Visualizing Java Objects, Threads, and Locks
research paper

Isabelle Attali, Denis Caromel, Marjorie Russo
INRIA Sophia Antipolis,
CNRS - I3S - Univ. Nice Sophia Antipolis,
BP 93, 06902 Sophia Antipolis Cedex - France
tel: 33 4 92 38 79 10
fax: 33 4 92 38 76 33
First.Last@sophia.inria.fr

relevant subject areas:
programming environments, visualization

Abstract

How to visualize the various entities that participate in the execution of a Java program ?

Both object-oriented and concurrent, the Java model features inter-related aspects that are critical for the understanding of an application: objects, static variables, threads, locks, etc. This article raises a few questions and observations regarding the visualization of such entities, and propose suitable solutions that we tested within a graphical environment for Java.

This environment allows to visualize objects and thread activities at execution in both a textual and a graphical manner, e. g. the topology of the object graph (trees, DAGs, cycles), thread activities and status, locks, and synchronizations. Synthetic views give relevant information on interaction between objects and threads.

The principles and solutions used here for graphical visualization would be applicable to other Java environments, as well as to other languages with object-oriented and concurrency features.

1 Introduction

How to visualize the various entities that participate in the execution of a Java program ?

Both object-oriented and concurrent, the Java model features interrelated aspects that are critical for the understanding of an application: objects, static variables, threads, locks, etc. This article raises a few questions and observations regarding the visualization of such entities, and propose suitable solutions that we tested within a graphical environment for Java.

The topology of instance variables within an object-oriented system can be rather complex. Is it mainly trees, are there DAGs, are there cycles, where ? Those are important questions for the understanding of an application. How can a programming environment provide that information to users in a simple and effective manner, without requiring them to compare reference values within a text-only debugger ? Static variables raise special concerns as they can be accessed from objects without an explicit reference to them.

When it comes to multi-threading and concurrent accesses, that information becomes even more crucial. How can we effectively visualize the objects shared by threads ? Is there any recursive call, either directly or through cycles ? Do several threads access a single object at the same time ?

Finally, crucial to any concurrent model is the ability to build critical sections. The powerful Java locking mechanism allows quite easily to specify synchronized blocks, resulting in potentially numerous locked objects within even a simple application. How can we visualize the threads and indirectly the objects waiting for a given lock, held on a specific object by a specific thread ? Can we find out easily about a thread successively locking several objects within a data structure (e.g a list, or a tree), resulting in a chain of locks ? How to visualize that a given object has been locked several times by the same thread ?

Not only the threads and objects state at a given time, but the chains of events are critical for the understanding of the concurrent features. How can we follow the threads turning away from the dormant state after beeing notifyied by some other thread ? Is it possible to get an animated view of such successive events ?

The environment proposed in this paper provides solutions to those problems. It allows to visualize objects and thread activities at execution in both a textual and a graphical manner, e.g. the topology of the object graph (trees, DAGs, cycles), thread activities and status, locks, and synchronizations. Synthetic views give relevant information on status of objects and interaction between objects and threads. Based on an executable operational semantics, no instrumentation is required from the user. Notifications from the semantics to the visualization engines (textual and graphical) make it possible to show, during execution, the changes in the structures.

The principles and solutions used here for graphical visualization of objects, threads, and locks would be applicable to other Java environments, as well as to other languages with object-oriented and concurrent features.

Section 2 gives a short introduction to the Java model characteristics relevant to the environment discussed in the paper, especially features related to threads, concurrency, and locks. Section 3 analyses the visualization issues, presents the solutions that we propose, and details the graphical environment that implements them. We successively cover objects and threads visualization in both textual and graphical forms, synthetic views for thread status and locks, synchronizations and animation[1]. The environment architecture and techniques used to achieve such a graphical and interactive environment are presented at Section 4. The paper terminates with related work and conclusions.

2 Object, Concurrency, and the Java Model

Java [11] is a concurrent class-based object-oriented programming language, that includes quite classical concurrent, multi-thread, monitor-like locks and synchronization features.

We present in this section the main features of the model that are directly relevant for this article, and use a classical producer-consumer program [5] as an illustration.

2.1 Java classes and static variables

A class is defined by its name, its static initializers, its constructor list, and its members (attributes which are also called *fields* and methods). Fields can

[1] Color figures serve as illustrations throughout this article; referees may want to access an electronic colored version via the World Wild Web at the address: `http://www.inria.fr/croap/personnel/ia/oopsla_paper.ps`.

be instance variables (which are incarnated for each instance of the class) or class variables. Class variables (*static* variables) are incarnated once and so are common to all objects of that type.

2.2 Objects and threads

The execution of a Java program creates objects, modifies them and updates object references. The program behavior during execution is therefore described by the evolution of its object list. So the main entities of a Java program at execution are *Objects*. Objects are mainly characterized by their type name, and their field values, and can be identified by a single identifier.

We can distinguish particular objects named *Threads*. In Java, threads are objects whose class derives from a particular class named *Thread* (with the clause *extends Thread*). Threads have an *activity*, that is to say a stack of successive calls (a *continuation*) and a *status* (detailed in Section 2.3).

2.3 Synchronization

Java is a concurrent programming language. This implies that several entities (here threads) can execute themselves simultaneously (on a multi-processor architecture) or with interleaving (on a mono-processor computer).
Java provides a mechanism which enables programmers to get a deterministic behavior for a given Java program by "critical sections", using monitor-like synchronizations. A part of a program which is enclosed in a *synchronized* statement or method will be executed by only one thread at once. It relies on locks: there is exactly one lock associated with each Java object. Threads have a status which takes 7 possible values: created, runnable, executing, blocked, dormant, dead. When a thread is created by another thread, its status is *created*. When its *start* method is called, the created thread becomes *runnable*. At this step, the thread is able to execute itself. A *runnable* thread becomes *executing* each time it is actually executing its continuation.

When a thread Thr wants to execute a synchronized block on an object Obj, it has to lock this object. There are three possibilities:

- Obj is not locked: Thr locks Obj which becomes *locked*, and enters into the critical section. At the end of the critical section, Thr will release the lock on Obj.

- Obj is already locked by another thread: Thr becomes *blocked* (or disabled for thread scheduling purpose) and will not resume execution before the current lock on Obj is released.

- Obj is already locked by Thr: the number of locks is incremented and Thr enters into a new critical section. At the end of that synchronized block, the number of locks on Obj is decremented.

A *dormant* thread is a thread which has acquired a locked on an object and has executed a *wait* primitive. It has so released the lock on that object and has been placed in its *wait set*: when another thread will call the *notify* or the *notifyAll* primitive on this object, the thread will resume its execution, its first activity will be to re-acquire the lock on the object. A *suspended* thread is a thread on which the *suspend* method was called by another thread. It remains in this state (disabled for thread scheduling purposes) until the *resume* method is called on it by another thread. Each time a thread is no longer dormant, blocked or suspended, it is re-enabled for thread scheduling (runnable status). Finally, a *dead* thread is a thread which has completed normally (without error) its continuation.

