Semantic-Based Visualization for Parallel Object-Oriented Programming

Isabelle Attali, Denis Caromel, Sidi O. Ehmety, Sylvain Lippi

INRIA Sophia Antipolis
CNRS - I3S - Univ. Nice Sophia Antipolis
BP 93, 06902 Sophia Antipolis Cedex
tel: 33 93 65 79 10, fax: 33 93 65 76 33, email: First.Last@sophia.inria.fr
http://www.inria.fr/croap/eiffel-ll

Abstract

We present a graphical environment for parallel object-oriented programming. It provides visual tools to develop and debug object-oriented programs as well as parallel or concurrent systems. This environment was derived from a structural operational semantics of an extension of the Eiffel language, Eiffel//. Object-related features of the language (inheritance, polymorphism) are formalized using a big-step semantics, while the interleaving model of concurrency is expressed with small-step semantics.
Without user instrumentation, the interactive environment proposes features such as step-by-step animated executions, graphical visualization of object and process topology, futures and pending requests, control of interleaving, deadlock detection.

1 Introduction

We present a graphical environment for parallel object-oriented programming. This environment provides visual tools to develop and debug object-oriented programs as well as concurrent systems. This environment has been derived from a formal semantics of an extension of the Eiffel language [41], named Eiffel// (pronounce Eiffel parallel) [17].

We adopt a structural operational semantics [55] for both the Eiffel language itself, and the concurrent primitives. More specifically, we use the Natural Semantics [33] within the Centaur system [23], and the Typol formalism [21] which provides us with executable specifications. While currently used on a specific model of concurrency, other models could be specified, and visualization tools would directly follow from the semantic description. Moreover, the approach of applying formal semantics to visualization can be used for other object-oriented languages.

Graphical techniques have been used in program development in the setting of object-oriented programming on one hand [14], and parallel programming on the other [36]. Concurrent object-oriented programming uncovers specific aspects such as the visualization of active objects, requests, futures, synchronizations, etc.

Our work is specifically concerned with the graphical visualization of hand-written applications; method-based environments for interactive design with source code generation are slightly out of the scope of this paper. Our contribution is first to demonstrate that it is actually possible to derive visualization and debugging tools[1] directly from a formal semantics; secondly, the environment does not require any instrumentation from the user. The

[1]For a color version of the figures, see at the end of this volume.

outcome of such an approach is twofold: (i) a pedagogic environment to demonstrate concepts of object-oriented programming, actor computation, and formal semantics; (ii) a step towards environments for the formal study of parallel object-oriented programming.

Our interactive environment, based on a concurrent model of execution of parallel activities through interleaving, presents features such as step-by-step animated executions, graphical visualization of object and process topology, futures and pending requests, control of interleaving, deadlock detection.

The next section of this paper is a discussion of related work. Section 3 illustrates, with a simple example, the language and model used for concurrent programming. Section 4 focuses on the operational semantic definition. From the formal semantic definition, graphical and interactive visualization tools are derived (Section 5). In Section 6, we compare and discuss our technique and tools with closely-related approaches. Finally, Section 7 briefly discusses our contribution and outlines future work.

2 Related Work

Our system being concerned with both semantics of concurrency and program visualization, related work in these two domains is presented and discussed here.

The seminal works of Hoare (CSP [30]) and Milner (CCS [42]) play a fundamental role for procedural programming, but they occurred to be not suited for object-oriented programming, where the configuration of systems changes dynamically. Milner has then proposed the π-calculus [43] as an extension of CCS; in the π-calculus, channels (for interaction between agents) can be dynamically generated and transmitted from one agent to another. On the other hand, object-based models for concurrent systems have been explored [29, 1, 73]. The actor model is an extension of the λ-calculus (well-suited to functional programming) and is based on the notion of configuration of actors. Autonomous actors communicate via asynchronous message-passing; each actor has its own behavior, reacts to a message and changes its configuration.

Models have been used and extended in many different ways: for instance, Honda and Tokoro proposed in [31] an object calculus for asynchronous communication built on Milner π-calculus. On the other hand, Nierstrasz and Papathomas combined the concurrency models from CCS, CSP and the actor models into a computational model of communicating agents [51]. More recently, Satoh and Tokoro developed RtCCS [59], a formalism for real-time object-oriented computing based on CCS.

For these different approaches, sound type systems were developed [4, 49, 65, 35] and formal semantics have been described in different frameworks (denotational [6], operational [5, 48, 3, 19], based on the π-calculus [67], or traces [64]).

Based on a formal model or not, several concurrent object-oriented programming languages have been designed, such as ConcurrentSmalltalk [72], Distributed Smalltalk [11], Hybrid [45], Pool [4], ABCL [68], Eiffel// [16], DROL [63], and more recently Java [26].

Program visualization is an active research area in several domains such as software design, performance monitoring, software training. Representative systems are Zeus [12] for algorithm animation, Polka [62] for program animation on parallel architectures and Pavane [22] for visualization of concurrent program execution (see [36] for a survey on parallel programs visualization). One interesting example is ToonTalk [34]: K. Kahn shows that concurrent constraint programming with an interactive animation becomes suitable for children.

Specifically related to the object paradigm, some systems are not directly language-based, but rather provide independent environments for interactive visualization. For instance, Object Design Exploratorium (ODE) [57] offers a learning environment for design principles. A few products, more oriented towards modeling methodologies and interactive environments, provide animated execution and code generation. Based on the Shlaer-Mellor method [61], SES/objectbench [60] takes advantage of a simulation engine to provide animated simulations and BridgePoint [56] has a model verifier for execution and a translation engine to generate source code. ObjecTime [39], using the ROOM method (Real-time Object-Oriented Modeling), includes visualization with active and passive objects. 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++ [27] builds a database of information of C++ programs which is used to display various views of the program structure. GraphLog [20], a generic tool with a visual query language, allows visualizing and querying software structures. Object Explorer [10] 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 [24], 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 [28], propose a framework for both class hierarchies and objects using the CLOS [52] meta-level architecture; an explicit association of visualization objects with application objects is needed in that case. Object Visualizer [53, 54] and HotWire [37] are based on program instrumentation mechanisms. Object Visualizer is accumulative, based on an event space, and need *renderer* classes for displaying and changing graphical elements, while HotWire is a visual debugger for C++ with custom visualization based on a scripting language. Recently, Lange and Nakamura, in [38], proposed to use interactive visualization to understand – and reuse – frameworks. They couple program instrumentation with a modeling of both static and dynamic program information (within a logical framework, using a Prolog notation). Stored in a database, the information can then be selected and filtered out for specific display. Program Explorer, their program visualization tool for C++, applies these principles.

A state of the art of applications of visual techniques in object-oriented programming is presented in [14]. Experiences are reported concerning two orthogonal issues: communication from the programmer to the computer (visual syntax, language issues) and vice-versa (visual environment, presentation of static and dynamic information, with an emphasis on animation). In this paper, we are only concerned with this second aspect.

Providing an interactive graphical environment for concurrent object-oriented programming leads to more specific visualization issues – e.g. how to show active objects (agents, actors), asynchronous and synchronous message passing, synchronizations between objects. More fundamentally, an important issue is to be able to ensure the consistency between the program execution and the visualization. A few early works can be found in [70], while a number of approaches have been explored in recent research.

One orientation consists of the development of a library. BEE++ [13] provides dynamic analysis of distributed systems through a library of classes to be extended by the end-users in order to monitor and to visualize applications. Vion-Dury and Santana, in [66], propose a 3D interactive animations for spatial visualization where objects have polyhedral colored shapes. The system is built on top of a debugging tool of Guide [9] using record and replay. With a different approach and goal, the ObjChart formalism and environment [25] proposes to specify reactive objects directly in a visual framework. The environment is founded on an executable compositional semantics. Based on equations over traces, it does not allow dynamic object creation.

Other systems are more integrated within a specific language. Agha and Astley [2] propose a visualization system for an extension of the actor model [1]. An abstraction for visualization, *visualizer*, monitors a list of the system components, and ensure consistency by calling themselves visualization functions. Implemented using synchronizers and reflection, the system does not require explicit modification of the application code.

The system we present focuses on the visualization of dynamic aspects, and shows the actual objects of the program during execution with two representations: textual and graphical. The system is based on a formal operational semantics, needs no instrumentation, and does not require the user to write any code or script. It permits an interactive visualization of both sequential and concurrent object systems.

3 Model and Language for Concurrent Programming: Eiffel//

In this section, we give a quick overview of a concurrent language, Eiffel// [17], defined as an extension of Eiffel [40] to support programming of parallel applications. These extensions are not concerned with syntax, but are purely semantic, which gives to both languages the same syntactic description (see Figure 1). Both are strongly typed, statically-checked class-based languages. Our purpose here is not to discuss the rational of Eiffel//, and issues related to object-based concurrent languages (the reader can refer to a recent and related design (C++// [18]), and to related research [47, 69, 15, 50, 71]).

3.1 Syntactic Constructors

Syntactic constructors are presented in Figure 1, and used in the semantic specifications in their concrete form. An expression $E \in$ Expr is a variable, a feature-call, an arithmetic or logic expression or a constant. A statement $S \in$ Stmt is an assignment, a sequence of statements, a selection, an iteration, a routine call[2], including the object creation. The identifier X is an attribute, a local variable, a formal parameter or one of the two pseudo-variables **current** and **result**. The identifier Y denotes an attribute or a local variable (including the **result**). K designates a constant value (integer, boolean, **void** (null reference), etc.) and Op_1 and Op_2 the usual unary and binary operators.

[2]The non-qualified feature-call $M(E_1, \ldots, E_n)$ is equivalent to the qualified call **current** $\cdot M(E_1, \ldots, E_n)$.

— **Expressions**	— **Statements**
$E ::= X$	$S ::= Y := E$
$\mid E.M(E_1, \ldots, E_n)$	$\mid S_1;\ S$
$\mid Op_1\ E$	$\mid$ **if** E **then** S_1 **else** S_2 **end**
$\mid E_1\ Op_2\ E_2$	$\mid$ **until** E **loop** S **end**
$\mid K$	$\mid E.M(E_1, \ldots, E_n)$

Figure 1: Statements and Expressions

3.2 Concurrent features

The Eiffel// model uses the following principles:

- heterogeneous model with both passive and active objects (processes, actors);
- sequential processes;
- unified syntax between message passing and inter-process communication;
- systematic asynchronous communications towards active objects;
- wait-by-necessity (automatic and transparent futures);
- no shared passive objects (call-by-value between processes);
- centralized and explicit control by default;
- polymorphism between objects and processes.

