HAL
archives-ouvertes.fr

An Operational Semantics for the Eiffel Language

Isabelle Attali, Denis Caromel, Sidi Ould Ehmety

▶ **To cite this version:**

Isabelle Attali, Denis Caromel, Sidi Ould Ehmety. An Operational Semantics for the Eiffel Language. RR-2732, INRIA. 1995. <inria-00073962>

HAL Id: inria-00073962
https://hal.inria.fr/inria-00073962

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An Operational Semantics for the Eiffel// Language

Isabelle Attali , Denis Caromel , Sidi Ould Ehmety

N° 2732

Novembre 1995

PROGRAMME 2

ISSN 0249-6399

An Operational Semantics for the Eiffel// Language

Isabelle Attali *, Denis Caromel **, Sidi Ould Ehmety ***

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projets CROAP et SLOOP

Rapport de recherche n° 2732 — Novembre 1995 — 24 pages

Abstract: This paper formally describes the dynamic semantic of the Eiffel// language (Eiffel parallel). Eiffel// is a parallel extension of Eiffel language based on passive and active objects, asynchronous communication and wait-by-necessity. In this semantics we use formal specifications of inheritance and dynamic binding which we have defined in Natural Semantics for the Eiffel language. While in the framework of Natural Semantics (big-step semantics), we specify the Eiffel// semantics in the SOS style (small-step semantics). From this specification we automatically generate a programming environment for Eiffel//.

Key-words: Operational semantics, parallel object-oriented languages, active objects, passive objects, asynchronous communications, wait-by-necessity, futures

(Résumé : tsvp)

* Isabelle.Attali@sophia.inria.fr
** Denis.Caromel@sophia.inria.fr
*** Sidi.Ould-Ehmety@sophia.inria.fr

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

Une Sémantique Opérationnelle pour le Langage Eiffel//

Résumé : Cet article décrit formellement la sémantique dynamique du langage Eiffel//, extension parallèle du langage Eiffel, dont les caractéristiques principales sont la notion d'objets passifs et actifs, l'asynchronisme, et l'attente par nécessité. Pour cette sémantique, nous utilisons la spécification formelle de l'héritage et de la liaison dynamique que nous avons définis en Sémantique Naturelle pour le langage Eiffel. Bien que dans le cadre de la Sémantique Naturelle (sémantique big-step) Nous spécifions la sémantique d'Eiffel// dans le style SOS (sémantique small-step). A partir de cette spécification, nous générons automatiquement un environnement de programmation pour Eiffel//.

Mots-clé : Sémantique opérationnelle, langages à objets parallèles, objets actifs, objets passifs, communications asynchrones, attente par nécessité, futurs.

1 Introduction

The Eiffel parallel (Eiffel//) language [Car91, Car93], an extension of Eiffel, is a parallel language supporting the programming of multi-processor architectures. The Eiffel// language belongs to the category of asynchronous languages with asynchronous communications. The main objective of this language is to help the writing of parallel programs thanks to the following features:

- reusability of sequential programs when designing parallel systems,
- transformation of sequential programs to obtain an equivalent system running in parallel,
- methodology for the designer of parallel applications,
- a set of classes makes it possible to tailor the programming of concurrency control to the particular application.

Our aim is to define a formal semantics for the Eiffel// language with the following objective: the transformation of Eiffel programs (sequential) to Eiffel// programs (parallel). For that, we have defined the dynamic semantic of the Eiffel language [ACO93] using a Natural Semantics [Kah83] with the Centaur system [BCD+88].

There is no syntactic differences between Eiffel and Eiffel//, so we can reuse the syntactic description (concrete and abstract syntax). We note that both languages are based on the class concept and inheritance, so we may reuse the semantic modules dealing with inheritance (simple, multiple and repeated) and dynamic binding [ACE95]. On the other hand, for the semantics of parallel aspects (concurrent execution, communication), we use a *small step* semantics (Structural Operational Semantics [Plo81]) rather than a *big step* semantics, because the small step semantics is more appropriate to describe the interleaving.

The semantics of parallelism is widely studied in the literature. Many models are proposed, for example the processes algebra (CCS [Mil80], CSP [Hoa78]). These models are useful for procedural programming, but they are not suited for object-oriented programming where the configuration of systems change dynamically. Milner has proposed the π-calculus [MPW89a, MPW89b] as an extension of CCS with the dynamic generation of channels and the possibility to be viewed as a transmissible value. The π-calculus seems well-suited for parallel object-oriented language but its terms become difficult to manipulate as soon as we have to handle a real language (a semantics of Eiffel// in π-calculus is proposed in [Sag95]).

Finally, the actor model [AMST92] is an extension of λ-calculus with call-by-value and some primitives for the creation and manipulation of actors. This model is based on the notion of configuration of actors (a community of actors, autonomous entities of calculus which interact in concurrency and communicate by message sending). This model is more adapted to functional programming.

The formal semantics of some parallel object-oriented languages have been studied, namely the POOL family [Ame89]. For example, in [ABKR86] an operational semantics for a

member of this family is given while [ABKR89] presents a denotational semantics based on metric spaces. A rewriting in π-calculus of a POOL language is given in [Wal91].

The semantics of another parallel extension of Eiffel (Concurrent Eiffel) has been defined using an operational semantics [JP93].

In the future, from our formal specification of the semantics of Eiffel and Eiffel//, we will focus on the formalization of transformations of sequential programs to parallel programs. We will study (and prove) their validity using the classical tools of equivalences such as trace [Hoa78] and bisimilation [Mil80].

This report is structured as follow. Section 2 is an informal introduction to Eiffel//. Section 3 presents the abstract syntax of Eiffel// statements and expressions. Section 4 gives the abstract syntax of semantic structures used for describing the dynamic semantics of Eiffel//. Section 5 provides the operational semantics. Section 6 is a brief overview of the generated environment for programming in Eiffel//. Section 7 is the conclusion.

2 The Eiffel// language

The Eiffel// language, is an extension of Eiffel. As Eiffel, Eiffel// is a strongly typed, statically-checked class-based language supporting simple, multiple and repeated inheritance. Intuitively, Eiffel// may be viewed as Eiffel in which new classes are added. These new classes take care of the parallelism and processes management. The language has been defined in [Car91, Car93], as an extension of Eiffel.

2.1 Eiffel// Programs

As in Eiffel, the text of an Eiffel// program (a system) is a set of classes, with a distinguished class, the root class. The root class is the main program. A class is the definition of an abstract data type implementation: an object is an instance of a class. A class has a set of *features* related to its instances. A feature can be either an attribute or a routine (a procedure or a function). In Eiffel// (as in Eiffel), the keyword **result** denotes the result returned by the current function and **current** denotes the current object (the equivalent of **self** in Smalltalk and **this** in C++). The execution of a system consists in the creation of an instance of the root class (root object), and the execution of the **create** routine (a predefined routine) of the root class. The execution of the **create** routine on the root object may create/alter other objects.

A difference with Eiffel is that, at the execution, an Eiffel// system is composed of processes and objects:

- a process or process object is an instance of a class inheriting, directly or indirectly, from a particular class: the class PROCESS.
- object (passive object): all the others.