2.4 Example

As an illustration throughout the paper we use a well-known Java program: the producer-consumer program (see Figure 1). In this program, there are two consumers and one producer of a data (here an integer). The producer must put successive data in a cubbyhole which are then read by two consumers. The cubbyhole is a buffer, a shared object, synchronized. Consumers can not read a data before the producer puts its first data in the cubbyhole and the producer must not put a new data if the previous one has not been read by the two consumers. This example introduces a lot of Java concepts: object and thread creation, concurrent execution, synchronized methods, notifications.

3 Visualization

At run-time, the main entities are objects. Visualization of objects and their characteristics during program execution is an important feature for the programmer.
In this section we present the different issues related to visualization of an object-oriented and concurrent application and the solutions provided by our environment.

prod_cons_time.java

File Display Edit Selections Editing-Tools java

```
public class  prod_cons_time {

   Producer  p1;
   Consumer  c1;
   Consum_T  c2;
   /*creation and start of the three threads*/
   public static void  main(String[] args) {
      ...
   }

}
public class  Producer  extends Thread {

   private CubbyHole  cubbyhole;
   public  Producer(CubbyHole c)  {
      cubbyhole = c;
   }
   public void  run()  {
      int i;
      i = 0;
      while (i < 3)  {
         cubbyhole.put(i);
         i = i + 1;
      }
   }

}
public class  CubbyHole {

   private int  contents;
   private int  available = 0;
   CubbyHole()  {
   }
   public synchronized void  put(int value)  {
      while (available > 0)  {
         wait();
      }
      contents = value;
      available = 2;
      notifyAll();
   }
   /*"get" method to get the value.*/
   public synchronized int  get()  {
      ...
   }

}
public class  Consumer  extends Thread {

   private CubbyHole  cubbyhole;
   static Addressee  add;
   public  Consumer(CubbyHole c)  {
      cubbyhole = c;
      add = new Addressee();
   }
```

Figure 1: Producer-Consumer Example Program.

3.1 Object List: Textual Visualization

First of all, it is often a problem to know, during execution, the value of a given field, or if an object is currently locked. To help the programmer, our environment provides a window (Figure 2) which gives a *textual presentation* of the current object list during program execution.

A programmer can know at first sight if the considered object is a thread or not: a thread has a *status* which appears in green and objects have no activity field. He can also visualize each object type (for example "CubbyHole" for the object 1), field list (two fields for the object 1: "contents" and "available", which both have the value 0) and if the object is locked or not, and what object locks it.

The activity of each thread is represented through a continuation and an execution environment (method name, parameter and local variable list).

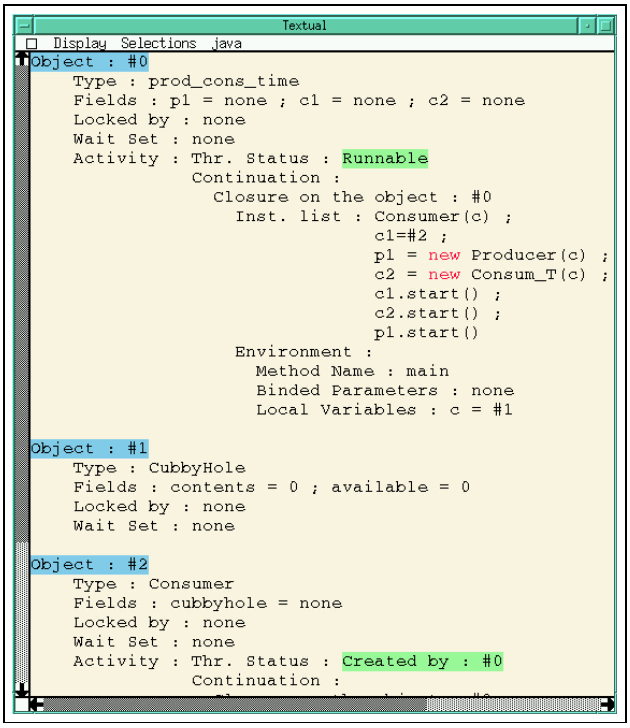

Figure 2: Textual Objects and Continuations at Execution.

3.2 Objects, Threads, Classes and Static Variables: Graphical Presentation

Another important problem encountered by programmers which use object-oriented languages is related with the evolution of the object topology. With a textual presentation, it could be really difficult to discover which objects are connected, directly or indirectly, by a reference. So it is very useful to have a *graphical presentation* to depict references which exist between objects.

The Figure 3 shows our graphical presentation which enables the programmer to easily envision the topology and its evolution during program execution. It represents the object list (left part of Figure 3) using a graph where nodes represent objects and arrows are the references between objects. Objects and threads are distinguished by the presence of a rectangle around thread names. This presentation highlights references between objects, the object type and kind (thread or not). Threads *status* are made visible with different color codes (see Section 3.3).
Visualization of static variables requires a specific treatment. We use a solution that displays classes which have static variables, as for instance on the right hand side of Figure 3. The objects which are referenced by such a class are accessible to every object of this type. We don't show a link between an object and its class (but it would be a possibility, for instance on request). This keeps the topology simpler and more synthetic, but the programmer can, using this technique, easily understand where such static references are.

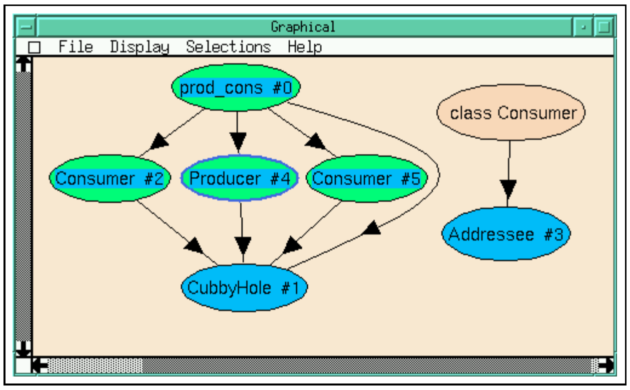

Figure 3: Objects, Threads, Classes and Static Variables.

For example, while the thread 2 does not have a reference on the object 3, we can see that its type (Consumer) gives it access to object 3 through its static variable.

A zooming process makes it possible to examine, on request, attribute values of a given object node. Clicking on a node expands its field values (see Figure 4), clicking again make it return to default.

