# Chapitre 4

# Jitan, un Environnement de Visualisation

L'apprentissage d'un langage de programmation est toujours délicat pour les programmeurs néophytes. Dans ce cadre-là, un environnement de programmation permettant de visualiser l'exécution des programmes peut apporter un plus réel au niveau de la compréhension. De nombreux environnements ont donc tout d'abord été développés dans un but pédagogique. D'autre part, lorsque des programmeurs ont à reprendre des programmes et ne disposent pas d'une bonne documentation, un tel environnement peut aussi apporter une aide précieuse en facilitant la compréhension du programme, ainsi que sa mise au point.
Notre environnement ne souhaite en aucun cas se placer en concurrence avec les environnements commerciaux existant pour le langage Java comme Visual J++ de Microsoft [78] ou Kawa [165], mais plutôt introduire des aspects innovants marquant une avancée par rapport à ces derniers.
Java, avec sa double orientation mêlant concepts objets et concurrence, introduit un nouveau degré de difficulté au niveau de la programmation, principalement par la complexité des mécanismes liés à la concurrence. Ces nouveaux aspects augmentent le besoin d'un environnement de programmation, que ce soit pour aider à l'écriture des programmes ou leur mise au point.

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

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

Notre environnement, appelé Jitan, est obtenu directement et gratuitement à partir de notre spécification sémantique de Java. En effet, la sémantique opérationnelle que nous avons définie est exécutable (après une simple compilation) au sein du système Centaur. L'interpréteur associé est ainsi obtenu gratuitement sans aucune manipulation de l'utilisateur. Notre sémantique, de part son aspect formel, introduit une confiance dans la spécification. Cette confiance se retrouve alors au niveau de l'interpréteur. Le programmeur peut en effet être certain du respect de la définition informelle de Sun [76]. Nous avons ensuite enrichi cet interpréteur de vues textuelles et graphiques dynamiques pour obtenir un véritable environnement de programmation et d'exécution. Notre interprète est basé sur une sémantique du source du langage (contrairement aux interpréteurs travaillant à partir du byte-code) et est totalement indépendant par rapport aux machines virtuelles Java existantes.
Jitan constitue une réponse aux attentes des programmeurs, qu'ils soient débutants dans la programmation objet et la concurrence, ou confirmés et souhaitant mettre au point des programmes concurrents. En effet, il fournit une visualisation textuelle et surtout graphique des objets (création, mise à jour, etc.) au cours de l'exécution d'un programme Java et offre des vues synthétiques ou d'abstraction très utiles lorsque le programme crée un grand nombre d'objets.

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

## 4.1 D'une spécification formelle à un environnement de visualisation

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

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

Notre environnement montre la liste des objets grâce à deux moteurs de visualisation, tous deux étant basés sur les structures sémantiques permettant de modéliser la liste d'objets comme cela est présenté dans la Section 3.1.2. Les vues textuelles sont des formatages directs de la liste des objets et des threads avec leurs attributs (noms et valeur) pour la première vue, et les activités des threads dans la seconde, ainsi que les statuts des différents objets et threads et leurs interactions au sein de deux vues de résumé. La vue graphique, qui présente la topologie du graphe d'objets, est visualisée avec le serveur de graphes de Centaur [92].
Les noeuds et les arêtes du graphe sont construits en utilisant un parcours de l'arbre de syntaxe abstraite représentant la liste des objets : pour chaque objet, un noeud le modélisant est créé, et pour chaque attribut ou variable locale dont la valeur est une référence, une arête est créée entre les deux objets impliqués. Nous ne contrôlons pas directement le dessin du graphe. C'est la responsabilité du serveur de graphes d'utiliser une heuristique particulière pour le placement des noeuds et des arêtes. Mais le programmeur peut interactivement modifier le placement des noeuds et des arêtes sur le graphe à l'aide de la souris. Les formes des noeuds et des arêtes peuvent quant à elles être spécifiées au sein d'un fichier de ressources.

Pour afficher un objet sur le graphe, la structure graphique correspondant à cet objet est envoyée au serveur de graphes. De même, des instructions sont envoyées au serveur afin de créer des arêtes et des flèches entre les différents noeuds du graphe. Nous avons défini des structures au niveau du serveur de graphes correspondant à chaque entité du graphe (noeud ou arête). Par exemple, la structure graphique d'un objet est constituée de :
- son nom
- son type
- son statut
- sa liste d'attributs
- sa liste de variables locales
- un booléen qui indique si l'objet est zoomé ou non (afin de lire les valeurs des attributs et des variables locales).

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

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

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

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

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

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

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

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

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

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

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

`java_environment.ty`

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

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

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

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

## 4.2 Vue générale de l'environnement

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

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

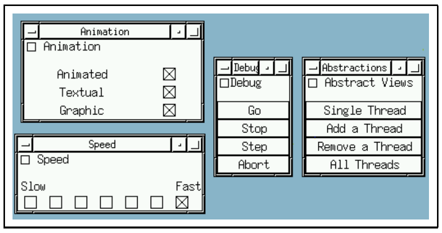
La vue 4.2(b) permet d'étudier les activités des différents threads. La liste des objets (incluant les threads qui sont avant tout des objets) est présentée sous forme textuelle (Figure 4.2(c)) avec pour chaque objet son type et ses attributs. La vue graphique (Figure 4.2(d)) présente la topologie de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

### 4.2.1 Visualisation textuelle de la liste d'objets

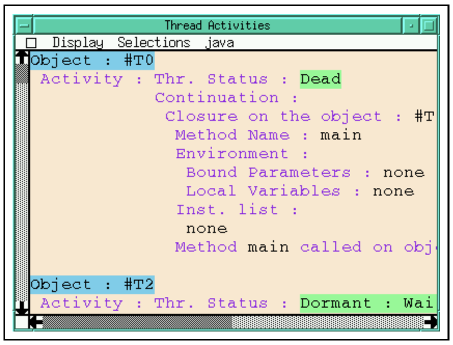
Tout d'abord, il est souvent difficile de vérifier à un point donné de l'exécution d'un programme la valeur d'un champ donné, ou si un objet est actuellement verrouillé. Pour aider le programmeur, notre environnement fournit deux vues (Figures 4.2(b) et 4.2(c)) qui donnent une *présentation textuelle*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

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

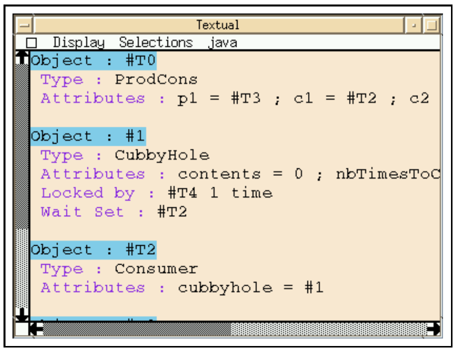
Une façon pour les programmeurs de traquer un problème dans un programme est de savoir ce qui a déjà été exécuté et ce qui ne l'a pas encore été, et ce pour chaque thread.



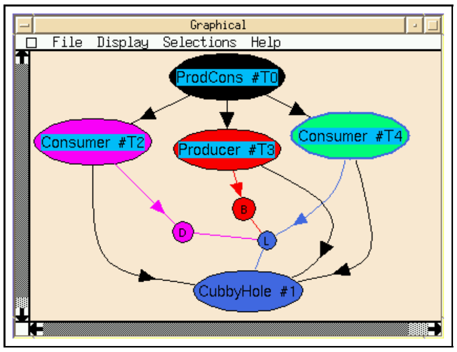

(a) Vues de contrôle.