So, in Eiffel//, a process is an object but not every object is process.

2.2 The PROCESS class

Figure 1 presents the PROCESS class. After creation, a process executes its `Live` routine. This routine describes the process script or body. By inheritance, the class PROCESS gives to a process a default behavior defined in the routine `Live`: requests to the process entry points are treated one at a time in a FIFO order. However, heirs of the class PROCESS may specify a new behavior by redefining the routine `Live`.

```
class PROCESS
      -- Each heir of the current class
      -- has process object instances
feature
      Live is
           -- Process body
           do
               -- Default policy for entry
               -- points is  FIFO
           end;
end -- PROCESS
```

Figure 1: The PROCESS class

2.3 Processes and Objects

A process is an autonomous object (*active*) executing a behavior, going through successive phases. On the other hand, an object remains a classical data structure (*passive*), waiting for calls to execute its routines. To manage the cohabitation of these two kinds of objects two choices have been done:

- it is possible to do a polymorphic assignment between an object and a process: an *entity* (a variable) which is not declared of a type process can dynamically refer to a process. Then a feature call can dynamically become a communication between processes.
- there are no shared objects between processes. So, to avoid shared objects, references on passive objects are passed by copy between processes.

2.4 Communication and Synchronization

A process is an object, so it has exported routines. During execution, any routine call on a process gives rise to an *Inter-Processes Communication* (IPC): syntactically an IPC is a routine call. The communication is asynchronous and based on the principle of handshake. More specifically, when an object calls a routine on a process:

- a handshake is established between the caller and the process,

- a request is sent to the process,
- the process resumes its previous activity, the caller continues its execution.

Because the caller does not wait for the execution of the routine, we say that the communication is asynchronous.

To synchronize the processes, the language uses the *wait-by-necessity* a data-driven synchronization: a process waits only when it attempts to use a value which is not yet available (future mechanism).

The wait-by-necessity is the only synchronization mechanism provided by Eiffel//. It provides a synchronization baased on dataflow. However, it is possible to make an explicite synchronization with the predefined routine **wait**, when applied to an object (**v.wait**).

2.5 Subsystems

An Eiffel// system may be viewed as a set of subsystems, each composed of:

- one process object, root of the subsystem;
- zero or more objects, they are passive objects referenced by the root process of the subsystem.

Figure 2 shows an Eiffel// system during execution. In this figure, processes are black and objects are gray; arrows represent references. Five subsystems are represented in this figure ((i), (ii), (ii), (iv) and (v)). Objects and processes may reference another process but not a passive object of another subsystem.

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 execution is parallel and communications are asynchronous: the target objet (which is always a process) stores the request as a pending request and the caller continues its execution. So, communications are synchronous in a single subsystem and asynchronous between two subsystems.

Any routine call (asynchronous or asynchronous) gives rise to a future; in the case of synchronous calls (arrows (3), (4) and (5)) the wait for the value of the future is immediate; in the case of asynchronous calls (arrows (1) and (2)) the wait is lazy: it is the wait-by-necessity.

2.6 Example

Figure 3 presents an example of reusability of Eiffel classes within the scope of Eiffel// programming. It provides a parallel version of the Eiffel class BINARY_TREE which describes a sorted binary tree:

- each node of the tree has a left child (`left`) and a right child (`right`);

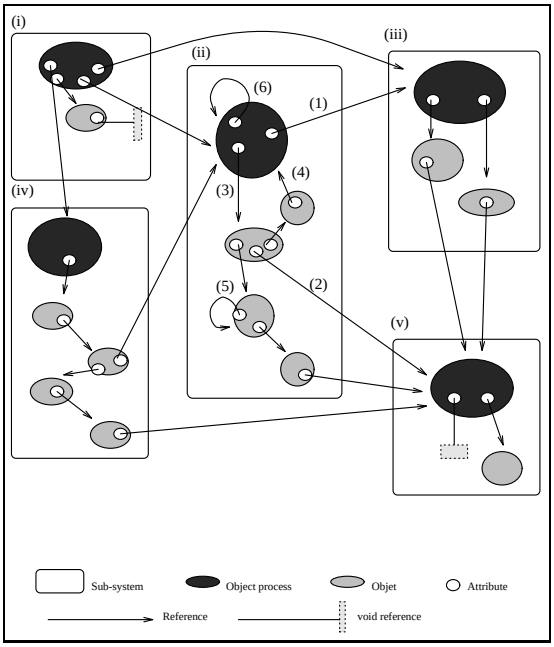

Figure 2: An Eiffel// system viewed as a set of subsystems

- an information (`info`) and its key (`key`) are stored in each node; in our example the information and the key are integers;
- all keys of the left subtree are smaller than the key of the current node;
- all keys of the right subtree are greater than the key of the current node.

The routine `insert` stores in the tree an information, in the right place according to a key and the routine `search` looks for the information associated with the key.

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.

Now we can use the parallel binary tree in an Eiffel// program by declaring a variable of type P_BINARY_TREE, and by using polymorphism (see Figure 4). In this case, we reuse existing sequential code (Eiffel code).

```
class BINARY_TREE
export   insert, search, left, reght
feature  key : INTEGER; info : INTEGER; left, right : BINRAY_TREE;
  insert (k : INTEGER; i : INTEGER) is
    do  --insert the information i ; k is the assiociated key
        if key = 0 then
           key := k; info := i;
           left.create; right.create;
        elsif key = k then
             info := i;
        elsif k < key then
             left.insert(k, i);
        else
             right.insert(k, i);
        end;
    end; --insert

 search (k : INTEGER) : INTEGER is
        -- the value associated to the key  k
    do
        if key = 0 then
           result := 0; -- not found
        elsif k = key then
           result := info;
        elsif k < key then
           result := left.search(k);
        else
           result := right.search(k);
        end;
    end; --search

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
```

Figure 3: Example of reusability in Eiffel//

Figure 5 shows a graphical representation of the systems from Figure 4 during execution; processes are black and arrows represent asynchronous communication. In this example all objects are processes.

```
class EXAMPLE
feature
    v: INTEGER;  bt: BINARY_TREE;
    Create is
       local p_bt: P_BINARY_TREE;
       do
         p_bt.create;
         bt := p_bt;
         build_binary_tree(bt);
         -- searching the value of the key 4
         v := bt.search(4);
         v.print
         -- wait by necessity
   end; --Create

    build_binary_tree(bt: BINARY_TREE) is
       do
          -- creation of the binary tree
          bt.insert(3, 6);
          bt.insert(1, 2);
          bt.insert(2, 4);
          bt.insert(4, 8);
          bt.insert(6, 12);
    end ; -- binary_tree
end -- Example
```

Figure 4: Example of an Eiffel// program

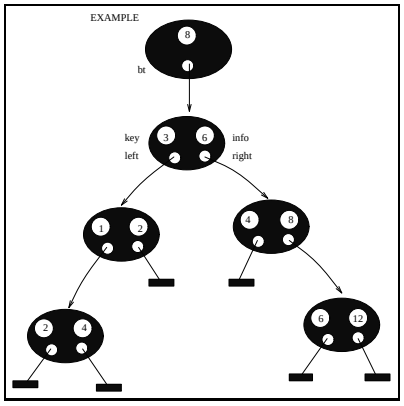

Figure 5: Execution of an Eiffel// system

3 Syntactic structures

We present here the abstract syntax of expressions and statements of Eiffel// (see Figure 6). There is no syntactic difference between Eiffel and Eiffel//, then we have the same expres-

sions and statements in both languages. Eiffel// has the specificity of offering a library of predefined services and routines. We model the semantics of the service **serve_oldest** (FIFO service) and the routine **wait** (explicit wait). For modeling synchronous communications, we consider all routines as functions (the value returned by a procedure is the current object) and we achieve the synchronization using the explicit wait.

