, souvent complexe.

On peut distinguer au sein de ces travaux l'étude de deux grandes catégories de propriétés : celles qui portent sur le langage lui-même, et celles qui portent sur des programmes donnés. Un autre point important est la distinction entre les propriétés déterminées statiquement comme les propriétés de sûreté du typage, et celles portant sur les aspects dynamiques du langage ou sur l'exécution des programmes.
Ayant fait initialement le choix de définir une sémantique dynamique du langage Java, nous ne nous sommes pas concentrés sur les propriétés de typage, mais plutôt sur les propriétés dynamiques du langage et des programmes. Nous nous sommes de plus particulièrement intéressés aux propriétés liées à la concurrence et aux mécanismes de synchronisation.
Nous avons d'autre part étudié la possibilité d'ajout d'assertions au sein de notre sémantique (voir Section 6.2.1.3), sur un modèle proche de celui de JML.

5.2 Propriétés du langage Java liées à la concurrence

De nombreuses propriétés de Java sont liées à la concurrence et à la synchronisation. Voici les différentes propriétés que nous avons choisi d'étudier au sein de cette section :

1. Un thread ne peut pas être à la fois *blocked* et *dormant* sur un même lock.
2. Un thread ne peut pas être dans le statut *blocked* en attente de deux locks distincts.
3. Un thread ne peut pas être dans le statut *dormant* sur deux locks distincts.
4. Deux threads ne peuvent pas détenir un lock sur un même objet à une même étape de l'exécution.
5. Un thread ne peut pas être lancé plus d'une fois sans être stoppé de façon abrupte (par la levée d'une exception) à moins qu'il n'ait déjà achevé son exécution.
6. Un thread ne peut pas relâcher un lock qu'il ne possède pas.
7. Lorsque l'exécution d'un bloc *synchronized* s'achève, que ce soit normalement ou non (si une exception est levée), le lock considéré est relâché.
8. Un thread ne peut quitter le *wait set* d'un objet (et ainsi quitter le statut *dormant*) sans qu'un autre thread n'ait appelé la méthode *notify* (ou *notifyAll*) sur cet objet.
9. Un thread ne peut pas être *blocked* ou *dormant* sur un lock qu'il possède.
10. Un thread ne peut pas s'exécuter tant qu'il n'a pas été lancé par un autre thread.

Les propriétés intrinsèques à notre modélisation sont étudiées dans un premier paragraphe. Un second traite des propriétés pouvant et devant être exprimées en Typol. Un dernier paragraphe développe une preuve d'une propriété.

5.2.1 Propriétés assurées par notre modélisation

Ce chapitre, qui traite des propriétés directement prises en compte par notre modélisation, se découpe en deux parties : nous présentons tout d'abord les propriétés modélisées au niveau de la syntaxe abstraite que nous avons définie, puis dans un second paragraphe nous étudions les propriétés modélisées par notre spécification Typol.

5.2.1.1 Propriétés induites par la syntaxe abstraite des structures sémantiques

Certaines propriétés de Java liées à la concurrence (propriétés 1 à 3) sont traduites directement au sein de notre spécification par la syntaxe abstraite des structures sémantiques que nous avons définies.

Propriété 1

Un thread ne peut pas être à la fois blocked et dormant sur un même lock.
Cette propriété porte sur l'unicité du statut d'un thread.
La syntaxe abstraite du statut d'un thread est définie comme suit :

```
ThreadStatus ::= Created Runnable Executing Blocked Suspended Dormant
                                    Dead ;
```

Par définition, le type du statut d'un thread sera donc un des sous-types du phylum *ThreadStatus*, chacun de ces phyla constituant le type correspondant à un statut donné d'un thread. La syntaxe abstraite définie reflète donc l'unicité de ce statut et assure ainsi *par construction* le respect de la propriété 1.

Propriétés 2 et 3 :
Considérons la propriété 3 :
Un thread ne peut pas être dans le statut blocked en attente de deux locks distincts.
Prenons un thread dont le statut est *blocked*, c'est-à-dire qui a essayé d'acquérir un lock déjà détenu par un autre thread. La syntaxe abstraite définissant le statut *blocked* est la suivante :

```
Blocked ::= blocked;
blocked → ObjectId;
```

ce qui permet de construire des arbres de la forme :

```
blocked(ObjId)
```

où `ObjId` représente la référence de l'objet que le thread considéré a voulu verrouiller. L'opérateur *blocked* est défini au sein de la syntaxe abstraite que nous avons spécifiée comme étant un opérateur unaire, ce qui impose la contrainte que le thread ne puisse être dans le statut *blocked* que sur un unique objet. De plus, la sémantique garantit que cet objet est unique, c'est-à-dire qu'elle ne modifie pas la référence de l'objet sur lequel le thread est *blocked*. Les règles de typage induites par la syntaxe abstraite ainsi définie et l'unicité du statut d'un thread assurent donc par construction le respect de la propriété 2.

Une réflexion identique peut être conduite au sujet de la propriété 3 :
Un thread ne peut pas être dans le statut dormant sur deux locks distincts.
La syntaxe abstraite correspondant au statut *dormant* est :

```
Dormant ::= dormant;
dormant → ObjectId Val;
```

et permet d'écrire des arbres de la forme :

```
dormant(ObjId, NbLocks)
```

où `ObjId` désigne la référence de l'objet sur lequel le thread possédait `NbLocks` avant d'appeler sur celui-ci la méthode *wait*. L'opérateur *dormant* est binaire, mais n'accepte qu'un seul identifiant d'objet, ce qui traduit le fait que le thread ne puisse pas être *dormant* sur plus d'un objet. De plus nous avons vu précédemment que l'unicité du statut d'un thread est assurée par la syntaxe abstraite que nous avons définie, ce qui permet de refléter par construction la propriété 3.

Propriété 4 :
Deux threads ne peuvent pas détenir un lock sur un même objet à une même étape de l'exécution.
Cette propriété est induite directement par la syntaxe abstraite des structures sémantiques que nous avons définies.

La syntaxe abstraite correspondant au fait pour un objet `ObjId` d'être verrouillé `NbLocks` fois par un thread `OThId` donné est la suivante :

```
Locked ::= locked unlocked ;
  locked → ObjectId Val ;
```

Cette définition permet d'écrire des arbres de syntaxes abstraite de la forme :

locked(OThId, NbLocks)

L'opérateur *locked* est donc binaire, mais n'accepte qu'un seul identifiant d'objet, ce qui traduit le fait qu'un objet ne puisse pas être verrouillé par deux threads différents à la fois. De plus, il y a unicité du statut de l'objet qui est soit *locked*, soit *unlocked* d'après la définition du phylum *Locked* :

```
Locked ::= locked unlocked ;
```

La propriété 4 est donc modélisée par construction de notre syntaxe abstraite.

Les quatre propriétés que nous avons présentées dans ce paragraphe sont donc directement assurées par notre modélisation à travers la définition de la syntaxe abstraite typée de nos structures sémantiques.

5.2.1.2 Propriétés induites par notre spécification Typol

Certaines propriétés (propriétés 5 à 7), si elles ne sont pas directement assurées par la syntaxe abstraite des structures sémantiques, le sont par contre par notre spécification Typol.

Propriété 5 :
Un thread ne peut pas être lancé plus d'une fois sans être stoppé de façon abrupte (par la levée d'une exception) à moins qu'il n'ait déjà achevé son exécution.
Cette propriété porte essentiellement sur la possibilité d'une erreur de programmation : le fait d'appeler la méthode *start* sur un même thread plusieurs fois au cours d'une même exécution. Cette erreur peut avoir différents résultats selon l'état du thread quand on appelle à nouveau sa méthode *start*. Si le thread est dans le statut *dead* (il a fini son exécution), l'appel de la méthode *start* est tout simplement ignoré. Si le thread est en cours d'exécution, l'appel de la méthode *start* lève une exception.

Cette propriété est reflétée par les trois règles sémantiques présentées en Figure 5.1 qui expriment exactement le lancement correct d'un thread (première règle), le lancement d'un thread dont le statut est *dead* et qui a donc achevé son exécution (deuxième règle) et enfin le lancement d'un thread qui est déjà en train de s'exécuter et donc la levée de l'exception correspondante. La spécification Typol que nous avons définie assure donc directement le respect de la propriété 4.

Propriété 6 :
Un thread ne peut pas relâcher un lock qu'il ne possède pas.
De nouveau, le respect de cette propriété est directement assuré par nos règles Typol.

```
Start:
   --RC: Causes the thread to pass from the created status into the runnable
   --RC: status to begin its execution (the JVM calls the "run" method).
   --
   --CS: OThId2.start();
   getThreadStatus(OThId |- ObjL1 -> created(_))  &
   setNewThreadStatus(OThId, runnable(), ClVarL |- ObjL1 -> ObjL1_1)
   ----------------
   ObjL1, ClVarL, _, OThId
      |- identifier "start", vals[] -> ObjL1_1, ClVarL, insts[], runnable() ;

Start_ThreadAlreadyDead_DoNothing:
   --
   --CS: OthId2.start();
   getThreadStatus(OThId |- ObjL1 -> dead())
   ----------------
   ObjL, ClVarL, _, OThId
      |- identifier "start", vals[] -> ObjL, ClVarL, insts[], runnable() ;

Start_ThreadAlreadyStarted_ExceptionThrown:
   --RC: The IllegalThreadStateException is thrown because the given thread
   --RC: was already started.
   --
   --CS: OthId2.start();
   getThreadStatus(OThId2 |- ObjL1 -> ThStatus)  &  not_eq(ThStatus, created(_))
   &  not_eq(ThStatus, dead())  &
   createExceptionObject(ObjL1, OThId1
                         |- identifier "IllegalThreadStateException",
                            exceptText
                               "Method 'start' called on an already started thread."
                               ->
                               _, ObjL1_1, InstL)
   ----------------
   ObjL1, ClVarL, OThId1, OThId2
      |- identifier "start", vals[] -> ObjL1_1, ClVarL, InstL, runnable() ;
```

FIG. 5.1 – *Règles Typol spécifiant la sémantique de la méthode start.*

```
SynchronizedStatement_GetTheLockOrBeBlocked:
   --CS: synchronized(#8) Statement
   getObjectLockInformation(ObjId2 |- ObjL1 -> OThId2, nbLocks)  &
   getTheLockOrBeBlocked(OThId1, ObjId2, Statement, ObjL1, ClVarL
                           |- OThId2, nbLocks -> ObjL1_1, Inst, ThStatus)
   ----------------
   ObjL1, ClVarL, OThId1
      |- clr(
            ObjId1, Mode, MethName, Env,
            insts[synchroLock(ObjId2, Statement).InstL]) ->
            ObjL1_1, ClVarL, clr(ObjId1, Mode, MethName, Env, insts[Inst.InstL]),
            ThStatus ;

SynchronizedStatement_BeginBlockExecution:
   appendtree(insts[Statement, synchroEnd(ObjId2)], InstL1, InstL2)
   ----------------
   ObjL, ClVarL, _
      |- clr(
            ObjId1, Mode, MethName, Env,
            insts[synchroExecution(ObjId2, Statement).InstL1]) ->
            ObjL, ClVarL, clr(ObjId1, Mode, MethName, Env, InstL2), runnable() ;
```

FIG. 5.2 – *Règles Typol spécifiant l'introduction de la continuation* `synchroEnd`.