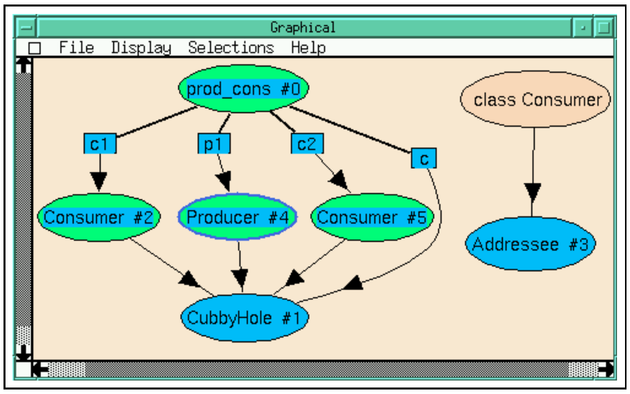

Figure 4: Looking at attributes.

For readability, no other information (such as thread activity) is available in the graphical presentation.

In order to debug a program or to understand its behavior it is very important to visualize DAGs (Directed Acyclic Graphs) and cycles between objects. Indeed such information can enable the programmer to detect possible deadlocks and to avoid data access problems by using synchronization mechanisms (see Section 3.4). The topology visualization also points out non reachable objects or groups of objects (no reference from an active thread to those objects), which are subject to garbage collection.

On the example shown on Figure 5, we can see a cycle between the objects 8, 16 and 17. The reference possessed by the object 10 on itself means that a field of the considered object currently references itself.

3.3 Synthetic Views

In current and actual systems, there are many threads, objects, and as such, many complex interactions between them. A solution used to tackle complexity relies on synthetic views of objects and threads. Understanding complex

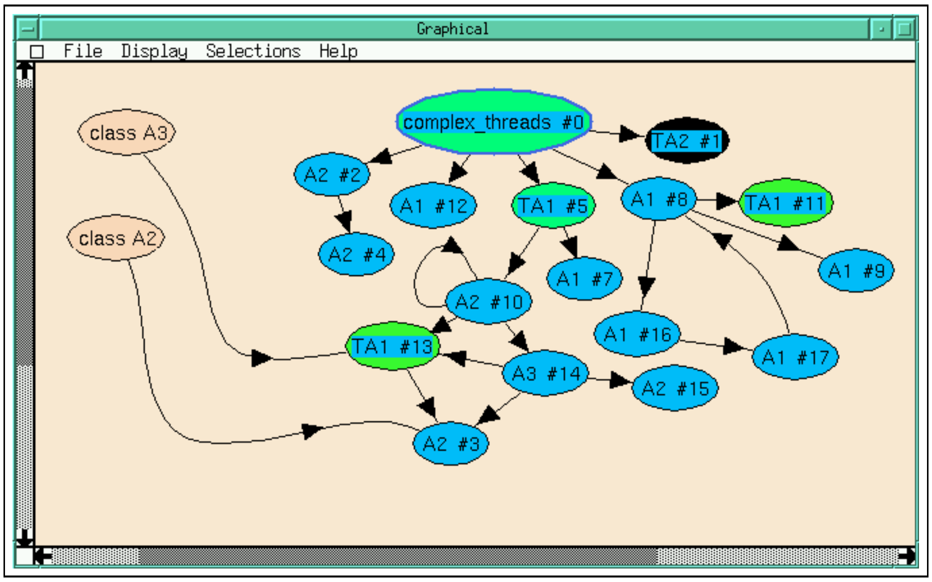

Figure 5: Object Topology.

concurrent behaviors (deadlocks and livelocks, needs for synchronizations) greatly benefits from those views.

First, Figure 6 shows the different objects and threads classified by their status. It can be very useful for the programmer to easily realize (in a synthetic way) that no thread can proceed execution, or that there is at least one thread which is not *dead*. We can directly see that thread 4 is *runnable*, thread 5 is *blocked*, that the thread 2 is *dormant*, and that the thread 0 is *dead*. It also shows which objects are *locked* (here the object 1).

A succession of method calls on a same object or a cycle in the method calls may be an important information in order to understand or debug a program. How can we visualize such behavior ? Figure 7 shows a window which synthetically presents the thread stacks and the object activations. On top of the window, the programmer can see on which object each thread is currently working and through which objects it has got this reference (and the chain of dynamic calls).

For example thread 2 is executing a method on object 3 and within this method, the thread executes another method call on object 4 then on object 1 and on the object 3 again. This represents an indirect recursive call.

A crucial problem when writing concurrent applications is to know where

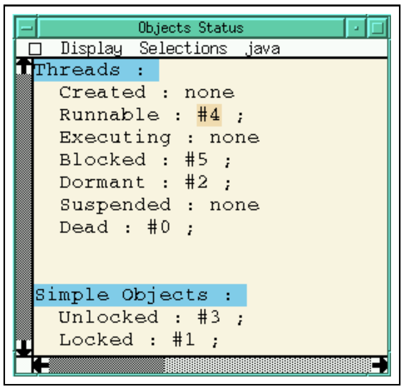

Figure 6: Object and Thread Status.

synchronizations have to be placed. Very often methods of an object which can be accessed by several threads have to be declared as critical sections. It is therefore very useful to figure out concurrent accesses to objects. For each object in the system (including threads that are objects), our environment gives all the threads that are currently accessing it. The lower part of the window shown on Figure 7 exhibits that the object 3 is currently accessed by the threads 6, 5 and 2. Moreover, we witness that this object is accessed two times by a single thread 2 (i.e. there are currently two method calls on this object coming from the same thread).

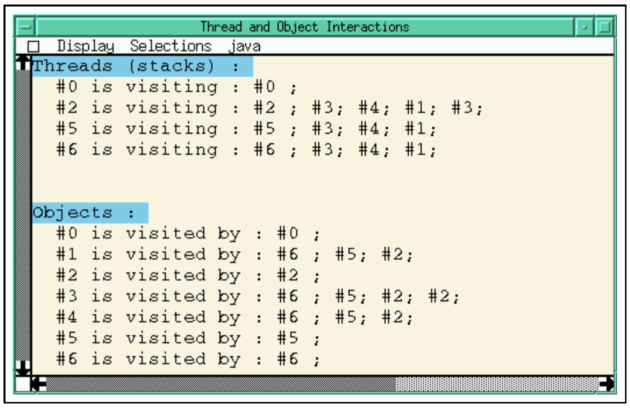

Figure 7: Thread Stacks and Object Activations.

3.4 Synchronizations and Locks

Every Java programmer who had programmed multi-threaded applications would agree on the fact that debugging such programs is at least difficult: how to know, at a given execution step, which objects are locked, or which threads are blocked, in order to understand why there is a deadlock or if a critical section is necessary.

Textual and synthetic information presented above is a first answer to that question, but is not enough alone, so we provide a way to graphically visualize synchronization mechanisms and locks, in order to give the programmer an integrated view of the topology, thread status, and locks altogether.