For the presentation of the semantics of the Eiffel// language we use the concrete representation of the syntax rather than the abstract one. An expression $\mathrm{E} \in$ Expr is a variable, a routine call, an arithmetic or logic expression or a constant. A statement $\mathrm{S} \in$ Stmt is an assignment, a sequence of statements, a selection, an iteration, a routine call or an object creation statement. We do not present the non-qualified call routine $(\mathrm{M}(\mathrm{E}_1, \ldots, \mathrm{E}_n))$ because it is equivalent to the qualified call **current** $\cdot\, \mathrm{M}(\mathrm{E}_1, \ldots, \mathrm{E}_n)$. 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 **result**). K designates a constant value (integer, boolean, **void** (null reference), etc.) and Op_1 and Op_2 the unary and binary operators.

— **Expressions**	— **Instructions**
E ::= X	S ::= Y := E
| E.M($\mathrm{E}_1, \ldots, \mathrm{E}_n$)	| S_1; S
| Op_1 E	| **if** E **then** S_1 **else** S_2 **end**
| E_1 Op_2 E_2	| **until** E **loop** S **end**
| K	| E.M($\mathrm{E}_1, \ldots, \mathrm{E}_n$)
	| Y.**create**
	| **serve_oldest**

Figure 6: Eiffel// Statements and Expressions

4 Semantic structures

For specifying the dynamic semantics of the Eiffel// language, we need to define some structures which describe the global state of a system. During execution, an Eiffel// system is composed of objects. Each object in the system has a state and a behavior; the collection of all object states is the state of the system. So, we need a structure for modeling objects (with their activity) during execution. On the other hand, because each communication in Eiffel// gives rise to a future, we need a structure to store the futures and their values. In this section, we present such structures.

4.1 Objects

We model a system of Eiffel// objects $\Omega \in \text{Objs}$ with a list $\Omega ::= \{\Omega_i\}^*$ where each element (an object $\Omega_i \in \text{Obj}$) of this list is a quintuplet: $\Omega_i ::= \langle \alpha, \kappa, \rho, \text{C}, \text{R} \rangle$.

The value $\alpha \in \text{OName}$ is the identifier of the object, $\kappa \in \text{CName}$ is the name of the object class (its dynamic type), $\rho \in \text{Pairs} ::= \{\rho_i\}^*$ is a list of pairs (attribute, value), $\text{C} \in \text{Clrs} ::= \{\text{C}_i\}^*$ is the closure list (modeling object activity) and $\text{R} \in \text{Rqsts} ::= \{\text{R}_i\}^*$ a list of requests to serve.

A closure $\text{C}_i = \langle \text{S}, \eta \rangle$ is defined by a sequence of statements $\text{S} \in \text{Stmt}$ and a context $\eta \in \text{Cxt}$. The context is 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 (with **result**) and their values.

Finally, a request $\text{R}_i \in \text{Rqsts}$ is modeled by a quadruplet $\text{R}_i = \langle \text{M}, \widetilde{\text{V}}, \phi, \alpha \rangle$ with M the name of routine to serve, $\widetilde{\text{V}} = (\text{V}_1, \ldots, \text{V}_n)$ the effective parameters, ϕ the future that will contain the value of **result** after the completion of the routine and α the sender identifier (for sending back the result).

4.2 Futures

For modeling futures we add a new value: the awaited value (or future). So, a valeur V may be an effective value noted V (integer, boolean, reference, etc.) or a future ϕ[1].

At any time, we can distinguish between an awaited value and the effective returned value. This distinction allows us a straightforward specification of the wait-by-necessity.

Futures created during the execution of an Eiffel// system are stored in an environment shared by all objects. This environment is a list of pairs (future name, value):
$\Phi \in \text{Ftrs} ::= \{\Phi_i\}^*$. Each element $\Phi_i \in \text{Ftr}$ is defined by $\Phi_i = \langle \phi, \text{V} \rangle$ where $\phi \in \text{FName}$ is a future name and $\text{V} \in \text{Val}$ is its value.

4.3 Continuations

In a small step operational semantics, it is necessary to describe continuations. A continuation is what an objet has to do after some elementary step of execution. This leads us to define new syntactic constructors (unknown by the programmer).

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

Intuitively, $\text{E} \Rightarrow$ and $\text{E} \Leftarrow$ are useful for passing the current result from a closure to another in the same object;
$\text{E} \Rightarrow \phi$ returns the result of a service from an object to another;
$\text{E} \rightsquigarrow \text{M}(\text{E}_1, \ldots, \text{E}_n)$ is used for modeling the evaluation of parameters (by copy or by reference);

[1] This solution has been used in [JP93] for the specification of the semantics of Concurrent Eiffel.

null is the statement which does nothing;
$\text{E} \cdot \textbf{clone}(\alpha)$ makes a copy of E and $\textbf{clone\_attrs}(\rho, \alpha)$ makes a copy of each attribute value of ρ.

5 Dynamic Semantics

In this section we describe the operational semantics of Eiffel//. 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, as it is defined for Eiffel.

We then describe the operational semantics of Eiffel// in terms of a transition system, modeling possible transitions from one configuration to another. Configurations represent states of an Eiffel// system and transitions represent global actions. Each global action is an internal action or an interaction action between two objects. Then we present rules describing global actions of an Eiffel// system. We will show later how these global actions are expressed in terms of local actions on objects.

5.1 Inheritance and Dynamic Binding

Because, Eiffel and Eiffel// have the same concepts of inheritance, classes, and dynamic binding, we can reuse the semantics of inheritance (with renaming and redefinition) and dynamic binding defined for Eiffel [ACE95]. In this semantics, we use a natural deduction style, which means we do not build, for every class, effective inherited features, the attributes list, and so on. Instead of building some intermediate data structures (the equivalent of symbol table or memory state for imperative languages), we prefer to use the source program, looking for information in the current class, or in ancestors of the current class, when we need it. In our semantic model, the dynamic binding between a routine name and its actual body is specified when a feature call occurs, according to the current object, polymorphism and inheritance.