As in Eiffel, the text of an Eiffel// program (a system) is a set of classes, with a distinguished class, the root class. Parallelism is introduced via a particular class named PROCESS. Instances of classes inheriting (directly or not) from the PROCESS class are processes. Processes inherit a default behavior (which ensures that requests to the process entry points are treated in a *fifo* order), but this behavior can be redefined with the overriding of the **live** routine. All other objects are passive; a process is an object but not every object is process. Polymorphism between objects and processes is possible: an entity which is not declared of a type process can dynamically refer to a process. In that case, a feature-call dynamically becomes an asynchronous communication between processes (*Inter-Process Communication*).

Cohabitation of active and passive objects leads to an organization in subsystems. Each subsystem contains a root process and the passive objects it references. Within a subsystem, the execution is sequential and communications are synchronous: the target object immediately serves the request and the caller waits for the return of the result. Between subsystems, the executions are parallel and communications are asynchronous: the target object (a process) stores the request in a list of pending requests, and the caller carries on execution. There is no shared object between subsystems: references on passive objects are passed by copy between subsystems. As a consequence, when polymorphism between an object and a process occurs, two changes happen: asynchronous calls and copy transmission of parameters. Of course, this modifies the local semantics of the call (reflected in the formal description, Figure 5, rules (`I11`) to (`I15`)). However, this is often a desired change when parallelizing and, in many cases, does not affect the global semantics of the application (see for instance the speech recognition application, Figure 13).

Synchronization is handled via the *wait-by-necessity*, a data-driven mechanism which automatically triggers a wait when an object attempts to use the result of an awaited value (transparent future). The wait-by-necessity, by automatically adding some synchronization, tends to maintain the behavior of a sequential program when doing the parallelization. Explicit synchronizations can be expressed with the predefined routine **Wait** (`v.Wait` triggers a wait if v is a future). Another primitive **Awaited**, is a boolean feature with returns *true* when the considered object is awaited (`if v.Awaited then` "`do something in the meantime`"). It is also possible to wait for requests: the predefined routine **wait_a_request** permits to block a process until a new request arrives, e.g. inside a loop construct of a server.

3.3 An example

As an illustration, Figure 2 presents an Eiffel// system. It provides a parallel version of the sequential class BINARY_TREE, which describes the management of a sorted binary tree with two routines `insert` and `search`: each node of the tree has two

```
class BINARY_TREE
  export   insert, search, left, right
  feature
    key : INTEGER;
    info : INTEGER;
    left, right : BINARY_TREE;
    insert (k : INTEGER; i : INTEGER) is
        ... - inserts information i with key k
    search (k : INTEGER) : INTEGER is
        ... - searches for the value of key k
end - BINARY_TREE

class P_BINARY_TREE
  export insert, search, left, right
  inherit PROCESS;
          BINARY_TREE redefine left, right;
  feature
    left, right : P_BINARY_TREE;
end - P_BINARY_TREE
```

(a) Sequential and Parallel Binary Trees

```
class EXAMPLE
  feature v: INTEGER;  bt: BINARY_TREE;
    create is
      local p_bt: P_BINARY_TREE;
        do
          p_bt.create;
          bt := p_bt;      - polymorphism
          build_binary_tree(bt);
          v := bt.search(2);
          v.print         - wait-by-necessity
        end; - Create
    build_binary_tree(bt: BINARY_TREE) is
        do  - building the binary tree
            bt.insert(3, 6);
            bt.insert(1, 2);
            bt.insert(2, 4);
            bt.insert(4, 8);
            bt.insert(6, 12);
        end ; - build_binary_tree
end - EXAMPLE
```

(b) Using Active Objects

Figure 2: An Eiffel// system

children (`left` and `right`), an information (`info`) and an associated key (`key`); keys of the left (resp. right) subtree of a node are smaller (resp. greater) than the key of this node.

To parallelize the binary tree we define the P_BINARY_TREE class. It inherits from the PROCESS class and the BINARY_TREE class; no other programming is necessary; the full version of the class is actually shown in the Figure 2.a. Polymorphism between processes (`p_bt`) and objects (`bt`) makes it possible to reuse existing sequential code (here `build_binary_tree` for instance). In that example, the default fifo behavior and the wait-by-necessity ensure that all insertions are handled in a correct order, and before the search; the parallel system preserves the semantics of the sequential one.

4 Operational Semantics

In this section, we describe the operational semantics of Eiffel//. This operational semantics simulates parallelism with a non-deterministic interleaving of (activities of) concurrent objects.

The semantics of inheritance and dynamic binding is expressed in Natural Semantics [33]. Although, the modules describing the actual execution of statements (loops, feature calls, assignments, ...) are expressed in Structural Operational Semantics (SOS) [55]).

Natural Semantics (*big-step* semantics) is opposed to SOS (*small-step* or *transitional* semantics) in the sense that intermediate steps of the execution of programs are hidden in a big-step semantics. The general idea of a semantic definition in Natural Semantics is to provide axioms and inference rules that characterize semantic behaviors to be defined on language constructs. Behaviors are expressed with sequents in a logical style. These two styles of semantic description cohabit well in the logical framework of the Typol formalism [21].

We assume that the source program, an abstract syntax tree noted Π, is correctly type-checked. We briefly present the semantics related to inheritance and dynamic binding (Section 4.2), as it is defined for Eiffel (the reader can refer to a detailed version of the semantics of Dynamic Binding [7]).

We need to define some structures which describe the global configuration of a system. During execution, an Eiffel// system is composed of objects. Each object in the system has a configuration (attribute values, activity, pending requests); the collection of all object configurations is the configuration of the system. For modeling objects (with their activity) during execution, we need a structure (based on an abstract syntax). We also need a structure to store the futures and their values.

We then describe the operational semantics of the language in terms of a transition system, modeling possible transitions (global actions) from one configuration to another. We present rules describing global actions of a system; these global actions are expressed in terms of local actions on objects or interactions between objects.

4.1 Semantic Structures

We present the semantic structures used to model objects: static type, attributes, local variables, pending requests, and their activity (a list of closures).

4.1.1 Objects

We model a system of objects $\Omega \in$ Objs with a list of objects $\Omega ::= \{\Omega_i\}^*$. Each object $\Omega_i \in$ Obj is a quintuplet: $\Omega_i ::= \langle \alpha, \kappa, \rho, \textsc{c}, \textsc{r} \rangle$, defined as follows:

The value $\alpha \in$ OName is the identifier of the object, $\kappa \in$ CName is the name of the object class (its static type), ρ is a list of pairs (attribute, value), $\textsc{c} \in$ Clrs is a list of closures (modeling object activity) and $\textsc{r} \in$ Rqsts a list of requests to serve.

A closure $\textsc{c}_i = \langle \textsc{s}, \eta \rangle$ is defined by a sequence of statements $\textsc{s}$ of Stmt and a context η, formed by two lists of pairs: $\eta = \langle \rho_1, \rho_2 \rangle$.

The lists ρ_1 and ρ_2, respectively manage the association between formal and effective parameters and local variables and their values.

Finally, a request $\textsc{r}_i \in$ Rqst is modeled by a quadruplet $\textsc{r}_i = \langle \textsc{m}, \widetilde{\textsc{v}}, \phi, \alpha \rangle$ with $\textsc{m}$ the name of routine to serve, $\widetilde{\text{V}} = (\text{V}_1, \ldots, \text{V}_n)$ the effective parameters, ϕ the future for the value of **result** after the routine completion and α the sender identifier.

4.1.2 Futures

For modeling futures we add a new value, the awaited value ϕ, so we can distinguish between an awaited value and the effective returned value at any time.

The environment of futures $\Phi ::= \{\Phi_i\}^*$, is shared by all objects; each future $\Phi_i = \langle \phi, \text{V} \rangle$ is defined with a name $\phi \in$ FName, and a value $\text{V} \in$ Val, defined as an effective value $\underline{\text{V}} \in$ EVal (integer, boolean, reference) or a future ϕ.

4.1.3 Continuations

In a small step operational semantics, it is necessary to describe *continuations*: the actions an objet has to perform. This leads us to define new constructors.

$$\text{E} ::= \ldots \mid \text{V} \mid \;\Leftarrow\; \mid \text{E} \leadsto \text{M}(\text{E}_1, \ldots, \text{E}_n) \mid \text{E} \cdot \mathbf{clone}(\alpha)$$
$$\text{S} ::= \ldots \mid \mathbf{null} \mid \text{E} \Rightarrow \;\mid \text{E} \Rightarrow \phi \mid \mathbf{clone\_attrs}(\rho, \alpha)$$

Intuitively, $\text{E} \Rightarrow$ and $\text{E} \Leftarrow$ are used to transmit the current result between closures, in a single object; $\text{E} \Rightarrow \phi$ returns the result of a service between objects; $\text{E} \leadsto \text{M}(\text{E}_1, \ldots, \text{E}_n)$ is used for modeling the evaluation of parameters (transmitted by copy or by reference); we also use the notation $\widetilde{\text{E}}$ for the list of parameters $(\text{E}_1, \text{E}_2, \ldots, \text{E}_n)$; **null** is the statement which does nothing; $\text{E} \cdot \mathbf{clone}(\alpha)$ makes a copy of the expression E and $\mathbf{clone\_attrs}(\rho, \alpha)$ makes a copy of each attribute value in ρ.

4.2 Inheritance and Dynamic Binding

We do not build, for every class, an intermediate data structure for all inherited features, attributes, etc. Instead, we use the source program, looking for information in the current class, or in ancestors (see [7] for more details). From the semantics of Eiffel, we use the following predicates:

- feature$(\text{M}, \kappa, \Pi) =$
  $\text{M}'(Decs_1)$: T **is local** $Decs_2$ **do** S_M **end;**
  determines in the program Π, the effective declaration of the routine M according to possible renamings and redefinitions (M' is the version of the routine named M in the class κ).
- $bind(Decs, \widetilde{\text{V}}) = \rho_1$
  builds the ρ_1 environment: the list of pairs (formal parameter, value) where each value comes from the list of effective parameters $\widetilde{\text{V}}$.
- $init(Dec) = \rho_2$
  builds the ρ_2 environment: the list of pairs (local variable (including **result**), initial value) where each initial value depends on the type of the variable (**0** for integer, **void** for references, etc.).

