Operational semantics of a distributed object-oriented language and its Z formal specification

Marc Benveniste

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1230

Programme 3
Réseaux et Systèmes Répartis

OPERATIONAL SEMANTICS OF A DISTRIBUTED OBJECT-ORIENTED LANGUAGE AND ITS Z FORMAL SPECIFICATION

Marc BENVENISTE

Mai 1990

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Operational Semantics of a Distributed Object-Oriented Language and Its Z Formal Specification

Sémantique Opérationnelle d'un Langage de Programmation Distribuée à Objets et sa Spécification Formelle Exprimée en Z*

Marc Benveniste†
IRISA/INRIA-Rennes
Campus de Beaulieu
35042 Rennes Cédex. FRANCE.
e-mail: mbenveni@irisa.irisa.fr

Publication Interne n° 532 - Avril 1990 - 100 Pages

*Travail effectué dans le cadre du projet LSP à l' IRISA.

†Supported by a grant from the XII CONACyT-CEFI fellowship program jointly held by MEXICO and FRANCE.

Abstract

PolyGoth is a distributed programming language that integrates the class abstraction of languages like Simula, SmallTalk, Pool or Eiffel with a parallel block structuring concept and its associated notations, namely *multiprocedures* and *fragments*. The concepts and notations introduced in PolyGoth enable programmers to address distribution issues in a novel and structured way. To avoid useless complexity, a kernel of the language is used to exhibit the semantics of its most interesting constructs. The semantics is given in an operational way that is based on a transition system. The **Z** specification language is used to describe the semantics. This novel use of a specification language greatly enhances the structure of the description, its rigorousness, and perhaps, its usefulness.

Key words and phrases:

Class-based programming, block structuring, multiprocedure, fragmentation, distributed systems, parallelism, coordinated call, transition systems, **Z**, specification language, language design.

Résumé

PolyGoth est un langage de programmation distribuée dans lequel s'intègrent le concept de classe, présent dans des langages tels que Simula, SmallTalk, Pool ou Eiffel, et une structure de blocs parallèles et ses notations associées : les *multiprocédures* et les *fragments*. Les concepts et notations introduits dans PolyGoth permettent de traiter de façon novatrice et structurée les aspects distribution d'un programme. Afin d'éviter d'inutiles complications, un noyau du langage est employé pour fournir la sémantique de ses commandes les plus intéressantes. Un système de transitions définit la sémantique de façon opérationnelle. Le langage de spécification **Z** est utilisé pour décrire cette sémantique. Ce nouvel usage d'un langage de spécification améliore la structure de la description, renforce la rigueur du discours tenu et le rend peut-être même plus utile.

Mots et phrases clés

Langage à objets, structure de bloc, multiprocédure, fragmentation, systèmes distribués, parallélisme, appel coordonné, systèmes de transitions, langage de spécification, **Z**, conception de langage.

Contents

1 Introduction

In this paper, we present the formal definition of the main concepts and notations introduced in the POLYGOTH programming language. The language itself is defined elsewhere [Ployette et al., 1987, Lecler, 1989]. The *multiprocedure* concept proposed in [Banâtre et al., 1986] and revisited in [Banâtre and Benveniste, 1989] plays a major role in POLYGOTH. The multiprocedure concept conciliates parallelism and procedures. It generalizes the procedure concept so that parallelism is cast in its structure. As a direct consequence, the communication mechanism furnished by the procedure –parameter passing and result return– gets a broader significance with the multiprocedure. The coordinated call –a natural extension of the procedure call– turns out to be a powerful synchronisation construct. It could be compared to a multiway ADA-like *rendez-vous*, but the coordinated call is a more general mechanism. For instance, together with the multiprocedure, it provides an interesting parallel recursion.

Concurrency in object-oriented languages is an important issue. Viewing an object as a context "surviver" of a multiprocedure execution leads to a novel and highly structured way of introducing parallelism into an object oriented language. The formalization of the fragmentation mechanism derived from this thinking is another contribution of this paper. Fragmentation addresses both parallelism and (logical) distribution in a unified manner. A fragmented object is a distributed entity whose components are tied by the multiprocedures provided as its methods.

In order to exhibit the operational behaviour of the notations associated to the multiprocedure and the fragmentation –as introduced in POLYGOTH– only a small subset of the language has been considered for this exercise. The semantics is given through the definition of a syntax-driven transition system, in the G. Plotkin style. Although the usefulness of a formal semantics definition could be doubtful for an already implemented language, this definition is a must to the thoroughness of a language presentation, and a sound basis to undertake its evaluation. This work is geared for the latter purpose. The **Z** specification language [Spivey, 1989] is used to ensure a rigorous description of the different syntactic and semantic entities. Besides the clear advantage of writing the semantics in a mechanically checkable formalism, a formal specification discloses subtleties as well as difficulties that are otherwise often "swept under" an imprecise notation. The semantics is chosen operational because it is an easy step towards a more abstract definition and because of implementation concerns. Its formal specification serves several purposes. It is a contract for the compiler writer, although the identity equivalence is the only one provided. It is an abstract, yet detailed, language manual for the programmer. Last, but not least, it is a valuable working paper both for the language and the implementation designers: the former may foresee implications of a design modification throughout the language, and the latter may employ a stepwise refinement methodology to locate where decisions have to be made and to evaluate their repercussions. The presentation is self-contained and no previous knowledge of **Z** is required to understand it (at least we hope so).

POLYGOTH should be seen more as an implementation of new ideas about distributed object-oriented programming than as a new programming language. An informal presentation of these ideas is given in section 2. Follows a (still informal) description of the language in section 3. Syntax and semantics of the language's kernel are presented in sections 4 and 5. It should be noted that the description of POOL's operational

semantics, reported in [America et al., 1986], has greatly influenced ours. Section 6 shows a very simple expression of the two levels of parallelism present in our language. The meaning of a program is defined at the end of that section. In section 7, some early comments about the language are made, further research is suggested and related work is mentioned. Conclusions are exposed in section 8. Finally, the complete formal specification of the semantics is included in appendix A. It has been type-checked and printed with the *fuzz* package [Spivey, 1988]. For briefness' sake, only a representative excerpt of the specification is reproduced in the main sections of this paper. In contrast with the lack of explanations in the appendix, they are duly accompanied with a reasonable amount of explaining text. We are fairly confident in that the interested reader is provided with enough guidance to successfully find his/her way throughout the appendix.

2 Motivations

The ideas proposed in the POLYGOTH language are centered around the procedure relevancy in dealing with parallelism. Traditionally, parallelism has been kept "outside" the procedure. The procedure concept has evolved towards modules, processes, objects and agents mainly led by two forces: abstraction and parallelism. This complex evolution –referred here in an oversimplified manner– has been carried out by successive extensions of the procedure. The procedure concept itself has not been revisited. The *multiprocedure* concept is the outcome of a generalizing approach to the integration of parallelism and procedures undertaken in [Banâtre, 1980]. A brief overview of that generalization process is given hereafter.

2.1 Multiprocedures

The procedure is a powerful program-structuring concept mainly because it enables abstraction and it is nestable. Abstraction is achieved because once a procedure is defined, that is to say its pre- and post-conditions are established, there is no need to know how its body is implemented in order to use it. Nesting comes in two flavours: lexical and dynamic. Lexical nesting refers to the declaration of a procedure within another procedure. Dynamic nesting deals with the execution of a procedure within another procedure's execution or its own, in which case it is known as recursion. The basic way to use a procedure is to call it. The call leads to (dynamic) nesting. The relationship between the caller and the callee –nesting "one in one"– is quite interesting. In terms of dependence, it is clear that the called procedure is "locally alive" within the caller. The called procedure may not be active together with the calling one. Communication relies solely on parameter passing and result restitution. Besides, those communications are unidirectional: parameter passing goes along the caller-callee direction and result restitution goes the other way around. In terms of correctness,

substitutions and logical implications formally describe the relationship[1]:

$$\frac{\begin{array}{l}\textbf{proc } \mathcal{Q}(\text{fp}) : \text{r};\\ \text{Body}\\ \textbf{end},\\ P \Rightarrow R[\,{}^{fp}/_{ap}\,],\ R\ \{\ \text{Body}\ \}\ T,\ T[\,{}^{r}/_{c}\,] \Rightarrow Q\end{array}}{P\ \{\ c := \mathcal{Q}(ap)\ \}\ Q.}$$

Naturally, one is entitled to ask: if nesting is such a good structuring concept, can it be married with parallelism?