En effet, seules deux instructions permettent de relâcher un lock sur un objet, il s'agit de l'exécution de l'instruction de fin de bloc *synchronized* et de l'exécution de la méthode *wait* sur l'objet considéré.
Traitons tout d'abord du cas de l'instruction de fin de bloc *synchronized*. Cette instruction est représentée par la continuation *synchroEnd(ObjId)* où `ObjId` représente la référence de l'objet sur lequel porte le bloc *synchronized*. Cette continuation est ajoutée automatiquement en fin du bloc lorsque le thread courant acquiert le lock sur l'objet et commence à exécuter le bloc. Cela est spécifié par les règles présentées en Figure 5.2.
La première règle permet au thread courant représenté par la référence `OThId1` d'acquérir le lock sur l'objet `ObjId2` (sinon le thread passe dans le statut *blocked*). Quand le thread a acquis le lock, la seule règle Typol qui peut s'appliquer est la deuxième règle du set de la Figure 5.2. Son unique prémisse introduit la continuation *synchroEnd(ObjId2)* à la fin de la liste des instructions du bloc *synchronized*. Cette règle est la seule à introduire cette continuation, et elle ne s'applique que lorsque le thread courant a déjà acquis le lock. La propriété est donc respectée dans le premier cas.
La seule autre instruction permettant de relâcher un lock sur un objet est l'exécution de la méthode *wait* sur l'objet verrouillé. La Figure 5.3 présente la règle Typol spécifiant l'exécution de la méthode *wait*. La première prémisse de cette règle permet d'assurer directement le respect de la propriété en vérifiant que l'objet verrouillé l'a bien été par le thread appelant la méthode *wait*. En effet, grâce à cette prémisse appelant le set `getObjectLockInformation`, on vérifie que l'objet est verrouillé par le thread `OThId`, qui est aussi le thread courant. Ce n'est que dans ce cas que le (ou les) lock acquis sur l'objet est relâché (prémisse `releaseObjectLock`).

```
Wait:
    --RC: Makes the current thread "dormant", adds it to the wait set of the current
    --RC: object and releases the locks on the current object. The current thread is
    --RC: disabled for thread scheduling until another thread call the notifyAll method
    --RC: on the current object (or the notify method).
    --
    --CS: ObjId.wait();
    --PC: Checks that the object 'ObjId' is locked by the current executing
    --PC: thread 'OThId' and determines how many times the object is locked
    --PC: (nb times).
    --PC: Adds the thread identifier 'OThId' to the wait set of the object 'ObjId'.
    --PC: Releases the nb locks on the object 'ObjId'.
    getObjectLockInformation(ObjId |- ObjL1 -> OThId, nb)  &  diff(nb, 0)  &
    addThreadToTheObjectWaitSet(OThId, ObjId |- ObjL1 -> ObjL1_1)  &
    releaseObjectLock(ObjId |- ObjL1_1 -> ObjL1_2)
    ----------------
    ObjL1, ClVarL, OThId, ObjId
        |- identifier "wait", vals[]
               ->
               ObjL1_2, ClVarL, insts[synchroWaitEndRelock(ObjId, integer nb)],
               dormant(ObjId, integer nb) ;
               do ENVchangeObjectStatusInStatusList(ObjId);
```

FIG. 5.3 – *Règles Typol spécifiant l'exécution de la méthode wait.*

Notre spécification Typol permet donc de s'assurer du respect de la propriété 5.

Propriété 7 :
Lorsque l'exécution d'un bloc "synchronized" s'achève, que ce soit normalement ou non (si une exception est levée), le lock considéré est relâché.
Dans le cas où le bloc *synchronized* s'achève de façon "normale" (c'est-à-dire sans levée d'une exception), la dernière instruction du bloc est la continuation *synchroEnd*. En effet, comme cela a été expliqué précédemment, cette continuation est ajoutée en fin de bloc dès le début de l'exécution de celui-ci. Or, l'exécution de cette continuation est spécifiée par la règle Typol présentée en Figure 5.4. La première prémisse de cette règle permet de vérifier que l'objet sur lequel porte le bloc est bien verrouillé par le thread courant (représenté par la référence `OThId`). La deuxième prémisse se charge de relâcher le lock par l'appel du set `releaseObjectLock`. Le lock est donc bien relâché en cas de sortie "normale" du bloc *synchronized* ce qui assure la prise en compte de la propriété dans ce premier cas.
Traitons maintenant le cas de la sortie abrupte d'un bloc *synchronized* (suite à la levée d'une exception). Afin de prendre en compte la propriété dans notre sémantique, nous avons défini le set Typol `getSynchroEndInstructions` présenté en Figure 5.5.
Les règles Typol de ce set permettent de parcourir la liste des instructions d'un bloc et de former la liste des instructions *synchroEnd* présentes au sein de ce bloc dans l'ordre de leur présence. La liste ainsi obtenue est transmise à la règle ayant appelé ce set. Ce set permet quand une instruction de levée d'exception est rencontrée au sein d'un bloc de récupérer les instructions de relâchement des locks afin de les exécuter avant

```
EndSynchronizedStatementWithOneLock_RemoveLock:
   getObjectLockInformation(ObjId2 |- ObjL1 -> OThId, 1)  &
   releaseObjectLock(ObjId2 |- ObjL1 -> ObjL1_1)
   ----------------
   ObjL1, ClVarL, OThId
      |- clr(ObjId1, Mode, MethName, Env, insts[synchroEnd(ObjId2).InstL]) ->
            ObjL1_1, ClVarL, clr(ObjId1, Mode, MethName, Env, InstL), runnable() ;
            do
               ENVsendObjectAndClassVariableLists(ObjL1_1, ClVarL)  &
               ENVupdateEnvIfNonDormantOrBlockedThreadOnObject(ObjId2,
                                                                ObjL1_1, 0)
               &  ENVchangeObjectStatusInStatusList(ObjId2);
```

FIG. 5.4 – *Règles Typol spécifiant l'exécution de la continuation* `synchroEnd`.

```
set getSynchroEndInstructions is
--SC: Retrieves from an instruction list, all the "synchroEnd" instructions
--SC: useful for removing locks set for a synchronized block or method.
--SC: (the "synchroEnd" instructions are added during the program
--SC: interpretation)
--
--JC: Instruction list
--JC: -> Instruction list that only contains the synchroEnd instructions.
judgement (Insts -> Insts);

NoMoreInstruction: (insts[] -> insts[]) ;

SynchroEndAddedToTheInstrL_RecursiveRule:
   (InstL1 -> InstL2)
   ----------------
   (insts[synchroEnd(ObjId1).InstL1] -> insts[synchroEnd(ObjId1).InstL2]) ;

NotASynchroEndInst_RecursiveRule:
   (InstL1 -> InstL2)
   ----------------
   (insts[Inst.InstL1] -> InstL2) ;
            provided not_eq(Inst, synchroEnd(_));
end getSynchroEndInstructions;
```

FIG. 5.5 – *Règles Typol spécifiant le set* `getSynchroEndInstructions`.

```
Except:
   getSynchroEndInstructions(InstL1 -> InstL2)  &
   appendtree(InstL2, insts[exception(OThId, ObjId1, MethName, except(ClName, ObjId2, ExceptText))],
              InstL3)
   ----------------
   ObjL, ClVarL, OThId
      |- clr(ObjId1, Mode, MethName, Env, insts[except(ClName, ObjId2, ExceptText).InstL1]) ->
            ObjL, ClVarL, clr(ObjId1, Mode, MethName, Env, InstL3), runnable() ;
```

FIG. 5.6 – *Règle Typol utilisant le set* `getSynchroEndInstructions`*.*

de propager l'exception, et donc d'assurer le respect de la propriété 6. La Figure 5.6 présente une règle Typol utilisant le set `getSynchroEndInstructions` (ce set est aussi utilisé par les autres règles spécifiant la levée d'une exception).
Cette règle s'applique lorsqu'une exception (opérateur *except*) est présente en tête de la liste des instructions `InstL1` à exécuter dans une closure. La liste `InstL2` des instructions de relâchement des locks est obtenue par l'utilisation du set `getSynchroEndInstructions` dans l'ordre d'imbrication des blocs. Elle est alors placée (par la seconde prémisse de la règle) en tête de la nouvelle liste d'instructions de la closure, immédiatement suivie par l'instruction de levée d'exception. Cela forme la nouvelle liste d'instructions `InstL3` de la closure.
La propriété 7 est donc bien modélisée par notre spécification Typol.

Cette section nous a permis de mettre en évidence le lien très fort qui unit spécification Typol et propriétés du langage. En effet certaines propriétés du langage Java sont directement modélisées par les règles Typol, ce qui assure leur respect.

5.2.2 Expression et vérification des propriétés en Typol

Certaines propriétés du langage Java (propriétés 8 à 10) liées à la concurrence, bien qu'elles soient bien prises en compte par notre sémantique, ne sont pas directement apparentes ni lisibles au sein de la définition des structures sémantiques ou des règles Typol que nous avons définies. Nous avons donc choisi de les exprimer en Typol dans un set dédié à la vérification des propriétés, comme nous le montrons dans cette section.
La vérification de ces propriétés a lieu lors du changement de statut d'un thread, dans le même pas d'exécution. Par exemple si l'on vérifie qu'un thread quitte le statut *dormant* suite à l'appel de la méthode *notify* sur l'objet considéré, la vérification a lieu dans le même pas d'exécution que celui de l'appel effectif de la méthode, avant la validation de cet appel. Il s'agit donc d'une vérification expérimentale ayant lieu durant l'exécution d'exemples de programmes Java à l'aide de notre interpréteur.
Les règles Typol de ce set ne sont pas encore intégrées au sein de notre sémantique, ce qui explique leur présentation différente des règles Typol précédentes.

```
set setNewThreadStatus is
--SC: sets the new status to the given thread.
--
--JC: Thread reference that will have the new status, the new status,
--JC: class variables (only used for the environment)
--JC: |- objects
--JC: -> modified object list
judgement ObjectId, ThreadStatus, ClassVariablesL |- Objects -> Objects;
--iJC: (same as above without the class variables)
judgement ObjectId, ThreadStatus |- Objects -> Objects;

EntryPoint_SendInfoToEnv:
   OThId, ThStatus |- ObjL1 -> ObjL1_1
   ----------------
   OThId, ThStatus, ClVarL |- ObjL1 -> ObjL1_1 ;
            do
               ENVsendObjectList(ObjL1_1)  &
               ENVsendObjectAndClassVariableLists(ObjL1_1, ClVarL);

SetNewStatusToTheGivenThread:
   OThId, NewThStatus
      |- objects[
            object(OThId, ClName, AttrL, Lock, WaitSet, activity(OldThStatus, Clr)).
            ObjL] ->
            objects[
               object(
                  OThId, ClName, AttrL, Lock, WaitSet,
                  activity(NewThStatus, Clr)).ObjL] ;
            do ENVupdateThreadStatusList(OThId, OldThStatus, NewThStatus);

LookForTheGivenThread:
   OThId, ThStatus |- ObjL1 -> ObjL1_1
   ----------------
   OThId, ThStatus
      |- objects[object(ObjId, ClName, AttrL, Lock, WaitSet, Act).ObjL1] ->
            objects[object(ObjId, ClName, AttrL, Lock, WaitSet, Act).ObjL1_1] ;
            provided diff(OThId, ObjId);
end setNewThreadStatus;
```

FIG. 5.7 – *Set Typol* `setNewThreadStatus` *mettant à jour le statut d'un thread.*

5.2.2.1 Préalable

Le changement de statut d'un thread est spécifié par le set `setNewThreadStatus` représenté en Figure 5.7, qui, étant donné la référence `OThId` d'un thread et son nouveau statut `ThStatus` modifie la structure sémantique du thread en mettant à jour son statut au sein de son activité.