We also define a new predicate *inheritprocess*(κ, Π) simply based on the existing predicate $inherit(\kappa, \kappa')$ which states whether a class κ inherits from class κ'; it will be used, at object creation, to specify if an object is active.

4.3 The Transition System

Our operational semantics is based on a transition system whose states represent global configurations of a set of objects. The execution of a program is modeled by a sequence of configurations, starting from a suitable initial configuration. A global configuration changes into another global configuration when a global action is applied on the whole system of objects. A global action is for instance a communication between objects or the creation of a new object. A global action is defined in terms of a local action in a given object. This object is determined arbitrarily in the set of objects (see Section 5.4 for more details) to perform some activity (this object is actually working during one elementary interleaving transition). A local action of the working object may be an internal action or an interaction with another object.

The semantics of a program is given by a transition system which represents all its possible executions. A global configuration of a system is a triplet $\langle \Pi, \Phi, \Omega \rangle$ where Π is the source program, a list of classes, Φ is the environment of futures and Ω is the list of objects.

The transitions between configurations are given with rules which describe global actions of the system. These rules are of the form: $\langle \text{System, Ftrs, Objs} \rangle \longrightarrow \langle \text{System, Ftrs, Objs} \rangle$ which is interpreted as follows:

A system in a configuration $\langle \Pi, \Phi, \Omega \rangle$ *performs a global action and changes its configuration into* $\langle \Pi, \Phi', \Omega' \rangle$.

Note that during execution, the system Π is never modified. Execution of a system is a sequence of transitions:

$\langle \Pi,\, \Phi_0,\, \Omega_0 \rangle \longrightarrow \langle \Pi,\, \Phi_1,\, \Omega_1 \rangle \longrightarrow \cdots$

where the initial configuration is given by:

$$\langle \Pi, \Phi_0, \Omega_0 \rangle = \langle \Pi,\, [\,],\, \{\langle \alpha_0, \text{ROOT},\, \rho_0,\, \{\langle \textbf{create},\, \langle [\,],\, \textit{init}(\text{Decs}) \rangle \rangle\}, [\,] \rangle\} \rangle$$

In the initial configuration, the list of objects contains one object (the root object, instance of the root class of Π). Attribute values are initialized in ρ_0. The root object has to execute its **create** predefined routine and has no request to serve.

4.3.1 Global Actions

The global actions of systems are given in Figure 3; they show how a configuration evolves according to local actions of objects.

Rule (G1) describes how the global configuration may change as a result of an internal action in one object. It reads as follows, from bottom to top: an Eiffel// system Π, a list of futures Φ, a list of objects Ω with a selected object $\langle \alpha, \kappa, \rho, \text{C}, \text{R} \rangle$ to be activated, are changed into a new configuration (right of the arrow) where only ρ', C', R' (attributes, closures, and requests) are possibly different *if and only if* the top part of the rule can be proved, i.e. the selected object can perform an internal action and change its state with the values ρ', C', R'.

Rules (G2) and (G3) deal with the creation of objects: a new identifier is generated and a new process (resp. object) is created depending on whether the definition class κ_1 of the object to create inherits from the PROCESS class (the predicate ss, when applied to an object identifier, returns its root process).

Rule (G4) describes the deep copy of a passive object γ asked by the α object (copy of the object itself and all the referenced passive objects). A new object is created in the subsystem of object β (given as a parameter of the action `cln`), with a new identifier λ. The fields Cname, Attrs and Rqsts of λ are those of γ. The activity of the λ object starts by making a copy of its attributes (**clone_attrs**(ρ_γ, β)), and a future continuation ($\lambda \Rightarrow \phi$) for the object α. Then, α waits for the future ϕ (see `(I15)`), and receives the identifier λ.

Rule (G5) describes the synchronization between `send`$(\text{M}, \widetilde{\text{V}}, \phi, \beta)$ (action performed by α) and `rcv`$(\alpha, \text{M}, \widetilde{\text{V}}, \phi)$ (action performed by β). All communications are semantically specified with a new future ϕ which handles the wait-by-necessity. A synchronous communication is then modeled with an immediate wait on the future.

$$\frac{\Pi,\, \Phi \vdash \langle \alpha,\, \kappa,\, \rho,\, \text{C},\, \text{R} \rangle \xrightarrow{\texttt{int}} \rho',\, \text{C}',\, \text{R}'}{\langle \Pi,\, \Phi,\, \Omega \cup \{\langle \alpha,\, \kappa,\, \rho,\, \text{C},\, \text{R} \rangle\} \rangle \longrightarrow \langle \Pi,\, \Phi,\, \Omega \cup \{\langle \alpha,\, \kappa,\, \rho',\, \text{C}',\, \text{R}' \rangle\} \rangle} \quad (\texttt{G1}\text{:internal})$$

$$\frac{\Pi,\, \Phi \vdash \langle \alpha,\, \kappa,\, \rho,\, \text{C},\, \text{R} \rangle \xrightarrow{\texttt{new}(\kappa_1, \beta)} \rho',\, \text{C}',\, \text{R}'}{\begin{array}{l}\langle \Pi,\, \Phi,\, \Omega \cup \{\langle \alpha,\, \kappa,\, \rho,\, \text{C},\, \text{R} \rangle\} \rangle \longrightarrow \\ \qquad \langle \Pi,\, \Phi,\, \Omega \cup \{\langle \alpha,\, \kappa,\, \rho',\, \text{C}',\, \text{R}' \rangle, \langle \beta,\, \kappa_1,\, [\,],\, [(\text{S},\, ())],\, [\,] \rangle\} \rangle\end{array}} \quad (\texttt{G2}\text{:process creation})$$

provided $\textit{inheritprocess}(\kappa_1, \Pi)$, $\text{gen}(\beta)$
where $\textit{feature}(\texttt{live}, \kappa_1, \Pi) = \texttt{live} \;:\; \text{T}$ **is do** S **end**, $ss(\beta) = \beta$

$$\frac{\Pi,\, \Phi \vdash \langle \alpha,\, \kappa,\, \rho,\, \text{C},\, \text{R} \rangle \xrightarrow{\texttt{new}(\kappa_1, \beta)} \rho',\, \text{C}',\, \text{R}'}{\begin{array}{l}\langle \Pi,\, \Phi,\, \Omega \cup \{\langle \alpha,\, \kappa,\, \rho,\, \text{C},\, \text{R} \rangle\} \rangle \longrightarrow \\ \qquad \langle \Pi,\, \Phi,\, \Omega \cup \{\langle \alpha,\, \kappa,\, \rho',\, \text{C}',\, \text{R}' \rangle, \langle \beta,\, \kappa_1,\, [\,],\, [\,],\, [\,] \rangle\} \rangle\end{array}} \quad (\texttt{G3}\text{:object creation})$$

provided *not* $\textit{inheritprocess}(\kappa_1, \Pi)$, $\text{gen}(\beta)$, where $ss(\beta) = ss(\alpha)$

$$\frac{\Pi,\, \Phi \vdash \langle \alpha, \kappa_\alpha, \rho_\alpha, \text{C}_\alpha, \text{R}_\alpha \rangle \xrightarrow{\texttt{cln}(\gamma, \phi, \beta)} \rho'_\alpha, \text{C}'_\alpha, \text{R}'_\alpha}{\begin{array}{l}\langle \Pi,\, \Phi,\, \Omega \cup \{\langle \alpha,\, \kappa_\alpha,\, \rho_\alpha,\, \text{C}_\alpha,\, \text{R}_\alpha \rangle\} \rangle \longrightarrow \\ \langle \Pi,\, \langle \phi,\, \phi \rangle \cdot \Phi,\, \Omega \cup \{\langle \alpha,\, \kappa_\alpha,\, \rho'_\alpha,\, \text{C}'_\alpha,\, \text{R}'_\alpha \rangle, \langle \lambda,\, \kappa_\gamma,\, \rho_\gamma,\, \text{C},\, \text{R}_\gamma \rangle\} \rangle\end{array}} \quad (\texttt{G4}\text{:clone})$$

provided
$\exists \langle \gamma, \kappa_\gamma, \rho_\gamma, \text{C}_\gamma, \text{R}_\gamma \rangle \in \Omega \cup \{\langle \alpha, \kappa_\alpha, \rho_\alpha, \text{C}_\alpha, \text{R}_\alpha \rangle\}$, $\text{gen}(\lambda)$, $\text{gen}(\phi)$
where $\text{C} = \langle \textbf{clone\_attrs}(\rho_\gamma, \beta);\; \lambda \Rightarrow \phi,\, () \rangle$, $ss(\lambda) = ss(\beta)$

$$\frac{\begin{array}{l}\Pi, \Phi \vdash \langle \alpha, \kappa_\alpha, \rho_\alpha, \text{C}_\alpha, \text{R}_\alpha \rangle \xrightarrow{\texttt{snd}(\beta, \text{M}, \widetilde{\text{V}}, \phi)} \rho'_\alpha, \text{C}'_\alpha, \text{R}'_\alpha \\ \Pi, \Phi \vdash \langle \beta, \kappa_\beta, \rho_\beta, \text{C}_\beta, \text{R}_\beta \rangle \xrightarrow{\texttt{rcv}(\text{M}, \widetilde{\text{V}}, \phi, \alpha)} \rho'_\beta, \text{C}'_\beta, \text{R}'_\beta\end{array}}{\begin{array}{l}\langle \Pi, \Phi, \Omega \cup \{\langle \alpha, \kappa_\alpha, \rho_\alpha, \text{C}_\alpha, \text{R}_\alpha \rangle, \langle \beta, \kappa_\beta, \rho_\beta, \text{C}_\beta, \text{R}_\beta \rangle\} \rangle \longrightarrow \\ \qquad \langle \Pi, \Phi', \Omega \cup \{\langle \alpha, \kappa_\alpha, \rho'_\alpha, \text{C}'_\alpha, \text{R}'_\alpha \rangle, \langle \beta, \kappa_\beta, \rho'_\beta, \text{C}'_\beta, \text{R}'_\beta \rangle\} \rangle\end{array}} \quad (\texttt{G5}\text{:comm.})$$

provided $\text{gen}(\phi)$, where $\Phi' = \langle \phi,\, \phi \rangle \cdot \Phi$

$$\frac{\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \text{C}, \text{R} \rangle \xrightarrow{\texttt{rep}(\phi, \text{V})} \rho', \text{C}', \text{R}'}{\langle \Pi, \Phi, \Omega \cup \{\langle \alpha, \kappa, \rho, \text{C}, \text{R} \rangle\} \rangle \longrightarrow \langle \Pi, \Phi[\phi \leftarrow \text{V}], \Omega \cup \{\langle \alpha, \kappa, \rho', \text{C}', \text{R}' \rangle\} \rangle} \quad (\texttt{G6}\text{:result})$$

Figure 3: Semantic Rules for Global Actions

Finally, Rule (G6) assigns the result value V to the associated future ϕ in the futures environment.