First, we visualize all the thread status by color codes (red for blocked threads, black for dead threads). During execution, we also show locked objects (light blue for unlocked objects, dark blue for locked objects). All status are visible in Figure 8: a complete legend presenting all color and graphic codes.

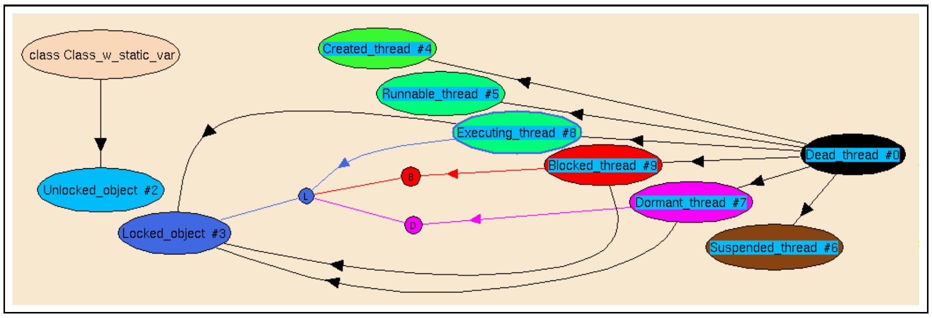

Figure 8: Objects, Classes, Locks, and Thread Status.

About information related to locks, our first idea was to make the lock visible on the reference between the thread and the concerned object. But a thread that has no direct reference on an object can hold a lock on it, indirectly, through other objects, see Figure 9.
In this example, the thread 0 holds a lock on the object 3 but in fact this lock is obtained through three successive method calls (via the objects 1, 2 and 3).

This implies that the information can not be attached on the references but must have its own representation. The graphical presentation of the synchronization mechanisms is detailed in the following section.

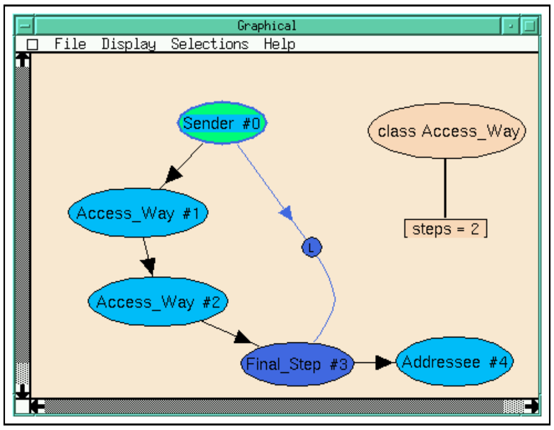

Figure 9: Example of Indirect Locked Object.

Figure 10 presents a more common communication topology where all the objects of the access way are locked by the sender thread in order to protect the transmitted data.

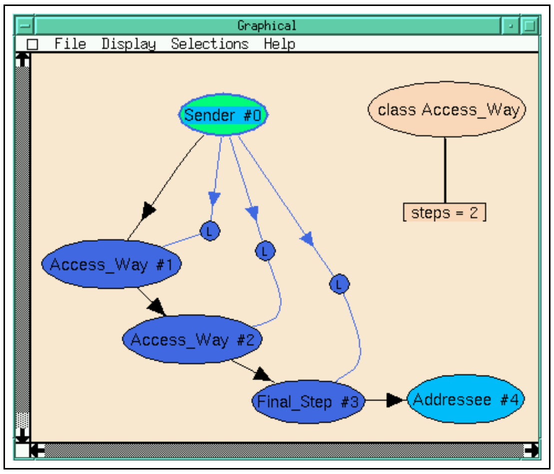

Figure 10: A chain of locked objects.

3.5 Synchronization Animation

How to visualize synchronizations during execution ? Information to study and understand synchronization can be found in the varied textual views but we chose to highlight them in the graphical presentation, and moreover in an animated fashion. This feature requires to introduce new kinds of arrows between objects.

- L-arrows are arrows labeled with a blue L label (L for locked); this arrow exhibits a lock between the origin thread and the destination (locked) object;
- B-arrows are arrows labeled with a red B label (B for blocked); this arrow links a thread and a lock on an object; it establishes that the origin thread is blocked, waiting for a lock on an object which is already locked by another thread;
- D-arrows are arrows labeled with a purple D label (D for dormant); this arrow links a dormant thread and a lock on an object. It indicates that the dormant thread will compete in order to hold this lock as soon as it will be re-enabled for thread scheduling.

We now demonstrate how these informations are shown within the animated graphical view (see Figures 11,12), during execution of the producer-consumer program (presented in Figure 1).

In this program, there are 4 threads and one object. In the top window, the main thread has created the three other threads (two consumers ans one producer) and the object (CubbyHole), and has started the thread Consumer 2. The thread Consumer 2 is currently executing (thicker blue border on the node).

Later during execution (second window), the main thread is dead (black node). The three threads have been started. Consumer 2 and Producer 4 are runnable and Consumer 5 is executing. CubbyHole is locked by Consumer 2 (presence of a L-arrow and dark blue color for CubbyHole).

Later again (third window), Consumer 5 tried to access CubbyHole (call to the synchronized *get* method) and is now blocked (shown in red and B-arrow). Consumer 5 is waiting for the lock on CubbyHole (which is still owned by Consumer 2).

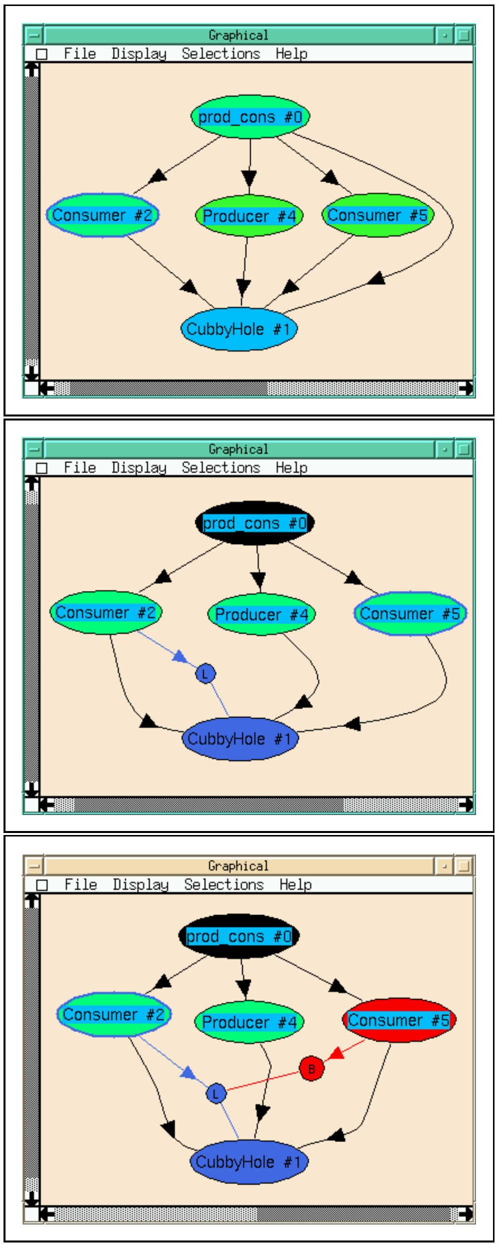

Figure 11: Synchronizations (1).