Ce set comprend trois règles Typol :

- La première règle est la règle d'entrée dans le set, elle se charge de lancer la recherche du thread parmi la liste des objets, et communique la nouvelle liste des objets obtenue après la mise à jour du statut du thread considéré.
- La deuxième règle de ce set (règle appelée `SetNewStatusToTheGivenThread`),

traite le cas où l'on a trouvé le thread dont on doit mettre à jour le statut. Elle permet de connaître l'ancien statut du thread considéré qui est représenté par `OldThStatus`.

– La troisième règle est la règle récursive qui permet le parcours de la liste des objets à la recherche du thread considéré. Cette règle s'applique lorsque l'objet en tête de liste n'est pas le thread recherché (la clause *provided* assure cette condition). Le set est rappelé avec en sujet la queue de la liste des objets.

Afin de spécifier en Typol les propriétés 8 à 10, nous écrivons un set appelé `threadPropertyVerificati` qui constitue un nouveau module de notre spécification. Ce set va être appelé au sein du set `setNewThreadStatus`.
Nous devons donc modifier la seconde règle du set `setNewThreadStatus` afin de demander la vérification des propriétés sur le thread dont le statut est mis à jour. Cela est effectué par l'ajout d'une clause *provided* avec appel du set `threadPropertyVerification`. La vérification de la propriété est ainsi préalable à la mise à jour du statut du thread considéré. La règle `SetNewStatusToTheGivenThread` devient alors :
`SetNewStatusToTheGivenThread:`

$OThId, NewThStatus, ObjL \vdash objects[object(OThId, ClName, AttrL,$
$Lock, WaitSet, activity(OldThStatus, Clr)).ObjL] \rightarrow objects[object(OThId,$
$ClName, AttrL, Lock, WaitSet, activity(NewThStatus, Clr)).ObjL];$
provided threadPropertyVerification$(OThId, ObjL \vdash$
$OldThStatus, NewThStatus);$
`do` ENVupdateThreadStatusList$(OThId, OldThStatus, NewThStatus);$

Le statut du thread est nécessaire afin de savoir quelle propriété doit être vérifiée. Afin toutefois de ne pas apporter plus de modifications (autre que l'ajout de la clause *provided*) aux règles du set `setNewThreadStatus`, nous testons le statut du thread dans le set `threadPropertyVerification`.

Il est tout de même nécessaire d'ajouter la liste globale des objets dans les hypothèses de la règle `SetNewStatusToTheGivenThread`, et donc dans l'ensemble des règles du set `setNewThreadStatus`, et par là même de modifier son jugement interne (le deuxième) qui devient :
`judgement ObjectId, ThreadStatus, Objects ⊢ Objects =>Objects;`
Pour éviter cette modification, il serait nécessaire de conserver la liste des objets à un niveau global (dans une variable globale) ce qui va à l'encontre de la pûreté logique de notre spécification formelle. Les structures sémantiques principales comme la liste des objets doivent figurer, si elles sont nécessaires, au sein des hypothèses des règles, et non être stockées dans une variable globale, ce qui est juste une possibilité spécifique au formalisme Typol.

5.2.2.2 Expression et vérification des propriétés

Cette section montre comment nous pouvons intégrer une vérification pratique de propriétés exprimées en Typol dans un nouveau module de notre spécification. Cette vérification a lieu lors de l'interprétation de programmes Java par l'interpréteur généré à partir de notre spécification Typol.

Propriété 8 :
Un thread ne peut quitter le wait set d'un objet (et ainsi quitter le statut dormant) sans qu'un autre thread n'ait appelé la méthode notify (ou notifyAll) sur cet objet.

La vérification de la propriété 8 a lieu lorsque l'ancien statut du thread est *dormant*. La syntaxe abstraite de l'opérateur *dormant* est la suivante :

```
dormant → ObjectId Val;
```

ce qui permet de construire des arbres de syntaxe abstraite de la forme :

```
dormant(ObjId, NbLocks)
```

où `ObjId` désigne la référence de l'objet sur lequel le thread possédait `NbLocks` avant d'appeler sur celui-ci la méthode *wait*.

Nous connaissons donc l'objet sur lequel la méthode *notify* ou *notifyAll* a dû être appelée pour que la propriété 8 soit respectée. Il reste à vérifier que le thread qui est en train de s'exécuter (statut *executing*) est bien en train d'appeler une de ces deux méthodes sur cet objet. Si c'est le cas, la propriété sera vérifiée.
Voici donc la définition du set `threadPropertyVerification` :

```
set threadPropertyVerification is
judgement ObjectId, Objects ⊢ ThreadStatus, ThreadStatus;
```

`DormantThread_NotifyCall:`

$$\frac{\begin{array}{l}\text{getCurrentExecutingThread}(\vdash ObjL \rightarrow OThId2)\&\\ \text{getThreadCurrentMethod}(OThId2 \vdash ObjL => \_,\\ \text{methodCall1}(\_, ObjId, MethodName, arguments[]))\&\\ \text{notifyMethod}(MethodName)\end{array}}{OThId1, ObjL \vdash dormant(ObjId, \_), \_;}$$

```
end threadPropertyVerification;
```

Le sujet de la règle `DormantThread_NotifyCall` de ce set permet d'imposer à la règle de ne s'appliquer que lorsque l'on considère un thread *dormant*, la première prémisse détermine quel est le thread en train de s'exécuter, la seconde détermine quelle est l'instruction que ce thread est en train d'exécuter, et la troisième vérifie que cette instruction est l'appel de la méthode *notify* ou de la méthode *notifyAll*.

Le set `getCurrentExecutingThread` qui est utilisé parmi les prémisses de la règle `DormantThread_NotifyCall` dans le set `threadPropertyVerification` est défini par :

```
set getCurrentExecutingThread is
judgement ⊢ Objects → ObjectId;
```

ExecutingThread:

$$\vdash objects[object(OThId, \_, \_, \_, \_, activity(executing(), \_)).\_] \rightarrow OThId;$$

NotTheExecutingThread:

$$\frac{\vdash ObjL \rightarrow OThId}{\vdash objects[object(\_, \_, \_, \_, \_, activity(ThStatus, \_)).ObjL] \rightarrow OThId;}$$

$provided$ diff$(ThStatus, executing())$;

NotAThread:

$$\frac{\vdash ObjL \rightarrow OThId)}{\vdash objects[object(\_, \_, \_, \_, \_, empty()).ObjL] \rightarrow OThId;}$$

```
end getCurrentExecutingThread;
```

Ce set traite successivement les trois cas :

- L'objet en tête de la liste d'objets figurant en sujet de la règle est le thread dans le statut *executing*, on donne donc sa référence `OThId` en conclusion de la règle qui ne comporte pas de prémisse.
- L'objet en tête de la liste d'objets figurant en sujet de la règle est un thread, mais son statut n'est pas *executing* (cela est assuré par la clause *provided*). Le set est alors rappelé de façon récursive avec la fin de la liste d'objets.
- L'objet en tête de la liste d'objets figurant en sujet de la règle n'est pas un thread car il n'a pas d'activité (valeur *empty()*). Le set est alors rappelé de façon récursive avec la fin de la liste d'objets.

Le set Typol `notifyMethod` est simple à définir, il ne comporte que deux règles (une pour chaque méthode) :

```
set notifyMethod is
judgement Identifier;
NotifyMethod:
```

$identifier$ "$notify$";

```
NotifyAllMethod:
```

$identifier$ "$notifyAll$";

```
end notifyMethod;
```

Le set `threadPropertyVerification` que nous avons défini pour spécifier la propriété 8 en Typol est un set qui peut être utilisé pour spécifier d'autres propriétés portant sur le statut d'un thread.

Propriété 9 :

Un thread ne peut pas être blocked ou dormant sur un lock qu'il possède.

Cette propriété porte également sur le statut d'un thread. Sa spécification peut donc être incluse dans le set `threadPropertyVerification`.

Les règles Typol suivantes permettent de spécifier qu'un thread donné `OThId1` n'est pas dans le statut *blocked* en attente d'un lock qu'il possède déjà. La première règle traite le cas où le thread `OThId1` est dans le statut *blocked* en attente d'un lock sur l'objet `ObjId`, lock possédé par un autre thread `OThId2`.
`BlockedOnObjectLockedByAnotherThread:`

$$\frac{\begin{array}{l}\text{getObjectLockInformation}(ObjId \vdash ObjL \rightarrow OThId2, integer\ nb)\ \&\\ \text{diff}(nb, 0)\ \&\\ \text{diff}(OThId1, OThId2)\end{array}}{OThId1, ObjL \vdash \_, blocked(ObjId);}$$

`BlockedOnUnlockedObject_WillCompeteToAcquireTheLock:`

$$\frac{\text{getObjectLockInformation}(ObjId \vdash ObjL \rightarrow \_, integer\ 0)}{OThId1, ObjL \vdash \_, blocked(ObjId);}$$

Dans ces deux règles, nous ne considérons pas le statut précédent des threads qui est alors représenté par le symbole `_`. Cette deuxième règle s'applique lorsque l'objet `ObjId` n'est pas verrouillé (le nombre de locks est nul). Dans ce cas, au prochain pas d'exécution, le thread `OThId1` pourra passer dans le statut *executing* si il est choisi par l'ordonnanceur.