From the semantics of Eiffel, we use the following predicates:

- $feature(\text{M}, \kappa, \Pi) = \text{M}'(Vdecs_1)$: T **is local** $Vdecs_2$ **do** S_M **end;**
  which determines the effective declaration of the routine according to possible renamings and redefinitions (M' is the version of M in κ).
- $type(\text{X}, \kappa, \Pi) = \text{T}$, which determines the static type of the attribute X.
- $init(Vdec) = \rho$, which builds the ρ environment: the list of pairs (local variable, initial value) where each initial value depends on the type of the variable (**0** for integer, **void** for references, etc.).
- $bind(Vdecs, \widetilde{\text{V}})$, which builds the ρ environment: the list of pairs (formal parameter, value) where each value comes from the list of effective parameters $\widetilde{\text{V}}$.

We also define a new boolean predicate $inheritprocess(\kappa, \Pi)$ simply based on the existing predicate $inherit(\kappa, \kappa')$ which states whether a class inherit from another.

5.2 Modeling subsystems

We need some predicates dealing with subsystems. Because, there is only one process per subsystem we identify each subsystem by its root process. The predicate ss applied on an object identifier return its process. So, if α and β are two identifiers of objects:

- $ss(\alpha) = \alpha$ means that α is a process,
- $ss(\alpha) = ss(\beta)$ means that α and β are in the same subsystem.

5.3 The Transition System

Our operational semantics is based on a transition system whose states represent global configurations of systems of objects. The global transition relation is defined in terms of a family of labelled transition relations which describe the possible actions of objects. The execution of a program is modeled by a sequence of configurations with transitions in between, starting from a suitable initial configuration. The semantics of a program is given by a transition system which represents all its possible executions.

Formally, a global configuration "state" of a system Eiffel// is a triplet $\langle \Pi, \Phi, \Omega \rangle$ where Π is the Eiffel//system (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 have a judgement of the form:

$$\langle \text{System, Ftrs, Objs} \rangle \longrightarrow \langle \text{System, Ftrs, Objs} \rangle$$

which may be interpreted as follow:

A system in a state $\langle \Pi, \Phi, \Omega \rangle$ *performs a global action and changes its state into* $\langle \Pi, \Phi', \Omega' \rangle$. Note that the system Eiffel// Π (the source program) is not modified during execution. So, the execution of an Eiffel// system is a sequence of transitions:

$$\langle \pi_0 \cdot \Pi,\, \Phi_0,\, \Omega_0 \rangle \longrightarrow \langle \pi_0 \cdot \Pi,\, \Phi_1,\, \Omega_1 \rangle \longrightarrow \langle \pi_0 \cdot \Pi,\, \Phi_2,\, \Omega_2 \rangle \longrightarrow \cdots$$

where the initial configuration is given by:

$$\langle \pi_0 \cdot \Pi, \Phi_0, \Omega_0 \rangle = \langle \pi_0 \cdot \Pi,\, [\,],\, \{\langle \alpha_0, \textsc{root},\, \rho_0,\, \{\langle \textbf{create},\, \langle [\,],\, \mathit{init}(\text{Vdecs}) \rangle \rangle\}, [\,] \rangle\} \rangle$$

At the beginning, the list of object contains only one object (the root object, instance of π_0). Its attributes are initialized in ρ_0. The object should execute its **create** predefined routine and has no requests to serve.

The global actions of an Eiffel// system are defined in terms of local actions of these objects (see Figure 7). A local action of an object α may by an internal action to α or an action of α which expresses an interaction with another object of the system (synchronized action).

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

$$\frac{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\mathtt{new}(\kappa_1,\beta)} \rho',\ \mathrm{C}',\ \mathrm{R}'}{\langle\, \Pi,\ \Phi,\ \Omega \cup \{\langle \alpha,\ \kappa,\ \rho,\ \mathrm{C},\ \mathrm{R}\rangle\}\rangle \longrightarrow \langle\, \Pi,\ \Phi,\ \Omega \cup \{\langle \alpha,\ \kappa,\ \rho',\ \mathrm{C}',\ \mathrm{R}'\rangle, \langle \beta,\ \kappa_1,\ [\,],\ [(\mathrm{S},\ ())],\ [\,]\rangle\}\rangle} \qquad (\mathtt{G2})$$

provided $inheritprocess(\kappa_1, \Pi) = \mathbf{true}$, $\mathrm{gen}(\beta)$, $ss(\beta) = \beta$

where $feature(\mathtt{live}, \kappa_1, \Pi) = \mathtt{live}\ :\ \mathrm{T}\ \mathbf{is\ do}\ \mathrm{S}\ \mathbf{end}$

$$\frac{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\mathtt{new}(\kappa_1,\beta)} \rho',\ \mathrm{C}',\ \mathrm{R}'}{\langle\, \Pi,\ \Phi,\ \Omega \cup \{\langle \alpha,\ \kappa,\ \rho,\ \mathrm{C},\ \mathrm{R}\rangle\}\rangle \longrightarrow \langle\, \Pi,\ \Phi,\ \Omega \cup \{\langle \alpha,\ \kappa,\ \rho',\ \mathrm{C}',\ \mathrm{R}'\rangle, \langle \beta,\ \kappa_1,\ [\,],\ [\,],\ [\,]\rangle\}\rangle} \qquad (\mathtt{G3})$$

provided $inheritprocess(\kappa_1, \Pi) = \mathbf{false}$, $\mathrm{gen}(\beta)$, $ss(\beta) = ss(\alpha)$

$$\frac{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa_\alpha,\ \rho_\alpha,\ \mathrm{C}_\alpha,\ \mathrm{R}_\alpha\rangle \xrightarrow{\mathtt{cln}(\beta,\phi,\gamma)} \rho'_\alpha,\ \mathrm{C}'_\alpha,\ \mathrm{R}'_\alpha}{\langle\, \Pi,\ \Phi,\ \Omega \cup \{\langle \alpha,\ \kappa_\alpha,\ \rho_\alpha,\ \mathrm{C}_\alpha,\ \mathrm{R}_\alpha\rangle\}\rangle \longrightarrow \langle\, \Pi,\ \langle \phi,\ \phi\rangle \cdot \Phi,\ \Omega \cup \{\langle \alpha,\ \kappa_\alpha,\ \rho'_\alpha,\ \mathrm{C}'_\alpha,\ \mathrm{R}'_\alpha\rangle, \langle \lambda,\ \kappa_\beta,\ \rho_\beta,\ \mathrm{C},\ \mathrm{R}_\beta\rangle\}\rangle} \qquad (\mathtt{G4})$$

provided $\exists \langle \beta, \kappa_\beta,\ \rho_\beta,\ \mathrm{C}_\beta,\ \mathrm{R}_\beta\rangle \in \Omega \cup \{\langle \alpha,\ \kappa_\alpha,\ \rho_\alpha,\ \mathrm{C}_\alpha,\ \mathrm{R}_\alpha\rangle\}\, \mathrm{gen}(\lambda)$, $\mathrm{gen}(\phi)$, $ss(\lambda) = ss(\gamma)$

where $\mathrm{C} = \langle \mathbf{clone\_attrs}(\rho_\beta, \gamma);\ \lambda \Rightarrow \phi,\ ()\rangle$