Traditionally, parallelism is expressed with a compound statement like Dijkstra's **Parbegin ...Parend**. From a context point of view, the **Parbegin ...Parend** statement enables various sequential instruction streams to share a single data context. The lack of *identity* of the **Parbegin ...Parend** is mainly responsible for this state of affairs. Since it has no name it cannot be called. Therefore, lexical nesting is the only one available with its known limitations. It seems natural then to introduce an abstraction of the compound statement similar, in its genesis, to the procedure concept. We name that abstraction a *multiprocedure*. Multiprocedures are declarable, callable and nestable concurrent entities.

2.1.1 Multiprocedure declaration

A multiprocedure declaration consists of a *header* and a *body*. The header part specifies the name, the formal parameters and eventually a formal result. The header part of a multiprocedure is identical to a procedure header declaration. The body is made of a collection of blocks. Each block specifies the subset of formal parameters it deals with and its contribution to the result. A block is a set of declarations –known as local to the block– and a sequence of statements. If we adopt a call-by-value parameter passing mechanism, multiprocedure blocks are totally independent. For simplicity's sake, we do not consider lexical nesting here.

2.1.2 Multiprocedure activation

A multiprocedure is called as a procedure is. Nevertheless, the execution is carried out differently: a multi-context is created, parameters are made available to the multi-context, statements of the different blocks are executed in parallel, results are built out of each block's contribution and made available to the caller and finally, the multi-context is destroyed. The relationship between the caller and the different callee blocks –from now on *p-callees* for *p*artial *callees*– is a nesting "many in one". Various blocks, the p-callees that make up the multiprocedure, are nested within the caller. In order to achieve full composability, we still have to define how a multiprocedure issues a call.

2.1.3 Coordinated call

The *coordinated call* is derived from the traditional procedure call mechanism by generalizing it. Since a multiprocedure is made out of a set of blocks, a call should mean

[1]The presented description is not formally correct. The reader is referred to a complete axiom system for verifying procedural programs reported in [Cook, 1978].

that all those blocks perform a unique call. The proposed calling mechanism joins the calling multiprocedure blocks –from now on *p-callers*– to perform a unique call. They are all synchronized at the calling point. Actual parameters to the call may be supplied by any block of the calling party as long as formals are rightly covered. Therefore, a formal to actual parameter assignment is explicitly specified within the call denotation. When the call is terminated, results –if any– are made available to all the p-callers before resuming their parallel activities. Naturally, the coordinated call for a single-blocked multiprocedure is the traditional procedure call. The relationship between the p-callers and the p-callees is a "many in many" nesting.

2.1.4 Remarks

The multiprocedure concept bears the notions of distribution and fragmentation in its parameter passing mechanism and in its structure. Indeed, parameters are collected from the multiprocedure p-callers and distributed to the p-callees. Results are collected from the p-callees and multicasted to the p-callers. Multiprocedure blocks denote quite independent computations. Therefore, each block may be executed by a distinct agent. A multiprocedure may be thought of as a distributed and fragmented entity. These remarks are developed further in the following section.

2.2 Fragmentation

Programming in the large requires linguistic support in order to cope with the ever-growing complexity of the problems faced. Building blocks, larger than procedures, are needed to enable teams of programmers to work cooperatively and to deal with huge projects. From a software engineering standpoint, the key properties looked for in a programming notation are abstraction power and composability. Objects and their inheritance mechanism have been widely accepted with the advent of the object-oriented approach.

On the other hand, recent hardware achievements, such as multiprocessor workstations and fast communication means, are clear invitations to exploit parallelism. The object paradigm seems to fulfill both programming needs. Parallel object-oriented languages have emerged over the last few years to take advantage of these new architectures [Bal et al., 1989]. However, parallelism is not a settled issue among the object oriented community.

2.2.1 Objects and parallelism

Three programming models –procedure, message and operation oriented–, identified in the classical survey [Andrews and Schneider, 1983], may be used to explain different integrations of parallelism into object-oriented languages. Languages like TRELLIS/OWL [Schaffert et al., 1986] or SMALLTALK80 [Goldberg and Robson, 1983] rely on the procedure-oriented programming model. Processes are differentiated from objects. Objects are shared passive data that can implement the monitor abstraction [Hoare, 1974]. Others, like POOL [America, 1987], assimilate objects to processes. Active objects have their own activity and explicitely state their willingness to accept *rendez-vous*. These languages are derived from the message-oriented model. Actor languages, like ACT 1 [Lieberman, 1987] or CANTOR [Athas and Seitz, 1988], support

the message-oriented model, although objects have a simpler and more abstract definition. The operation-oriented model advocates for the best of the two formers: as in a message-oriented language, each object is a process; as in a procedure-oriented language, operations are performed by calling a procedure. The difference is that the calling process and the callee are synchronized while the operation is executed. Languages founded on this model like EMERALD [Black et al., 1986] provide remote procedure calls to communicate and to create parallel activities. Other languages like HYBRID [Nierstrasz, 1987] are based on hybrids derived from these three basic ones.

Back in section 2.1.4, we emphasized the logical distribution present in the multiprocedure concept. Thus, it is tempting to bring parallelism to objects through the replacement of their (sequential) methods by multiprocedures. Even though attractive, this proposition turns the object state into a large shared data for the different multiprocedure blocks. This monolithic embedding is in direct conflict with multiprocedure fragmented contexts. Indeed, a fragmented context supposes independent activities of the same lifelength that only share parameters.

2.2.2 Fragmented objects

A solution to the posed conflict suggests itself if we recall that an object is a "special extension" of a procedure whose context survives its execution. Now, we can define a new kind of object, called a fragmented object, that "extends" a multiprocedure whose (fragmented) context survives its execution. Each block of this new object is called a *fragment*. The monolithic object state becomes a logically distributed set of partial states. The replacement of (sequential) methods by multiprocedures is now a desirable proposition. Methods can be thought of as procedures (lexically) nested in the object (state). This nesting extends the scope of state variables throughout the procedure, giving it the opportunity to modify them. Multiprocedures need to be nested in our fragmented object in order to modify its state. Data dependency is an excellent candidate to guide the nesting of multiprocedure blocks in fragments. Indeed, if a multiprocedure block has to modify data belonging to a particular fragment, the block should be nested in that fragment. This calls for a *distributed* declaration of multiprocedures: blocks are declared within distinct fragments. A direct structural correspondence between state and behavior should be noted here: the parallel computation denoted by a multiprocedure *induces* a (logically) distributed data structure and conversely. A fragmented object captures all the potential parallelism found in a data structure, think of a vector or a matrix for instance. Furthermore, multiprocedures appear as an elegant and familiar notation to express operations on these distributed data structures. They are elegant because distributed operations are usualy expressed in a rather cumbersome manner that involves a large message manipulation burden. They are familiar since they generalize the well known and heavily used procedure. Protection and locality are salient enforcements of the combined structuring tool: parallelism is data driven, fragments represent (mostly) local processing, multiprocedures provide cohesion and composability in a unified control structure.