Voici les règles correspondant au statut *dormant* du thread `OThId1` :
`DormantOnObjectLockedByAnotherThread:`

$$\frac{\begin{array}{l}\text{getObjectLockInformation}(ObjId \vdash ObjL \rightarrow OThId2, integer\ nb)\ \&\\ \text{diff}(nb, 0)\ \&\\ \text{diff}(OThId1, OThId2)\end{array}}{OThId1, ObjL \vdash \_, dormant(ObjId, integer\ \ \_);}$$

`DormantOnAnUnlockedObject:`

$$\frac{\text{getObjectLockInformation}(ObjId \vdash ObjL \rightarrow \_, integer\ 0)}{OThId1, ObjL \vdash \_, dormant(ObjId, integer\ \ \_);}$$

Ces règles sont construites sur le même modèle que les deux règles précédentes. Le sujet de la règle permet de vérifier le statut du thread considéré, et d'obtenir ainsi l'identifiant de l'objet sur lequel le thread est *blocked* ou *dormant*. Les prémisses sont utilisées pour vérifier que cet objet est soit non verrouillé (deuxièmes règles), soit verrouillé par un autre thread (premières règles). Ce dernier cas est traité par une succession de prémisses : la première prémisse permet de déterminer les informations sur le lock liées à cet objet, la seconde s'assure que l'objet est verrouillé (nombre de locks différent de zéro) et la troisième vérifie que le thread possédant le verrou sur l'objet n'est pas le thread considéré (les identifiants des threads sont différents).

Propriété 10 :
Un thread ne peut pas s'exécuter tant qu'il n'a pas été lancé par un autre thread.

Cette propriété porte également sur le statut d'un thread. Elle est donc définie au sein du set `threadPropertyVerification`.

Voici sa spécification :

`CreatedThread_StartCall:`

$$\frac{\text{getCurrentExecutingThread}(\vdash ObjL \rightarrow OThId2)\& \quad \text{getThreadCurrentMethod}(OThId2 \vdash ObjL => \_, \quad \text{methodCall1}(\_, OThId1, identifier``start", arguments[]))}{OThId1, ObjL \vdash created(\_), \_;}$$

Ici, si l'ancien statut du thread `OThId1` était *created* (ce qui est assuré par unification avec le sujet de la règle), nous déterminons quel est le thread qui est en train de s'exécuter (première prémisse), puis vérifions que ce thread est bien en train d'exécuter la méthode *start* sur le thread `OThId1`. Dans ce cas, le changement d'état du thread de *created* à *runnable* est validé car la propriété 10 est vérifiée.

5.2.2.3 Set de vérification d'expression et de vérification des propriétés

En résumé, nous avons maintenant le set `threadPropertyVerification` suivant :

```
set threadPropertyVerification is
judgement ObjectId, Objects ⊢ ThreadStatus, ThreadStatus;
```

`DormantThread_NotifyCall:`

$$\frac{\text{getCurrentExecutingThread}(\vdash ObjL \rightarrow OThId2)\& \quad \text{getThreadCurrentMethod}(OThId2 \vdash ObjL => \_, \quad \text{methodCall1}(\_, ObjId, MethodName, arguments[]))\& \quad \text{NotifyMethod}(MethodName)}{OThId1, ObjL \vdash dormant(ObjId, \_), \_;}$$

`BlockedOnObjectLockedByAnotherThread:`

$$\frac{\text{getObjectLockInformation}(ObjId \vdash ObjL \rightarrow OThId2, integer\ nb)\ \& \quad \text{diff}(nb, 0)\ \& \quad \text{diff}(OThId1, OThId2)}{OThId1, ObjL \vdash \_, blocked(ObjId);}$$

`BlockedOnUnlockedObject_WillCompeteToAcquireTheLock:`

$$\frac{\text{getObjectLockInformation}(ObjId \vdash ObjL \rightarrow \_, integer\ 0)\ \&}{OThId1, ObjL \vdash \_, blocked(ObjId);}$$

`DormantOnObjectLockedByAnotherThread:`

$$\frac{\text{getObjectLockInformation}(ObjId \vdash ObjL \rightarrow OThId2, integer\ nb)\ \& \quad \text{diff}(nb, 0)\ \& \quad \text{diff}(OThId1, OThId2)}{OThId1, ObjL \vdash \_, dormant(ObjId, integer\ \_);}$$

`DormantOnAnUnlockedObject:`

$$\frac{\text{getObjectLockInformation}(ObjId \vdash ObjL \rightarrow \_, integer\ 0)\ \&}{OThId1, ObjL \vdash \_, dormant(ObjId, integer\ \_);}$$

`CreatedThread_StartCall:`

$$\frac{\begin{array}{l}\text{getCurrentExecutingThread}(\vdash ObjL \rightarrow OThId2)\& \\ \text{getThreadCurrentMethod}(OThId2 \vdash ObjL => \_, \\ \text{methodCall1}(\_, OThId1, identifier\text{“}start\text{''}, arguments[]))\end{array}}{OThId1, ObjL \vdash created(\_), \_;}$$

`end threadPropertyVerification;`

Pour toutes les propriétés (8 à 10) spécifiées en Typol dans cette section, il faut prévoir une possibilité d'arrêt de l'exécution, ou du moins une fenêtre d'avertissement, si les propriétés ne sont pas vérifiées.

Cette section a montré la puissance de notre spécification qui facilite l'expression des propriétés en Typol, et qui permet d'autre part l'intégration de leur vérification expérimentale à l'interprétation des règles Typol.

5.2.3 Preuves de propriétés

Ce paragraphe présente un schéma de preuve de propriété de notre sémantique.
Il s'agit ici de réaliser une preuve par induction d'une propriété comme dans le travail effectué par Egon Börger et al. [15].

L'énoncé de la propriété 4 est le suivant :
Deux threads ne peuvent pas détenir un lock sur un même objet à une même étape de l'exécution.
Il s'agit d'un invariant qui assure que l'on ne rencontrera jamais pendant l'exécution d'un programme le cas d'un lock détenu simultanément par deux threads.

5.2.3.1 Remarques préalables

Afin d'effectuer cette preuve, il est bon de faire quelques remarques sur l'exécution, les threads et les locks :

1. Tout d'abord, quand un objet est créé, son lock est libre. Cela est exprimé par l'élément *unlocked()* (intégré à la structure sémantique de l'objet nouvellement créé) visible au sein de la conclusion de la règle Typol de création d'un objet simple présentée dans la Sectionobjcreationsection, Figure 3.14.
2. Quand l'interprétation d'un programme commence, un unique thread est créé et peut s'exécuter : le thread principal, c'est-à-dire un thread qui n'a pas été créé de manière explicite par le programmeur, mais de façon implicite, afin d'exécuter la méthode *main*.
3. Une règle Typol ne peut pas être interrompue : chaque règle Typol est atomique.

5.2.3.2 Schéma de la preuve

La preuve de la propriété 4 se fait par induction.

Cas de base

La remarque 2 nous permet de déduire que la propriété 4 est vérifiée au début de l'exécution d'un programme. La remarque 1 permet de vérifier que la propriété 4 est respectée de fait à la création d'un objet.
Nous pouvons en déduire que la propriété 4 est vérifiée au début de l'exécution et ce tant qu'un seul thread existe.

Etape d'induction

Prenons comme hypothèse que la propriété 4 est vérifiée à une certaine étape i de l'exécution. Il faut montrer qu'elle reste vérifiée au pas d'exécution suivant.

Etudions les différentes possibilités pouvant introduire un conflit avec la propriété 4 à l'étape i+1 de l'exécution sachant qu'il n'y avait pas de conflit à l'étape i. Montrons alors que ces possibilités ne peuvent pas se produire.
La remarque 1 nous permet d'établir que la propriété 4 reste vérifiée tant qu'il n'y a qu'un thread en train de s'exécuter. De plus, le changement d'état pouvant affecter le respect de la propriété 4 se produit lorsque non seulement il y a plusieurs threads en train de s'exécuter, mais plus précisément lorsqu'un de ces threads veut acquérir un lock sur un objet.
Le cas à étudier se situe donc lorsqu'un thread possède un lock sur un objet et qu'un autre thread veut acquérir ce lock. En effet, si le lock de l'objet est libre, il est acquis directement par le thread le demandant, qui devient alors l'unique possesseur de ce lock.

Les deux seules instructions permettant d'acquérir un lock sont :
- l'instruction *synchronized* avec comme argument la référence de l'objet dont le thread veut acquérir le lock.
- la continuation *synchroWaitEndRelock* qui est exécutée quand un thread vient de quitter le statut *dormant* et qu'il souhaite acquérir à nouveau les locks qu'il possédait sur l'objet considéré avant d'appeler la méthode *wait* sur celui-ci.

Etudions donc la sémantique de chacune de ces instructions.
Instruction synchronized

Les règles définissant le mécanisme de synchronisation sont présentées dans la Section 3.5.3, Figures 3.30, 3.31, 3.33 et 3.34. Rappelons tout d'abord leur principe à travers une description informelle de leur action :
1. Règle `SynchronizedStatement_HasTheObjectReference` : calcul de la référence de l'objet que l'on veut verrouiller (cette référence est donnée sous forme d'une

```
set getObjectLockInformation is
--SC: Returns how many times the given object is locked and by which thread.
--
--JC: The object reference for which the lock information is required
--JC: |- objects
--JC: -> The reference of the thread that has locked the object,
--JC: lock(s) number
judgement ObjectId |- Objects -> ObjectId, integer;

LockedObject:
    getObject(ObjId |- ObjL ->
                        object(ObjId, _, _, locked(OThId, integer nbLocks), _, _))
    ----------------
    ObjId |- ObjL -> OThId, nbLocks ;

UnlockedObject:
    getObject(ObjId |- ObjL -> object(ObjId, _, _, unlocked(), _, _))
    ----------------
    ObjId |- ObjL -> _, 0 ;
end getObjectLockInformation;
```

FIG. 5.8 – *Obtention des informations sur le lock d'un objet.*

expression en argument de la clause *synchronized*).

2. Règle `SynchronizedStatement_GetTheLockOrBeBlocked` : tentative de verrouillage de l'objet : le statut du thread courant devient *blocked* si l'objet est déjà verrouillé par un autre thread.
3. Règle `SynchronizedStatement_BeginBlockExecution` : exécution du corps du bloc *synchronized* dès que le lock est acquis.
4. Règles `EndSynchronizedStatement_...` : libération du verrou à la sortie du bloc. Il y a deux cas afin de déterminer si l'objet n'est plus verrouillé ou si son compteur de locks est juste décrémenté de 1.

Le seul cas critique est celui d'un thread réussissant à acquérir le lock d'un objet alors que ce lock est déjà possédé par un autre thread. La seule règle pouvant introduire un tel conflit avec la propriété 4 est la règle de la Figure 3.31. En effet, les autres règles n'ont soit aucune action sur le lock de l'objet (règles des Figures 3.30 et 3.33), soit libèrent ce lock (règle de la Figure 3.34).
La règle de la Figure 3.31 est composée de deux prémisses. La première, l'appel du set `getObjectLockInformation`, permet de savoir si l'objet considéré est verrouillé, et dans ce cas par quel thread, et combien de fois. La seconde prémisse, l'appel du set `getTheLockOrBeBlocked`, permet soit au thread courant d'acquérir le lock demandé, soit de faire passer le statut de ce thread de *runnable* à *blocked* dès qu'il aura fini le pas d'exécution courant.