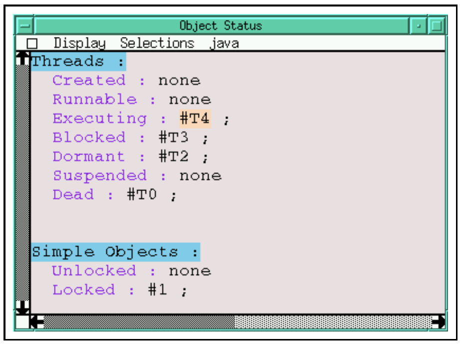

(b) Activités des threads.



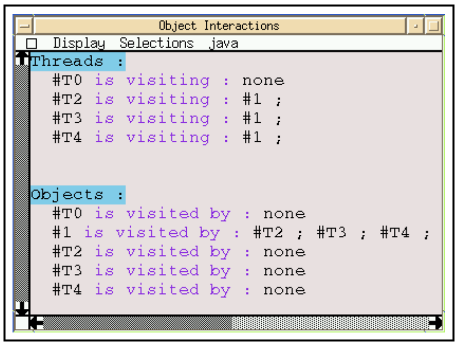

(c) Vue textuelle des objets.




(d) Vue graphique des objets.




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




(f) Interactions entre les objets et les threads.

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

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

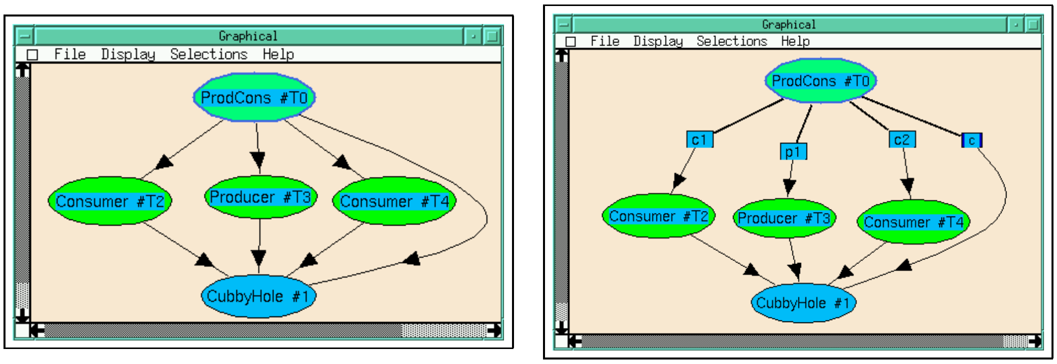
### 4.2.2 Présentation graphique des objets, threads, classes et variables statiques

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

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

#### 4.2.2.1 Représentation des objets

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




(a) Avant le zoom. (b) Après le zoom.

