Université de Nice-Sophia Antipolis
UFR SCIENCES

École Doctorale : STIC

THÈSE

Présentée pour obtenir le titre de :
DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ DE NICE SOPHIA ANTIPOLIS

Spécialité : INFORMATIQUE

par

Marjorie RUSSO

Équipe d'accueil : OASIS - INRIA Sophia Antipolis

Titre de la thèse :

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

Soutenue publiquement le 13 juillet 2001 devant le jury

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

Remerciements

En tout premier lieu, je tiens à remercier Isabelle Attali et Denis Caromel pour m'avoir encadrée pendant ma thèse. Je souhaite ici leur témoigner ma reconnaissance pour l'aide qu'ils m'ont apportée et la patience dont ils ont fait preuve. Ils ont su en effet me guider et m'indiquer certaines grandes orientations que pouvait suivre mon travail tout en me permettant d'effectuer mes propres choix. Je les remercie aussi des possibilités qu'ils m'ont offertes de présenter mes recherches, notamment dans les équipes des départements d'informatique de San Luis (UFMA) et de Recife (UFPE).
Le temps qu'ils m'ont accordé, leur disponibilité et leur soutien scientifique et moral (surtout dans certains moments difficiles) m'ont indiscutablement été d'un grand secours.

Egon Börger a accepté d'être rapporteur de cette thèse. Ses travaux sur la sémantique formelle de Java (entre autres) en font un expert reconnu du domaine, et je lui suis très reconnaissante d'avoir acceptée la charge de rapporteur.

C'est avec plaisir que je compte Traian Muntean parmi mes rapporteurs. Au delà de sa réputation scientifique, on ne peut qu'apprécier ses qualités humaines et d'enseignement dont j'ai eu l'occasion de profiter durant mes études d'ingénieur à l'ESIL.

Je remercie Michel Riveill qui me fait l'honneur de présider ce jury.

Je souhaite aussi remercier Jean Marc Jezequel qui a accepté de juger mon travail.

Je voudrais finalement témoigner de ma reconnaissance envers l'équipe du projet OASIS et les membres de l'ex-projet CROAP qui m'ont accueillie très chaleureusement et ont permis à ma thèse de se dérouler dans une ambiance remarquable.

Je souhaite enfin remercier mes parents et le reste de ma famille qui m'ont soutenue durant toute la durée de ma thèse.
Je dédie ce document à mon beau-père.

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 [51, 130], 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 [101, 102], Eiffel// [40], et Java [76].

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 [76] 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 [33].
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 [32, 164] pour mieux comprendre les bases de cette différentiation).
Les langages de prototypes [16] (comme Pargon [152], Self [42], et Omega [15], 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 [71] 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 [101, 102], C++ [158], Eiffel [117] et naturellement Java [76].

