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 [76].
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 [66, 122]. Cette approche est basée sur la modélisation du programme Java considéré sous forme d'un graphe des flux.
Dans [66], 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 [122], 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 [75] et CSP [104]) et des langages de programmation synchrones (comme Esterel [14] et Lustre [89]) 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 [157], 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 [45] 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 [47], les auteurs décrivent les sémantiques opérationnelle, dénotationnelle et logique de Triveni et prouvent des théorèmes de correspondance entre ces trois sémantiques.
Dans [46], 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 [134], 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 [91] 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 [50] 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 [87, 88, 34] (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 [87] est bâti autour de l'exemple d'un buffer borné auquel accèdent des producteurs et des consommateurs. L'article [88] propose une application de JPF à la recherche de problèmes de synchronisation au sein d'un serveur de jeu d'échec chinois. L'article [34] présente l'extension de JPF en un environnement de vérification, d'analyse et de tests pour Java.

L'article [53] 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 [61, 62, 63, 64, 127, 128, 126, 162, 163, 138, 29], et qui peut se résumer en *l'exécution de programmes bien typés ne peut pas mal se passer*. Dans [126], 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 [138], Müller et Poetzsch-Heffter présentent une logique de programmation dans un style proche de celui d'Hoare [103] 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.
Dans [29], Börger et al. établissent la sûreté des types pour l'intégralité du langage Java en s'appuyant sur des preuves de certaines propriétés structurelles du langage.

Drew Dean [52], 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 [109] présente brièvement le langage JML (Java Modeling Language) et le projet LOOP. JML [108, 110] 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 [140] 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 [139] 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 sé-

mantique (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

```
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`*.*

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. 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

```
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.*

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`).
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

```
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`.

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 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

```
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`.

```
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.*

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.

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 avons écrit un set appelé `threadPropertyVerification` 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:`

$$
\begin{array}{l}
OThId, NewThStatus, ObjL \vdash objects[object(OThId, ClName, AttrL,\\
Lock, WaitSet, activity(OldThStatus, Clr)).ObjL] \rightarrow objects[object(OThId,\\
ClName, AttrL, Lock, WaitSet, activity(NewThStatus, Clr)).ObjL];\\
\textbf{provided threadPropertyVerification}(OThId, ObjL \vdash\\
OldThStatus, NewThStatus);\\
\texttt{do}\ \text{ENVupdateThreadStatusList}(OThId, OldThStatus, NewThStatus);
\end{array}
$$

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:`

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

```
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:`

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

`NotTheExecutingThread:`

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

$provided\ \mathrm{diff}(ThStatus, executing());$

`NotAThread:`

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

```
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:`

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

`BlockedOnUnlockedObject_WillCompeteToAcquireTheLock:`

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

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:`

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

`DormantOnAnUnlockedObject:`

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

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:`

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

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:`

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

`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:`

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

`DormantOnAnUnlockedObject:`

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

`CreatedThread_StartCall:`

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

```
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. dans [29] où les auteurs effectuent des preuves de théorèmes concernant des invariants portant sur les threads.

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 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 `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

```
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.*

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, 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 *synchro-*

```
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.*

```
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.*

WaitEndRelock. 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 [76] 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.