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

#### 4.2.2.2 Représentation des variables locales et des attributs

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

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

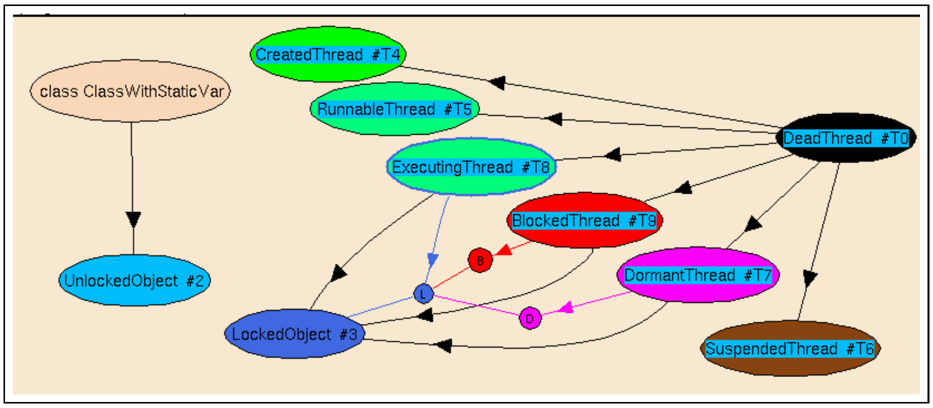



FIG. 4.4 – *Objets, classes, verrous, et statuts des threads.*

#### 4.2.2.3 Représentation des variables statiques

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

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

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

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

### 4.2.3 Vues synthétiques

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

#### 4.2.3.1 Récapitulation des statuts des threads et des objets

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

#### 4.2.3.2 Interactions entre threads

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

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

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

## 4.3 Visualisation de la concurrence et des synchronisations

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

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

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

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

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

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

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

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

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

## 4.4 Exemples d'animation

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

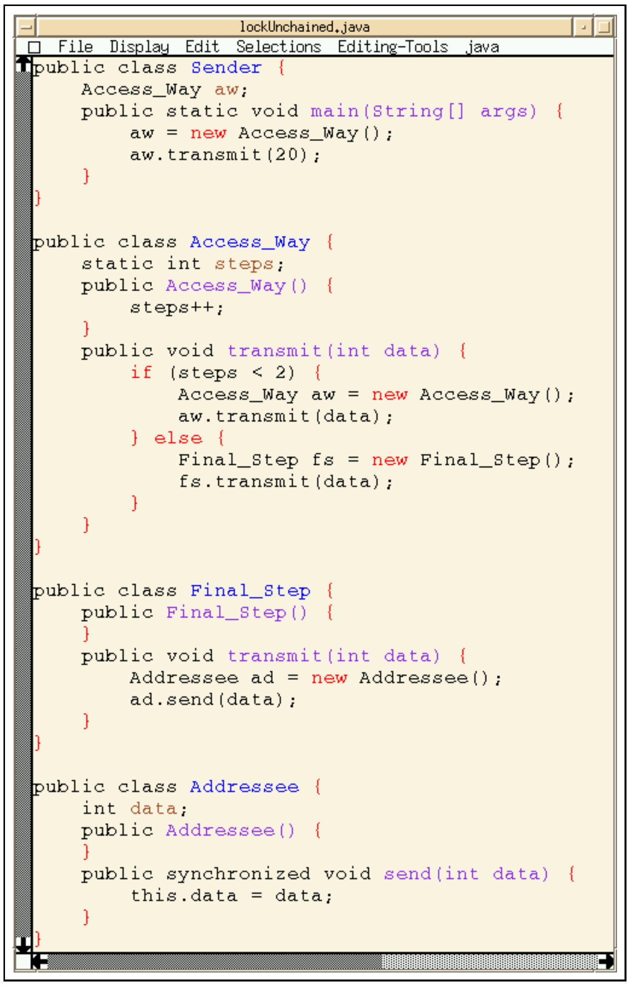



FIG. 4.5 – *Une chaîne de communication.*

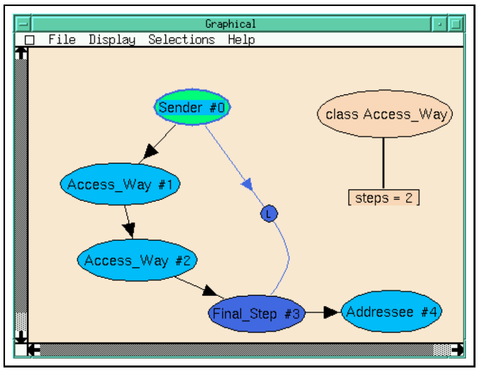