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 [153, 7, 73] et plus particulièrement à [8] 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 [169, 176, 38, 124, 174], [2, 4, 3] pour le modèle d'acteurs, et [123, 125] 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// [9].
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.

```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`.

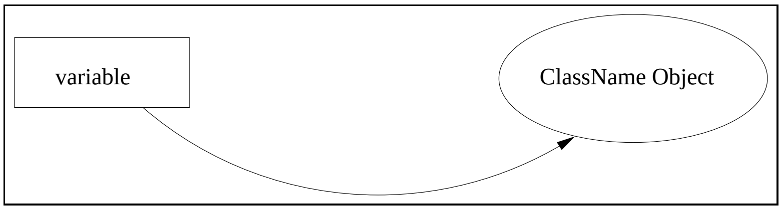

FIG. 2.2 – *Visualisation d'une référence vers un objet.*

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 [107, 106] pour une étude détaillée des différents mécanismes mis en jeu) s'exprime au moyen de *threads* (le lecteur intéressé

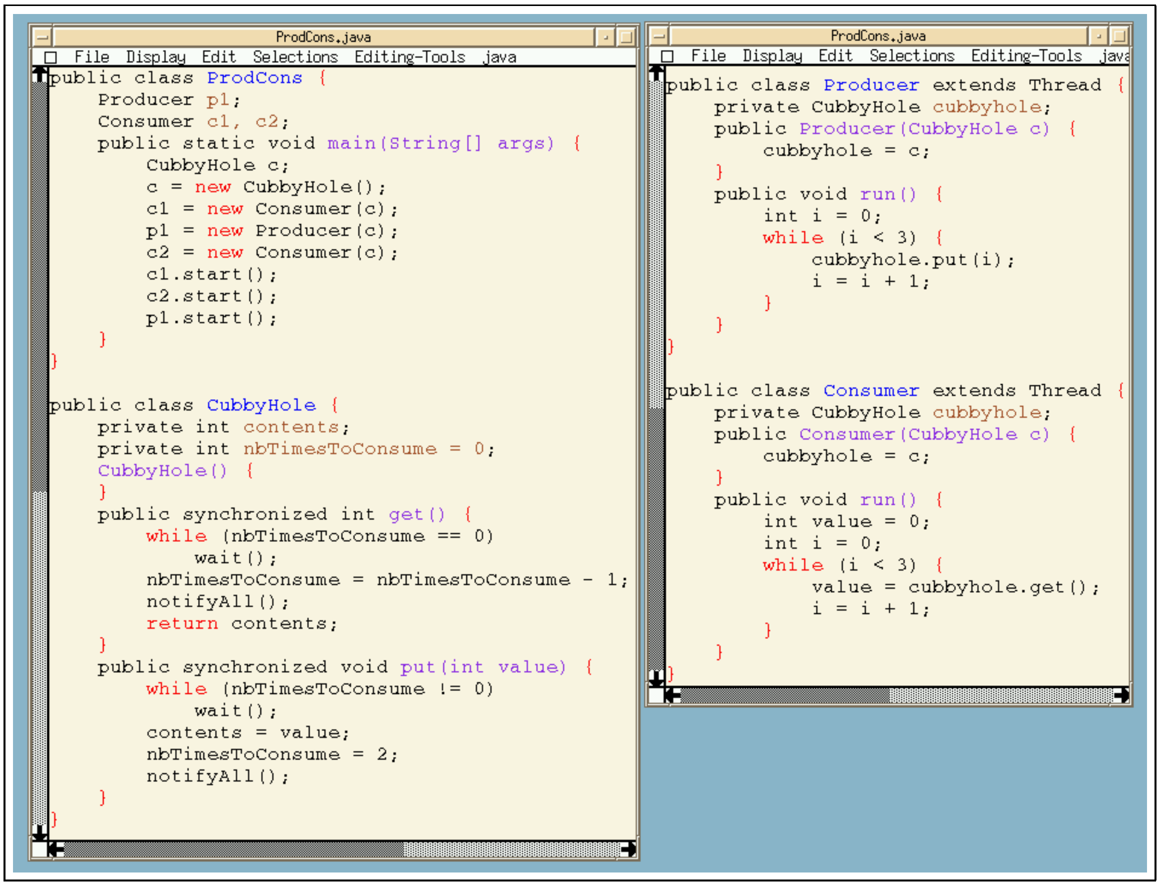

FIG. 2.3 – *Programme Producteur-Consommateur.*

peut se reporter à [131, 90, 60]), 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 [115] 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*.

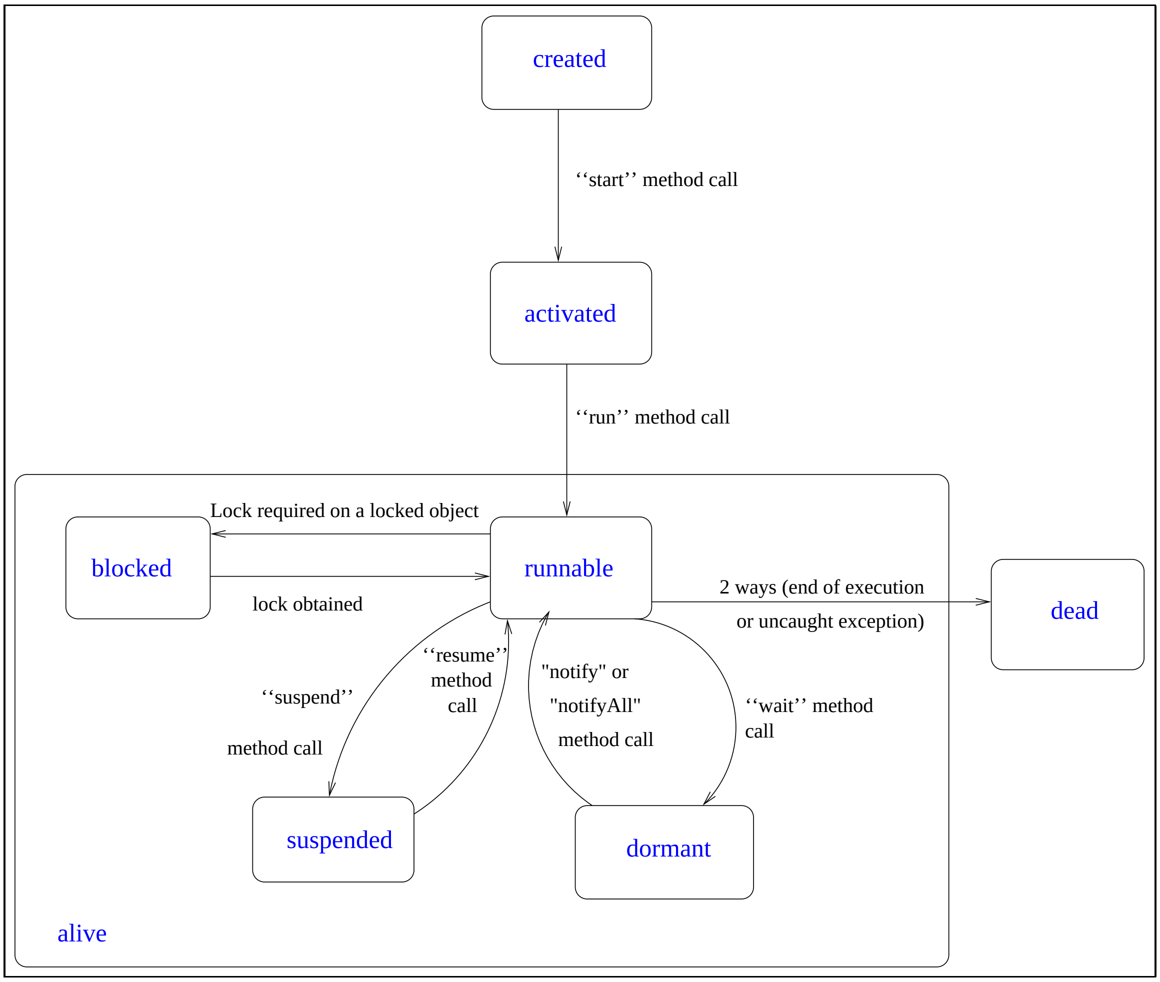

FIG. 2.4 – *Les différents états d'un thread.*

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 [9]).

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 ([76]) 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 [76]. 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, l'utilisation des machines à états abstraits, la sémantique opérationnelle et la sémantique naturelle.

2.2.1.1 Sémantique algébrique

La sémantique algébrique (voir [80, 170, 171] 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 [55] pour un compilateur du langage Pascal, [43] pour un compilateur Lisp, etc.), ou pour prouver des propriétés de programmes (comme dans [148] pour faire des preuves de terminaison de programmes). Le lecteur intéressé pourra se reporter à [79] pour avoir des détails sur l'utilisation de la sémantique algébrique.

2.2.1.2 Sémantique des grammaires attribuées

Les grammaires attribuées (voir [54, 132, 168]) 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 [172] pour le langage Pascal, [65] pour ADA, etc.).

2.2.1.3 Sémantique dénotationnelle

La sémantique dénotationnelle (voir [119, 173, 141] 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 ([146]), de langages parallèles (comme OCCAM [147]) ou de langages de spécification de systèmes concurrents ou distribués (comme LOTOS [37]). Ce type de sémantique a aussi été employé afin de spécifier des langages logiques (comme dans [57] pour définir un langage de programmation logique par contraintes). Watt [173] 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 Machines à états abstraits (ASM)