2.2.3 Remarks

Active or passive objects, message passing or procedure call, synchronous or asynchronous communications, are dimensions that define a *space* where parallel object-

oriented languages may be positioned. The rationale for choosing a position in that space is often grounded in the underlying system architecture. Quoting the authors of [Bal et al., 1989]:

> The unit of parallelism in a language ranges from the process [...] to the expression [...]. In general, the higher the cost of communication in a distributed system, the larger the appropriate granularity of parallelism.

Fragmented objects offer two units of parallelism: fragments giving rise to an intra-object parallelism, and objects bringing inter-object parallelism. Intra-object parallelism yields inexpensive communications, for locality may be identified; fragmented objects are well suited for tightly coupled architectures. Inter-object communications are costlier; sets of objects are well adapted for loosely coupled systems. POLYGOTH is an experimental language to implement the ideas just exposed on a network of multiprocessors [Banâtre et al., 1988].

3 Informal presentation of the kernel

A POLYGOTH program describes the behaviour of a (logically) distributed system in terms of fragmented objects interacting by exchanging messages. Objects have a morphology, that is to say they are made of subobjects called *fragments*. These fragments hold two kinds of data: private and partial data. Private data is only available to the fragment that holds it. Partial data results from splitting a global data amongst the (object) fragments. All the (global) data an object may possess must be placed under the custody of its fragments and splitted if more than one fragment is concerned. A global data is always fragmented –possibly in one piece– and its constituents reside in the object fragments. Additional variables are introduced in order to name the data components issued from splitting the global variables.

Objects are protection units. Their data cannot be accessed by any other object of the system. Moreover, their morphology is known only to themselves. Objects manage their data using *variables* called instance variables. Variables contain references either to other objects –since all data are objects–, to the object they belong to or to a special object denoted by *nil*. Assigning an object to a variable makes it refer to the assigned object.

Objects are characterized by the *class* they belong to, except that *nil* belongs to all classes. A class is the description of a set of objects that share the same morphology (or body), global variables and methods. A class also describes what should be done at instancing time: each fragment of its body specifies a block of the multiprocedure that is called whenever an instance is created. As for any multiprocedure, parameters may be declared; they are known as class parameters. The *new* operation provided for classes enables dynamic object creation. Multiprocedures are the sole source of parallelism in the language.

Objects have the ability to act on their data obeying very strict protocols known as their *methods*. Methods are multiprocedures. The blocks of these multiprocedures are *placed* on the object fragments. The placement of a block on a fragment expands the scope of the fragment variables throughout the block. Hence, method blocks can access instance variables. They can have local variables as well. Additionally, they may call,

either with a simple or with a coordinated call, any method offered by their object. These calls are known as internal and they are immediately executed. In order to achieve a high degree of local processing, method blocks are generally placed to match the fragmentation of the data they treat.

Objects may only interact by sending messages to each other. The sending object must specify the receiving one. A message is a request for the designated object. If understood by the receiver –i.e. if a method is defined to accept that request–, it will be handled as soon as all the fragments concerned by the corresponding method are found idle. Meanwhile, the sender is suspended, waiting for a result message.

There are some primitive objects in the language. Only integer and boolean objects will be presented in this semantics. The semantics described in this paper is a formalization of what is reported in [Ployette et al., 1987, Lecler, 1989]. When in doubt, the actual implementation, presented in [Lecler, 1989], served as the reference.

4 Syntax

This section deals with the syntax adopted to define the presented kernel. It differs from the actual (concrete) syntax of POLYGOTH. It is chosen abstract and concise to simplify the description of the semantics. Moreover, additional syntactic elements have been introduced to enhance clarity. Technical paragraphs, labeled with the $\boxed{\mathrm{Z}}$ sign, explain the used notation.

4.1 Variables and names

The different sets of syntactic elements named hereafter are assumed to be given. Let Var_{Glob} be the set of global variables with typical elements sx, sy or sz[2]. These variables are known to the different fragments of the object. Similarly, we introduce:

- the set of partial variables, ($fx, fy, fz \in$) Var_{Part}. These variables are used to name the various "pieces" issued from splitting the global data,
- the set of private variables, ($x, y, z \in$) Var_{Priv}, that is, fragment's local "state" variables, and
- the set of local variables, that is, multiprocedure's variables, ($u, v, w \in$) Var_{Loc}.

We introduce these names into our **Z** specification with the following notation:

$[Var_{Glob}, Var_{Part}, Var_{Priv}, Var_{Loc}]$.

We also need to distinguish different sets of names:

$[Name_{Method}, Name_{Class}]$

- the set of method names, ($mf \in$) $Name_{Method}$, and finally,
- the set of class names, ($C \in$) $Name_{Class}$.

[2]From now on, we use the ($a, b, c \in$) A notation to introduce a set A whose typical elements are a, b, c.

Methods and classes have fragmented bodies. The corresponding components, namely blocks and fragments, are identified with a natural number. We introduce the following type renamings in order to clarify their meaning in future uses:

$$Count == \mathbb{N}$$
$$BodyNum_{Meth} == \mathbb{N}_1$$
$$BodyNum_{Class} == \mathbb{N}_1$$

[Z] The above expressions are *abbreviation definitions*. They introduce global constants in the specification. The identifier on the left of the == symbol has the value and the type of the expression on the right of it.

[Z] $\mathbb{N}_1$ stands for the set of natural positive numbers ($\mathbb{N} - \{0\}$). ◇

Objects are not fully characterized. Instead, the naming scheme used to denote them is presented in the next section.

4.2 Objects and all that jazz

We rely on the reader's intuition of what an object is. A mathematical description of its nature is not of our concern here. We are mainly interested in showing the effect programs have on objects. We restrict ourselves to their denotation through a sound naming scheme. In the sequel of this paper, whenever we write "object" we really mean its name. This somehow misleading abbreviation is only motivated by the otherwise great number of "name of the object" awkward occurrences.

We define the set of non-standard objects, *NObj*, with typical elements α,α' or α_i, by taking sequences of natural numbers, $\mathbb{N}$,

$$NObj == \mathrm{seq}\,\mathbb{N}$$

[Z] $\mathrm{seq}\,X$ is the set of finite sequences over X. These are finite functions from $\mathbb{N}$ to X whose domain is a segment $1\,..\,n$ for some natural number n. We write $\langle a_1, .., a_n \rangle$ as a shorthand for the set $\{1 \mapsto a_1, .., n \mapsto a_n\}$. The empty sequence is noted $\langle\rangle$. The i^{th} element of a sequence s is noted $s(\ i\ )$. ◇

A non-standard object is uniquely named with a sequence of natural numbers. This naming scheme is explained in section 5.4.2. *NObj* is the set of names given to objects built by program. These objects are called non-standard because they have to be built. By contrast, standard objects are those which are already built in the system. They correspond to the primitive types in most languages (integer, boolean etc.). Let *tt* and *ff* denote the boolean objects *true* and *false*. Let *nil* stand for the special object which belongs to all classes.

We now introduce some structure to our objects. Indeed, splitting objects presupposes they are composed or structured because the issued "pieces" are also objects. We retain the sequence as the only object composer. So, an object is:

- one of the three constants denoted *nil*, *tt* or *ff*, or

- an integer n denoted $int(\ n\ )$, or
- a non-standard object α denoted $obj(\ \alpha\ )$, or
- a finite non-empty sequence of objects $\langle \gamma_1 \,..\, \gamma_k \rangle$ denoted $seqobj(\ \langle \gamma_1, .., \gamma_k \rangle\ )$.