FIG. 4.6 – *Un exemple d'objet verrouillé indirectement.*

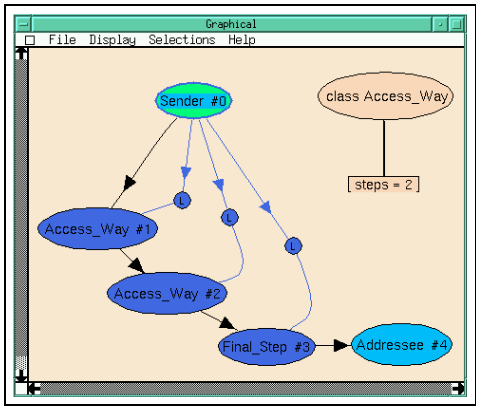



FIG. 4.7 – *Une chaîne d'objets verrouillés.*

### 4.4.1 Animation graphique de l'interprétation d'un programme

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

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

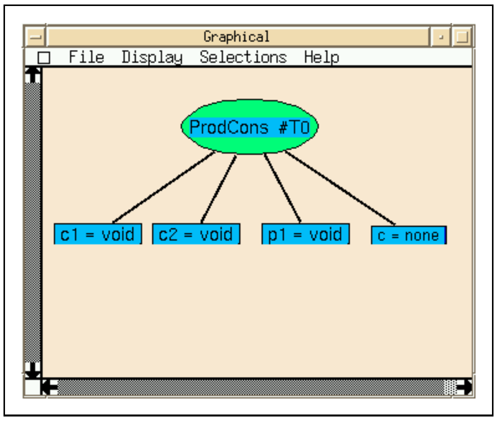



FIG. 4.8 – *Animation de la vue graphique (1).*

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

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

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

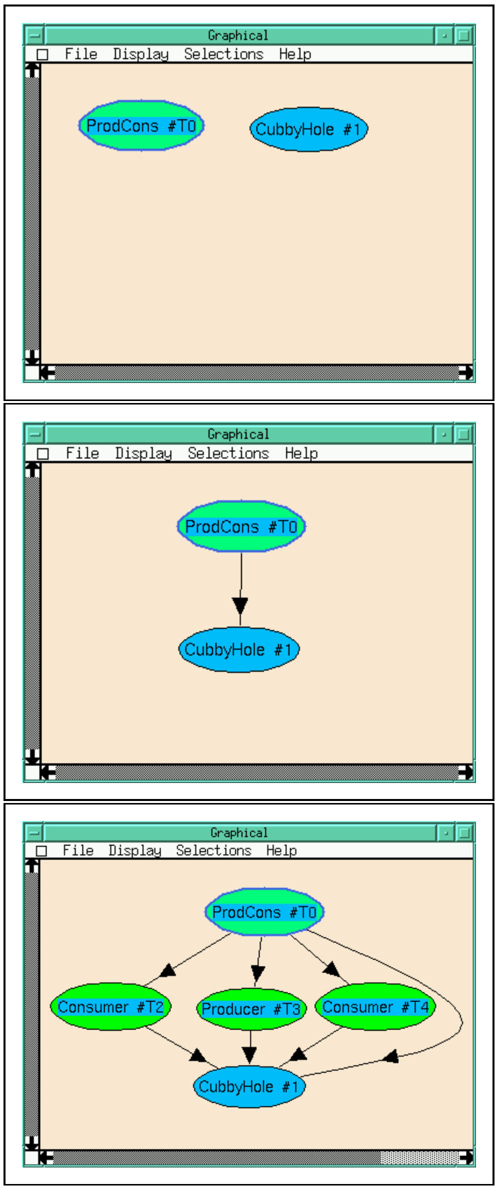



FIG. 4.9 – *Animation de la vue graphique (2).*

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

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

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

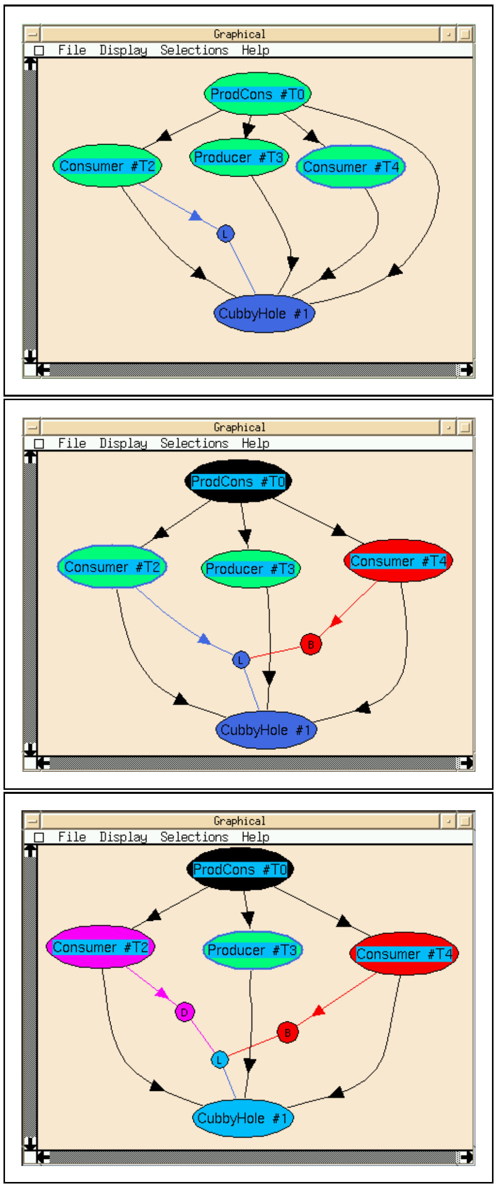



FIG. 4.10 – *Animation de la vue graphique (3).*

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

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

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

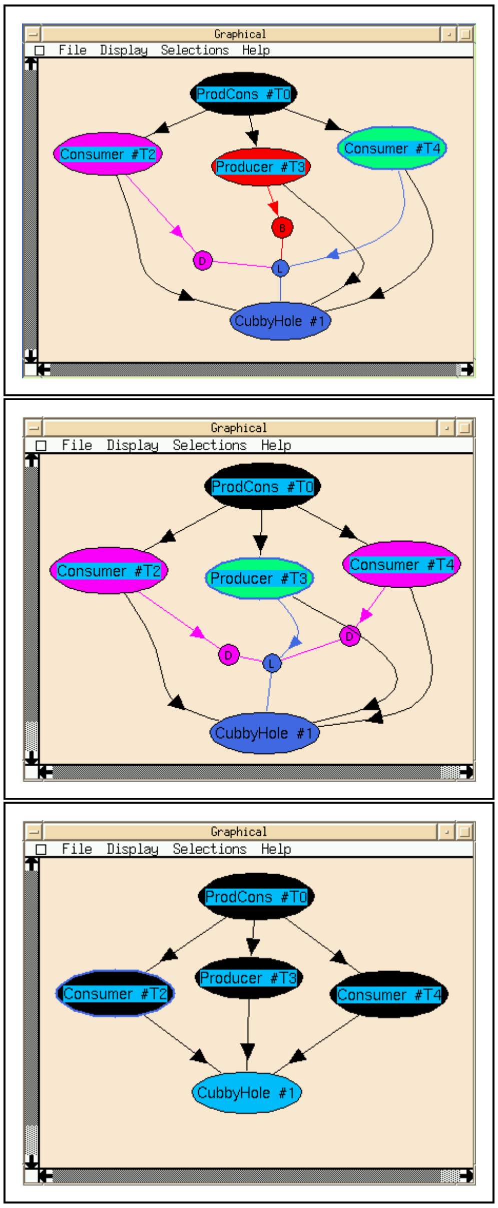



FIG. 4.11 – *Animation de la vue graphique (4).*

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

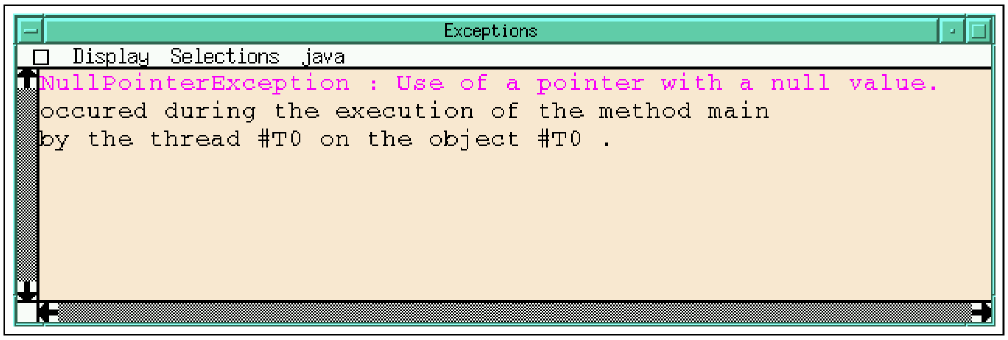


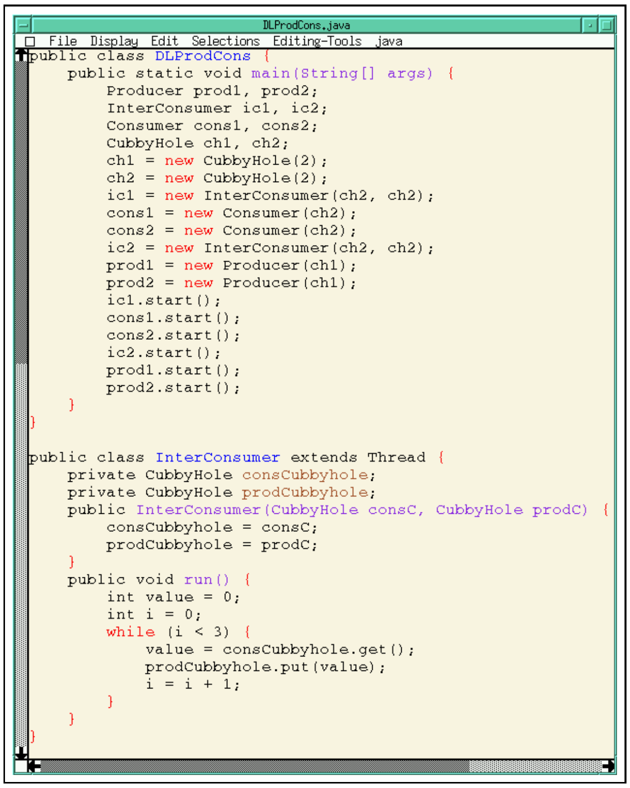

FIG. 4.12 – *Exemple d'exception non rattrapée.*

### 4.4.2 Détection et correction d'un deadlock

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

**Détection**

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




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

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

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

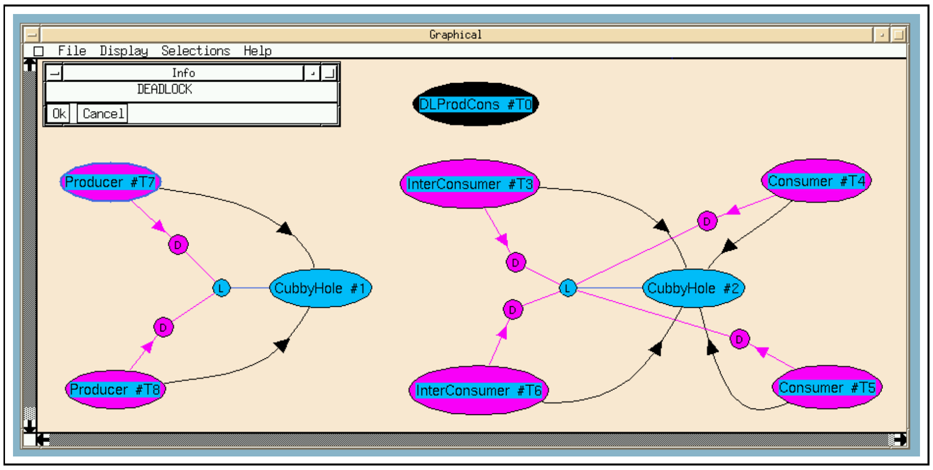