Further execution of the program (top window from Figure 12) turns Producer 4 into the blocked status, waiting for the lock on CubbyHole (still owned by Consumer 2). So, there are now two blocked threads (visible in red).

In the middle window, a lock is released (object CubbyHole and its lock are now in light blue) by Consumer 2. Consumer 2 becomes dormant (purple color and link to the CubbyHole lock by a D-arrow) because it has locked the CubbyHole before the Producer 4 has put a value in it (and therefore has no value to read). So Consumer 2 has called the wait method in order to enable Producer 4 to get the lock on CubbyHole. There are still two blocked threads waiting for a lock on CubbyHole.

Finally, as shown in the bottom window, one of them will resume execution arbitrarily and will hold this lock (Producer4). Eventually, both Producer 4 and Consumer 5 will successively release the lock and Consumer 2 will resume.

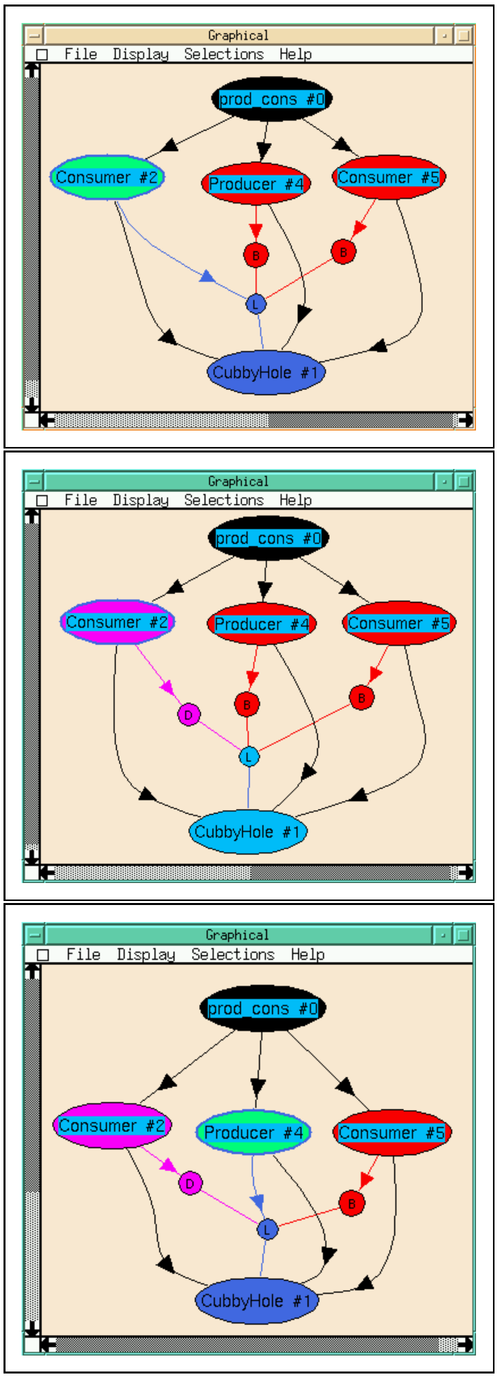

Figure 12: Synchronizations (2).

4 The Environment Architecture

While based on formal structures, the design principles used in our tools offer a general model to handle Java run-time entities. These principles can be reused in other interactive environments.

Our environment is derived from Centaur [3], a generic programming environment, which is based on Natural Semantics [17] and its implementation formalism Typol [7]. The Centaur system permits, from a set of formal specifications (both syntactic and semantic), to derive a dedicated minimal programming environment.

Program editing and visualization is part of the so-called *editing server* (possibly running on one machine) as opposed to the *semantic server* which handles execution of the semantics (possibly running on another machine); the model is based on a client-server architecture, with asynchronous communications.

4.1 Syntactic Specifications

From syntactic specifications, Centaur automatically generates a structure editor, with syntax coloring, using an abstract syntax (a set of 128 operators and 56 types). This abstract syntax eases the editing process: according to the current selection (a point in the program), abstract syntax patterns can be chosen, without actually being annoyed by the concrete syntax of the language.

As an illustration, we give in Figure 13 the abstract syntax tree corresponding to the expression: `Obj.m_name(Expr1, Expr2)`.
Operator names are given in lower-case, while types names start with a capital letter.

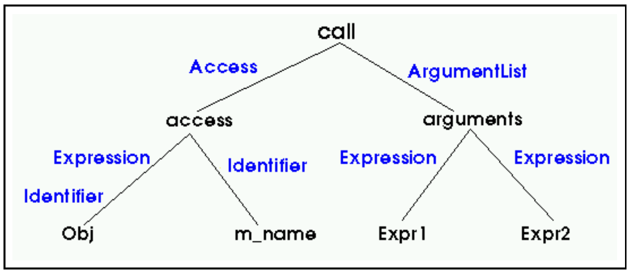

Figure 13: Abstract Syntax for Method Call.

4.2 Semantic Specifications

From semantic specifications, Centaur automatically generates an interpretor which takes as input a syntactically correct Java program (in fact, an abstract syntax tree). The result of the semantic evaluation is a list of objects and threads, which denotes the behavior (the meaning) of the program. We do not detail the semantic specification here; instead we prefer to focus on semantic structures (the model for objects and threads) and give an overview of the architecture of the semantic description (organized in modules).

4.2.1 Semantic Structures

During execution, a Java program creates, uses, and updates objects and threads. Figure 14 establishes the structure of objects and threads. In case of a simple object (not thread), the only difference is that the activity is *nil*.

The activity is composed of a status and a continuation, which is made of:

- a thread identifier,
- the name of the current method,
- an instruction list (language statements as well as closures for method calls);
- an execution environment made of parameters (name-value pairs) and local variables (name-value pairs).