We define the set of all objects, $(\ \gamma, \gamma', \gamma_i \in\ )$ OBJ, as follows:

$$
\begin{aligned}
OBJ ::= & \ nil \mid tt \mid ff \\
& \mid int \ \langle\!\langle \mathbb{Z} \rangle\!\rangle \\
& \mid obj \ \langle\!\langle NObj \rangle\!\rangle \\
& \mid seqobj \ \langle\!\langle \ \mathrm{seq}_1 \ OBJ \ \rangle\!\rangle
\end{aligned}
$$

Z The OBJ set is defined as a *free type*. The notation for free types adds nothing to the power of the **Z** language, but it makes it easier to describe recursive structures. The above definition introduces a new basic type OBJ, 3 constants of type OBJ (namely *nil*, *tt* and *ff*) and 3 new variables (namely *int*, *obj* and *seqobj*). Given OBJ, *int* is interpreted as an injection, noted $\rightarrowtail$, from $\mathbb{Z}$ to OBJ, *obj* as $NObj \rightarrowtail OBJ$ and *seqobj* as $\mathrm{seq}_1\ OBJ \rightarrowtail OBJ$.

Z $\mathrm{seq}_1\ X$ stands for the set of non empty sequences of X. The size operator over finite sets is written $\#$, for instance $\#\langle a, b, c \rangle = 3$ and $\#\langle\rangle = 0$. The following holds $\mathrm{seq}_1\ X == \{\ s : \mathrm{seq}\ X \mid \#s > 0\ \}$. $\diamond$

4.3 Expressions

We now define the set $(\ e \in\ )$ Exp of expressions followed by an informal explanation of their intended meaning.

$$
\begin{aligned}
Exp ::= & \ svar \ \langle\!\langle \ Var_{Glob} \ \rangle\!\rangle \mid fvar \ \langle\!\langle \ Var_{Part} \ \rangle\!\rangle \\
2 \quad & \mid pvar \ \langle\!\langle \ Var_{Priv} \ \rangle\!\rangle \mid lvar \ \langle\!\langle \ Var_{Loc} \ \rangle\!\rangle \\
3 \quad & \mid call \ \langle\!\langle \ Name_{Method} \times \mathrm{seq}\ Exp \ \rangle\!\rangle \\
4 \quad & \mid cocall \ \langle\!\langle \ Name_{Method} \times \mathrm{seq}(\ Var_{Loc} \times Exp) \ \rangle\!\rangle \\
5 \quad & \mid meth \ \langle\!\langle \ Exp \times Name_{Method} \times \mathrm{seq}\ Exp \ \rangle\!\rangle \\
6 \quad & \mid cometh \ \langle\!\langle \ Exp \times Name_{Method} \times \mathrm{seq}(\ Var_{Loc} \times Exp) \ \rangle\!\rangle \\
7 \quad & \mid new \ \langle\!\langle \ Name_{Class} \times \mathrm{seq}\ Exp \ \rangle\!\rangle \\
8 \quad & \mid conew \ \langle\!\langle \ Name_{Class} \times \mathrm{seq}(\ Var_{Priv} \times Exp) \ \rangle\!\rangle \\
9 \quad & \mid name \ \langle\!\langle \ OBJ \ \rangle\!\rangle \\
10 \quad & \mid self \\
11 \quad & \mid wait \ \langle\!\langle \ NObj \times Name_{Method} \ \rangle\!\rangle
\end{aligned}
$$

4.3.1 Variables

The first and second lines enable the use of variables as expressions. They yield as their value the name of the object they currently store. The shared variable is peculiar since it typically stores a composed object. Its evaluation then yields a value composed of the objects stored by the various partial variables it is associated to.

4.3.2 Method call

A call expression –line 3– of the form *call*($mf, \langle e_1, .., e_n \rangle$) results in evaluating parameters e_1 through e_n, from left to right, and then executing the method named mf. Local variables are initialized in every method fragment either to their corresponding parameter value, if any according to the parameter function definition, or to *nil*. Then, the statements found in the definition of mf are executed in parallel, yielding a composed value which is the result of the expression.

A coordinated call expression –line 4– is a partial expression. All concerned fragments of the caller have to evaluate a matching coordinated call in order to proceed. Coordinated calls match if they all call the same method and they provide the required parameters. The set of coordinated calls is complete when each fragment evaluates the parameters it provides. All of them are synchronized to issue a single call. They all get the result which is the (composed) value yield by the execution of the defining statements of the called method mf.

4.3.3 Communications

An external method call expression –line 5– of the form *meth*($e, mf, \langle e_1, .., e_n \rangle$) results in evaluating the destination object, e. Evaluation of parameters e_1 through e_n follows and then, a request is sent to the destination object for it to execute its mf method. When the body fragments of the destination object on which the method mf is allocated are all idle, the request is served as if it were a local call. The value which results from this evaluation is sent back to the caller and it constitutes the value of the external method call expression.

The external method coordinated call expression –line 6– results in evaluating the destination object, e. When all the concerned fragments of the caller reach their matching calls and each fragment has evaluated the parameters it provides, all of them are synchronized to send a single request to the destination object for it to execute its mf method. When the body fragments of the destination object on which the method mf is allocated are all idle, the request is served as if it were a local call. The value which results from this evaluation is sent back to the caller. It constitutes the value of the external method coordinated call expression and it is available to all the concerned fragments of the caller.

4.3.4 Object creation

The object creation expression –line 7– of the form *new*($C, \langle e_1, .., e_n \rangle$) creates an instance of the class named C. Parameters e_1 through e_n are evaluated then, a new instance of the C class is created. Its instance variables are initialized either to the corresponding parameter values, if any, or to *nil*. Its body statements are executed and when they are all terminated, the name of the created object is returned as the result of the object creation expression.

The coordinated object creation expression –line 8– is quite similar to the previous one. This expression synchronizes all the concerned fragments at their matching point. Each fragment evaluates the parameters it provides and then, a new object of the C class is created. Its instance variables are initialized either to the corresponding parameter values, if any, or to *nil*. Its body statements are executed. When they all

terminate, the name of the created object is returned to all the creating fragments as the result of the expression.

4.3.5 Object identity

Direct naming of objects is available for any known object in the surrounding environment. The direct naming expression –line 9– of the form *name*(γ) names object γ. The expression *self* –line 10– always yields the name of the object evaluating it.

4.3.6 Sugar

The expression appearing in line 11 does not belong to the actual syntax of the kernel. It cannot be written by a programmer. It is included here to ease the writing of the semantic rules in section 5. The wait expression, line 11, of the form *wait*(α, mf) indicates that the results of an execution of mf are awaited from α.

4.4 Statements

The set ($s \in$) *Stmt* of statements is described hereafter. An informal explanation of their meaning follows.

$$
\begin{array}{rl}
 & Stmt ::= fassign \langle\!\langle\ Var_{Part} \times Exp\ \rangle\!\rangle \\
2 & \quad \mid passign \langle\!\langle\ Var_{Priv} \times Exp\ \rangle\!\rangle \\
3 & \quad \mid lassign \langle\!\langle\ Var_{Loc} \times Exp\ \rangle\!\rangle \\
4 & \quad \mid massign \langle\!\langle\ Var_{Glob} \times Exp\ \rangle\!\rangle \\
5 & \quad \mid exp \langle\!\langle\ Exp\ \rangle\!\rangle \\
6 & \quad \mid seqcomp \langle\!\langle\ Stmt \times Stmt\ \rangle\!\rangle \\
7 & \quad \mid if \langle\!\langle\ Exp \times Stmt \times Stmt\ \rangle\!\rangle \\
8 & \quad \mid do \langle\!\langle\ Exp \times Stmt\ \rangle\!\rangle \\
9 & \quad \mid send \langle\!\langle\ Stmt \times CallId\ \rangle\!\rangle
\end{array}
$$

4.4.1 Assignment

The four first statements –lines 1 through 4– are assignments. They assign the result of evaluating the expression they apply to respectively, to a partial variable, a private variable, a local variable and a global variable. The assignment is executed by evaluating the expression and making the variable refer to the resulting object. Only results of expressions are assignable. This is the only semantic difference between statements and expressions. The execution of an assignment yields the object it makes the (assigned) variable refer to.