FIG. 4.14 – *Détection du deadlock.*

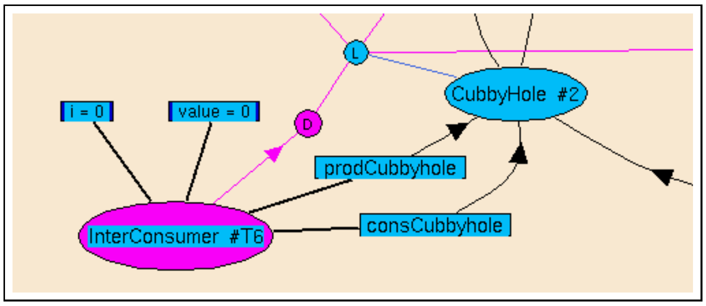



FIG. 4.15 – *Zoom sur la cause du deadlock.*

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

**Correction**

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

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

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

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

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

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

## 4.5 Passage à l'échelle

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

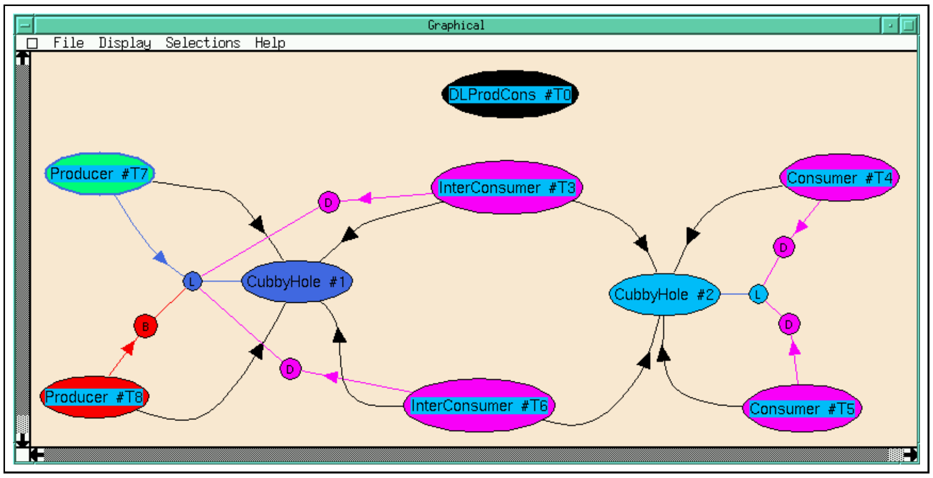



FIG. 4.16 – *Topologie obtenue avec la version corrigée.*

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

### 4.5.1 Etude des différentes possibilités

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

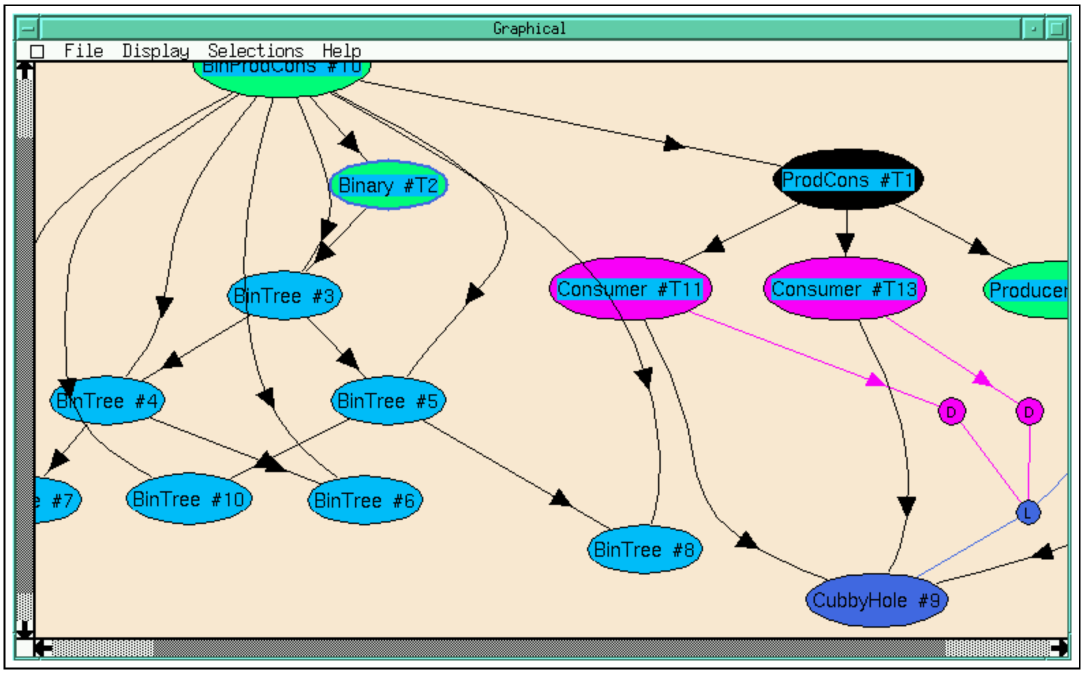



FIG. 4.17 – *Vue globale du graphe d'objets.*

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

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

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

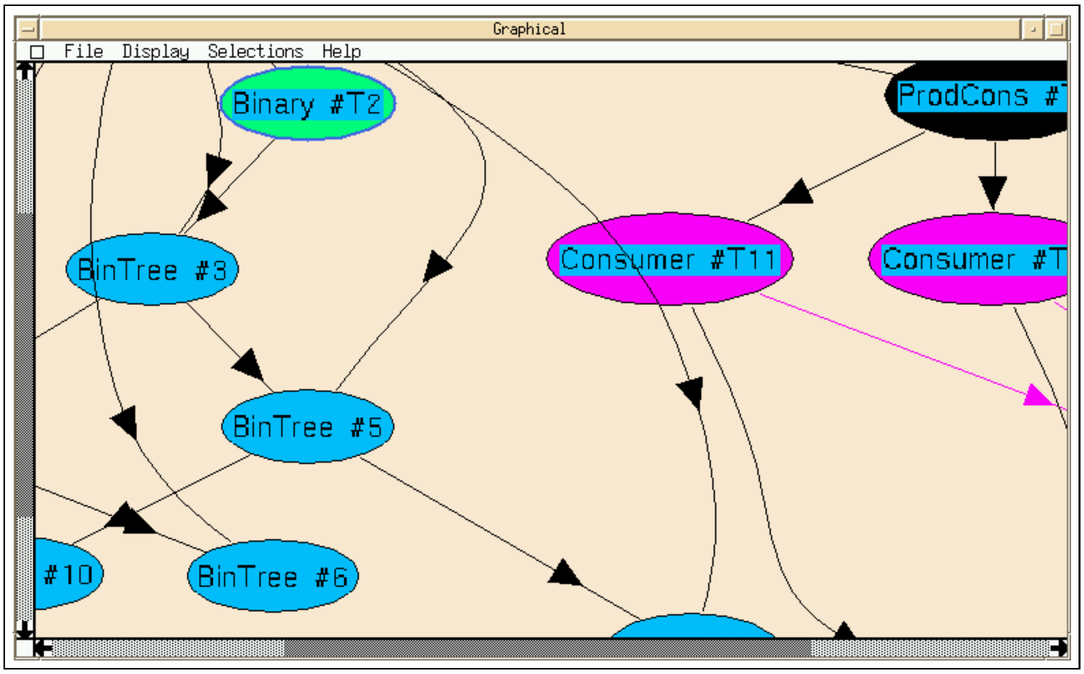



FIG. 4.18 – *Vue du graphe d'objets après l'action Zoom In.*

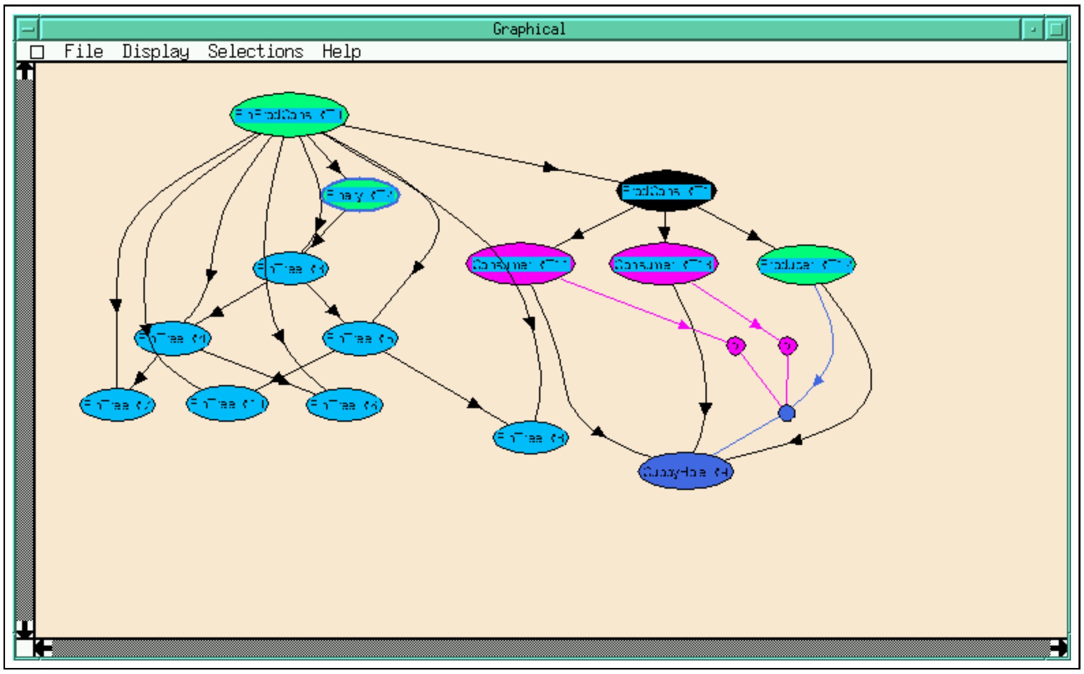



FIG. 4.19 – *Vue du graphe d'objets après l'action Zoom Out.*

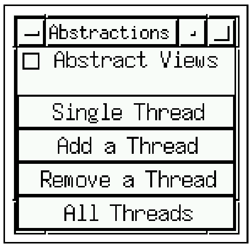



FIG. 4.20 – *Menu des abstractions.*

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

### 4.5.2 Application à Jitan : les abstractions

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

#### 4.5.2.1 Fonctionnalités offertes

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

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

Cette section présente successivement ces quatre possibilités.