$$\frac{\Pi,\ \Phi \vdash \langle \alpha, \kappa_\alpha, \rho_\alpha, \mathrm{C}_\alpha, \mathrm{R}_\alpha\rangle \xrightarrow{\mathtt{snd}(\beta,\mathrm{M},\widetilde{\mathrm{V}},\phi)} \rho'_\alpha, \mathrm{C}'_\alpha, \mathrm{R}'_\alpha \qquad \Pi,\ \Phi \vdash \langle \beta, \kappa_\beta, \rho_\beta, \mathrm{C}_\beta, \mathrm{R}_\beta\rangle \xrightarrow{\mathtt{rcv}(\mathrm{M},\widetilde{\mathrm{V}},\phi,\alpha)} \rho'_\beta,\ \mathrm{C}'_\beta,\ \mathrm{R}'_\beta}{\begin{array}{c}\langle \Pi, \Phi, \Omega \cup \{\langle \alpha, \kappa_\alpha, \rho_\alpha, \mathrm{C}_\alpha, \mathrm{R}_\alpha\rangle, \langle \beta, \kappa_\beta, \rho_\beta, \mathrm{C}_\beta, \mathrm{R}_\beta\rangle\}\rangle \longrightarrow \\ \langle \Pi, \Phi', \Omega \cup \{\langle \alpha, \kappa_\alpha, \rho'_\alpha, \mathrm{C}'_\alpha, \mathrm{R}'_\alpha\rangle, \langle \beta, \kappa_\beta, \rho'_\beta, \mathrm{C}'_\beta, \mathrm{R}'_\beta\rangle\}\rangle\end{array}} \qquad (\mathtt{G5})$$

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

$$\frac{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\mathtt{rep}(\phi,\mathrm{V})} \rho',\ \mathrm{C}',\ \mathrm{R}'}{\langle\, \Pi,\ \Phi,\ \Omega \cup \{\langle \alpha,\ \kappa,\ \rho,\ \mathrm{C},\ \mathrm{R}\rangle\}\rangle \longrightarrow \langle\, \Pi,\ \Phi[\phi \leftarrow \mathrm{V}],\ \Omega \cup \{\langle \alpha,\ \kappa,\ \rho',\ \mathrm{C}',\ \mathrm{R}'\rangle\}\rangle} \qquad (\mathtt{G6})$$

Figure 7: Rules describing global actions

5.4 Global Actions

The global actions of Eiffel// systems are given in Figure 7. They show how a configuration may change depending on the local actions of objects. The rule (`G1`) describes a internal action in an object, (`G2`) and (`G3`) describe the creation of a new object, (`G4`) describes the copy of an object, (`G5`) describes the inter-objects communication and (`G6`) describes the return of the result of a service.

Internal action (`G1`): This rule describes how the global state may change as a result of the independent activity of one object.

Creation of a new object (`G2`) and (`G3`): For the creation of an object, a new identifier is generated. There are two possibles cases:

- the class of the object κ_1 inherit from the PROCESS class: the new object is a process, it becomes the root of a new subsystem. The object starts the execution of the `live` routine.
- the class of the object κ_1 does not inherit from the PROCESS class: the new object is created in the same subsystem. The object (a passive object) does not execute any statement; it waits for calls to execute its routines.

Copy of an object (`G4`): This rule describes the deep copy of an object β for the own use of γ. The copy of an object (passive object) is necessary to transmit it between subsystems. In this rule, a deep copy of β will be transmitted between α and γ. We create a new object λ, in the subsystem of γ, and update its fields from the fields of β.

The new object starts the execution of **clone_attrs**(ρ, γ) for copying its attributes. This statement is composed sequentially with $\lambda \Rightarrow \phi$, where ϕ is a new future on which α is set to wait. The execution of $\lambda \Leftarrow \phi$ has the effect of updating the value of ϕ with λ.

Communication (`G5`): This rule describes the synchronization between the `send`$(\text{M}, \widetilde{\text{V}}, \phi, \beta)$ action performed by α and the `rcv`$(\alpha, \text{M}, \widetilde{\text{V}}, \phi)$ action performed by β. The communication gives rise to a new future ϕ which handles the wait-by-necessity.

Return of the result (`G6`): The return of the result of a service is done by assigning the value to the future.

5.5 Local Actions

The rules describing local actions have a judgement of the form:

$$\text{Ftrs, System} \vdash \text{Obj} \xrightarrow{\texttt{local}} \text{Pairs, Clrs, Rqsts}$$

This judgment may be interpreted as follow:

An object $\alpha \in \mathrm{Obj}$ *performs some local action and modifies its own environment (formed by its attributes, its closures and its requests).*

In the following, we describe the semantics of Eiffel// statements[2] and show how they are subdivided into elementary actions which are internal to an object, and elementary actions of communication between two objects. The label `local` may have the following values: `int`, `new`, `cln`, `snd`, `rep`.

5.5.1 Non Elementary Statements

In a small step semantics, each Eiffel// statement is executed in one or more elementary steps. After one elementary step, one obtains an intermediary state and a continuation. We have defined the semantics of each statement Eiffel// and each continuation. Because we can not know in advance which action will be applied the label `local` is a free variable l (which will be instanciated when an axiom is applied). We give in Figure 8 the semantics of Eiffel// statements. For each statement, additional rules are required for elimination and introduction of a closure.

5.5.2 Internal actions

The axioms describing internal actions are given in Figures 9 and 10.

The axioms for evaluation of a **current** (`I1`) variable, a formal parameter (`I2`), a local variable (`I3`), an attribute (`I4`) and an unary and binary arithmetic or logic expression (`I5` and `I6`) are straightforward with $[\![Op_1]\!]$ and $[\![Op_2]\!]$ being, respectively, the interpretations of Op_1 and Op_2.

The axiom (`I7`) describes the evaluation of a future ϕ. This axiom is not applicable if the value of ϕ in the futures environment is not an effective value $\underline{\mathrm{v}}$. In this case, the absence of any action for the object (other than receipt of requests) models the wait-by-necessity.

The axioms (`I8`) and (`I9`) describe the local call of a routine: an object calls one of its own routines. In this case a new closure is created with the statements of the routine and its context. The statements of the routine is composed sequentially with **result** $\Rightarrow$. The execution of **result** $\Rightarrow$ has the effect of evaluating **result**, returning the value to the next closure and eliminating the closure. The $\Leftarrow$ in the second closure acts as a place-holder for the value to be returned by the call (value of **result** in the first 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.

The axioms (`I11`) to (`I15`) present how the parameters are evaluated according to the call. For a synchronous call (a call between two subsystems), parameters are passed by reference. For an asynchronous call (a call without a subsystem), parameters are passed by copy. In the axiom (`I12`), $\mathrm{E} \cdot \widetilde{\mathbf{clone}}(\beta)$ denotes $\mathrm{E}_1.\mathbf{clone}(\beta), \mathrm{E}_2.\mathbf{clone}(\beta), \ldots \mathrm{E}_n.\mathbf{clone}(\beta)$, where $\widetilde{\mathrm{E}} = \mathrm{E}_1, \mathrm{E}_2, \ldots, \mathrm{E}_n$ is the list of parameters of the routine M.