4.4.2 Expression

An expression –line 5– of the form *exp*(e) where $e \in Exp$, is also a statement and it yields e's evaluation.

4.4.3 Composition

Sequential composition of statements –line 6– of the form *seqcomp*(s_1, s_2) where s_1 and $s_2 \in Stmt$, is executed as usual. The value produced by s_2 is yield as the result.

4.4.4 Branch and loop

Branches and loops –lines 7 and 8– are executed as usual. A branch of the form $if(\ e, s_1, s_2\ )$ produces the result of the selected statement according to e's evaluation. The loop produces a *nil* result when it ends.

4.4.5 Send

The statement in line 9 is not available to the programmer. It does not belong to the syntax of the language. It is included to enable the writing of semantic rules to express both the return of a local method call and the return of an external one. *CallId* is a set of encoded names to identify the calling object and the fragments waiting for the answer.

$$CallId == NObj \times \mathbb{F}_1\ BodyNum_{Class}$$

Z $\mathbb{F}\ X$ is the set of finite subsets of X. $\mathbb{F}_1\ X$ is the non-empty version of $\mathbb{F}\ X$. ◇

In a statement of the form $send(\ s, (\alpha, \{i_1, .., i_k\})\ )$, statement s is executed and the result it yields is sent to fragments numbered $i_1, .., i_k$ of the non-standard object α.

4.5 Declarations

4.5.1 Blocks

A block is made of a finite subset of variables, called its *Var* component, drawn from a generic set named *VAR* and of a statement, called its *s* component.

BLOCK[*VAR*]

$Var : \mathbb{F}\ VAR$

$s : Stmt$

Z A schema notation introduces a new schema name, in this case *BLOCK*. A schema is made of a *declaration part* and an optional *predicate part*. In this case, there is no predicate part, but there is a *generic parameter*, *VAR*. The declaration part introduces identifiers and associates them with types. The colon binds the identifier preceding it to the type following it. The introduced identifiers are known as the schema components. Let b be a *BLOCK*; then $b.Var$ and $b.s$ denote the two components of block b. ◇

The set of method blocks, *Block*, is the set of blocks whose variables are drawn from the set of local variables Var_{Loc}.

Block

$BLOCK[Var_{Loc}]$

Z This schema is an instance of the generic one above with Var_{Loc} as the actual parameter. It is equivalent to the schema below:

Block

$Var : \mathbb{F}\ Var_{Loc}$

$s : Stmt$

◇

The set of *fragments* or class blocks, *Fragment*, is the set of blocks whose variables are drawn from the set of partial variables Var_{Part} and that are extended with a *Priv* component. The extra component of a class block is a finite set of variables drawn from the set of private variables Var_{Priv}.

Fragment

$BLOCK[Var_{Part}]$

$Priv : \mathbb{F}\ Var_{Priv}$

4.5.2 Method

A method being a multiprocedure is composed of parallel blocks that are placed on different fragments.

METHOD

$Param$: iseq Var_{Loc}

$Alloc : BodyNum_{Class} \rightarrowtail\!\!\!\!\!\!\rightarrow BodyNum_{Meth}$

$Body$: $\text{seq}_1\ Block$

Z iseq X is the set of injective finite sequences over X: these are precisely the finite sequences over X which contain no repetitions. ◇

Method blocks, organized as a non-empty sequence, form the *Body* component of a method. Parameters are specified with a (possibly empty) sequence of local variables in the *Param* component. No repetitions are allowed since it is necessary to uniquely name formal parameters in a coordinated call. The placement of method blocks on class fragments is described by the *Alloc* component. It is a finite injection, denoted by the $\rightarrowtail\!\!\!\!\!\!\rightarrow$ symbol, from class body numbers to method body numbers. The body number of a block is simply its index in the sequence of the body it belongs to. For instance, if a method *mf* has a body $\langle b_1, b_2, b_3 \rangle$ whose blocks are placed on fragments number 4, 1 and 5 respectively, then the *Alloc* component of *mf*, noted *mf.Alloc*, would be:

$$\{\ 4 \mapsto 1, 1 \mapsto 2, 5 \mapsto 3\ \}.$$

Some restrictions apply to the components of this schema:

- every method block has to be placed on a different fragment:

  $\text{ran}\ Alloc = \text{dom}\ Body;$

Z The *domain* (dom) of a binary relation between X and Y is the set of members of X which are related to at least one member of Y by that relation. The *range* (ran) is the set of members of Y to which at least one member of X is related. ◇

- all local variables appearing in the *Param* sequence have to belong to at least one block of the method body:

$$\forall i : 1 \mathinner{..} \#Param \bullet \exists j : 1 \mathinner{..} \#Body \bullet \\ Param(\, i\,) \in (Body(\, j\,)).Var.$$

Z The large dot, •, is used in **Z** to bind declarations to expressions. Like many programming languages, **Z** has a system of nested scopes. ◇

We incorporate these restrictions into the *METHOD* schema with a *predicate part* written as follows:

$$\begin{array}{l} \underline{METHOD} \\ Param : \mathrm{iseq}\ Var_{Loc} \\ Alloc : BodyNum_{Class} \rightarrowtail\!\!\!\twoheadrightarrow BodyNum_{Meth} \\ Body : \mathrm{seq}_1\ Block \\ \hline \mathrm{ran}\ Alloc = \mathrm{dom}\ Body \\ \forall i : 1 \mathinner{..} \#Param \bullet \exists j : 1 \mathinner{..} \#Body \bullet \\ \quad Param(\, i\,) \in (Body(\, j\,)).Var \end{array}$$

Z The *predicate part* of a schema expresses constraints on the values of the declared identifiers. It is separated from the declaration part with an horizontal bar. Values are constrained by the schema's *invariant*, that is to say, the logical conjunction of the predicates appearing in this part of the schema. ◇

4.5.3 Class

A class is specified by four components: the class parameters modeled with a (possibly empty) sequence of private variables, the global variables and their associated partial variables, the set of methods provided and the body which is defined as a non-empty sequence of fragments.

$$\begin{array}{l} \underline{CLASS} \\ Param : \mathrm{iseq}\ Var_{Priv} \\ Virtual : Var_{Glob} \twoheadrightarrow\!\!\!\!\!+ \mathrm{seq}_1\ Var_{Part} \\ Method : Name_{Method} \twoheadrightarrow\!\!\!\!\!+ METHOD \\ Body : \mathrm{seq}_1\ Fragment \end{array}$$

Class parameters are not fragmentable. However, they may be shared by several fragments. That is why private variables are used. It is required in section 3 that all global

variables are fragmented in partial variables. The *Virtual* component of the schema specifies a mapping from the declared global variables to their corresponding partial variables, structured as a non-empty sequence. It is called virtual because, as the formal semantics reveals, these global variables are not necessarily a concrete reality; they rather appear as a global access mechanism to partial variables. The *Method* component binds method names to method declarations. Let *init* be the name of the special method used to initialize objects variables. That name is reserved, in this specification, for the multiprocedure defined by the *Body* component of a class. Furthermore, every class provides the *init* method name and binds it to a special method declaration:

$$| \; init : Name_{Method}$$

Z The specification paragraph above is known as an *axiomatic description.* It introduces one or more global variables, and optionally specifies a constraint on their values. These variables must not have a previous global declaration, and their scope extends from their declaration to the end of the specification. Just as in the schema notation, an horizontal bar separates the declaration from the predicates formalizing the constraints, if any. ◇

Some constraints have to be posed on the *CLASS* schema for a later need of the following properties: partial variables are uniquely bound to global variable fragments, and private variables used as class parameters belong to some class fragment. These constraints are formalized in the predicate part of the schema and explained hereafter.