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

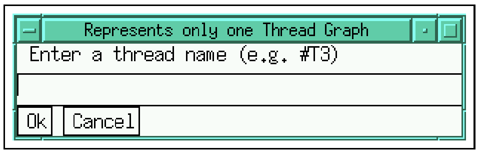



FIG. 4.21 – *Fenêtre de choix du thread dont on veut le sous-graphe.*

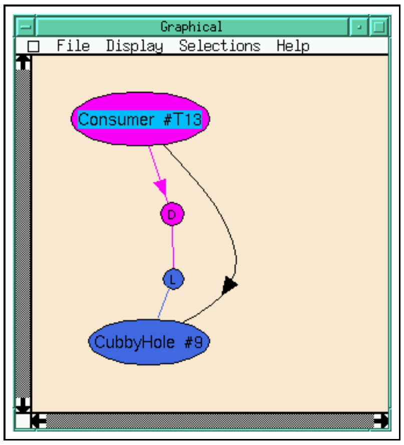



FIG. 4.22 – *Vue du sous-graphe du thread* `#T13`.

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

**Ajout d'un sous-graphe**

Notre environnement permet d'ajouter des sous-graphes à la vue graphique lorsqu'elle visualise déjà un sous-graphe associé à un thread.
Lorsque l'utilisateur clique sur l'option *Add a Thread* de la fenêtre de la Figure 4.20, une fenêtre du même type que celle présentée en Figure 4.21 apparaît. Elle permet à l'utilisateur de choisir le thread dont il veut visualiser le sous-graphe. Ce dernier est alors visualisé dans la même vue graphique.
Si l'on reprend l'exemple ci-dessus, et que l'on demande l'ajout du sous-graphe associé au thread `#T2`, la vue graphique est modifiée et visualise les deux sous-graphes comme cela est présenté dans la Figure 4.23.
Cette vue permet de réaliser que les deux threads `#T2` et `#T13` sont indépendants : aucun ne possède une référence, directe ou non, sur l'autre. On peut d'autre part remarquer de façon directe le pattern d'arbre binaire présenté par le sous-graphe du thread `#T2`, ce qui n'était pas évident sur la vue globale de la Figure 4.17.

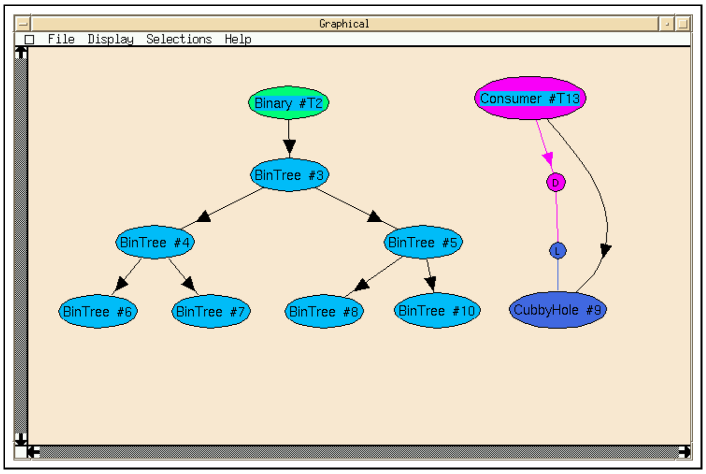



FIG. 4.23 – *Ajout du sous-graphe du thread* #T2.

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

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

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

### 4.5.2.2 Spécification des abstractions

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

`getObjectsForGraphic`

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

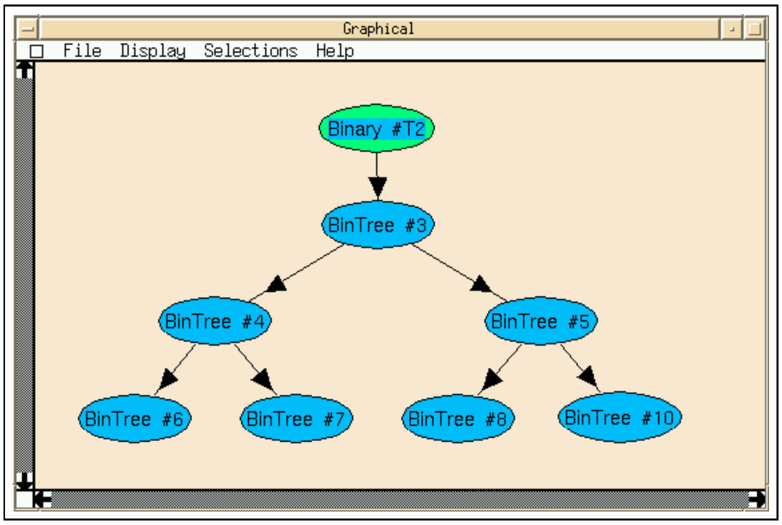



FIG. 4.24 – *Suppression du sous-graphe du thread* #T13.

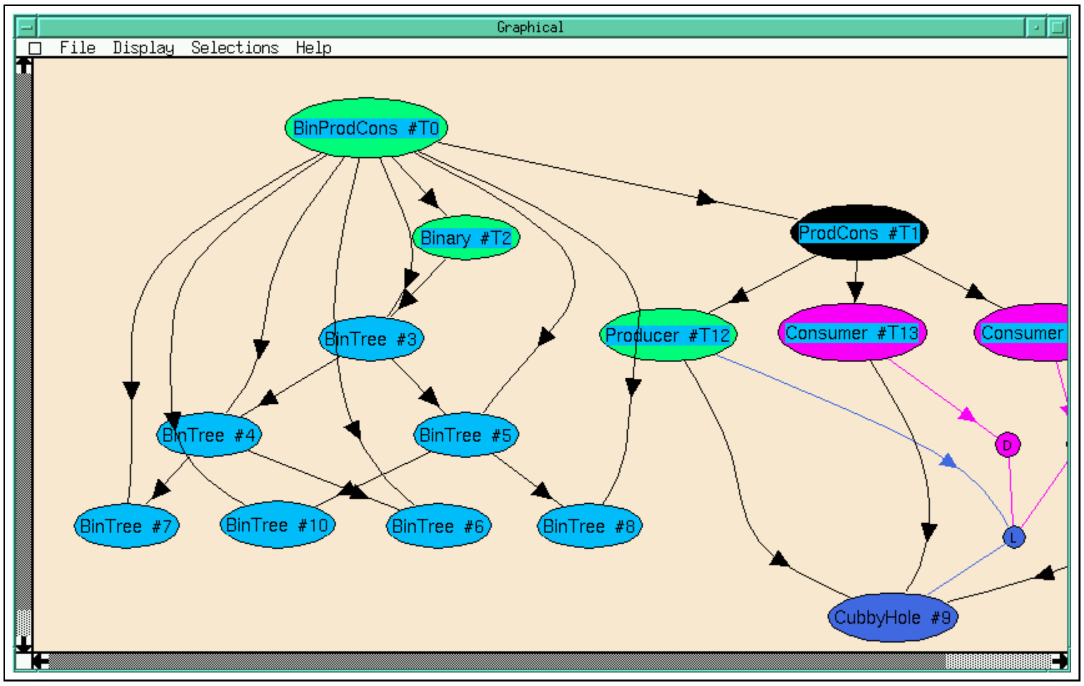



FIG. 4.25 – *Retour à la vue globale.*

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

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

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

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

après abstraction. En effet, les vues textuelles continuent de permettre la visualisation des informations sur tous les objets créés, même lorsque l'utilisateur a demandé une abstraction de la vue graphique. Grâce aux vues textuelles, l'utilisateur peut ainsi disposer d'une vue globale de l'exécution, même si tous les objets n'apparaissent pas sur le graphique.