4.3.2 Local Actions

Due to space limitation, we can not present the full description of local actions but focuse on the most specific aspects. The reader can refer to [8] for more details. Semantic rules describing local actions are of the form:

System, Ftrs $\vdash$ Obj $\xrightarrow{\texttt{local}}$ Pairs, Clrs, Rqsts
which is interpreted as follows:

An object performs some local action and modifies its configuration (attributes, closures, requests).

Semantics of Statements
We focus in Figure 4 on the rules for assignment, selection, and message passing, which can be decomposed into elementary actions (both internal to an object, and serving a communication between two objects).

For instance, the rule *assign* describes an elementary step during the execution of an assign statement: the right-hand side of the assign is still an expression to be evaluated E. An elementary step consists in evaluating one step of E, which gives a new expression E', so the continuation is an assign statement with E' in the right-hand side.

$$\frac{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{E},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathrm{E}',\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'}{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{Y} := \mathrm{E},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathrm{Y} := \mathrm{E}',\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'} \quad (\mathit{assign})$$

$$\frac{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{E},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathrm{E}',\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'}{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathbf{if}\ \mathrm{E}\ \mathbf{then}\ \mathrm{S}_1\ \mathbf{else}\ \mathrm{S}_2\ \mathbf{end},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathbf{if}\ \mathrm{E}'\ \mathbf{then}\ \mathrm{S}_1\ \mathbf{else}\ \mathrm{S}_2\ \mathbf{end},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'} \quad (\mathit{selection})$$

$$\frac{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{E},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathrm{E}',\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'}{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{E} \cdot \mathrm{M}(\widetilde{\mathrm{E}}),\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathrm{E}' \cdot \mathrm{M}(\widetilde{\mathrm{E}}),\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'} \quad (\mathit{call\_object})$$

$$\frac{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{E},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathrm{E}',\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'}{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \beta \leadsto \mathrm{M}(\widetilde{\mathrm{E}}),\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \beta \leadsto \mathrm{M}(\widetilde{\mathrm{E}'}),\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'} \quad (\mathit{call\_param})$$

$$\frac{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{E},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathrm{E}',\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'}{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{E} \Rightarrow \phi,\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathrm{E}' \Rightarrow \phi,\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'} \quad (\mathit{result})$$

Figure 4: Rules for statements

Semantics of Internal Actions
Axioms for internal actions are given in Figure 5. Axioms for evaluation (resp. assignment) of an attribute (`I4`) (resp. (`I17`)) are straightforward. The axiom (`I7`) describes the evaluation of a future ϕ. This axiom does not apply if the value of ϕ is the value ϕ itself; the future is unknown. In this case, the absence of any action for the object (other than receipt of requests or results) models the wait-by-necessity.

Axioms (`I8`) and (`I9`) describe the local call of a routine. In this case, a new closure is created with the routine body plus the continuation **result** $\Rightarrow$ with its context. After the completion of the routine, the result value will be sent to the next closure.

The axiom (`I10`) describes the explicit **Wait**: when the value of an expression is an effective value $\underline{\mathrm{V}}$ the wait terminates and the value is returned.

Axioms (`I11`) to (`I15`) present the transmission of parameters, depending on the type of the call. For a synchronous call (`I11`), parameters are passed by reference. For an asynchronous call (`I12`), parameters are passed by copy. Axioms (`I13`) to (`I15`) deal with the actual copy of parameters depending on whether it is a constant (`I13`), a process (`I14`), or a passive object (`I15`) for which a deep copy is required.

The axiom (`I21`) expresses the creation of an object of type κ_1 (the type of Y). The new object is referenced by Y (as expressed in the assignment statement returned as a continuation). The axiom (`I23`) is a terminal action for sending back a result (the continuation is **null**).

Semantics of Communications
Figure 6 gives axioms for the description of communications. Axioms (`C1`) and (`C2`) may be applied when all subexpressions have been evaluated and respectively describe a call between objects of two

–Axioms for expressions

$$\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle I,\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\texttt{int}} \rho,\ \langle \rho[\mathrm{A}],\ \eta\rangle \cdot \mathrm{C}, \mathrm{R} \qquad (\texttt{I4})$$

$$\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \phi,\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\texttt{int}} \rho,\ \langle \Phi[\phi],\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R} \qquad (\texttt{I7})$$
$$\text{provided } \Phi[\phi] \neq \phi$$

$$\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \alpha \rightsquigarrow \mathrm{M}(\widetilde{\mathrm{V}}),\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\texttt{int}} \qquad (\texttt{I8})$$
$$\rho,\ \langle \mathrm{S}_M\, ;\, \mathbf{result} \Rightarrow,\ \eta_M\rangle \cdot \langle \Leftarrow,\ \eta\rangle \cdot \mathrm{C}, n\,\mathrm{R}$$

where
$feature(\mathrm{M}, \kappa, \Pi) = \mathrm{M}'(Decs_1) : \mathrm{T}$ **is local** $Decs_2$ **do** S_M **end;**
$bind(Decs_1, \widetilde{\mathrm{V}}) = \rho_1, init(Decs_2) = \rho_2, \eta_M = \langle \rho_1, \langle \mathbf{result}, \mathbf{void}\rangle \cdot \rho_2\rangle$

$$\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{V} \Rightarrow,\ \eta\rangle \cdot \langle \Leftarrow,\ \eta_1\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\texttt{int}} \rho,\ \langle \mathrm{V},\ \eta_1\rangle \cdot \mathrm{C},\ \mathrm{R} \quad (\texttt{I9})$$

$$\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \underline{\mathrm{V}} \cdot \mathbf{Wait},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\texttt{int}} \rho,\ \langle \underline{\mathrm{V}},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R} \quad (\texttt{I10})$$

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle \beta \cdot \mathrm{M}(\widetilde{\mathrm{E}}), \eta\rangle \cdot \mathrm{C}, \mathrm{R}\rangle \xrightarrow{\texttt{int}} \rho,\ \langle \beta \rightsquigarrow \mathrm{M}(\widetilde{\mathrm{E}}), \eta\rangle \cdot \mathrm{C}, \mathrm{R} \quad (\texttt{I11})$$
$$\text{provided } \beta \neq \mathbf{void},\ ss(\alpha) = ss(\beta)$$

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle \beta \cdot \mathrm{M}(\widetilde{\mathrm{E}}), \eta\rangle \cdot \mathrm{C}, \mathrm{R}\rangle \xrightarrow{\texttt{int}} \rho, \langle \beta \rightsquigarrow \mathrm{M}(\widetilde{\mathrm{E} \cdot \mathbf{clone}}(\beta)), \eta\rangle \cdot \mathrm{C}, \mathrm{R}$$
$$\text{provided } \beta \neq \mathbf{void},\ ss(\alpha) \neq ss(\beta) \qquad (\texttt{I12})$$

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle \mathrm{K} \cdot \mathbf{clone}(\beta), \eta\rangle \cdot \mathrm{C}, \mathrm{R}\rangle \xrightarrow{\texttt{int}} \rho, \langle \mathrm{K}, \eta\rangle \cdot \mathrm{C}, \mathrm{R} \quad (\texttt{I13})$$

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle \gamma \cdot \mathbf{clone}(\beta), \eta\rangle \cdot \mathrm{C}, \mathrm{R}\rangle \xrightarrow{\texttt{int}} \rho, \langle \gamma, \eta\rangle \cdot \mathrm{C}, \mathrm{R} \quad (\texttt{I14})$$
$$\text{provided } \gamma \neq \mathbf{void},\ ss(\gamma) = \gamma$$

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle \gamma \cdot \mathbf{clone}(\beta), \eta\rangle \cdot \mathrm{C}, \mathrm{R}\rangle \xrightarrow{\texttt{cln}(\gamma,\phi,\beta)} \rho,\ \langle \phi \cdot \mathbf{Wait}, \eta\rangle \cdot \mathrm{C}, \mathrm{R}$$
$$\text{provided } \gamma \neq \mathbf{void},\ ss(\gamma) \neq \gamma \qquad (\texttt{I15})$$

– Axioms for instructions

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle I := \mathrm{V}, \eta\rangle \cdot \mathrm{C}, \mathrm{R}\rangle \xrightarrow{\texttt{int}} \rho[I \leftarrow \mathrm{V}],\ \langle null, \eta\rangle \cdot \mathrm{C}, \mathrm{R} \qquad (\texttt{I17})$$

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle \mathrm{Y} \cdot \mathbf{create}, \eta\rangle \cdot \mathrm{C}, \mathrm{R}\rangle \xrightarrow{\texttt{new}(\kappa_1,\beta)} \rho, \langle \mathrm{Y} := \beta, \eta\rangle \cdot \mathrm{C}, \mathrm{R}$$
$$\text{where } feature(\mathrm{Y}, \kappa, \Pi) = Decs : \kappa_1,\ \mathrm{Y} \in Decs \qquad (\texttt{I21})$$

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle \mathrm{V} \Rightarrow \phi, \eta\rangle \cdot \mathrm{C}, \mathrm{R}\rangle \xrightarrow{\texttt{rep}(\phi,\mathrm{V})} \rho, \langle null, \eta\rangle \cdot \mathrm{C}, \mathrm{R} \quad (\texttt{I23})$$

Figure 5: Axioms for internal actions

different subsystems and a call between objects of the same subsystem. In both cases, a request is created and sent to the callee, and a new future ϕ represents the return value.