La remarque 3 nos permet d'affirmer que les informations acquises lors de l'appel de la première prémisse sont toujours vraies lorsque la deuxième est exécutée. Le set

```
SynchronizedStatement_AfterANotify:
   --RC: The current executing thread retrieves the lock(s) that it had
   --RC: before the wait() method call on the given object.
   getObjectLockInformation(ObjId2 |- ObjL1 -> _, 0)   &
   setLock(ObjId2, OThId, nbLocks |- ObjL1 -> ObjL1_1)
   ----------------
   ObjL1, ClVarL, OThId
      |- clr(
           ObjId1, Mode, MethName, Env,
           insts[synchroWaitEndRelock(ObjId2, nbLocks).InstL]) ->
           ObjL1_1, ClVarL, clr(ObjId1, Mode, MethName, Env, InstL), runnable() ;
           do ENVsendObjectAndClassVariableLists(ObjL1_1, ClVarL);
```

FIG. 5.9 – *Spécification de l'exécution de la continuation synchroWaitEndRelock.*

`getObjectLockInformation` (Figure 5.8) est constitué d'un simple parcours de la liste des objets afin de trouver l'objet considéré et de rendre en conclusion les informations liées au lock de cet objet sous la forme d'un arbre de syntaxe abstraite construit sur le modèle :

`locked(OThId, nbLocks)`

où `OThId` est la référence du thread détenant le lock (valeur quelconque si l'objet n'est pas verrouillé), et `nbLocks` le nombre de locks que le thread `OThId` détient sur l'objet (valeur nulle si l'objet n'est pas verrouillé). Le set `getObjectLockInformation` ne modifie pas ces informations, et ne peut donc pas introduire un conflit avec la propriété 4. Le conflit ne peut donc se produire qu'au niveau du set `getTheLockOrBeBlocked`. Le set `getTheLockOrBeBlocked` représenté en Figure 3.32 analyse les valeurs retournées par l'appel du set `getObjectLockInformation`. Le lock de l'objet n'est modifié que lorsque la première ou la troisième règle s'applique, ce qui traite respectivement des cas suivants :

- règle `UnlockedObject` : l'objet considéré n'est pas verrouillé. Le thread courant peut alors acquérir ce lock sans créer de conflit avec la propriété 4.
- règle `ObjectLockedByTheCurrentThread` : l'objet est déjà verrouillé par le thread courant, le nombre de locks qu'il détient sur l'objet est alors augmenté de 1.

Le seul cas pouvant créer un conflit avec la propriété 4 est traité par la règle appelée `ObjectLockedByAnotherThread`, et dans ce cas, le thread courant se voit refuser l'acquisition d'un lock sur l'objet et passe dans le statut *blocked*. La propriété 4 est donc respectée par l'exécution de l'instruction *synchronized*.

Continuation synchroWaitEndRelock

L'exécution de cette continuation est spécifiée par la règle Typol présentée en Figure 5.9.

Considérons un thread qui était *dormant* sur un objet, et qui vient de quitter ce statut lorsqu'un autre thread a appelé la méthode *notify* ou *notifyAll* (voir Section 3.5.4,

```
HasFoundTheNextExecutingThread_Blocked:
   isTheObjectLocked(ObjId |- ObjL1 -> false())
   ----------------
   ObjL1 |- objects[
               object(
                  OThId, Type, AttrL, Lock, WaitSet, activity(blocked(ObjId), Clr)).
               ObjL2] =>
               OThId,
               objects[
                  object(
                     OThId, Type, AttrL, Lock, WaitSet, activity(executing(), Clr)).
                  ObjL2] ;
            do
               ENVupdateEnvIfNonDormantOrBlockedThreadOnObject(ObjId, ObjL1,
                                                                1)  &
               ENVupdateThreadStatusList(OThId, blocked(ObjId), executing());
```

FIG. 5.10 – *Spécification du passage du statut blocked au statut executing.*

Figure 3.37). Il est alors passé dans le statut *blocked*. Ce thread peut alors être choisi par l'ordonnanceur pour s'exécuter, sous réserve que l'objet sur lequel il était *dormant* et maintenant *blocked* ne soit plus verrouillé (il l'était par le thread ayant appelé la méthode *notify*). Cela est spécifié par la règle Typol de la Figure 5.10 qui fait partie du set `getTheNewExecutingThread` qui définit notre ordonnanceur. Le fait que l'objet ne soit pas verrouillé constitue donc un préalable à l'exécution de la continuation *synchroWaitEndRelock*. Cela est d'autre part vérifié par la première prémisse de la règle de la Figure 5.9 qui appelle le set `getObjectLockInformation`. Ces deux actions (vérification de l'absence de lock sur l'objet et exécution de la continuation) sont effectuées en un seul pas d'exécution, donc sans conflit avec la propriété 4.

Conclusion de la preuve

Les instructions *synchronized* et *synchroWaitEndRelock* sont les seules permettant à un thread d'acquérir un lock sur un objet en Java. Or, nous venons de voir que leurs exécutions étaient conformes à la propriété 4. Cela montre que quelle que soit l'instruction exécutée à l'étape i+1, si la propriété 4 était vérifiée à l'étape i, elle le demeure à l'étape i+1. La propriété 4 était d'autre part vérifiée à l'étape initiale de l'exécution du programme, elle donc vraie par induction.

5.3 Conclusion

Il existe de nombreuses propriétés énoncées de manière informelle au sein du livre référence [61] de Sun. Notre spécification définissant une sémantique formelle dynamique du langage Java, nous avons choisi de considérer principalement les propriétés liées à la concurrence, aux threads et aux mécanismes de synchronisation.

La syntaxe abstraite des structures sémantiques que nous avons définies intègre une bonne partie des propriétés de bases liées par exemple à l'unicité du statut d'un thread ou du possesseur d'un lock. Notre spécification Typol exprime directement d'autres propriétés, liées plus directement à la synchronisation.
Il existe cependant des propriétés plus complexes, liées par exemple au changement de statut d'un thread, qui ne sont pas induites directement par nos structures sémantiques ou notre spécification sémantique. Ces propriétés doivent alors être vérifiées lorsque le changement de statut se produit. Le langage Typol nous permet alors, de part sa modularité, de spécifier des sets chargés de vérifier ces propriétés, propriétés qui sont exprimées en Typol. La vérification de ces propriétés intervient alors en cours d'interprétation du programme.

Il est possible de prévoir une action de l'utilisateur, qui pourrait alors demander ou supprimer la vérification de toutes ou d'une partie des propriétés. La vérification des propriétés deviendrait alors une partie intégrante de notre environnement. Cela constitue une de nos perspectives.

Chapitre 6

Bilan et perspectives

Cette thèse a présenté le langage Java, un langage à objets concurrent. Nous avons successivement étudié les aspects objets, la concurrence et ses mécanismes pour en proposer une spécification formelle en utilisant le formalisme Typol. Ce formalisme fait partie du système Centaur, un environnement de développement d'environnement de programmation. Cela a le grand avantage de permettre la définition d'une sémantique exécutable sous Centaur après une simple compilation des spécifications Typol.
Cette exécutabilité nous a conduit tout naturellement à développer un environnement d'exécution particulièrement axé sur la visualisation graphique de la topologie des objets et des mécanismes de synchronisation. Un autre aspect que nous avons étudié est l'utilisation de cette formalisation comme une base formelle pour la vérification de propriétés du langage Java.

Ce chapitre constitue en quelque sorte un bilan de notre étude, et présente ensuite des perspectives multiples, liées tout d'abord à des développements de notre travail, puis s'élargissant vers des domaines connexes.

6.1 Bilan

Cette première section permet de dresser un bilan de notre étude, que ce soit au niveau de la définition d'une spécification formelle d'un large sous-ensemble du langage Java, du développement de l'environnement d'exécution associé, ou de la vérification de propriétés du langage.

6.1.1 Une sémantique formelle d'un large sous-ensemble de Java

L'étude (présentée dans la Section 2.1)des concepts objets et des mécanismes liés à la concurrence nous a permis de mettre en avant des difficultés présentées par l'écriture et la mise au point de programmes Java. Cette complexité croissante des langages est

une des raisons du développement de spécifications formelles définissant une sémantique claire et surtout précise des langages par opposition aux sémantiques informelles des tutoriaux ou d'autres ouvrages de présentation des langages.
Nous avons donc défini une sémantique formelle dynamique du langage Java (étudiée dans le Chapitre 3. Pour cela, nous avons tout d'abord écrit une syntaxe concrète et abstraite du langage, puis étudié la définition de structures sémantiques permettant d'une part de représenter l'état de l'exécution à chaque étape, et d'introduire des continuations utilisées par notre sémantique *small-step*.

Les spécifications sémantiques sont liées directement au source, ce qui signifie que nous ne sommes pas limités à une machine virtuelle particulière (standard, propriétaire ou modifiée). A la place, nous interprétons directement, en utilisant des règles sémantiques liées aux constructeurs du source Java, le source du programme (et non le byte-code). Cette spécification, écrite dans le formalisme Typol, est originale par l'utilisation conjointe du style *big-step* pour définir les aspects de Java liés aux objets (comme leur création, l'héritage, etc), et du style *small-step* pour spécifier les aspects liés à la concurrence (exécution concurrente des threads, synchronisation, etc). L'exécution concurrente des threads est simulée par l'introduction d'entrelacement au niveau de l'exécution des threads, ce qui permet de rendre compte des mêmes mécanismes.

Une autre originalité de cette spécification est constituée par le fait qu'elle soit exécutable au sein du système Centaur après une simple compilation. La génération automatique de l'interpréteur à partir de ces spécifications ne requiert des changements qu'au sein des spécifications, et non dans le code de l'environnement (par exemple pour passer d'une version du langage Java à une autre).

6.1.2 Un environnement d'exécution innovant

Notre spécification formelle étant exécutable, nous l'avons utilisée comme base d'un environnement d'exécution de programmes Java présenté dans le Chapitre 4.
Les effets de l'exécution du programme sont montrés à la volée, grâce à la visualisation et à l'animation de l'interprétation du programme. L'environnement que nous avons développé propose :

- des vues textuelles permettant la visualisation de :
  - la liste des objets avec pour chacun leur type, leurs attributs, et leur lock
  - la liste des threads avec leur activité
- des vues synthétiques proposant :
  - la liste des objets classés selon leur statut
  - la liste des interactions entre threads et objets
- une vue graphique représentant :
  - la topologie de l'ensemble des objets
  - les statuts des objets grâce à des codes de couleur

  - les synchronisations
- la détection de deadlocks
- un mécanisme d'abstraction permettant le passage à l'échelle.

Chaque vue est obtenue directement par un pretty-printing des structures sémantiques utilisées au sein de notre spécification. La vue graphique demande cependant un traitement plus complexe, avec une traduction des structures sémantiques vers des structures graphiques. Cette vue graphique représente la topologie de l'ensemble des objets et des threads. Grâce à la visualisation immédiate des références, des locks et des synchronisations, elle permet en particulier de détecter facilement tout objet partagé par plusieurs threads, et donc de protéger ses attributs par des méthodes *synchronized*.
Cette visualisation est animée et réalisée de façon incrémentale afin de gagner en efficacité. Elle est rendue possible grâce à un système de notification de le sémantique vers l'environnement l'informant de chaque modification et mise à jour des structures sémantique.