Le set *getObjectsForGraphic* traite les différentes possibilités offertes par les abstractions :

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

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

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

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

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

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

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

– Le cas, spécifié par la règle de la Figure 4.30, où l'utilisateur a demandé de supprimer le sous-graphe du thread *#T OThNumber* à la vue graphique précédente

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

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

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

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

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

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

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

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

## 4.6 Visualisation du masquage des attributs : des pistes

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

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

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

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

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

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

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

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

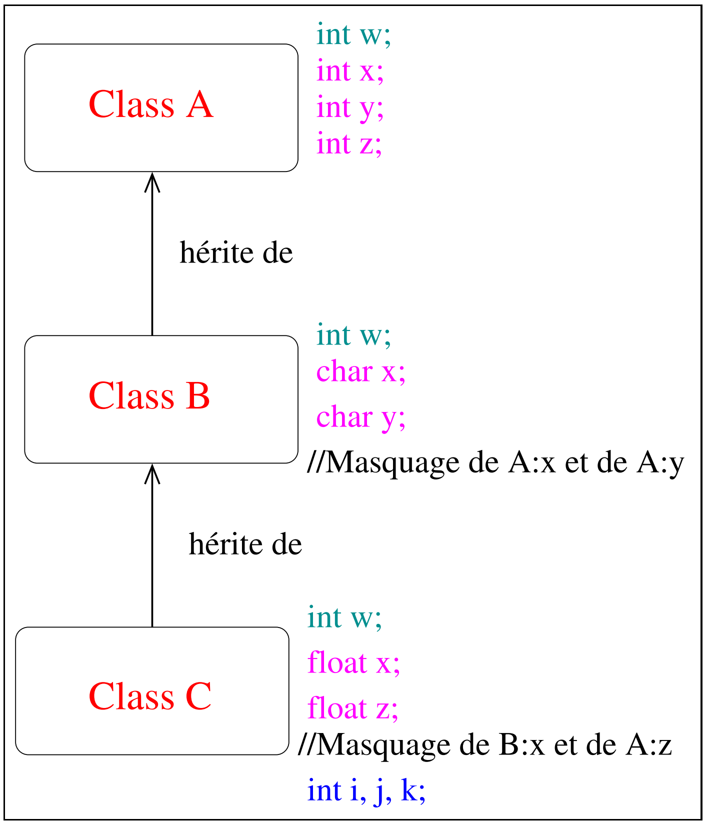



FIG. 4.32 – *Exemple de masquage des attributs.*

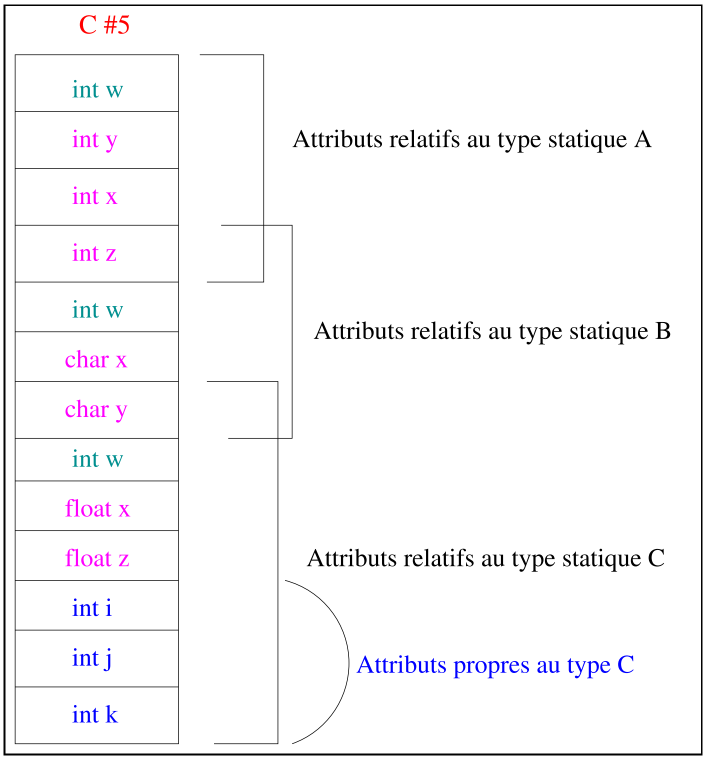



FIG. 4.33 – *Première solution pour la visualisation du masquage des attributs.*

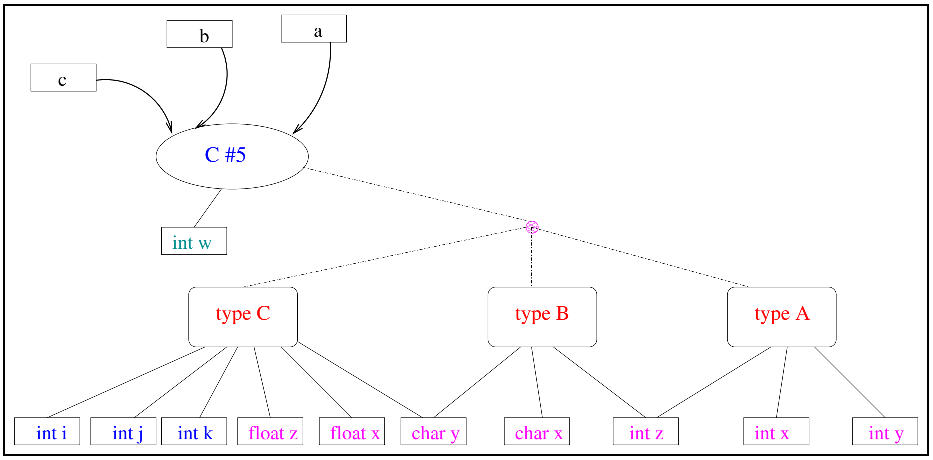



FIG. 4.34 – *Deuxième solution pour la visualisation du masquage des attributs.*

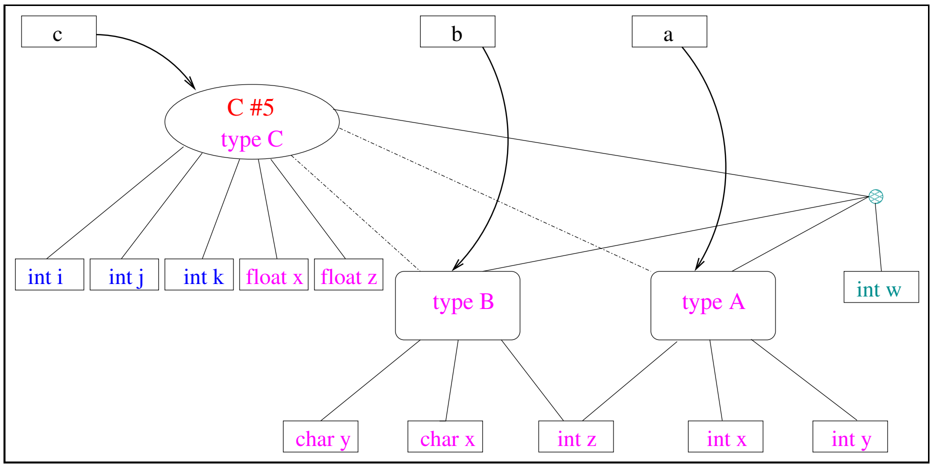



FIG. 4.35 – *Troisième solution pour la visualisation du masquage des attributs.*