On the caller side, in the case of two different subsystems, an asynchronous call occurs and the caller carries on its execution. On the other hand, within a subsystem, a synchronous call is achieved: the caller immediately waits for the value of the future (explicit **Wait**).

On the callee side, when a request is received, if it comes from a different subsystem (asynchronous call, axiom (C3)), the request is appended to the current list of pending requests, and will be treated later. On the other hand, when the request comes from the same subsystem (synchronous call, axiom (C4)), it is immediately treated. This is possible because within a subsystem, at most one object is ready for execution. This technique is also used to deal with recursion within a subsystem.

$$\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho\, \langle \beta \rightsquigarrow \mathrm{M}(\widetilde{\mathrm{V}}),\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\texttt{snd}(\beta,\mathrm{M},\widetilde{\mathrm{V}},\phi)} \rho,\ \langle \phi,\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}$$
$$\text{provided } ss(\alpha) \neq ss(\beta) \qquad (\texttt{C1})$$

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle \beta \rightsquigarrow \mathrm{M}(\widetilde{\mathrm{V}}), \eta\rangle \cdot \mathrm{C}, \mathrm{R}\rangle \xrightarrow{\texttt{snd}(\beta,\mathrm{M},\widetilde{\mathrm{V}},\phi)} \rho, \langle \phi \cdot \mathbf{Wait}, \eta\rangle \cdot \mathrm{C}, \mathrm{R}$$
$$\text{provided } ss(\alpha) = ss(\beta) \qquad (\texttt{C2})$$

$$\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\texttt{rcv}(\mathrm{M},\widetilde{\mathrm{V}},\phi,\beta)} \rho,\ \mathrm{C},\ \mathrm{R} \cdot \langle \mathrm{M}, \widetilde{\mathrm{V}}, \phi, \beta\rangle \qquad (\texttt{C3})$$
$$\text{provided } ss(\alpha) \neq ss(\beta)$$

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \mathrm{C}, \mathrm{R}\rangle \xrightarrow{\texttt{rcv}(\mathrm{M},\widetilde{\mathrm{V}},\phi,\beta)} \rho, \langle \mathrm{S}_M\, ; \mathbf{result} \Rightarrow \phi, \eta_{\mathrm{M}}\rangle \cdot \mathrm{C}, \mathrm{R}$$
provided $ss(\alpha) = ss(\beta)$ (C4)
where
$feature(\mathrm{M}', \kappa, \Pi) = \mathrm{M}(Decs_1) : \mathrm{T}$ **is local** $Decs_2$ **do** S_M **end;**
$bind(Decs_1, \widetilde{\mathrm{V}}) = \rho_1,$
$init(Decs_2) = \rho_2,$
$\eta_M = \langle \rho_1, \langle \mathbf{result}, \mathbf{void}\rangle \cdot \rho_2\rangle$

Figure 6: Axioms for communications

5 Semantics to Visualization

From the syntactic and semantic definition of Eiffel//, using the Centaur system and the Typol formalism, we derive an interactive environment for parallel object-oriented programming. Note that this environment is also suitable for sequential object-oriented programming and includes all of standard Eiffel (since Eiffel// semantics includes the semantics of Eiffel).

The principle which permits to go from semantics to visualization is as follows. First, the semantic structures (semantic domains) are directly used in the visualization. The list of objects, futures, and continuations (specified in section 4.1) will be directly used as an intermediate format by two visualization engines (one textual and one graphical, detailed below). Second, the semantics is equipped with notifications for the visualization engines. On appropriate semantic rules, when a rule is successfully applied (proved), the notification (if it exists) is triggered and the visualization engines become aware of some modification in the semantic structures. For instance, on rule (G1) Figure 3, the new list of objects has changed and is transmitted as a notification:

$$\frac{\Pi,\Phi\vdash\langle\alpha,\kappa,\rho,\mathrm{C},\mathrm{R}\rangle \xrightarrow{\mathtt{int}} \rho',\mathrm{C}',\mathrm{R}'}{\langle\Pi,\Phi,\Omega\cup\{\langle\alpha,\kappa,\rho,\mathrm{C},\mathrm{R}\rangle\}\rangle \longrightarrow \langle\Pi,\Phi,\Omega\cup\{\langle\alpha,\kappa,\rho',\mathrm{C}',\mathrm{R}'\rangle\}\rangle} \quad (\mathtt{G1})$$

$$\textbf{notify} \quad \Omega\cup\{\langle\alpha,\kappa,\rho',\mathrm{C}',\mathrm{R}'\rangle\}$$

Altogether, less than 10 semantic rules needed to be equipped with such notifications.

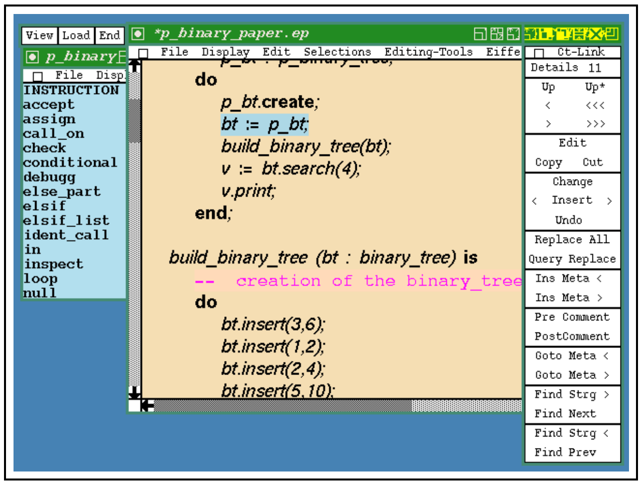
The environment obtained can be used by two kinds of users:

- novice programmers can build basic programs (without actually knowing the syntax of the language), compute their result using the generated interpreter, and visualize program execution with animation tools, including active objects;
- language designers can express and understand operational semantics of various concurrent object-oriented models.

In this section, we focus on the functional aspects of the program development and debugging environment: we present the structure editor, the interpreter, with two different granularities of interleaving. We also describe the visualization tools that allow animation and debugging of programs (textual and graphical presentation of objects, control over the execution, etc). Finally we explain how language designers can access and visualize the semantic model during execution of a given program.

5.1 Editing and Interpreting

The programming environment includes a parser and a pretty-printer which compose a structure editor (see Figure 7). This syntactic editor provides a guided editing mode (based on abstract syntax, via tree manipulation) as well as an in-line textual editing mode (based on concrete syntax, using a parsing process).



```
    do
        p_bt.create;
        bt := p_bt;
        build_binary_tree(bt);
        v := bt.search(4);
        v.print;
    end;

  build_binary_tree (bt : binary_tree) is
    --  creation of the binary_tree
    do
        bt.insert(3,6);
        bt.insert(1,2);
        bt.insert(2,4);
        bt.insert(5,10);
```

Figure 7: Editing and Visualizing a Program

The main window ("p_binary_paper.ep") shows, after parsing, a concrete representation of the abstract syntax tree of the example program presented in Figure 2. This concrete representation uses specific fonts and colors for keywords and comments. During editing, placeholders can be easily introduced (using the "Ins Meta" button in the "editing" box on the right). Placeholders can then be filled up by selection of an abstract syntax operator (the window on the left-hand side shows only possible operators of type INSTRUCTION, the current selection in the program window). The abstract syntax definition ensures that the whole abstract syntax tree for the program is syntactically correct. Finally, the editing window on the right provides help for navigation in the tree, changing the level of details, cut-and-paste operations, and so on. 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.

To trigger the interpretation of a program, we provide a specific popup menu, which is a call to the operational semantics of the language. The abstract syntax tree of the source program is transmitted from the editing server to the semantic server. The result of the execution is an abstract syntax tree modeling the final list of objects which is sent to the servers for visualization.

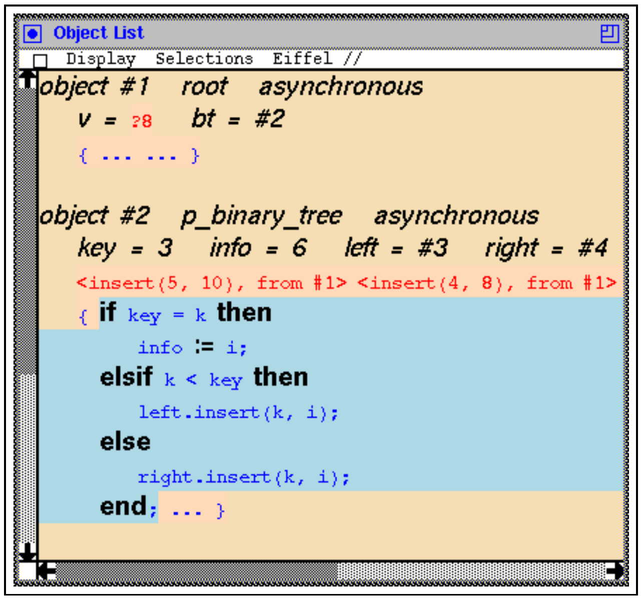
5.2 Visualization and Animation Tools

We provide two different visualizations of objects, using two visualization engines, both of them based on the semantic structure for modeling the list of objects presented in Section 4.1.

In one window, we present a textual representation of objects with their configuration: process or object, static type, current attribute values, current activity, current pending requests. Figure 8 shows the root object and a process of type P_BINARY_TREE. The `left` and `right` attribute values are references to other objects (♯3 & ♯4 are not shown here, the scroll-bar must be used). This process has currently 2 pending requests (in red, between ⟨ and ⟩). Last, its current activity (the highlighted continuation), is a `if` statement. The question mark '?' in object ♯1 is discussed in Section 5.3.

Such a presentation does not give a global view of the graph of objects. So, in another window, we show the complete topology of the system in a graphical representation. This graph is visualized thanks to the graph displaying package of Centaur [32]. 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. Two kinds of object nodes exist, distinguished with specific colors: objects in blue and processes in green. Figure 9 displays all processes during execution of the binary tree example.