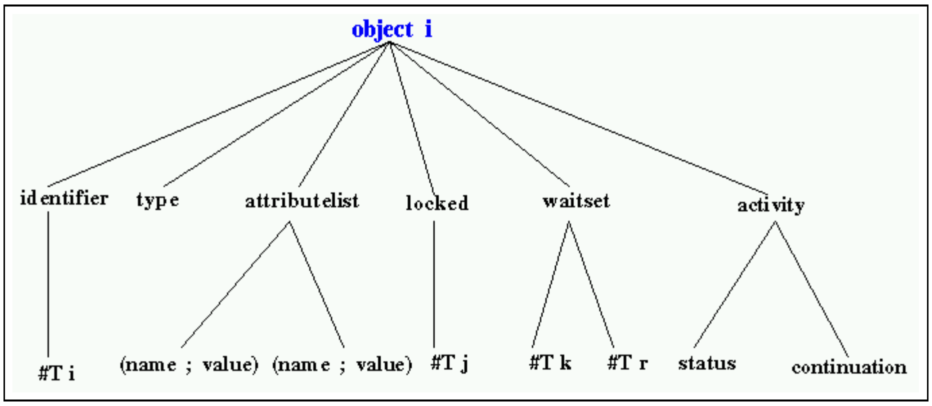

Figure 14: Abstract Syntax for Objects and Threads.

This structure is changing during program execution, as shown in Figure 15: the call `o2.foo()` is replaced by the closure representing the body of the method `foo`.

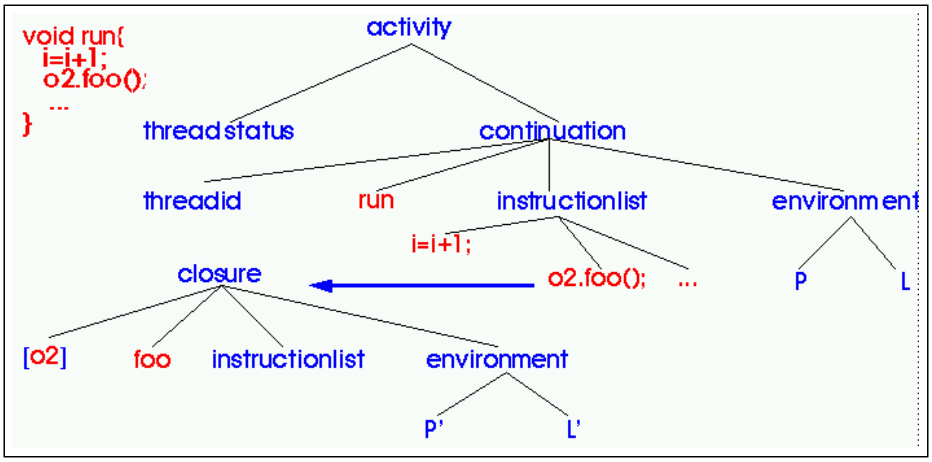

Figure 15: Activity of a Thread: an example.

4.2.2 Semantic Modules

The semantic specification is composed of 400 inference rules describing an operational semantics of Java. These rules are both highly declarative and executable. They are organized in modules as shown in Figure 16, which enhances design, readability, and debug. Some modules are expressed in a Natural Semantics style [17] (especially object-oriented features, e.g. `java_inheritance.ty`). Some others are expressed in a Structural Operational Semantics style [26] (especially concurrent features, e.g. `java_stat_execution.ty, java_expr_evaluation.ty`).

4.3 Textual and Graphical Visualization

The two different views of the list of objects use two visualization engines, both of them based on the semantic structure modeling the list of objects as presented in Section 4.2.1.

The textual view is a direct pretty-printing of the list of objects and threads (Figure 2). The graphical view shows the complete topology of the

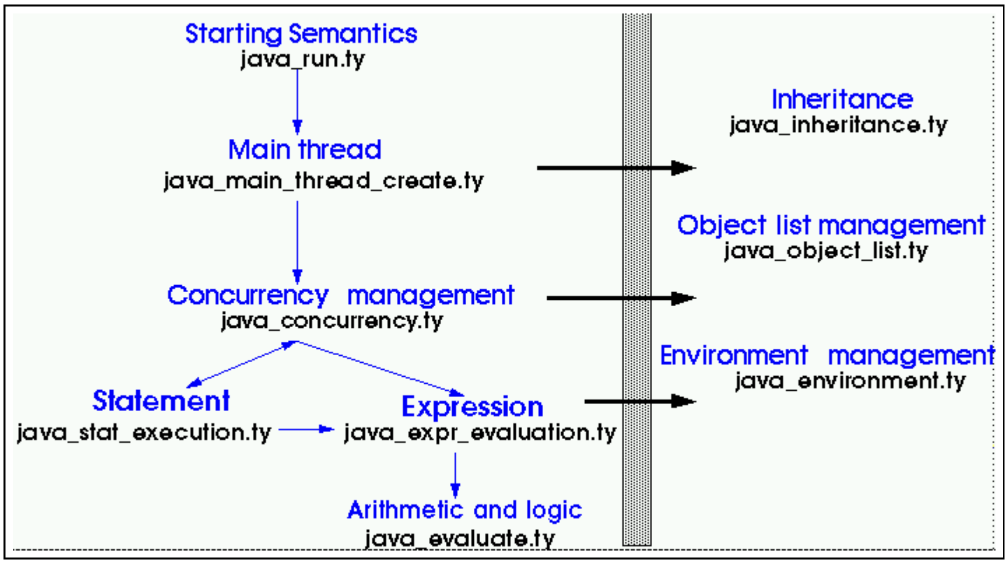

Figure 16: Semantic Modules.

object graph (Figure 5). This graph is visualized thanks to the graph server of Centaur [15]. Nodes and edges of the graph are built using a traversal of the abstract syntax tree representing the list of objects: for each object, a node representing an object is created, and for each attribute value which is a reference, an edge is created between two object nodes.

4.4 Programming Animation

Our environment provides animation to visualize objects *during* program execution, and so have a better understanding of the behavior of the program. Without any program instrumentation from the user, execution can be done in a step-by-step mode, stopped and resumed, or aborted at any time, so animation is controlled by the programmer, and occurs at his pace. As an extra feature, when needed, the user can slow down the execution using a control menu (see Figure 17).

Animation is performed with notifications from the semantics to the two textual and graphical servers.

For that purpose, the semantics is equipped with notifications. On some appropriate semantic rules, when successfully applied (proved), the notification (if it exists) is triggered and the visualization engines become aware of some modification in the semantic structures. Such notifications tell the servers to change the value of a field, the status of a thread, to add or remove a lock on a given object, an arc between two objects. Altogether, less than 10 semantic rules needed to be equipped with such notifications. In the case of the graphical server, the rules where we need to send notifications are the following:

1. object and thread creation,
2. thread status change (runnable, executing, locked, etc.),
3. object status change (locked, unlocked),
4. method calls and returns,
5. assignments.

Note that the notifications do not require to be scattered all over the architecture, but are actually concentrated in only two modules: object list and concurrency managements (Figure 16). Another critical aspect is incrementality: in order to have efficient and quality visualizations (avoiding flashing) the changes are done in an incremental manner in both views.