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

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

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

## 4.7 Travaux connexes et discussion

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

### 4.7.1 Travaux connexes

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

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

férents domaines tels que la conception de logiciels, l'évaluation des performances, et l'apprentissage de la programmation.
On peut citer comme systèmes représentatifs Zeus [35] pour l'animation d'algorithmes, Incense [121] pour l'affichage des structures de données, Pastis [120] pour l'animation de programmes, Polka [154] pour l'animation de programmes sur des architectures parallèles, et Pavane [145] pour la visualisation de l'exécution de programmes concurrents (voir [99] pour une synthèse sur la visualisation des programmes parallèles).

#### 4.7.1.1 Visualisation de la structure des programmes

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

CodeCenter [41] fournit des browsers graphiques affichant une trace des appels de fonctions et de l'utilisation des variables afin de détecter des erreurs statiques, et d'examiner et contrôler les différents modules et bibliothèques au sein d'un projet. CIA++ [77] bâtit une base de données d'informations sur des programmes C++, qui est ensuite utilisée pour afficher différentes vues de la structure des programmes. GraphLog [49], qui est un outil générique muni d'un langage de requêtes, permet de visualiser les différentes structures d'un logiciel et d'effectuer des requêtes sur celui-ci. Object Explorer [12] de Kent Beck est quant à lui dédié à la visualisation de programmes SmallTalk.

Felleisen et al., dans [70, 69], proposent un environnement graphique de développement de programmes Scheme nommé DrScheme. Cet environnement permet de visualiser le source du programme et fournit un outil interactif d'analyse statique appelé MrSpidey. Cet outil détecte des violations potentielles de sécurité avant l'exécution du programme, et explique le problème potentiel en affichant un graphe de flots de valeurs au sein du texte du programme.

#### 4.7.1.2 Visualisation de l'exécution des programmes

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

ObjView [74], qui est un système de conception de tableaux de bord électroniques basé sur un modèle écrit en C++, offre une interface avec à la fois une vue réaliste des objets du problème et des objets C++, avec une bonne visualisation des instances, des données membres et des fonctions. CodeCenter semble présenter les mêmes possibilités. Haarslev et Moeller, dans [86], proposent une structure représentant à la fois la hiérarchie des classes du programme, et les objets créés en utilisant l'architecture de niveau méta CLOS [133] ; une association explicite entre les objets visualisés et les objets de l'application est alors nécessaire. Object Visualizer [135, 136] et HotWire [100] sont quant à eux basés sur des mécanismes d'instrumentation de programmes et offrent les mêmes possibilités.

Plus récemment est apparu le besoin de traiter de grandes quantités d'objets, ce qui pose des problèmes, notamment au niveau graphique comme nous venons de le voir (tous les objets ne rentrent pas dans la vue graphique, ou alors à une trop petite échelle pour être utilisables). Kent a proposé une notation visuelle des invariants (à la fois au niveau des classes et au niveau des instances) pour des modèles à objets [98]. Lencevicius, Hölzle, et Singh développent le débuggage à base de requêtes qui permet d'après eux des manipulations efficaces d'une grande quantité d'objets [112]. Jerding, Stasko, et Ball se concentrent sur la compréhension des interactions entre classes et objets durant l'exécution des programmes [95].

#### 4.7.1.3 Environnements de visualisation Java

En ce qui concerne le langage Java, plusieurs systèmes commerciaux d'aide au développement de programmes existent [1] : Visual J++ de Microsoft [78], Kawa [165], JavaWorkshop [111], Javelin [156], VisualAge pour Java d'IBM [93], Metrowerks CodeWarrior [116], Assure [67], Asymetrics SuperCede Java [161], et Visual Threads de Compaq [48] qui propose un graphique décrivant de façon continue l'état des threads du programme, et la fréquence des évènements liés aux thread. Le lecteur intéressé peut se reporter à [68] pour une présentation assez exhaustive des différents outils du commerce, et [58] pour une revue des différents outils de visualisation Java.

### 4.7.2 Discussion

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

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

1. JDB [159] qui n'est pas un environnement de développement à proprement parler, mais plutôt un débugger en ligne, et les profileurs tels JProbe [129] basés sur une étude "off-line" du graphe des appels et de l'utilisation de la mémoire sont omis dans cette liste.

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

Le deuxième aspect est lié au débuggage interactif : aucune compilation du programme n'est nécessaire avec notre outil, à l'opposé de la re-compilation incrémentale nécessaire habituellement après toute modification ou addition d'un point d'arrêt. Avec Jitan, quand un programme est modifié (après arrêt de l'exécution), il suffit de relancer l'exécution. L'exception est SuperCede Java qui permet de modifier les programmes en cours d'exécution, sans pour autant recommencer celle-ci entièrement.
Cette possibilité offerte par Jitan provient du fait que notre environnement n'est pas basé sur l'exécution du byte-code instrumenté, ou n'utilise pas une machine virtuelle particulière (à la différence de la plupart des outils du commerce), mais utilise directement une sémantique opérationnelle, instrumentée avec des notifications (comme cela est expliqué dans le paragraphe 4.1). Cela ne demande donc aucune instrumentation de la part de l'utilisateur à l'inverse de l'outil JDI [160] qui est une bibliothèque Java qui peut seulement être utilisée comme base pour la construction d'un environnement de débuggage.

Enfin notre environnement repose directement sur une définition formelle du langage Java. L'outil le plus proche de Jitan de ce point de vue là est celui développé par Börger et al. en utilisant ASMGofer [29]. Les spécifications ASM de Java sont exécutables au sein de ASMGofer, une extension de TkGofer [44]. Cet outil permet entre autres de visualiser dans une même fenêtre sous forme textuelle le source du programme Java, l'état de la pile et du tas et présente une interface de gestion de l'exécution (points d'arrêt, exécution pas à pas, etc. Il se démarque ainsi de Jitan en fournissant des informations statiques sur le programme : liste de ses classes, listes des méthodes d'une classe, etc. Cependant, même s'il fournit de façon dynamique des informations sur l'exécution des programmes, il ne semble pas proposer de visualisation graphique de la topologie des objets et des mécanismes de synchronisation, ce qui constitue deux grandes orientations suivies dans Jitan.

## 4.8 Conclusion

Ce chapitre nous a permis de présenter notre environnement de visualisation de l'exécution de programmes Java basé sur la sémantique formelle opérationnelle de Java

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

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