[2] We regard expressions also as statements.

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

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

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

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

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

$$\frac{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{E},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathrm{E}',\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'}{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{E} \cdot \textbf{clone}(\beta),\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathrm{E}' \cdot \textbf{clone}(\beta),\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'} \quad (clone)$$

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

$$\frac{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{E},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathrm{E}',\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'}{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle Op_1\ \mathrm{E},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle Op_1\ \mathrm{E}',\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'} \quad (unary\_op)$$

$$\frac{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{E}_1,\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathrm{E}_1',\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'}{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{E}_1\ Op_2\ \mathrm{E}_2,\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathrm{E}_1'\ Op_2\ \mathrm{E}_2,\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'} \quad (binray\_op)$$

$$\frac{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{E}_2,\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathrm{E}_2',\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'}{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{V}_1\ Op_2\ \mathrm{E}_2,\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathrm{V}_1\ Op_2\ \mathrm{E}_2',\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'} \quad (binray\_op)$$

$$\frac{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{S}_1,\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathrm{S}_1',\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'}{\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{S}_1;\ \mathrm{S},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{l} \rho',\ \langle \mathrm{S}_1';\ \mathrm{S},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}'} \quad (sequence)$$

Figure 8: Rules for statements

$$\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathbf{current},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\mathtt{int}} \rho,\ \langle \alpha,\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R} \qquad \text{(I1)}$$

$$\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{P},\ \langle \rho_1,\ \rho_2\rangle\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\mathtt{int}} \rho,\ \langle \rho_1[\mathrm{P}],\ \langle \rho_1,\ \rho_2\rangle\rangle \cdot \mathrm{C},\ \mathrm{R} \qquad \text{(I2)}$$

$$\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \mathrm{L},\ \langle \rho_1,\ \rho_2\rangle\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\mathtt{int}} \rho,\ \langle \rho_2[\mathrm{L}],\ \langle \rho_1,\ \rho_2\rangle\rangle \cdot \mathrm{C},\ \mathrm{R} \qquad \text{(I3)}$$

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

$$\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle Op_1\ \underline{\mathrm{V}},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\mathtt{int}} \rho,\ \langle \underline{\mathrm{V}}',\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R} \quad \text{where } \underline{\mathrm{V}}' = [\![Op_1]\!](\underline{\mathrm{V}}) \qquad \text{(I5)}$$

$$\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \underline{\mathrm{V}}_1\ Op_2\ \underline{\mathrm{V}}_2,\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\mathtt{int}} \rho,\ \langle \underline{\mathrm{V}},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R} \quad \text{where } \underline{\mathrm{V}} = [\![Op_2]\!](\underline{\mathrm{V}}_1, \underline{\mathrm{V}}_2) \qquad \text{(I6)}$$

$$\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \phi,\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\mathtt{int}} \rho,\ \langle \underline{\mathrm{V}},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R} \quad \text{where } \underline{\mathrm{V}} = \Phi[\phi] \qquad \text{(I7)}$$

$$\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \alpha \cdot \mathrm{M}(\widetilde{\mathrm{V}}),\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\mathtt{int}} \rho,\ \langle \mathrm{S}_M;\ \mathbf{result} \Rightarrow,\ \eta_M\rangle \cdot \langle \Leftarrow,\ \eta\rangle \cdot \mathrm{C},\ n\,\mathrm{R} \qquad \text{(I8)}$$
$$\text{where } \mathit{feature}(\mathrm{M}, \kappa, \Pi) = \mathrm{M}'(\mathit{Vdecs}_1)\ :\ \mathrm{T}\ \mathbf{is\ local}\ \mathit{Vdecs}_2\ \mathbf{do}\ \mathrm{S}_M\ \mathbf{end;}$$
$$\mathit{bind}(\mathit{Vdecs}_1, \widetilde{\mathrm{V}}) = \rho_1,\ \mathit{init}(\mathit{Vdecs}_2) = \rho_2, \eta_M = \langle \rho_1,\ \langle \mathbf{result}, \mathbf{void}\rangle \cdot \rho_2\rangle$$

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

$$\Pi,\ \Phi \vdash \langle \alpha,\ \kappa,\ \rho,\ \langle \underline{\mathrm{V}} \cdot \mathbf{wait},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\mathtt{int}} \rho,\ \langle \underline{\mathrm{V}},\ \eta\rangle \cdot \mathrm{C},\ \mathrm{R} \qquad \text{(I10)}$$

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

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

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

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

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

Figure 9: Axioms for internal actions: expressions

The execution of E · **clone**(β) has the effect of evaluating E and cloning the value as follows: in the case of a constant value or an identifier of a process object, the value is returned (axioms (I13) and (I14)); in the case of an identifier of a passive object, a deep copy of this object is required (axiom (I15)). The label on the arrow shows that the object which will be copied is γ, that a new future ϕ should be generated and that the copy of γ should belong to the subsystem of β. The copy of an object is terminated upon the return of its identifier to α using the same mechanism than returning the result of a service.

In the Figure 10, the axioms for assignments (local variable (I16) and attribute (I17)), conditional (I18) and (I19) and iterative (I20) statements are straightforward.

The axiom (I21) expresses the creation of an object. The execution of the create statement induces in the global rule the creation of an object. The label emntions that the type of this object is κ_1 (the type of the variable Y) and that its identifier instantiated with the free variable β. The assignment statement returned as continuation has the effect of referencing the new object with the variable Y.

When an object executes **serve_oldest**, one request is served (axiom (I22)). This is a first request in the request list. A new closure is created with the routine statements and its context. The statements of the routine is composed sequentially with **result** $\Rightarrow \phi$. The execution of the **result** $\Rightarrow \phi$ has the effect of evaluating **result** and assigning the value to the future ϕ (axiom (I23)).

In the axioms (I24) and (I25), an object copies its attributes. Axiom (I26) gives the semantics of the *null* statement, (I27) the semantics of a value as a statement and (I28) terminates a closure.

5.5.3 Elementary Statements of Communications

Figure 11 gives axioms describing communications. Axioms (C1) and (C2) may be applied when all subexpressions have been evaluated and respectively describe an asynchronous and a synchronous call. In the two cases, the future ϕ is a new future generated in the global rule. In the synchronous call α waits immediately for the value of the future (explicit wait). This value is the result of the service. In the global rules these actions will be synchronized with actions of receiving messages (C3) and (C4). In the case of an asynchronous call, a request is created and stored at the end of the request list; in the case of a synchronous call, the routine is executed immediately.

Note that only effective values are passed as arguments to asynchronous calls. We may deduce this from axioms.