5 Related Work

Program visualization is an active research area, widely used in several domains such as software design, performance monitoring, software training. Representative systems are Zeus [4] for algorithm animation, Polka [28] for program animation on parallel architectures and Pavane [27] for visualization of concurrent program execution (see [19] for a survey on parallel programs visualization).

Among systems more directly connected to programming languages, the visualization and probing of class/program structure represent a first category – the display of *static* program information, along the line of graphical browsers. CIA++ [12] builds a database of information of C++ programs which is used to display various views of the program structure. GraphLog [6], a generic tool with a visual query language, allows visualizing and querying software structures. Object Explorer [2] by Kent Beck is dedicated to SmallTalk.

Another important category features the visualization of *dynamic* information: an actual representation of objects created when executing a program. ObjView [10], a system for the design of electronic boards based on a model written in C++, offers an interface with both a realistic view of problem-domain objects and representation-domain C++ objects with visualization of instances, member data and functions. Haarslev and Moeller, in [14], propose a framework for both class hierarchies and objects using the CLOS [23] meta-level architecture; an explicit association of visualization objects with application objects is needed in that case. Object Visualizer [24, 25] and HotWire [20] are based on program instrumentation mechanisms. The present Java graphical environment reuses some techniques from previous work [1], that we do not detail here, namely how to derive a graphical environment from a formal semantics.

More recently, the need for probing the large amount of objects was discussed: Kent proposed a visual notation for invariants (both at the class and the instance level) in object-oriented models [18]. Lencevicius, Hölzle, and Singh promoted the so-called "query-based" debugging that allows efficient manipulation of large object space [22].

Related work also includes several commercial systems for Java development: Microsoft's Visual J++ [13], Kawa [31], JavaWorkshop [21], Javelin [29], IBM's VisualAge for Java [16], and Asymetrics SuperCede Java [30] (see [9] for an exhaustive review of Java tools, and [8] for a review of Java visual tools).

Comparing our environment with those development environments for Java, we can highlight several common features: assistance for syntactic constructs, interactive debugging, examination of data contents, and visualization of thread activities. These feature are rather minimal for any modern development environment (see for instance the syntax coloring of Kawa, or the fully graphical debugger of SuperCede Java).

The advanced features of our environment, that bring a significant contribution to the field of development environments are the following.

The first aspect is the graphical visualization of the object graph topology. We admit that graphical environments do exist: Javelin is, among commercial tools, the one that is radically graphical-oriented: classes are presented as boxes connected by lines representing inheritance. Boxes contains data and methods of the class and can be graphically edited. But these informations are static (e.g. class hierarchy), and we believe that graphical visualization of dynamic information such as references between objects is very useful for the programmer. Moreover, visualization of synchronizations between threads is of major importance in threaded programming. Dynamic information (objects) is shown in SuperCede Java or Kawa, but their textual presentation is tree-oriented (not graph-oriented); there is is no global view, so one needs to compare the values of variables to detect shared objects or cycles. This kind of information appears at first sight in our environment, thanks to the graph-oriented visualization of the topology.

The second aspect is related with interactive debugging: in our environment, no compilation is necessary at all, as opposed to incremental recompilation after modification or addition of a breakpoint which is usually necessary in commercial development tools. The exception is Supercede Java, where you can modify running programs, and incorporate the changes into the running program without restarting it. This valuable capability of our environment comes from the fact that our debugger is not based on byte-code execution, possibly instrumented, but instead, is directly using the operational semantics, instrumented with notifications (as explained in Section 4).

6 Conclusion

Figure 17 presents a global view of our environment during execution of the Producer-Consumer program from Section 2.4. Besides the program itself (top left window), there are two synthetic views: object and thread status (top middle), thread stacks and object activations (top right). The object list is first presented in a textual form (bottom left) where a detailed view of objects and threads is given, including the activity of each thread (stack or continuations).

The graphical view (bottom right) features the topology of the object graph, the threads status (from top, and left to right: dead, dormant, executing, blocked), together with the visualization of locks (object 1 is locked by object 4). A control panel (inside the graphical window) provides for a step-by-step execution.

The contribution of this work resides in several aspects. First, we intend to show the benefits of graphical visualization of the instance topology in an object-oriented system, especially when dealing with complex and interconnected entities as found in a multi-threaded environment. We stated both the problem and a solution for visualizing objects, static variables, threads, and locks within that context. Those principles can be reused in other environments, while our platform demonstrates the feasibleness of the approach. We do believe that even within mono-threaded systems such graphical visualization would be useful. One can imagine to automatically produce object topology for documentation purpose, to recognize object patterns for instance when doing reverse engineering. The environment might also be used for teaching concurrent and object-oriented concepts.