A zooming process makes it possible to show, on request, attribute values of a given object node.



```
object #1   root   asynchronous
   v = ?8   bt = #2
   { ... ... }

object #2   p_binary_tree   asynchronous
   key = 3   info = 6   left = #3   right = #4
   <insert(5, 10), from #1> <insert(4, 8), from #1>
   { if key = k then
        info := i;
     elsif k < key then
        left.insert(k, i);
     else
        right.insert(k, i);
     end; ... }
```

Figure 8: Textual Visualization of Objects

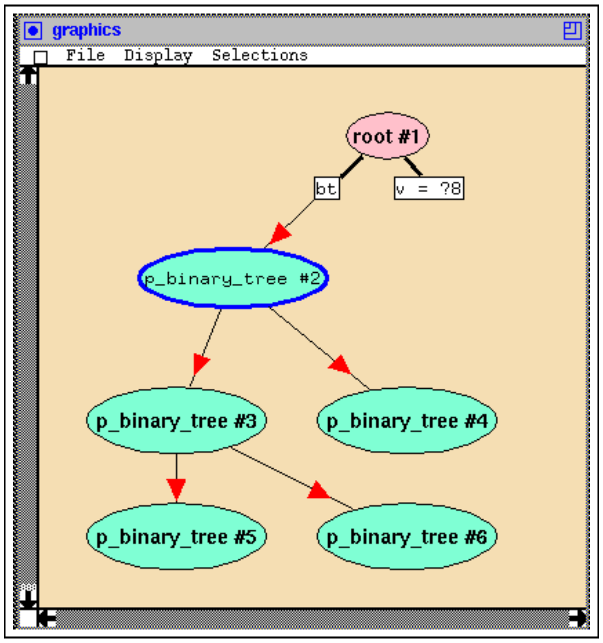


Figure 9: Graphical Visualization of Objects

This is done with an expansion of an object node into a subgraph made up with the object node itself and attribute nodes (containing attribute name and value, possibly a reference). References noted in Figure 8 are shown with arrows between the root process and its two sons (P_BINARY_TREE ♯3 & ♯4).

We also provide a selection mechanism in the graphical representation for modeling subsystems. Selecting an object highlights its root process (possibly itself) and related passive objects. This may be useful for a better understanding of concurrency aspects (communications, transmission of parameters, etc). For readability, no other information (such as pending requests, or activity) is available in the graphical representation.

These two visualizations are based on the same semantic structure modeling the list of objects and can be displayed after execution of the program. This technique also provides animation to visualize objects *during* program execution, and so have a better understanding of the behavior of the program.

One thing we chose to highlight is the current active object and activity, in the textual presentation (with a particular selection) and in the graphical presentation (with a thick blue border). This is done with a notification expressed in the semantic specification as a side effect of the application of rule (`G1`). This notification is sent from the semantic server to the editing server each time an object is selected for execution.

To visualize objects during execution and provide animation, it is also necessary to show the changes that occur (creation of a new object, update of an attribute value, etc). This is done with a notification in the appropriate semantic rule. For instance, axiom (`I17`) dealing with the assignment of an attribute, when applied, notifies the editing server that a change has occurred. Then, it is up to the textual and graphical presentations to show the change with an incremental redisplay (so the user is not bothered with screen flashes) and permits to focus the attention on the change (a new value, a new object) which is highlighted.

5.3 Visualizing Synchronizations

Our environment makes it possible to visualize automatic futures which are data-driven synchronizations. An example, from the binary tree example in Figure 2, is illustrated in Figures 8 and 9, with a question mark '`?8`' as the value of the attribute `v`. When the root object executes the statement `v:= bt.search(2)`, `bt` refers to a process object of type P_BINARY_TREE. The rule (`G5`) for communication is applied. Then the environment of futures Φ is updated by a new pair $\langle \phi, \phi \rangle$ and the future ϕ is transmitted to `bt` (which will later update on the future with the result). In the root object, the value ϕ (identified by the number 8), is assigned to the attribute `v`).

Execution continues, with the statement `v.print`, which starts by the evaluation of the attribute `v` (axiom (`I4`)). Because the value of `v` is the future ϕ, the continuation $\phi \cdot$`print` will be returned. The axiom for the evaluation of a future (`I7`) then may be applied, depending on whether or not the side condition of the axiom is verified. The value of the future ϕ may still be ϕ, otherwise, the value is an effective value (the integer 4). In the first case the root object is then waiting for the value of `v` and the execution continues with other processes. In the second case, the value is returned and the continuation is `4.print`.

5.4 Termination and Deadlock

Because operational semantics simulates parallelism with non-deterministic interleaving, there is only one working object at a time, chosen among all active objects. We use the following terminology for objects:

active: has an activity to complete (non-empty list of closures).

terminated: has no activity to carry on (empty list of closures).

waiting: is currently waiting either for the return of a future value (wait-by-necessity) or for a request (`wait_a_request` primitive).

In the initial configuration, only one object (root) exists and is active. To decide which object will proceed its execution (for one elementary step), we arbitrary (non deterministically) choose among active objects (see Section 5.7 for other alternatives). Activation and deactivation of objects occur, according to the following principles:

- when an object is created, it is activated with the **create** routine as continuation;
- after one transition step, one active object is *deactivated* if either this object is blocked, waiting for a request, or this object attempts to access the value of a future ϕ which is not yet returned (failure of axiom (`I7`)); in both cases, the object becomes waiting;
- after one transition step, a waiting object is *activated* if this object received a request (rule (`G5`) applies) or if this object received the value associated to a future ϕ (rule (`G6`)) applies).

Note that the deactivation of an object permits to handle waits without consuming cputime (*passive wait*).

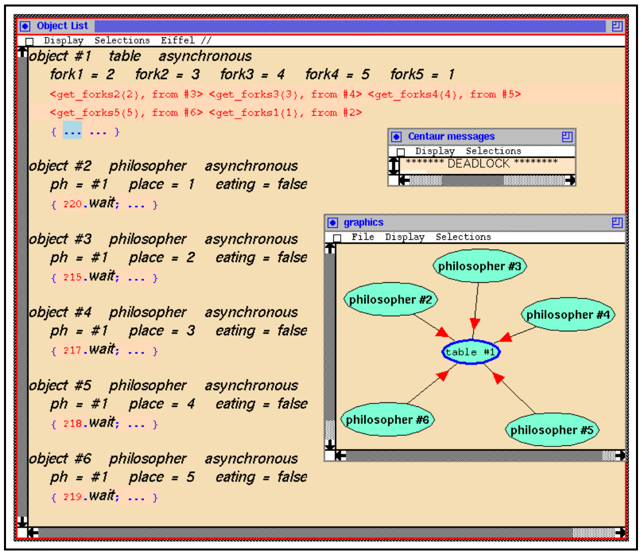
With this terminology, we can state termination and deadlock properties. An execution:

terminates: when every object is terminated;

deadlocks: if there is no more active object and at least one waiting object on a future.

Our semantic-based environment handles such properties and reports a message during execution when a deadlock is detected. Figure 10 illustrates the deadlock detection in a classical problem of resource allocation: the philosophers. In this version of the philosophers, as shown in the figure, it is possible that each philosopher grabs a fork and waits for the second one (the continuation is stopped on a future: `?i.wait`), and a deadlock occurs.

However, the fact that a program terminates normally (without deadlock) during one execution does not prevent from a deadlock occurring in another execution. To statically detect that a program is deadlock-free, we would have to consider all possible executions (with the combinatorial explosion problem) and make sure that no execution can lead to a deadlock.



```
object #1   table   asynchronous
   fork1 = 2   fork2 = 3   fork3 = 4   fork4 = 5   fork5 = 1
   <get_forks2(2), from #3> <get_forks3(3), from #4> <get_forks4(4), from #5>
   <get_forks5(5), from #6> <get_forks1(1), from #2>
   { ... ... }

object #2   philosopher   asynchronous
   ph = #1   place = 1   eating = false
   { ?20.wait; ... }

object #3   philosopher   asynchronous
   ph = #1   place = 2   eating = false
   { ?15.wait; ... }

object #4   philosopher   asynchronous
   ph = #1   place = 3   eating = false
   { ?17.wait; ... }

object #5   philosopher   asynchronous
   ph = #1   place = 4   eating = false
   { ?18.wait; ... }

object #6   philosopher   asynchronous
   ph = #1   place = 5   eating = false
   { ?19.wait; ... }
```

```
******* DEADLOCK ********
```

Figure 10: Detecting a deadlock

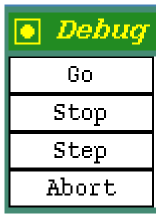
5.5 Controlling the execution

To control the execution, it is necessary to suspend (and resume) the semantic server on user request. This is done with the definition of a communication protocol between the editing server (user interactions) and the semantics server (program execution). Each transition of the system is conditioned by the reception of a message from the editing server. On the other hand, messages from the semantic server can be emitted to the editing server (to give information on the execution progress).

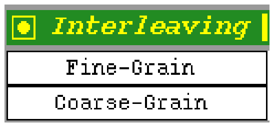
With these communications, it is quite straightforward to add a debug box to control execution (see Figure 11.a). Four buttons are provided in this tool:

- go: execution resumed without stopping;
- stop: execution stopped in the next configuration of the transition system;
- step: execution resumed for one transition (application of one global rule of Figure 3);
- abort: execution aborted.

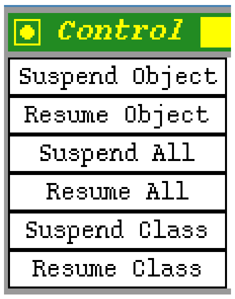
So, during the interpretation of a program, one can stop the execution, inspect the current state of the system (attribute values, requests, activities of objects), and resume execution, step by step, or continuously. Nothing in our approach prevents us from changing attribute values during execution; this possibility will be added in the future.



(a) Global control



(b) Control of granularity



(c) Control of interleaving

Figure 11: Control over the execution

5.6 Changing the granularity

Our semantics is based on a very fine-grain interleaving: every statement is decomposed into a large number of elementary steps (local actions, see Section 4.3.2); of course, a method execution is not atomic. This means that the number of system configurations during execution is important (e.g. 551 transitions for the binary tree example).