Aucune compilation du programme n'est nécessaire, ce qui rend possible un débuggage interactif. Il est cependant nécessaire de stopper l'exécution avant de modifier le source du programme et de la lancer à nouveau pour visualiser la nouvelle exécution. Aucune instrumentation n'est requise au niveau du source du programme. Comme l'interprétation a lieu au niveau du source, les développeurs acquièrent une meilleure compréhension du comportement de leur programme.

Les principales originalités de cet environnement sont la visualisation animée sous forme graphique de l'exécution du programme et surtout le mécanisme d'abstractions, qui permet de visualiser uniquement un sous-graphe donné ce qui est très utile en présence d'un grand nombre d'objets.

6.1.3 Une sémantique formelle comme base de vérification de propriétés

Un dernier chapitre (Chapitre 5) nous a permis de présenter une des fonctions d'une spécification formelle : servir de base à l'écriture de preuves de propriétés de programmes ou de langage.

Nous avons particulièrement étudié les propriétés du langage Java liées à la concurrence et aux mécanismes de synchronisation. Notre spécification traduit directement une grande partie de ces propriétés, que ce soit à travers la syntaxe abstraite des structures sémantiques que nous avons définies ou directement à travers nos règles Typol. Dans le cas où ce n'est pas directement évident, nous avons étudié comment exprimer ces propriétés directement en Typol, et intégrer leur vérification à l'interprétation de programmes. Cela nous a permis d'éviter une traduction vers un autre formalisme, source possible d'erreurs.
Enfin, un schéma de la preuve par induction d'une de ces propriété a été présenté.

6.2 Perspectives

6.2.1 Développements de notre travail

Dans un premier temps de nombreuses perspectives existent à partir de notre travail. Ce paragraphe présente des perspectives de développement de notre spécification et de notre environnement. Une dernière perspective est constituée par l'intégration de la vérification d'assertions dans notre spécification.

6.2.1.1 Développements de notre spécification

Nous avons vu dans la section 2.1.4 que certains éléments du langage Java n'étaient pas spécifiés dans notre sémantique.
D'autre part, notre sémantique prend pour hypothèse que le programme Java à analyser est syntaxiquement correct, mais surtout correct au niveau typage statique. Un bon complément de notre travail serait l'écriture d'une sémantique statique du langage qui pourrait ainsi vérifier cette correction des types du programme.

6.2.1.2 Développement des interactions sémantique-environnement

Nous avons vu dans la section 2.1.3.2 que notre ordonnanceur était déterministe. Il peut être intéressant d'introduire de l'indéterminisme à ce niveau afin de coller au plus près au modèle. D'autre part, permettre à l'utilisateur d'influer sur l'ordonnancement (comme cela a été fait en Eiffel// [9], en choisissant le thread devant s'exécuter, ou en bloquant artificiellement un thread, faciliterait la mise au point des programmes en mettant en évidence les différentes exécutions possibles.

6.2.1.3 Intégration d'assertions à notre spécification

Le traitement d'assertion pourrait compléter notre travail sur les propriétés du langage en cherchant à vérifier des propriétés exprimées directement au sein des programmes (sous forme de commentaires Java spécifiques afin de ne pas avoir à supprimer ces assertions lors de l'utilisation du programme avec un compilateur "classique"). L'intégration de la vérification d'assertions à notre spécification serait tout à fait réalisable, il faudrait pour cela définir une syntaxe concrète et abstraite de ces assertions, puis naturellement leur sémantique.

6.2.2 Un nouveau langage de spécification sémantique

L'utilisation du formalisme Typol nous a permis de l'étudier, et d'en découvrir les avantages et les faiblesses. Cette étude ouvre d'autres perspectives à travers la création d'un langage de spécification sémantique innovant.

6.2.2.1 Limites du formalisme Typol

Le formalisme Typol que nous avons utilisé pour définir notre spécification sémantique du langage Java possède comme nous l'avons vu des qualités certaines comme sa syntaxe proche de celle de la déduction naturelle, son exécutabilité et sa modularité. Cependant, il possède aussi des limites :

- Il n'est pas possible d'indiquer qu'une règle doit s'appliquer pour toute variable ne représentant pas un élément d'une liste sans citer explicitement tous les éléments pour lesquels la règle ne doit pas s'appliquer. Prenons l'exemple d'un set contenant des règles traitant successivement plusieurs cas (Cas1, Cas2, ...). Si l'on veut prévoir une règle traitant de tous les autres cas non encore cités, il n'est pas possible d'exprimer une condition du style `pour tous les cas différents des précédents`, il faut dire explicitement `pour les cas différents de Cas1, différents de Cas2, ...`. Ce problème possède une solution en Typol : il s'agit de l'utilisation de la clause `error recovery`, mais cela pollue la nature logique du cadre en supprimant la possibilité de backtrack.
- Les règles d'un set Typol doivent s'exclure mutuellement, et cela entraîne une augmentation de la complexité des règles, ou du moins la multiplication de celles-ci.
- Typol ne permet pas de représenter des ensembles autrement que par des listes : par exemple l'ensemble des objets est modélisé par une liste.
- Typol ne comprend pas les ensembles de types : il n'est pas possible de demander qu'une règle s'applique pour tous les sous-types d'un type donné.
- Le passage d'une sémantique formelle écrite dans le formalisme Typol vers les systèmes de preuves est très délicat et complexe. Ce passage ne peut se faire directement de façon automatique.

Une perspective intéressante peut alors être la création d'un langage de spécification sémantique en palliant ces faiblesses.

6.2.2.2 L'environnement SmartTools

Le système Centaur dans lequel est défini possède lui aussi ses propres faiblesses. En effet, son utilisation requiert l'apprentissage de multiples langages :

- Metal pour définir la syntaxe concrète et la syntaxe abstraite
- Ppml pour définir un pretty-printer. Ce langage permet d'afficher de manière lisible un programme stocké en mémoire sous forme d'arbre de syntaxe abstraite.
- Typol pour spécifier la sémantique
- Prolog : sa connaissance facilite la mise au point des spécifications Typol
- Lisp, langage dans lequel l'environnement d'exécution est développé

Cette multitude de langages au sein du système Centaur constitue un frein à son utilisation, et à sa diffusion en introduisant des problèmes de licences, ou de portabilité suivant les systèmes d'exploitation.

L'environnement SmartTools [10] en cours de développement dans le projet OASIS de l'INRIA Sophia Antipolis évite ces écueils. Il s'agit d'un générateur d'environnements de développement qui propose comme principaux composants un éditeur structuré et des outils sémantiques basé sur l'utilisation des visiteurs (pattern Java). SmartTools est facile à utiliser grâce à son interface utilisateur graphique développée avec les API Swing de Java. Le fait d'être basé sur Java et la technologie web (XML, XSSL, etc.) permet à SmartTools d'être portable, sous Unix comme sous Windows, et de pouvoir être connecté aisément à d'autres systèmes. SmartTools est en effet beaucoup plus ouvert sur l'extérieur en permettant par exemple de prendre en entrée une spécification syntaxique DTD au lieu d'avoir à utiliser le méta-langage AST de SmartTools dédié aux spécifications syntaxiques.

6.2.2.3 Un langage de spécification sémantique avec de multiples orientations

Des recherches sont actuellement en cours afin de définir un nouveau langage de spécification sémantique qui ait les mêmes qualités que le formalisme Typol sans en avoir les inconvénients. Ce langage doit prendre en compte les différentes orientations que l'on peut suivre à partir d'une définition formelle d'un langage : en faire la base d'un environnement de développement ou/et s'en servir comme base pour effectuer des preuves de propriétés du langage ou de programmes.

Une spécification sémantique exécutable

Ce langage de spécification doit être exécutable, permettant ainsi de servir de base à un environnement d'exécution tel que Jitan, mais réalisé avec les outils Java pour le rendre portable et plus aisé à développer. Ce formalisme doit donc permettre une communication avec l'environnement.

Une transition sémantique - preuve en Coq facilitée

Une perspective intéressante serait que ce même formalisme permette d'effectuer des preuves de propriétés de programmes. Nous visons la possibilité d'une traduction directe vers le système de preuves Coq.
Un tel formalisme de spécification sémantique permettrait alors à l'utilisateur d'une part de tester ses programmes grâce à l'environnement, et d'autre part d'effectuer des preuves de propriétés de son programme grâce au système de preuves Coq sans avoir de manipulation à effectuer.

Bibliographie

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[2] G. Agha. *Actors: A model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, 1986.

[3] G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. technical report, UIUC, 1995. To appear in Journal of Functional Programming.

[4] G. Agha, I. A. Mason, S. Smith, and C. Talcott. Towards a Theory of Actor Computation. *CONCUR'92, pages 565-580, Stony Brook, NY, USA, August*, 1992.

[5] Andrews. *Logic Programming: Operational Semantics and Proof Theory*. Cambridge University Press, Cambridge, UK, 1992.

[6] G. R. Andrews. *Concurrent programming*. Benjamin/Cummings Publishing Comp., Redwood City, 1 edition, 1991.

[7] G. R. Andrews and F. B. Schneider. Concepts and notations for concurrent programming. Technical Report TR82-520, Cornell University, Computer Science Department, September 1982.

[8] I. Attali, D. Caromel, and S. O. Ehmety. An operational semantics for the eiffel language. Technical Report RR-2732, Inria, Institut National de Recherche en Informatique et en Automatique.

[9] I. Attali, D. Caromel, S. O. Ehmety, and S. Lippi. Semantic-based visualization for parallel object-oriented programming. In *Proceedings of OOPSLA'96*, volume 31, 10 of *ACM Sigplan Notices*, San Jose, CA, October 1996. ACM Press.

[10] I. Attali, C. Courbis, P. Degenne, A. Fau, D. Parigot, and C. Pasquier. SmartTools: a generator of interactive environment tools. In *In Reinhard Wilhelm, editor, International Conference on Compiler Construction CC'01*, volume à paraître, Genova, Italy, avril 2001. Springer-Verlag. disponible à l'adresse http://www-sop.inria.fr/oasis/SmartTools/publications/ETAPS01/smartetap01.ps.

[11] K. Beck. Object explorer for visual works. Commercial Product available at <http://c2.com/ppr/about/author/kent.html>, First Class Software, Inc.

[12] G. Berry and G. Gonthier. The ESTEREL synchronous programming language : design, semantics, implementation. Technical Report RR-0842, Inria, Institut National de Recherche en Informatique et en Automatique.

[13] G. Blaschek. Type-safe object-oriented programming with prototypes – the concepts of omega. *Structured Programming*, 12:217–225, 1991.

[14] G. Blaschek. *Object-Oriented Programming with Prototypes.* Springer-Verlag, New York, NY, 1994.

[15] E. Börger, J. Schmid, and R. Stärk. *Java and the Java Virtual Machine - Definition, Verification, Validation.* Springer-Verlag, 2001.