CLASS

$$\begin{array}{l} Param : \text{iseq } Var_{Priv} \\ Virtual : Var_{Glob} \twoheadrightarrow \text{seq}_1 \; Var_{Part} \\ Method : Name_{Method} \twoheadrightarrow METHOD \\ Body : \text{seq}_1 \; Fragment \\ \hline \forall i : 1 \,..\, \#Body \bullet \\ \quad \forall fx : Var_{Part} \mid fx \in (Body(\, i \,)).Var \bullet \\ \quad\quad \exists sx : Var_{Glob} \mid sx \in \text{dom } Virtual \bullet fx \in \text{ran}(Virtual(\, sx \,)) \\ \forall sx, sy : Var_{Glob} \mid sx \in \text{dom } Virtual \wedge sy \in \text{dom } Virtual \bullet \\ \quad \forall i : 1 \,..\, \#(Virtual(\, sx \,)); \; j : 1 \,..\, \#(Virtual(\, sy \,)) \bullet \\ \quad\quad (Virtual(\, sx \,))(\, i \,) = (Virtual(\, sy \,))(\, j \,) \Rightarrow (sx = sy \wedge i = j) \\ \forall sx : Var_{Glob} \mid sx \in \text{dom } Virtual \bullet \\ \quad \forall i : 1 \,..\, \#(Virtual(\, sx \,)) \bullet \exists j : 1 \,..\, \#Body \bullet \\ \quad\quad (Virtual(\, sx \,))(\, i \,) \in (Body(\, j \,)).Var \\ \forall i, j : 1 \,..\, \#Body \mid i \neq j \bullet \\ \quad (Body(\, i \,)).Var \cap (Body(\, j \,)).Var = \varnothing \\ \forall i : 1 \,..\, \#Param \bullet \exists j : 1 \,..\, \#Body \bullet \\ \quad Param(\, i \,) \in (Body(\, j \,)).Priv \end{array}$$

- each partial variable is a fragment of exactly one global variable:

– that is, at least one,

$$\forall i : 1 \mathinner{..} \#Body \bullet \\ \forall fx : Var_{Part} \mid fx \in (Body(\ i\ )).Var \bullet \\ \exists sx : Var_{Glob} \mid sx \in \mathrm{dom}\ Virtual \bullet fx \in \mathrm{ran}(Virtual(\ sx\ ));$$

– and at most one.

$$\forall sx, sy : Var_{Glob} \mid sx \in \mathrm{dom}\ Virtual \wedge sy \in \mathrm{dom}\ Virtual \bullet \\ \forall i : 1 \mathinner{..} \#(Virtual(\ sx\ ));\ j : 1 \mathinner{..} \#(Virtual(\ sy\ )) \bullet \\ (Virtual(\ sx\ ))(\ i\ ) = (Virtual(\ sy\ ))(\ j\ ) \Rightarrow (sx = sy \wedge i = j);$$

- each partial variable composing a global variable must belong to exactly one fragment:

  – that is, at least one,

$$\forall sx : Var_{Glob} \mid sx \in \mathrm{dom}\ Virtual \bullet \\ \forall i : 1 \mathinner{..} \#(Virtual(\ sx\ )) \bullet \exists j : 1 \mathinner{..} \#Body \bullet \\ (Virtual(\ sx\ ))(\ i\ ) \in (Body(\ j\ )).Var;$$

  – and at most one.

$$\forall i, j : 1 \mathinner{..} \#Body \mid i \neq j \bullet \\ (Body(\ i\ )).Var \cap (Body(\ j\ )).Var = \varnothing;$$

- each private variable appearing in the parameter sequence must belong to at least one fragment:

$$\forall i : 1 \mathinner{..} \#Param \bullet \exists j : 1 \mathinner{..} \#Body \bullet \\ Param(\ i\ ) \in (Body(\ j\ )).Priv.$$

4.5.4 Program

We call $(u \in) UNIT$ the set of syntactically correct programs[3]. A *unit* is typically composed of a *Program*, which is a collection of named class declarations, and a *Main* class name, which is the program starting point. A brief explanation of the *UNIT* schema's invariant follows its definition.

UNIT

$$Program : Name_{Class} \nrightarrow CLASS \\ Main : Name_{Class}$$

$$Main \in \mathrm{dom}\ Program$$

$$(Program(\ Main\ )).Param = \langle\rangle$$

$$\forall \tau : Name_{Class} \mid \tau \in \mathrm{dom}\ Program \bullet \\ init \in \mathrm{dom}((Program(\tau)).Method) \wedge \\ (((Program(\tau)).Method)(\ init\ )).Alloc = \mathrm{dom}((Program(\tau)).Body) \lhd \mathrm{id}\ \mathbb{N}$$

[3]In fact, static semantics is also supposed correct for members of *UNIT*. Although PolyGoth is a strongly typed language, we do not get into its typing system in this paper. We assume all units are type-correct.

[z] The domain restriction $S \lhd R$ of a relation R to a set S relates x to y if and only if R relates x to y and x is a member of S. For instance,

$$\{ 1,4,6 \} \lhd \mathrm{id}\, \mathbb{N} = \{ 1 \mapsto 1, 4 \mapsto 4, 6 \mapsto 6 \}$$

◇

The starting point of the unit must be well-defined. The *Main* component of a unit must be in the domain of the *Program*. The starting class must be parameterless. Its *Param* component has to be the empty sequence. Every class of the program possesses a special initialization method, named *init*, whose statement components –those of the class body– are "placed" on each fragment of its body.

5 Semantics

Our operational semantics is based on a transition system "à la Plotkin" [Plotkin, 1981, Hennessy and Plotkin, 1979]. Informally –and hence unprecisely– a transition system is a structure that enables the description of a system's behaviour in terms of configurations and transitions. Loosely speaking, configurations are descriptions of (recognizable or discernible) states of the system and transitions are "elementary moves" between configurations. How elementary should the moves be is highly motivated by the expected notion of behavior. We want to exhibit the changes produced by the execution of the different syntactic constructs of the language. It seems natural then to choose the execution of syntactic constructs as the elementary moves between configurations of objects.

In the following sections, we provide a store representation for our objects and we define a chain of states and configurations to suit our purposes. We intend to show how each syntactic construct is to be understood. We use as little information as possible in doing so. We enrich our states and configurations gradually as more information is required. An outline of the three stages of our specification, namely the *fragment*, the *object* and the *system* levels, is given hereafter.

At the *fragment level*, those syntactic constructs whose execution concerns solely the (executing) fragment are dealt with. A partial state is characterized in section 5.2 and it is used to exhibit several basic constructs such as integer operations, local assignments, control statements, etc.

The *object level* covers those constructs whose execution concerns solely the (executing) object. A (local) configuration that incorporates the different partial states is defined in section 5.3. Changes in these local configurations describe the set of transitions involving only one object. For instance, global assignment, internal method call and *self* evaluation are characterized by these transitions.

The *system level* captures all possible interactions amongst objects. A (global) configuration is defined in section 5.4 in terms of local ones. Within that framework, object creation and external method call semantics are shown.

5.1 Storage representation

Let us take a closer look at objects. They are characterized by the class they are an instance of. The class declaration specifies the structure of all its instances: their

data and their associated methods. However, objects are active entities. They are responsible for actual data handling. They store their *permanent* and their *ephemeral* data. They change their internal state when they execute their methods.