This kind of granularity is useful when the user wants to see the detailed execution, inside an object or a process for instance. However, this granularity might be too fine in some cases, especially when the user is interested into the global behavior of the system, and interactions between processes. For this reason, we also provide a coarse-grain interleaving, which is handled just by providing a new set of rules for global actions; these rules are as expressed in Figure 3 except for the rule (`G1`), to be replaced with the following rule (`G1'`):

$$\frac{\Pi,\Phi \vdash \langle \alpha,\kappa,\rho,\mathrm{C},\mathrm{R}\rangle \xrightarrow[*]{\texttt{int}} \rho',\mathrm{C}',\mathrm{R}'}{\langle \Pi,\Phi,\Omega \cup \{\langle \alpha,\kappa,\rho,\mathrm{C},\mathrm{R}\rangle\}\rangle \longrightarrow \langle \Pi,\Phi,\Omega \cup \{\langle \alpha,\kappa,\rho',\mathrm{C}',\mathrm{R}'\rangle\}\rangle} \quad (\texttt{G1'})$$

Previously, atomic actions were object creation, object copy, communication, or an elementary internal action (every action dealing with only one object). In this coarse-grain version of the semantics, a suite of internal actions can be executed without interleaving until a global action involving two objects is reached. This is expressed by adding the transitive closure ($*$) of the internal action relation ($\xrightarrow{\texttt{int}}$) in rule (`G1'`).

The number of global configurations during execution is considerably reduced in coarse-grain execution (e.g. 73 transitions for the binary tree example instead of 551); note that method execution is still not atomic, but the interleaving is of a coarser grain. These two different sets of rules for global actions are accessible via two distinct entry points. One can then choose between these two modes of interleaving as shown with the menu of Figure 11.b.

5.7 Exploring the interleaving space

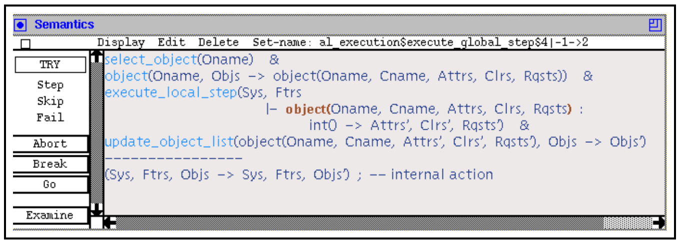
The selection of the next object to execute is, by default, arbitrary in the set of active objects.

This random selection can be changed on the programmer's request, after stopping the execution. This is done via a menu (see Figure 11.c); the user can decide to *suspend* (or *resume*) a given object, objects of a given class, or all objects. When clicking on object entries, the user is asked to select an object (in the textual or graphical presentation). When clicking on class entries, the user is asked to select a class in the program source.

Any action has a straightforward effect on the status of (individual or sets of) objects, switching an active object to inactive or the reverse. With this possibility, one can simulate fast or slow processes, different priorities on different families of processes, etc. For instance, one useful exploration is to suspend all objects and to resume one particular object. Thus, only this object can proceed with its activity as far as it does not access some awaited value. Thanks to this mechanism, very specific interleaving can be explored, e.g. looking for potential deadlock.

5.8 Understanding the semantics

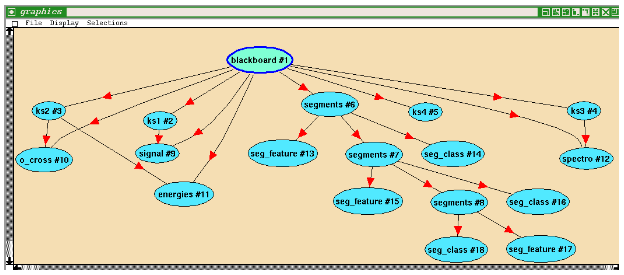
Thanks to a particular compilation mode of the semantic description, one can visualize the semantic rules: a specific window shows the inference rule currently applied (see Figure 12 for the application of rule (G1)). The `skip` button is an attempt to prove the rule without displaying the complete proof. One can control over the semantic interpretation; for instance, the `fail` button provokes a failure which has a direct impact on non-determinism and determines one particular execution path, at the meta-level of semantics. It is also possible to set breakpoints in the semantics and see the current value of a given variable (with the `examine` button).


Semantics
Display Edit Delete Set-name: al_execution$execute_global_step$4|-1->2
TRY
Step
Skip
Fail
Abort
Break
Go
Examine

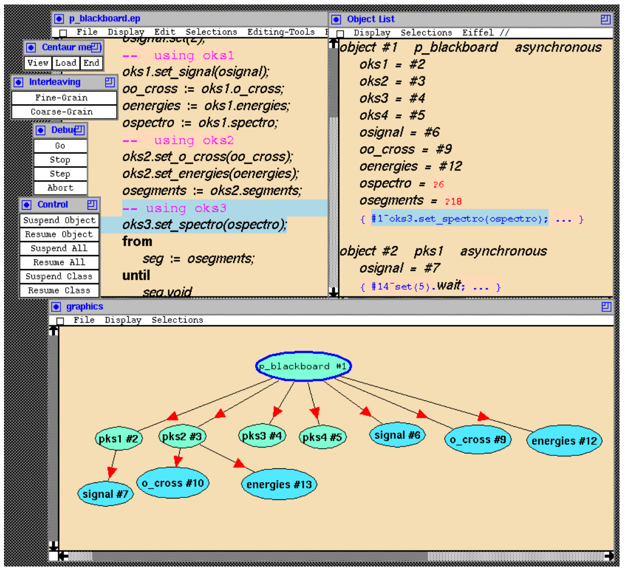
```
select_object(Oname)  &
object(Oname, Objs -> object(Oname, Cname, Attrs, Clrs, Rqsts))  &
execute_local_step(Sys, Ftrs
                        |- object(Oname, Cname, Attrs, Clrs, Rqsts) :
                               int() -> Attrs', Clrs', Rqsts')  &
update_object_list(object(Oname, Cname, Attrs', Clrs', Rqsts'), Objs -> Objs')
-----------------
(Sys, Ftrs, Objs -> Sys, Ftrs, Objs') ; -- internal action
```

Figure 12: Visualizing the Semantics

To conclude this section and as a general illustration of our environment, we show the sequential execution of an application in speech recognition [16] in Figure 13.a, and a parallel execution in Figure 13.b. We can deduce from both figures that the four knowledge sources became active objects and that some passive objects were duplicated (e.g. `signal`) because of the construction of several subsystems (no shared objects).



(a) Sequential Execution



```
osignal.set(2);
--  using oks1
oks1.set_signal(osignal);
oo_cross := oks1.o_cross;
oenergies := oks1.energies;
ospectro := oks1.spectro;
--  using oks2
oks2.set_o_cross(oo_cross);
oks2.set_energies(oenergies);
osegments := oks2.segments;
-- using oks3
oks3.set_spectro(ospectro);
from
    seg := osegments;
until
    seg.void
```

```
object #1   p_blackboard   asynchronous
   oks1 = #2
   oks2 = #3
   oks3 = #4
   oks4 = #5
   osignal = #6
   oo_cross = #9
   oenergies = #12
   ospectro = ?6
   osegments = ?18
   { #1~oks3.set_spectro(ospectro); ... }

object #2   pks1   asynchronous
   osignal = #7
   { #14~set(5).wait; ... }
```

(b) Parallel Execution

Figure 13: A Speech Recognition Application

6 Discussion

The technique we used, as presented in Sections 4 and 5, offers two main advantages:

(1) the environment generated is general enough so that programmers can develop and debug object-oriented programs in a sequential as well as in a concurrent framework;
(2) visualization tools are based on the semantic description of the concurrent object-oriented model, so the programmer is not required to instrument his programs to get a visualization of the execution.

If we take as a reference criteria proposed in the taxonomy of program visualizations [58], our system can be qualified as following. All aspects of the program, the *scope* criterion in [58], are visualized (code, data and control state, behavior): the source code is displayed and animated in one window, objects are displayed and animated in other windows. The *level of abstraction* is based on direct and structural representations: references are abstracted as an arc in the graph of objects. The *specification method* relies on logical predicates (as in Pavane [22]) at the semantics level (not at the code level). The *graphical interface* provides simple objects, events, multiple worlds, and control interaction: the graph server we use not only provides an abstract representation of objects (an output) but it also reacts to user actions (input): select an object, move an object, zoom, etc. Last, the *presentation of the visualization* is analytical and explanatory: we visualize concurrent computations as they occur, and also we can focus the attention of the user on a particular event (for instance, a synchronization is shown when the computation accesses an awaited value presented as a red question mark '?').

Agha et al. suggested to use predicate transition nets [44], and defined the so-called causal interaction model which captures causal behavior and coordination between actors [2]. In the latter work, one of the key points is the consistency between the events as they occur and the visualization; the authors developed a model that preserves this consistency (causal connection restriction). We do not have to deal with such questions because a semantic-based visualization is by definition consistent with the interpretation of the semantics. For the same reason, we do not need observers and coordinators, everything is centralized in the semantics definition and interpretation. The two approaches are rather complementary: [2] well-adapted for observation in situ and optimization, our approach for investigation of all possible behaviors and formal verifications.

In [46], Nierstrasz develops an executable notation, Abacus, allowing the specification of various concurrent object languages. The intended goal is to offer a generic platform; as an illustration, a specification of SAL [1] (a Simple Actor Language) is described within Abacus, and SAL programs can be interpreted using an Abacus to Prolog translation. Techniques similar to those described here could be used to provide Abacus with graphical visualization – a semantics of Abacus would have to be defined in Typol, leading to both genericity on the concurrent model, and non instrumented graphical visualization. However, since Abacus modeling occurs in term of agents and events which are composed to specify the semantics of a particular actor model, a general purpose visualization would reflect those building bricks. While interesting when designing and comparing concurrent models, they might not be relevant when experimenting with one given system[3]. Thus, an interesting direction for improvement might consist to add to the specification of a system, directly within the Abacus notation for instance, the necessary specification of how to visualize the concurrent language being modeled. This extra specification would provide an abstraction, relevant to the user of a concurrent language, over the agents and events being used to model it.

7 Conclusion and Future Work

In this paper, we presented how we can build a graphical visualization environment from an operational semantics of a (concurrent) object-oriented language. The visualization is not obtained from code instrumentation but automatically, using the semantic description. The graphical environment focuses on objects and their interactions (object topology, attribute values, concurrent activities, subsystems, synchronizations), provides a set of primitives for controlling and probing the execution (granularity of interleaving, step-by-step execution, control over the interleaving), and detects deadlock configurations when they occur. Due to non-determinism of concurrency, it is crucial to provide the user with the possibility to investigate the interleaving space of all possible executions.