[16] E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In *23rd International Symposium on Mathematical Foundations of Computer Science*, LNCS. Springer-Verlag, 1998. to appear.

[17] E. Börger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In *Formal Syntax and Semantics of Java*. Springer-Verlag, LNCS 1523, 1998.

[18] A. H. Borning. Classes versus prototypes in object-oriented languages. In *Proceedings of the ACM-IEEE Fall Joint Computer Conference, Montvale (NJ), USA*, pages 36–39, 1986.

[19] P. Borras and et al. Centaur: the System. In *SIGSOFT'88 Third Annual Symposium on Software Development Environments*, Boston, 1988.

[20] G. Brat, K. Havelund, S. Park, and W. Visser. Model Checking Programs. In *Proc of The IEEE International Conference on Automated Software Engineering (ASE)*, September 2000.

[21] M. H. Brown. Zeus: a system for algorithm animation and multiview editing. In *Proc. of the IEEE Workshop on Visual Languages*, 1991.

[22] K. B. Bruce, J. Crabtree, and G. Kanapathy. An operational semantics for TOOPLE: A statically-typed object-oriented programming language. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics 9th International Conference, New Orleans, LA, USA, Proceedings*, volume 802 of *Lecture Notes in Computer Science*, pages 603–626. Springer-Verlag, New York, NY, April 1993.

[23] J. Bryans, J. Davies, and S. Schneider. Towards a denotational semantics for ET-LOTOS. *Lecture Notes in Computer Science*, 962, 1995.

[24] *Concurrent Object-Oriented Programming.* Communications of the ACM, 36 (9), 1993. Special issue.

[25] J. R. Carbonell and A. M. Collins. Natural semantics in artificial intelligence. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 344–351, Standford, CA, August 1973. William Kaufmann.

[26] D. Caromel. *Programmation parallèle asynchrone et impérative: études et propositions.* Ph.D. thesis, Université de Nancy, February 1991.

[27] CenterLine Software, Cambridge, MA. *CodeCenter, The Programming Environment*, 1992.

[28] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self, a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Notices*, 24(10), October 1989.

[29] C. Choppy et al. A LISP compiler for FP language and its proof via algebraic semantics. In *Lecture Notes in Computer Science 185*. Springer-Verlag, New York, NY.

[30] C. Colby, L. Jategaonkar Jagadeesan, R. Jagadeesan, K. Läufer, and C. Puchol. Design and Implementation of Triveni: a Process-Algebraic API for Threads + Events. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 58–67. IEEE Computer Society Press, 1998.

[31] C. Colby, L. Jategaonkar Jagadeesan, R. Jagadeesan, K. Läufer, and C. Puchol. Objects and Concurrency in Triveni: a Telecommunication Case Study in Java. In *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS-98)*, pages 133–148, Berkeley, April 27–30 1998. USENIX Association.

[32] C. Colby, L. Jategaonkar Jagadeesan, R. Jagadeesan, K. Läufer, and C. Puchol. The Semantics of Triveni: a Process-Algebraic API for Threads + Events. In Rance Cleaveland, Michael Mislove, and Philip Mulry, editors, *Electronic Notes in Theoretical Computer Science*, volume 14. Elsevier Science Publishers, 2000.

[33] Compaq. Visual Threads. Commercial Product available at <http://www5.compaq.com/products/software/visualthreads/>.

[34] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and querying software structures. In *Proc. of the* 14^{th} *International Conference on Software Engineering*, pages 138–156, May 1992.

[35] J. Corbett, M. Dwyer, C. Pasareanu J. Hatcliff, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *2000 International Conference on Software Engineering*, Februar 2000.

[36] O.-J. Dahl and K. Nygaard. Simula Begin. Technical report, Norsk Regnesentral (Norwegian Computing Center), Oslo/N, 1967.

[37] D. Dean. The security of static typing with dynamic linking. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, Zurich, Switzerland, April 1997.

[38] C. Demartini, R. Iosif, and R. Sisto. Modeling and Validation of Java Multithreaded Applications using Spin. In *Proc. of the 4th SPIN Workshop, Paris, France*, November 1998.

[39] P. Deransart, M. Jourdan, and B. Lorho. *Attribute Grammars: Definitions, Systems and Bibliography*, volume 323 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, 1989.

[40] J. Despeyroux. An algebraic specification of a Pascal compiler. *ACM SIGPLAN Notices*, 18(12):34–48, December 1983.

[41] T. Despeyroux. Typol: A Formalism to Implement Natural Semantics. Research Report 94, INRIA, 1988.

[42] A. Di Pierro and C. Palamidessi. A logical denotational semantics for constraint logic programming. *Lecture Notes in Computer Science*, 788, 1994.

[43] C. Dichter. We test the top 6 Java visual IDEs. Product Reviews available at <http://www.javaworld.com/javaworld/jw-04-1998/jw-04-visual-ides.html>, JavaWorld: IDG's magazine for the Java community, 1998.

[44] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[45] D. G. Drake. Introduction to Java threads. *JavaWorld: IDG's magazine for the Java community*, 1(2), April 1996.

[46] S. Drossopoulou and S. Eisenbach. Is the Java Type System Sound? In *4th Int. Workshop Foundations of Object-Oriented Languages*, 1997.

[47] S. Drossopoulou and S. Eisenbach. Java is Type Safe - Probably. In *ECOOP'97*, LNCS 1241, pages 389–418. Springer Verlag, January 1997.

[48] S. Drossopoulou and S. Eisenbach. Towards an Operational Semantics and Proof of Type Soundness for Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, 1998.

[49] S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. *Lecture Notes in Computer Science*, 1523, 1999.

[50] S. Drossopoulou, J. Uhl, G. Persch, G. Goos, Dausmann, M., and G. Winterstein. An attribute grammar for Ada. *ACM SIGPLAN Notices*, 17(6):334–339, June 1982.

[51] M. Dwyer and L. A. Clarke. Data Flow Analysis for Verifying Properties of Concurrent Programs. In *Proc, of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 62–75, December 1994.

[52] D. Dyer. Can Assure save Java from the perils of multithreading? article available at <http://www.javaworld.com/javaworld/jw-10-1998/jw-10-assure.html>, JavaWorld: IDG's magazine for the Java community, 1998.

[53] D. Dyer. Capsule Reviews of JAVA tools. Product Reviews Available at <http://www.andromeda.com/people/ddyer/java/Reviews.html>, Andromeda Software, 1998.

[54] M. Felleisen, R. B. Findler, M. Flatt, and S. Krisnamurthi. The DrScheme project: An overview. *ACM SIGPLAN Notices*, 33(6):17–23, June 1998.

[55] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. DrScheme: A pedagogic programming environment for scheme. *Programming Languages: Implementations, Logics, and Program s*, 1292:369–388, September 1997.

[56] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, January 1998. ACM.

[57] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, 1998.

[58] K. A. Frenkel. Special issue on parallelism. *Communications of the ACM*, 29(12):1168–1169, December 1986.

[59] G. Friedrich, W. H., C. Stary, and M. Stumptner. Objview: A task-oriented, graphics-based tools for object visualization and arrangement. In *Proc. ECOOP '89*, pages 299–310, Nottingham, July 1989. Cambridge University Press.

[60] U. Goltz. CCS and petri nets. In *Semantics of Systems of Concurrent Processes*, pages 334–357, Berlin - Heidelberg - New York, April 1990. Springer.

[61] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison–Wesley, 1996.

[62] J. E. Grass. Object-oriented design archaeology with CIA++. *Computing Systems*, 5(1):5–67, 1992.

[63] P. Greg. *Using Visual J++*. Que, 1996. ISBN 078970899X.

[64] I. Guessarian. Some applications of algebraic semantics. In J. Winkowski, editor, *Proceedings of the 7th Symposium on Mathematical Foundations of Computer Science*, volume 64 of *LNCS*, pages 257–266, Zakopane, Poland, September 1978. Springer.

[65] I. Guessarian. *Algebraic semantics*, volume 99 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1981.

[66] V. Haarslev and R. Möller. A framework for visualizing object-oriented systems. In *Proc. OOPSLA/ECOOP '90, ACM SIGPLAN Notices 25 (10)*, pages 237–244, October 1990.

[67] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. In *International Journal on Software Tools for Technology Transfer (STTT)*, 1998.

[68] K. Havelund and J. Skakkebaek. Applying Model Checking in Java Verification. In *Proc. of the 5th International SPIN Workshop*, volume 1680, pages 216–231. Springer, LNCS, September 1999.

[69] L. Holenderski. Lustre. *Lecture Notes in Computer Science*, 891, 1995.

[70] A. Holub. Programming Java threads in the real world: Threading architectures. *JavaWorld: IDG's magazine for the Java community*, 3(9), September 1998.

[71] G. J. Holzmann. The Spin Model Checker. *IEEE Trans. on Software Engineering, Volume 23*, pages 279–295, May 1997.

[72] A. Le Hors. Graph: A directed graph displaying server, GIPE 2 Esprit project, 4th review report, workpackage 4, 1992.

[73] IBM Corporation. Visual Age for Java. Commercial Product available at <http://www.software.ibm.com/ad/vajava/>, 1997.

[74] T. Jensen, D. Le Métayer, and T. Thorn. Security and Dynamic Class Loading in Java: a Formalisation. In *Proceedings of the 1998 IEEE International Conference on Computer Languages*, pages 4–15, May 1998.

[75] D. F. Jerding, J. T. Stasko, and T.Ball. Visualizing interactions in program executions. In *Proc. of the* 17^{th} *International Conference on Software Engineering*, pages 360–370, May 1997.

[76] G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[77] G. Kahn, B. Lang, and B. Mélèse. Metal : A Formalism to Specify Formalisms. *Science of Computer Programming, Volume 3, North-Holland*, 1983.

[78] S. Kent. Constraint diagrams: Visualizing invariants in object-oriented models. In *Proc. OOPSLA'97 (Object-Oriented Programming: Systems, Languages, and Applications)*, volume 32, number 10. ACM Press, Sigplan Notices, Oct 1997.

[79] E. Kraemer and J. T. Stasko. The visualization of parallel systems: an overview. *Journal of Parallel and Distributed Computing, 18*, 1993.

[80] C. Laffra and A. Malhotra. Hotwire – A visual debugger for C++. In *USENIX Sixth C++ Technical Conference*, pages 109–122, Cambridge, MA, April 11-14 1994. USENIX.

[81] W. R. LaLonde and J. R. Pugh. *Inside Smalltalk: Volume I.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[82] W. R. LaLonde and J. R. Pugh. *Inside Smalltalk: Volume II.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[83] L. Lamport. The 'hoare logic' of concurrent programs. In Springer-Verlag, editor, *Acta Informatica 14*, pages 21–37, 1980.

[84] L. Lamport and F. B. Schneider. The "hoare logic" of CSP and all that. *TOPLAS*, 6(2):281–296, April 1984.

[85] J. Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, New York, NY, 1993. ACM.

[86] D. Lea. *Concurrent programming in Java: design principles and patterns.* The Java series. Addison-Wesley, Reading, MA, USA, 1996.