5.6 Wait-by-necessity

We finish this section with an illustration of the specification of the wait-by-necessity. For example, in Figure 4, when the root object executes the statement `v:= bt.search`, `bt` refers a process object of type P_BINARY_TREE. The rule which describes the communication (G5) is applied. Then the futures environment Φ is updated by a new pair $\langle \phi, \phi \rangle$ and the

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle \text{L} := \text{V}, \langle \rho_1, \rho_2 \rangle \rangle \cdot \text{C}, \text{R} \rangle \xrightarrow{\texttt{int}} \rho, \ \langle null, \langle \rho_1, \rho_2[\text{L} \leftarrow \text{V}] \rangle \cdot \text{C}, \text{R} \qquad \text{(I16)}$$

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

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle \textbf{if true then } \text{S}_1 \textbf{ else } \text{S}_2 \textbf{ end}, \eta \rangle \cdot \text{C}, \text{R} \rangle \xrightarrow{\texttt{int}} \rho, \langle \text{S}_1, \eta \rangle \cdot \text{C}, \text{R} \qquad \text{(I18)}$$

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle \textbf{if false then } \text{S}_1 \textbf{ else } \text{S}_2 \textbf{ end}, \eta \rangle \cdot \text{C}, \text{R} \rangle \xrightarrow{\texttt{int}} \rho, \langle \text{S}_2, \eta \rangle \cdot \text{C}, \text{R} \qquad \text{(I19)}$$

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle \textbf{until } \text{E} \textbf{ loop } \text{S} \textbf{ end}, \eta \rangle \cdot \text{C}, \text{R} \rangle \longrightarrow \rho, \langle \textbf{if } \text{E} \textbf{ then null else } \text{S}; \text{S}' \textbf{ end}, \eta \rangle \cdot \text{C}, \text{R} \qquad \text{(I20)}$$
$$\text{where } \text{S}' = \textbf{until } \text{E} \textbf{ loop } \text{S} \textbf{ end}$$

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

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle \textbf{serve\_oldest}, eta \rangle \cdot \text{C}, \langle \text{M}, \widetilde{\text{V}}, \phi, \beta \rangle \cdot \text{R}' \rangle \xrightarrow{\texttt{int}}$$
$$\rho, \langle \text{S}_M; \text{E} \Rightarrow \phi, \langle \rho_1, \langle \textbf{result}, \textbf{void} \rangle \cdot \rho_2 \rangle \rangle \cdot \langle null, \eta \rangle \cdot \text{C}, \text{R}' \qquad \text{(I22)}$$
$$\text{where } feature(\text{M}, \kappa, \Pi) = \text{M}'(Vdecs_1) \ : \ \text{T} \textbf{ is local } Vdecs_2 \textbf{ do } \text{S}_M \textbf{ end;}$$
$$bind(Vdecs_1, \widetilde{\text{V}}) = \rho_1, \ init(Vdecs_2) = \rho_2, \ \text{E} = \textbf{result} \cdot \textbf{clone}(\beta)$$

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

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle \textbf{clone\_attrs}(\langle \text{A}, \text{V} \rangle \cdot \rho_0, \beta), \eta \rangle \cdot \text{C}, \text{R} \rangle \xrightarrow{\texttt{int}}$$
$$\rho, \langle \text{A} := \text{V} \cdot \textbf{clone}(\beta); \textbf{clone\_attrs}(\rho_0, \beta), \eta \rangle \cdot \text{C}, \text{R} \qquad \text{(I24)}$$

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle \textbf{clone\_attrs}([\,], \beta), \eta \rangle \cdot \text{C}, \text{R} \rangle \xrightarrow{\texttt{int}} \rho, \langle null, \eta \rangle \cdot \text{C}, \text{R} \qquad \text{(I25)}$$

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle null; \text{S}, \eta \rangle \cdot \text{C}, \text{R} \rangle \xrightarrow{\texttt{int}} \rho, \langle \text{S}, \eta \rangle \cdot \text{C}, \text{R} \qquad \text{(I26)}$$

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle \text{V}; \text{S}, \eta \rangle \cdot \text{C}, \text{R} \rangle \xrightarrow{\texttt{int}} \rho, \langle \text{S}, \eta \rangle \cdot \text{C}, \text{R} \qquad \text{(I27)}$$

$$\Pi, \Phi \vdash \langle \alpha, \kappa, \rho, \langle [\,], \eta \rangle \cdot \text{C}, \text{R} \rangle \xrightarrow{\texttt{int}} \rho, \text{C}, \text{R} \qquad \text{(I28)}$$

Figure 10: Axioms for internal actions: statements

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

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

$$\Pi,\ \Phi \vdash \langle \alpha,\ \kappa\rho,\ \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\mathtt{rcv}(\mathrm{M},\widetilde{\mathrm{V}},\phi,\beta)} \rho,\ \langle \mathrm{S}_M;\ \mathbf{result} \Rightarrow \phi,\ \rho_{\mathrm{M}}\rangle \cdot \mathrm{C},\ \mathrm{R} \qquad \text{(C3)}$$
$$\textit{provided}\ \ ss(\alpha) = ss(\beta)$$
$$\text{where } feature(\mathrm{M}', \kappa, \Pi) = \mathrm{M}(Vdecs_1)\ :\ \mathrm{T}\ \mathbf{is\ local}\ Vdecs_2\ \mathbf{do}\ \mathrm{S}_M\ \mathbf{end;}$$
$$bind(Vdecs_1, \widetilde{\mathrm{V}}) = \rho_1,\ init(Vdecs_2) = \rho_2,\ \eta_M = \langle \rho_1,\ \langle \mathbf{result}, \mathbf{void}\ \rangle \cdot \rho_2\rangle$$

$$\Pi,\ \Phi \vdash \langle \alpha,\ \kappa\,\rho,\ \mathrm{C},\ \mathrm{R}\rangle \xrightarrow{\mathtt{rcv}(\mathrm{M},\widetilde{\mathrm{V}},\phi,\beta)} \rho,\ \mathrm{C},\ \mathrm{R} \cdot \langle \mathrm{M}, \widetilde{\mathrm{V}}, \phi, \beta\rangle \qquad \text{(C4)}$$
$$\textit{provided}\ \ ss(\alpha) \neq ss(\beta)$$

Figure 11: Axioms for communications

future ϕ is transmitted, for the return of the result, to the object referenced by `bt`. In the root object, the value ϕ is assigned to the attribute `v`.

The execution continues, with the statement `v.print`. The execution of such a statement starts by the evaluation of the attribute `v` (axiom (`I17`)). Because the value of `v` is the future ϕ, the continuation $\phi \cdot$ `print` will be returned. The axiom of the evaluation of a future (`I7`) then may be applied. Two cases are possible:

- fail, because the value of future ϕ is ϕ, and the side condition of the axiom requires that the value of the future should be an effective value.
- success, because the value of future ϕ is an effective value (the integer 8).

In the first case the root object is then waiting for the return of the result of `v`, and the execution continues with other processes (which may return the value of `v`). In the second case the value is returned and the next continuation is `8.print`.

6 Programming in Eiffel//

From the operational semantics of Eiffel//, using the Centaur system, we can generate an interactive environment for parallel object-oriented programming and visualization. We

provide features such as graphical representation of objects, visualization of closure and semantic rules, and animation tools to show the concurrent activities of objects (see Figure 12). While the textual representation of objects shows the behavior of objects (activity, pending request), the graphical visualization shows the graph of objects and processes (distinguished by two colours); references are indicated by edges of the graph. A zooming makes it possible to show the attribute values of a given object. Both representation highlight the current active object.