An important issue with visualization environment is scalability. Based on a formal semantics, our approach might raise some concerns. However,

[3]A similar phenomenon occurs when modeling parallel languages with the π-calculus: far too many agents are generated, not well representing the actor structure of a program.

the technique is viable, and our current system is actually operational; applications with more than 600 objects were handled and visualized graphically, and the execution speed is such that it is often necessary to slow down interpretation for the sake of visualization. To further improve these aspects and enable the user to apprehend complex systems, several paths are possible. Regarding visualization, a solution is to ask the user extra information in order to customize the layout. A first possibility consists in giving a partial view of the graph of objects (only processes, only objects from a given class, etc.); this solution is quite straightforward to add to our system, and we are currently working on that aspect. More sophisticated strategies, that shows an abstraction of the system topology, are also considered. Within our semantic framework, we see two possibilities for the specification of this visualization information: at the formal level using semantics rules, or at the target language level, writing visualization classes. These two options are probably complementary since they do not concern the same kind of user.

Another important improvement would be to provide a framework generic on the programming model, and especially the model used for concurrency. One possibility would be to use the approach discussed at the end of Section 6. Regarding visualization, if one describes the semantics of another model of concurrency, for instance using quasi-parallel and parallel processes, the visualization would directly follow from the new semantics, i.e. several continuations would be added to each active object.

Finally, expressing the behavior of a parallel system with transitional semantics inherently represents all possible executions, which can then be used to study program properties such as absence of deadlock, liveness, or equivalence using classical techniques based for instance on traces [30] or bisimulation [42]. A semantic-based visualization should permit to provide an integrated environment where users can both visualize program execution and study formal properties.

References

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

[2] G. Agha and M. Astley. A visualization model for concurrent systems. *International Journal of Information Science, Elsevier*, 1996. To appear.

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

[4] P. America. Inheritance and subtyping in a parallel object-oriented language. In *Proc. ECOOP '87*, LNCS 276, pages 234–242, Paris, France, June 1987.

[5] P. America, J. D. Bakker, J. N. Kok, and J. Rutten. Operational Semantics of a Parallel Object-Oriented Language (POOL). In *Proc. of the 13th Symposium on Principles of Programming Languages*, 1986.

[6] P. America, J. D. Bakker, J. N. Kok, and J. Rutten. Denotational Semantics of a Parallel Object-Oriented Language (POOL). *Information and Computation 83, 152-205*, 1989.

[7] I. Attali, D. Caromel, and S.O. Ehmety. A Natural Semantics for the Eiffel Dynamic Binding. To appear in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1996.

[8] I. Attali, D. Caromel, and S.O. Ehmety. A Natural Semantics for the Eiffel// Language. Research Report 2732, INRIA, 1995.

[9] R. Balter, S. Lacourte, and M. Riveill. The Guide language. *Computer Journal*, 37(6):519–530, 1994.

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

[11] J. K. Bennett. The design and implementation of Distributed Smalltalk. In *Proc. OOPSLA '87, ACM SIGPLAN Notices 22 (12)*, pages 318–330, December 1987.

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

[13] B. Bruegge, T. Gottschalk, and B. Luo. A framework for dynamic program analyzers. In *Proc. OOPSLA '93, ACM SIGPLAN Notices 28 (10)*, pages 65–82, October 1993.

[14] M. Burnett, A. Goldberg, and T. Lewis, editors. *Visual Object-Oriented Programming, Concepts and Environments.* Manning Publications, Greenwich, CT, 1995.

[15] *Concurrent Object-Oriented Programming.* Communications of the ACM, 36 (9), 1993. Special issue.

[16] D. Caromel. Concurrency and reusability: From sequential to parallel. *Journal of Object-Oriented Programming, 3(3)*, 1990.

[17] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM, 36 (9), pages 90-102*, 1993.

[18] D. Caromel, F. Belloncle, and Y. Roudier. The C++// system. In G. Wilson and P. Lu, editors, *Parallel Programing Using C++.* MIT Press, 1996. To Appear.

[19] P. Ciancarini, K. K. Jensen, and D. Yankelevich. On the operational semantics of a coordination language. In *Proc. Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pages 77–106. Springer-Verlag, 1995.

[20] 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.

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

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

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

[24] 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.

[25] D. Gangopadhyay and S. Mitra. Objchart: Tangible specification of reactive object behavior. In *Proc. ECOOP '93*, LNCS 707, pages 432–457, Kaiserslautern, Germany, July 1993.

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

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

[28] 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.

[29] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence, 8 (3)*, 1977.

[30] C.A.R Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[31] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proc. ECOOP '91*, LNCS 512, pages 133–147, Geneva, Switzerland, July 1991.

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

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

[34] K. Kahn. ToonTalk - an animated programming environment for children. In *Proc. of the National Educational Computing Conference*, Baltimore, MD, 1995.

[35] N. Kobayashi and A. Yonezawa. Type-theoric foundations for concurrent object-oriented programming. In *Proc. OOPSLA '94, ACM SIGPLAN Notices*, October 1994.

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

[37] 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.

[38] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proc. OOPSLA '95, ACM SIGPLAN Notices 30 (10)*, pages 342–357, October 1995.

[39] ObjecTime Ltd. ObjecTime. Commercial Product http://www.objectime.on.ca, Ontario, Canada.

[40] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[41] B. Meyer. *Eiffel, the Language.* Prentice-Hall, 1992.

[42] R. Milner. *A Calculus of Communicating Systems.* Springer Verlag, LNCS 92, 1980.

[43] R. Milner, J. Parrow, and D.J. Walker. *A Calculus of Mobile Processes.* Academic Press, 1989.

[44] S. Miriyala, G. Agha, and Y. Sami. Visualizing actor programs using predicate transition nets. *Journal of Visual Languages and Computing, 3 (2)*, 1992.

[45] O. Nierstrasz. Active objects in hybrid. In *Proc. OOPSLA '87, ACM SIGPLAN Notices 22 (12)*, pages 243–253, 1987.

[46] O. Nierstrasz. A guide to specifying concurrent behaviour with abacus. In *Object Management*, pages 267–293, Centre Universitaire d'Informatique, University of Geneva, July 1990.

[47] O. Nierstrasz. The next 700 concurrent object-oriented languages – reflections on the future of object-based concurrency. Object composition, Centre Universitaire d'Informatique, University of Geneva, June 1991.

[48] O. Nierstrasz. Towards an object calculus. In *Proc. of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, LNCS 612, pages 1–20. Springer-Verlag, 1992.

[49] O. Nierstrasz. Regular types for active objects. In *Proc. OOPSLA '93, ACM SIGPLAN Notices, 28 (10)*, pages 1–15, October 1993.

[50] O. Nierstrasz, P. Ciancarini, and A. Yonezawa, editors. *Rule-based Object Coordination.* LNCS 924. Springer-Verlag, 1995.

[51] O. Nierstrasz and M. Papathomas. Viewing objects as patterns of communicating agents. In *Proc. OOPSLA/ECOOP '90, ACM SIGPLAN Notices, 25 (10)*, 1990.

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

[53] 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.

[54] 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.

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

[56] Project Technology, Inc. BridgePoint. Commercial Product http://www.projtech.com, Berkeley, California.

[57] S. P. Robertson, J. M. Carroll, R. L. Mack, M. B. Rosson,
S. R. Alpert, and J. Koenemann-Belliveau. ODE: A self-guided, scenario-based learning environment for object-oriented design principles. In *Proc. OOPSLA '94, ACM SIGPLAN Notices*, pages 51–64, October 1994.

[58] G.-C. Roman and K. C. Cox. A Taxonomy of Program Visualization Systems. *IEEE Computer*, 1993.

[59] I. Satoh and M. Tokoro. A formalism for real-time concurrent object-oriented computing. In *Proc. OOPSLA '92, ACM SIGPLAN Notices, 27 (10)*, pages 315–326, October 1992.

[60] SES, Inc. SES/objectbench, SES/workbench. Commercial Product http://www.ses.com, Austin, Texas.

[61] S. Shlaer and S. Mellor. *Object Lifecyles: Modeling the World in States.* Prentice Hall, Englewood Cliffs, NJ., 1992.

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

[63] M. Tokoro and K. Takashio. Toward languages and formal systems for distributed computing. In *Proc. of the ECOOP '93 Workshop on Object-Based Distributed Programming*, LNCS 791, pages 93–110, 1994.

[64] V. Vasconcelos and M. Tokoro. Traces semantics for actor systems. In *Proc. of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, LNCS 612, pages 141–162, 1992.

[65] V. Vasconcelos and M. Tokoro. A typing system for a calculus of objects. In *Object Technologies for Advanced Software, First JSSST International Symposium*, LNCS 742, pages 460–474, November 1993.

[66] J.-Y. Vion-Dury and M. Santana. Virtual images: Interactive visualization of distributed object-oriented systems. In *Proc. OOPSLA '94, ACM SIGPLAN Notices 29 (10)*, pages 65–84, October 1994.

[67] D. Walker. Pi-Calculus Semantics of Object-Oriented Programming Langage. *Proc. TACS'91, Springer-Verlag, LNCS Vol. 526, pages 532-547*, 1991.

[68] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Proc. OOPSLA '88, ACM SIGPLAN Notices 23 (11)*, pages 306–315, November 1988.

[69] P. Wegner. Design issues for object-based concurrency. In *Proc. of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, LNCS 612, pages 245–256, 1992.

[70] P. Wegner, G. Agha, and A. Yonezawa, editors. *Workshop on Object-Based Concurrent Programming.* ACM SIGPLAN Notices 24 (4), San Diego, April 1989.

[71] G. Wilson and P. Lu, editors. *Parallel Programing Using C++.* MIT Press, 1996. To Appear.

[72] Y. Yokote and M. Tokoro. The design and implementation of Concurrent Smalltalk. In *Proc. OOPSLA '86, ACM SIGPLAN Notices, 21 (11)*, pages 331–340, November 1986.

[73] A. Yonezawa and M. Tokoro. *Object-Oriented Concurrent Programming.* MIT Press, Cambridge, Mass., 1987.