[87] D. Lea. *Concurrent Programming Using Java.* Addison-Wesley, 1996.

[88] G. T. Leavens, A. L. Baker, and C. Ruby. JML: a java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, October 1998. `http://www-dse.doc.ic.ac.uk/~sue/oopsla/cfp.html`.

[89] G. T. Leavens, K. Rustan, M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: Notations and Tools Supporting Detailed Design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106. ACM, October 2000.

[90] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[91] L. Lemay and Charlesworth. *Teach Yourself SunSoft Java Workshop in 21 Days.* SAMS, 1996. ISBN 1575211599.

[92] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *Proc. OOPSLA'97 (Object-Oriented Programming: Systems, Languages, and Applications)*, volume 32, number 10. ACM Press, Sigplan Notices, Oct 1997.

[93] J. J. Lukkien. An operational semantics for the guarded command language. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction, Second International Conference, Oxford, U.K.*, volume 669 of *Lecture Notes in Computer Science*, pages 233–249. Springer-Verlag, New York, NY, June/July 1993.

[94] W. Marek. A natural semantics for modal logic over databases. *Theoretical Computer Science*, 56(2):187–209, February 1988.

[95] C. McManis. Java in depth: Synchronizing threads in Java. *JavaWorld: IDG's magazine for the Java community*, 1(2), April 1996.

[96] Metrowerks. Metrowerks CodeWarrior. Commercial Product available at <http://www.metrowerks.com/>, Austin, TX, 1995.

[97] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[98] E. Morcos-Chounet and A. Conchon. Ppml: A General Formalism to Specify Pretty-printing. *Proc. of the IFIP Congress Dublin, North-Holland*, 1986.

[99] P. D. Mosses. Denotational semantics. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 11, pages 577–631. The MIT Press, New York, NY, 1990.

[100] H. Muller, J. Winckler, S. Grzybek, M. Otte, B. Stoll, F. Equoy, and N. Higelin. PASTIS-program animation using X. In Anonymous, editor, *European X Window System Conference*, pages 104–111, Edinburgh, UK, 1990. CEP Consultants.

[101] B. A. Myers. Incense: A system for displaying data structures. *Computer Graphics*, 17(3):115–125, July 1983.

[102] G. Naumovich, S. Avrunin, and L. A. Clarke. Data Flow Analysis for Checking Properties of Concurrent Java Programs. In *Proc. of the International Conference on Software Engineering*, May 1999.

[103] O. Nierstrasz. Active objects in hybrid. In *Proc. OOPSLA '87, ACM SIGPLAN Notices 22 (12)*, pages 243–253, 1987.

[104] O. Nierstrasz. Two models of concurrent objects. In *ACM SIGPLAN Notices 24 (4), Proc. Workshop on Object-Based Concurrent Programming (San Diego, Sept 1988)*, pages 174–176, April 1989.

[105] O. Nierstrasz and M. Papathomas. Towards a type theory for active objects. In *ACM OOPS Messenger, 2 (2), Proc. OOPSLA/ECOOP 90 Workshop on Object-Based Concurrent Systems*, pages 89–93, April 1991.

[106] T. Nipkow. Invited talk: Embedding programming languages in theorem provers. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *LNAI*, page 398, Berlin, July 7–10, 1999. Springer-Verlag.

[107] T. Nipkow and D. Von Oheimb. Java Light is Type Safe - Definitely. In *25st ACM Symp. Principles of Programming Languages*, 1998.

[108] T. Nipkow and D. Von Oheimb. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 119+37. SV, 1998.

[109] Nova Software Labs. JProbe Profiler. Commercial Product available at <http://www.nsl.fr/index.html>, Paris, France, 1998.

[110] K. Nygaard and O.-J. Dahl. Simula 67. In R. W. Wexelblat, editor, *History of Programming Languages*. acm press, 1981.

[111] S. Oaks and H. Wong. *Java Threads.* O'Reilly, 1997.

[112] J. Paakki. Attribute grammar paradigms: A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[113] A. Paepcke. *Object-Oriented Programming: The CLOS Perspective.* The MIT Press, 1993.

[114] D. Park, U. Stern, and D. Dill. Java Model Checking. In *First International Workshop on Automated Program Analysis, Testing and Verification, Limerick, Ireland*, June 2000.

[115] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proc. OOPSLA '93, ACM SIGPLAN Notices, 28 (10)*, pages 326–337, October 1993.

[116] W. De Pauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. In *Proc. ECOOP '94*, LNCS 821, pages 163–182, Bologna, Italy, July 1994. Springer-Verlag.

[117] G. D. Plotkin. A Structural Approach to Operational Semantics. Report, DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[118] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.

[119] E. Poll, J. van den Berg, and B. Jacobs. Formal Specification and Verification of JavaCard's Application Identifier Class. In *Proceedings of the JavaCard Workshop, Cannes, France*, September 2000.

[120] E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In *Fourth Smart Card Research and Advanced Application Conference (IFIP Cardis)*. Kluwer Academic Publishers, 2000.

[121] J. Power and G. Rosolini. A modular approach to denotational semantics. *Lecture Notes in Computer Science*, 1443, 1998.

[122] D. Prawitz. *Natural Deduction.* Almquist and Wiksell, Stockholm, 1965.

[123] Z. Qian. A Formal Specification of the Java Virtual Machine Instructions for Objects, Methods and Subroutines. In *Formal Syntax and Semantics of Java.* Springer-Verlag, LNCS 1523, 1998.

[124] G.C. Roman and et al. A System for Declarative Visualization of Concurrent Computations. *Journal of Visual Languages and Computing, 3 (2)*, 1992.

[125] M. Roncken and R. Gerth. A denotational semantics for synchronous and asynchronous behavior with multiform time. In Marta Zofia Kwiatkowska, Michael William Shields, and Richard Monro Thomas, editors, *Proceedings of the International BCS-FACS Workshop on Semantics for Concurrency*, Workshops in Computing, pages 21–37, Berlin, July 1990. Springer.

[126] A. W. Roscoe. Denotational semantics for OCCAM. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes on Computer Science*, pages 306–329. Springer-Verlag, Berlin, 1985.

[127] B. J. Ross. Using algebraic semantics for proving prolog program termination and transformation. In *Proceedings of the 1991 UK ALP meeting*, pages 63–70, Edinburgh, April 1991.

[128] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science*, 3(4):377–413, 1997.

[129] M. S. Sherman. *Paragon: A Language Using Type Hierarchies for the Specification, Implementation and Selection of Abstract Data Types.* PhD thesis, School of Computer Science, Carnegie Mellon University, August 1983. CMU-CS-83-147, also published by Springer-Verlag, LNCS, Volume 189, 1985.

[130] C. R. Snow. *Concurrent Programming*, volume 26 of *Cambridge Computer Science Texts*. Cambridge University Press, New York, NY, 1992.

[131] J. T. Stasko and E. Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing, 18*, 1993.

[132] Step Ahead Software Pty Ltd. Javelin. Commercial Product available at <http://www.ozemail.com.au/ stepsoft/javelin.htm>, NSW 2077 Australia, 1998.

[133] G. S. Stiles. Safe and Verifiable Design of Multithreaded Java Programs with CSP and FDR. In *Proceedings of Formal Underpinnings of Java, an OOPSLA'98 Workshop*, Vancouver, CA, October 1998. Technical Report, Princeton University.

[134] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, 1986.

[135] Sun. JDB The Java Debugger. Commercial Product available at <http://java.sun.com/products/jdk/1.1/docs/tooldocs/solaris/jdb.html>, Palo Alto, CA, 1997.

[136] Sun. Java Debugger Interface. Commercial Product available at <http://java.sun.com/products/jdk/1.3/docs/guide/jpda/jdi/index.html>, Palo Alto, Ca, 1998.

[137] SuperCede, Inc. SuperCede Java. Commercial Product available at <http://www.supercede.com>, Bellevue, WA, 1998.

[138] D. Syme. Proving Java Type Soundness. Technical report 427, University of Cambridge Computer Laboratory, 1997.

[139] D. Syme. Proving Java type soundness. *Lecture Notes in Computer Science*, 1523, 1999.

[140] A. Taivalssari. Classes versus prototypes: Some philosophical and historical observations. *Journal of Object-Oriented Programming*, 10(7):44–49, November 1997.

[141] Tek-Tools, Inc. Kawa. Commercial Product available at <http://www.tek-tools.com/kawa/>, Dallas TX, 1998.

[142] D. Terrasse. Encoding natural semantics in Coq. In V. S. Alagar, editor, *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology*, pages 230–244, Montreal, Canada, July 1995. Springer-Verlag LNCS 936.

[143] D. Terrasse. Vers un environnement de développement de preuves en Sémantique naturelle. Thèse de doctorat, ENPC, 1995.

[144] M. Tienari. *On the Definition of an Attribute Grammar*, volume 94 of *Lecture Notes in Computer Science*, pages 408–414. Springer-Verlag, Heidelberg, FRG, 1980.

[145] C. Tomlinson and M. Scheevel. Concurrent object-oriented programming languages. In Won Kim and Farederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 5, pages 79–124. Addison-Wesley Publishing Co., Reading, Mass., 1989.

[146] E. Wagner, J. Wright, J. Goguen, and J. Thatcher. Some fundamentals of order-algebraic semantics. In Andrzej Mazurkiewicz, editor, *Proceedings, Fifth Symposium on Mathematical Foundations of Computer Science*, volume 45 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 1976.

[147] E. G. Wagner, S. L. Bloom, and J. W. Thatcher. Why algebraic theories? In M. Nivat and J. C. Reynolds, editors, *Algebraic Semantics*, pages 607–634. Cambridge University press, 1985.

[148] D. A. Watt. An extended attribute grammar for PASCAL. *ACM SIGPLAN Notices*, 14(3):60–74, February 1979.

[149] D. A. Watt. Executable denotational semantics. *Software: Practice and Experience*, 16(1):13–43, 1986.

[150] P. Wegner. Design issues for object-based concurrency. In *Proc. of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, LNCS 612, pages 245–256, 1992.

[151] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* Foundations of Computing series. MIT Press, February 1993.

[152] A. Yonezawa and M. Tokoro, editors. *Concurrent Object-Oriented Programming.* The MIT Press, Cambridge, Mass., 1987.

Annexe 1 : syntaxe concrète et abstraite des constructeurs de Java

L'annexe 1 figurera in extenso dans la version finale. Elle est pour l'instant disponible à l'adresse :
http://www.inria.fr/oasis/personnel/Marjorie.Russo/Annexes/Annexe1.txt

Annexe 2 : syntaxe abstraite des structures sémantiques

L'annexe 2 figurera in extenso dans la version finale. Elle est pour l'instant disponible à l'adresse :
http://www.inria.fr/oasis/personnel/Marjorie.Russo/Annexes/Annexe2.txt

Annexe 3 : spécification d'une sémantique formelle de Java

L'annexe 3 figurera in extenso dans la version finale. Elle est pour l'instant disponible à l'adresse :
http://www.inria.fr/oasis/jitan/java_sem.html