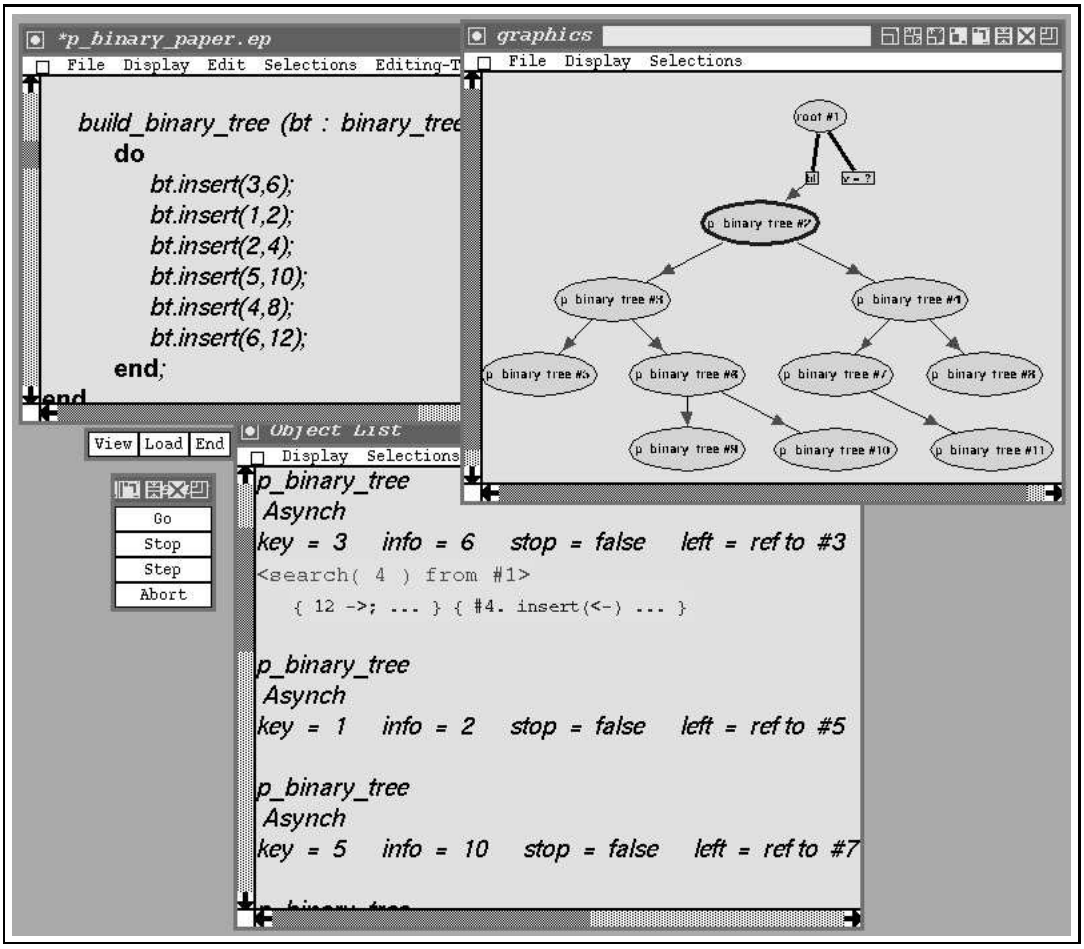

Figure 12: The Eiffel// environment

7 Conclusion

We have defined a formal operational semantics for the Eiffel// language. The parallel behavior is achieved by the creation of process objects. Communication between these objects is asynchronous with wait-by-necessity.

The semantics is based on a labelled transition system. We have expressed our semantics in the system Centaur using the Typol formalism.

The interest of this approach is to formally introduce parallelism within the framework of the Eiffel language, and more generally in object-oriented programming, leading towards formal transformations and parallelization for object-oriented languages. Our goal is to achieve formal transformations (parallelization) from sequential Eiffel programs to parallel Eiffel programs and prove the correctness of these transformations.

References

[ABKR86] P. America, J. D. Bakker, J. N. Kok, and J. Rutten. Oprerational Semantics of a Parallel Object-Oriented Language (POOL). *Conference Record of the 13th Symposium on Principles of Programming Languages, pages 194-205, St. Petersburg, Florida, January 13-15*, 1986.

[ABKR89] 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.

[ACE95] I. Attali, D. Caromel, and S.O. Ehmety. A Natural Semantics for the Eiffel Dynamic Binding. *Research Report I3S RR-95-60*, 1995.

[ACO93] I. Attali, D. Caromel, and M. Oudshoorn. A Formal Definition of the Dynamic Semantics of the Eiffel Language. In *Proceedings of ACSC'16*, Adelaide, Australia, February 1993.

[Ame89] P. America. Issues in the Design of a Pallel Object-Oriented Language. *Formal Aspects of Computing, N. 1, pages 366-411*, 1989.

[AMST92] G. Agha, I. A. Mason, S. Smith, and C. Talcott. Towards a Theory of Actor Computation. *CONCUR'92, pages 565-580, Stony Brook, NY, USA, August*, 1992.

[BCD+88] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the System. *INRIA Research Report 777, 1987 and SIGSOFT'88 Third Annual Symposium on Software Development Environments, Boston*, 1988.

[Car91] D. Caromel. Programmation Parallèle Asynchrone et Impérative: Etudes et propositions. *Thèse de Doctorat de l'Université de Nancy I, France, Fevrier*, 1991.

[Car93] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *CACM: Communication of the ACM, Volume 36, N. 9, pages 90-102*, 1993.

[Hoa78] C.A.R Hoare. Communicating Sequential Processes. *CACM, 21(8):666-69*, 1978.

[JP93] G. Jalloul and J. Potter. An Operational Semantics for a Model of Concurrent Eiffel, 1993. TR UTS-SOCS-93, 10.

[Kah83] G. Kahn. Natural Semantics. *Proceedings of Symposium on Theoretical Aspects of Computer Science, Passau, Germany, Volume 247 of Lecture Notes in Computer Science*, 1983.

[Mil80] R. Milner. A Calculus of Communicating Systems. *LNCS Vol. 92*, 1980.

[MPW89a] R. Milner, J. Parrow, , and D.J. Walker. A Calculus of Mobile Processes. *Part i, Academic press, Inc. pages 1-40*, 1989.

[MPW89b] R. Milner, J. Parrow, , and D.J. Walker. A Calculus of Mobile Processes. *Part ii, Academic press, Inc. pages 41-77*, 1989.

[Plo81] G.D. Plotkin. A Structural Approach to Operational Semantics. *Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark*, 1981.

[Sag95] D. Sagnol. π-calcul Pour les Langages à Objets Parallèles. *Rapport de DEA, Université de Nice-Sophia Antipolis*, 1995.

[Wal91] D. Walker. π-Calculus Semantics of Object-Oriented Programming Langage. *Proc. TACS'91, Springer-Verlag, LNCS Vol. 526, pages 532-547*, 1991.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399