Permanent data is the one referred by global -hence partial- and private variables. Ephemeral data is the one referred by local variables. Since we have introduced different syntactic sets of variables, it seems wise to structure the store of an object accordingly. We can define the store of an object as a (partial) function from its variables to a set of *values*. Values in our model are simply elements of the set of object names, OBJ, introduced in section 4.2. Therefore, we define the following *stores*:

$$Store_{Part} == Var_{Part} \nrightarrow OBJ$$

$$Store_{Priv} == Var_{Priv} \nrightarrow OBJ$$

$$Store_{Loc} == Var_{Loc} \nrightarrow OBJ$$

$$Store_{Glob} == Var_{Glob} \nrightarrow (Store_{Part})$$

Since methods may be recursively activated, it seems convenient to organize ephemeral store in a stack manner. Stacks are not provided as such by **Z**, but they can easily be expressed in terms of sequences (see A.1). Let AR be the set of activation records defined by the following schema:

AR

$Cur : Name_{Method}$

$Env : Store_{Loc}$

The name of the method being executed is recorded in the *Cur* component of the schema in order to express the synchronization requirement of the coordinated call semantics. We need to know the name of the method executing a coordinated call to identify all its components and to express that they all have to match the call, as it is explained in section 4.3.

5.2 Fragment level

A partial state is composed of the statement about to be executed, a partial store, a private store, a stack of activation records and a counter:

$State_\partial$

$s : Stmt$

$\delta : Store_{Part}$

$\phi : Store_{Priv}$

$\rho :$ stack(AR)

$\eta : Count$

The last component, *Count*, holds the number of objects that are created on the fragment. As it is explained in section 5.4.2, this information is needed in order to locally name objects and still ensure that unique names are provided.

All the transitions at this level involve two sets of values: the values of the partial state before and those after the transition. Several transitions induce only a change

in the statement component of the partial state, leaving all the others unchanged. A schema is defined to encompass this remark.

$\Xi\ State_\partial$

$$State_\partial$$
$$State_\partial'$$
$$\delta = \delta' \wedge \phi = \phi' \wedge \rho = \rho' \wedge \eta = \eta'$$

Z The above schema introduces all the identifiers declared in the schema $State_\partial$ and their *decorated* version introduced by the decorated schema $State_\partial'$. If A is a schema, A' is a copy of A in which all the component names have been decorated with $'$. When an identifier which already has a decoration is decorated, the two decorations are juxtaposed, with the new decoration on the right. There are three standard decorations used in describing operations: $'$ for labelling the final state of an operation, ? for labelling its input, and ! for labelling its outputs. Subscript digits may also be used as decorations.

Z The above schema's invariant is the logical conjunction of $State_\partial$'s invariant, the one of $State_\partial'$ and the extra requirement specified in the predicate part. ◇

5.2.1 Some standard object operations

Let *add*, *sub*, *div* and *mod* be the names of the usual methods provided for integer objects.

$$add, sub, div, mod : Name_{Method}$$

Operations on standard objects are formulated by message sending. For example, the addition $4 + 3$ is formulated with the following expression (concrete syntax could be $4 + 3$ or $4\ !\ add(\ 3\ )$):

$$meth(\ name(int(\ 4\ )), add, \langle\ name(int(\ 3\ ))\ \rangle\ ).$$

The successful evaluation of these integer binary operations are specified by the schema below.

IntBinOp

$\Xi\ State_\partial$
$i, j : \mathbb{Z}$
$Op : \{\ add, sub, div, mod\ \}$

$$s = exp(meth(\ name(int(\ i\ )), Op, \langle name(int(\ j\ ))\rangle\ ))$$
$$Op = add \Rightarrow s' = exp(name(int(\ i + j\ )))$$
$$Op = sub \Rightarrow s' = exp(name(int(\ i - j\ )))$$
$$(Op = div \wedge j \neq 0) \Rightarrow s' = exp(name(int(\ i \operatorname{div} j\ )))$$
$$Op = mod \Rightarrow s' = exp(name(int(\ i \bmod j\ )))$$

5.2.2 Chat about errors

Although the **Z** schema *calculus* is particularly well suited to specify them, error conditions are left unspecified in this paper. The exception handling mechanism for POLYGOTH is still in its design stage (see [Issarny, 1989] for foundations), and this specification is large enough for a first attempt. In future versions, error conditions will surely be taken care of. Just in order to give an idea of how error conditions may be dealt with, we provide a *DivideByZero* schema and indicate how it is incorporated into the global specification.

Let $Error_\partial$ be a special partial error state and $\Delta\ State_\partial$ be a schema used to introduce only the variables of the partial state before and after the transition.

$$Error_\partial : State_\partial$$

$\Delta\ State_\partial$

$State_\partial$

$State_\partial'$

Then, a division by zero can be specified as an error-leading transition:

DivideByZero

$\Delta\ State_\partial$

$i : \mathbb{Z}$

$s = exp(meth(\ name(int(\ i\ )), div, \langle name(int(\ 0\ ))\rangle))$

$\theta State_\partial' = Error_\partial$

[z] The value of the expression $\theta State_\partial'$ is a binding z with the following schema type: $\lhd s : Stmt;\ \delta : Store_{Part};\ \phi : Store_{Priv};\ \rho :$ stack(AR); $\eta : Count \rhd$; the value of each component of z is the value of the corresponding variable declared in the current environment. In the present case, those variables are introduced with the $State_\partial'$ declaration. The predicate $\theta State_\partial' = Error_\partial$ requires that the binding z be equal to the values corresponding to the partial error state, whichever those may be. Note that the types of the components of a binding θS, where S is a schema, are taken from the current environment, and not from the schema. ◇

A more robust specification of the integer binary operations can be written as follows:

$$RobIntBinOp \triangleq IntBinOp \vee DivideByZero$$

which is equivalent to the expanded version below:

RobIntBinOp

$$
\begin{array}{l}
\Delta\ State_\partial \\
i, j : \mathbb{Z} \\
Op : \{\ add, sub, div, mod\ \} \\
\hline
(\ \delta = \delta' \wedge \phi = \phi' \wedge \rho = \rho' \wedge \eta = \eta' \wedge \\
\quad s = exp(meth(\ name(int(\ i\ )), Op, \langle name(int(\ j\ ))\rangle\ )) \wedge \\
\quad (Op = add \Rightarrow s' = exp(name(int(\ i + j\ )))) \wedge \\
\quad (Op = sub \Rightarrow s' = exp(name(int(\ i - j\ )))) \wedge \\
\quad ((Op = div \wedge j \neq 0) \Rightarrow s' = exp(name(int(\ i \operatorname{div} j\ )))) \wedge \\
\quad (Op = mod \Rightarrow s' = exp(name(int(\ i \bmod j\ )))\ ) \\
\quad \vee \\
(\ s = exp(meth(\ name(int(\ i\ )), div, \langle name(int(\ 0\ ))\rangle)) \wedge \\
\quad \theta State_\partial' = Error_\partial\ )
\end{array}
$$

Integer and boolean relations are treated in appendix section A.3.1. The schemas describing them are quite similar to the one just presented; hence, no further explanations are provided.

5.2.3 Some expressions

We describe a private variable expression evaluation as an example of this kind of expressions.

PrivVarExp

$$
\begin{array}{l}
\Xi\ State_\partial \\
x : Var_{Priv} \\
\hline
x \in \operatorname{dom} \phi \\
s = exp(pvar(\ x\ )) \\
s' = exp(name(\phi(\ x\ )))
\end{array}
$$

The schema simply states that when a private variable, x, declared in the fragment appears as an expression, it yields the object stored by the fragment's private store, ϕ. Variations on this schema can be found in appendix section A.3.1, where appropriate store functions are used for each syntactic set of variables. Notice that only the semantics of local partial variables as expressions may be formalized at this level. The schema calculus is employed to complement this partial definition in section 5.3.

Assignments are treated in a similar way. We reproduce the local variable assignment hereafter to exemplify this group of schemas.