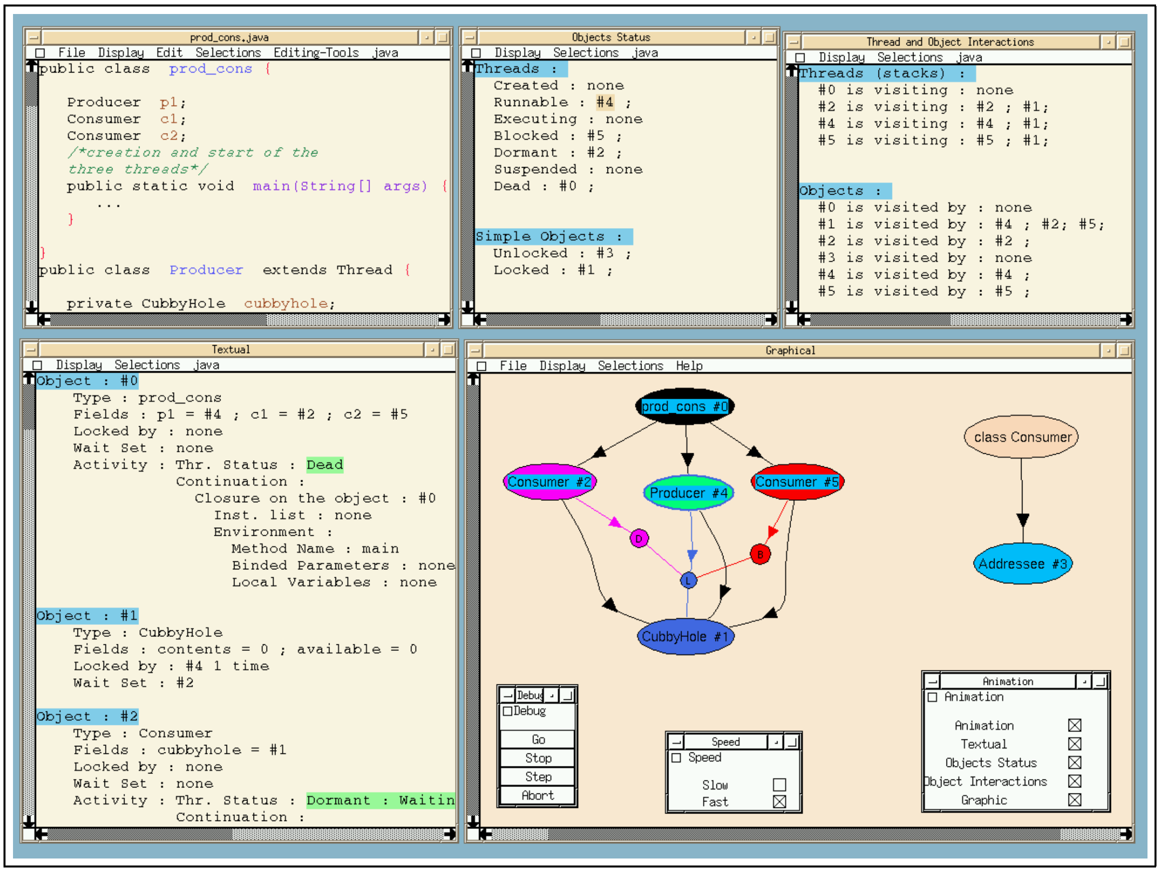

Figure 17: Global view of our graphical environment.

The architecture of the system itself, sending notifications to textual and graphical visualization servers, can be reused. While based on a formal semantics, the modeling and structuring of Java entities like objects, threads, and locks, together with the specific locations where the notifications have to take place, can be applied in other settings.

Further experimentations should focus on two main aspects. First, we would like to be able, when needed, to visualize only an abstract view of the object graph, allowing complex systems to be handled in an advantageous manner, and to gain in scalability. For instance, we are exploring techniques for displaying only threads (not objects), objects of certain types exclusively, or certain sub-graphs based on user information. Another feature that we would like to add to the environment should permit the user to control the interleaving between all threads (e.g. artificially blocking or slowing down threads), allowing to explore various potential behaviors, for instance in order to look for deadlocks.

References

[1] I. Attali, D. Caromel, S. O. Ehmety, and S. Lippi. Semantic-based visualization for parallel object-oriented programming. In *Proc. OOPSLA'96 (Object-Oriented Programming: Systems, Languages, and Applications)*, volume 31, number 10. ACM Press, Sigplan Notices, Oct 1996.

[2] K. Beck. Object explorer for visual works. Commercial Product http://c2.com/ppr/about/author/kent.html, First Class Software, Inc.

[3] P. Borras et al. Centaur: the System. In *SIGSOFT'88 Third Annual Symposium on Software Development Environments*, Boston, 1988.

[4] M. H. Brown. Zeus: a system for algorithm animation and multiview editing. In *Proc. of the IEEE Workshop on Visual Languages*, 1991.

[5] M. Campione and K. Walrath. *The Java Tutorial (Object-Oriented Programming for the Internet)*. Addison–Wesley, 1998.

[6] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and querying software structures. In *Proc. of the* 14^{th} *International Conference on Software Engineering*, pages 138–156, May 1992.

[7] T. Despeyroux. Typol: A Formalism to Implement Natural Semantics. Research Report 94, INRIA, 1988.

[8] C. Dichter. We test the top 6 Java visual IDEs. Product Reviews http://www.javaworld.com/javaworld/jw-04-1998/jw-04-visual-ides.html, JavaWorld Magazine, 1998.

[9] D. Dyer. Capsule Reviews of JAVA tools. Product Reviews http://www.andromeda.com/people/ddyer/java/Reviews.html, Andromeda Software, 1998.

[10] G. Friedrich, W. H., C. Stary, and M. Stumptner. Objview: A task-oriented, graphics-based tools for object visualization and arrangement. In *Proc. ECOOP '89*, pages 299–310, Nottingham, July 1989. Cambridge University Press.

[11] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison–Wesley, 1996.

[12] J. E. Grass. Object-oriented design archaeology with CIA++. *Computing Systems*, 5(1):5–67, 1992.

[13] P. Greg. *Using Visual J++.* Que, 1996. ISBN 078970899X.

[14] V. Haarslev and R. Möller. A framework for visualizing object-oriented systems. In *Proc. OOPSLA/ECOOP '90, ACM SIGPLAN Notices 25 (10)*, pages 237–244, October 1990.

[15] A. Le Hors. Graph: A directed graph displaying server, GIPE 2 Esprit project, 4th review report, workpackage 4, 1992.

[16] IBM Corporation. Visual Age for Java. Commercial Product http://www.software.ibm.com/ad/vajava/, 1997.

[17] G. Kahn. Natural Semantics. In *Proc. of Symposium on Theoretical Aspects of Computer Science, Passau, Germany, LNCS 247*, 1987.

[18] S. Kent. Constraint diagrams: Visualizing invariants in object-oriented models. In *Proc. OOPSLA'97 (Object-Oriented Programming: Systems, Languages, and Applications)*, volume 32, number 10. ACM Press, Sigplan Notices, Oct 1997.

[19] E. Kraemer and J. T. Stasko. The visualization of parallel systems: an overview. *Journal of Parallel and Distributed Computing, 18*, 1993.

[20] C. Laffra and A. Malhotra. Hotwire – A visual debugger for C++. In *USENIX Sixth C++ Technical Conference*, pages 109–122, Cambridge, MA, April 11-14 1994. USENIX.

[21] L. Lemay and Charlesworth. *Teach Yourself SunSoft Java Workshop in 21 Days.* SAMS, 1996. ISBN 1575211599.

[22] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *Proc. OOPSLA'97 (Object-Oriented Programming: Systems, Languages, and Applications)*, volume 32, number 10. ACM Press, Sigplan Notices, Oct 1997.

[23] A. Paepcke. *Object-Oriented Programming: The CLOS Perspective*. The MIT Press, 1993.

[24] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proc. OOPSLA '93, ACM SIGPLAN Notices, 28 (10)*, pages 326–337, October 1993.

[25] W. De Pauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. In *Proc. ECOOP '94*, LNCS 821, pages 163–182, Bologna, Italy, July 1994. Springer-Verlag.

[26] G. D. Plotkin. A Structural Approach to Operational Semantics. Report, DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[27] G.C. Roman et al. A System for Declarative Visualization of Concurrent Computations. *Journal of Visual Languages and Computing, 3 (2)*, 1992.

[28] J. T. Stasko and E. Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing, 18*, 1993.

[29] Step Ahead Software Pty Ltd. Javelin. Commercial Product http://www.ozemail.com.au/ stepsoft/javelin.htm, NSW 2077 Australia, 1998.

[30] SuperCede, Inc. SuperCede Java. Commercial Product http://www.supercede.com, Bellevue, WA, 1998.

[31] Tek-Tools, Inc. Kawa. Commercial Product http://www.tek-tools.com/kawa/, Dallas TX, 1998.