Les ASM [81, 82, 83, 151] constituent une autre approche concernant la création et l'analyse de systèmes formels. Elles permettent de représenter sous forme mathématique les fondements opérationnels des langages en utilisant des systèmes de transitions entre états abstraits. Ces états sont représentés sous forme de structures algébriques. Les données sont présentées comme des objets abstraits, éléments d'ensembles équipés d'opérations de base et de prédicats. La

notion d'exécution est alors décrite de façon classique par des systèmes de transition.
L'approche des ASM fournit une base uniforme à l'analyse à la fois mathématique et expérimentale des langages de programmation. Les ASM ont été utilisées pour spécifier la sémantique de langages, comme Parlog [26], Prolog [28], et plus récemment Java [85, 29]. De même, elles se révèlent très efficaces dans la vérification des systèmes logiciels complexes (voir [23, 29]).

2.2.1.5 Sémantique opérationnelle

La sémantique opérationnelle (voir [6] 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 [137] (aussi appelée "small-step") et la Sémantique Naturelle de Gilles Kahn [96] (aussi appelée "big-step).
La sémantique opérationnelle a été utilisée pour spécifier formellement des langages à objets (par exemple TOOPLE [36]), des langages à commandes gardées (voir [113]), etc.
Son principal avantage est d'être assez intuitive, car proche de la vision du programmeur.

2.2.1.6 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 [142], 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 [114], en intelligence artificielle [39], pour spécifier l'évaluation paresseuse [105], 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 [175]. 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 [33], 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 [97], 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 [118], pour les spécifications d'affichage ; il permet la transformation d'une représentation abstraite du programme en une représentation concrète.
- Typol [56], 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:`

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

provided getUserAction();
do updateWindow(Env");

`Binary_Expression_Rule:`

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

...

`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 [166, 167] a étudié une possibilité de traduction de spécifications en Sémantique Naturelle dans le système de preuves Coq [59].

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.