LocAssign

$\Delta\ State_\partial$
$u : Var_{Loc}$
$ar : AR$
$\gamma : OBJ$

$u \in \mathrm{dom}(\mathrm{top}(\ \rho\ )).Env$
$s = lassign(\ u, name(\ \gamma\ )\ )$
$ar.Cur = (\mathrm{top}(\ \rho\ )).Cur$
$ar.Env = (\mathrm{top}(\ \rho\ )).Env \oplus \{\ u \mapsto \gamma\ \}$
$s' = exp(name(\ \gamma\ ))$
$\delta' = \delta \wedge \phi' = \phi$
$\rho' = \mathrm{push}(\ \mathrm{pop}(\ \rho\ ), ar\ )$
$\eta' = \eta$

Z The $\oplus$ operation is known as *functional overriding*. Let f and g be two functions with same signatures. The domain of $f \oplus g$ is the union of the two domains and for all $x \in \mathrm{dom}\, f \oplus g$ we have:

$$f \oplus g(x) = \begin{cases} g(x) & \text{if } x \in \mathrm{dom}\, g \\ f(x) & \text{otherwise.} \end{cases}$$

Z In this schema, ar and γ are used as auxiliary variables. Normally, an existential quantifier is used in **Z** to declare auxiliary variables, but it is sometimes cumbersome to do so; this is one of those. They would have been introduced with the *let ... in* in VDM [Jones, 1986]. ◇

The current activation record, ar, is updated in its local store component, Env, to reflect the new value, γ, associated with this assignment to the local variable u. Notice how pre-conditions are grouped and written before the group of post-conditions.

5.2.4 A control statement

The *If* schema is used to illustrate the specification of a control statement.

If

$\Xi\ State_\partial$
$\beta : \{\ tt, ff\ \}$
$s_1, s_2 : Stmt$

$s = if(\ name(\ \beta\ ), s_1, s_2\ )$
$\beta = tt \Rightarrow s' = s_1$
$\beta = ff \Rightarrow s' = s_2$

5.2.5 Summarizing

The schema calculus is used to aggregate all the previously defined basic transitions into a single partial state change named $Axioms_{Basic}$:

$Axioms_{Basic}$

$\Delta\ State_\partial$

$(\exists_1 i, j : \mathbb{Z} \bullet$
$(\exists_1 Op : \{ add, sub, div, mod \} \bullet IntBinOp) \lor$
$(\exists_1 Rel : \{ equal, less, greater \} \bullet IntBinRel)) \lor$

$(\exists_1 \beta : \{ tt, ff \} \bullet$
$(\exists_1 \gamma : \{ tt, ff \};\ Op : \{ and, or \} \bullet BoolBinRel) \lor$
$BoolPreRel \lor$
$(\exists_1 s_1, s_2 : Stmt \bullet If)) \lor$

$(\exists_1 \gamma : OBJ \bullet$
$(\exists_1 fx : Var_{Part} \bullet LocPartVarExp \lor LocPartAssign) \lor$
$(\exists_1 x : Var_{Priv} \bullet PrivVarExp \lor PrivAssign) \lor$
$(\exists_1 u : Var_{Loc};\ ar : AR \bullet LocVarExp \lor LocAssign) \lor$
$(\exists_1 s_1 : Stmt \bullet Do \lor ValueDiscard))$

Z A schema used as an expression is satisfied whenever all its components are in scope and the values they are bound to by the current environment make its invariant hold. For instance,

$\Delta\ State_\partial$

$\exists_1 i, j : \mathbb{Z} \bullet$
$\exists_1 Op : \{ add, sub, div, mod \} \bullet IntBinOp,$

specifies the set of all partial state changes for which two integers, namely i and j, and a binary operation method name can be found such that *IntBinOp*'s invariant holds. Thus, it specifies all those transitions induced by a successful execution of the four usual integer binary operations. ◇

This larger mathematical object describes all the basic transitions a single fragment may handle on its own. This axiom, taken together with the rules exposed in section A.4.1 of the appendix, specifies the abstract machine a fragment is.

5.3 Object level

We introduce the set, $Config_{Local}$, of *local configurations*. A local configuration contains all the information about an active object needed to show the meaning of our programming language notations. We need to identify the object we describe, its morphology and its corresponding partial states. For this, we use the object's name and a non-empty sequence of partial states: one for each fragment. We also need to know the contents of its (global) store and the (name of the) class it belongs to. The environment of the object rests in a unit description. The following schema specifies the set of local configurations.

$Config_{Local}$

$\alpha : NObj$
$\psi : Store_{Glob}$
$\sigma : \mathrm{seq}_1\ State_\theta$
$\tau : Name_{Class}$
$UNIT$

Each local configuration characterizes an active object of the running program. Therefore, some restrictions are imposed. We show how to specify them in **Z**.

- the class has to be defined in the current environment:

  $$\tau \in \mathrm{dom}\ Program$$

- storage is provided exactly for those global variables declared by the class the object belongs to:

  $$\mathrm{dom}\,\psi = \mathrm{dom}((Program(\tau)).Virtual)$$

- an object has as many partial states as the number of fragments its associated class body is declared to be made of:

  $$\#\sigma = \#((Program(\tau)).Body)$$

- every partial state provides storage for exactly those partial and private variables its corresponding class body fragment declares. Furthermore, we naturally choose the correspondence between fragments and partial states to be given by the identity of their respective sequence numbers:

  $$\begin{array}{l}\forall i : 1\,..\,\#\sigma \bullet \\ \quad \mathrm{dom}((\sigma(i)).\delta) = (((Program(\tau)).Body)(i)).Var \wedge \\ \quad \mathrm{dom}((\sigma(i)).\phi) = (((Program(\tau)).Body)(i)).Priv\end{array}$$

- the stack of activation records of a partial state is never empty. The method name kept in the top (activation) record is declared by the class the corresponding object belongs to:

  $$\begin{array}{l}\forall i : 1\,..\,\#\sigma \bullet \\ \quad (\ (\sigma(i)).\rho \neq \mathrm{empty}\ ) \wedge \\ \quad (\ \exists_1\ ar : AR \mid ar = \mathrm{top}(\ (\sigma(i)).\rho\ ) \wedge ar.Cur \in \mathrm{dom}(Program(\tau)).Method \bullet\end{array}$$

  Either that name is *init*, the universal creation-time method name,

  $$ar.Cur = init$$

  or all the blocks of the corresponding method are being executed exactly by the fragments they have been placed on,

  $$\begin{array}{l}\exists_1 M : METHOD \mid M = ((Program(\tau)).Method)(\ ar.Cur\ ) \bullet \\ \quad (\forall k : BodyNum_{Class} \mid k \in \mathrm{dom}\ M.Alloc \bullet \exists n : 0\,..\,\mathrm{size}(\ (\sigma(k)).\rho\ ) \bullet \\ \quad\quad (\mathrm{top}(\ \mathrm{pop}^n(\ (\sigma(k)).\rho\ )\ )).Cur = ar.Cur)\end{array}$$

and ephemeral storage is provided for exactly those local variables declared in the corresponding method block.

$$i \in \mathrm{dom}\, M.Alloc \wedge$$
$$\mathrm{dom}\, ar.Env = ((M.Body)((M.Alloc)(i))).Var$$

- lastly, the global store of an object is always coherent with the partial stores of its fragments:

$$\bigcup\{\ sx : Var_{Glob} \mid sx \in \mathrm{dom}\, \psi \bullet \psi(\ sx\ )\ \} = \bigcup\{\ i : 1\,..\,\#\sigma \bullet (\sigma(i)).\delta\ \}$$

Summarizing, we get the following schema:

Config$_{Local}$

$\alpha : NObj$
$\psi : Store_{Glob}$
$\sigma : \mathrm{seq}_1\ State_{\partial}$
$\tau : Name_{Class}$
$UNIT$

$\tau \in \mathrm{dom}\ Program$

$\mathrm{dom}\, \psi = \mathrm{dom}((Program(\tau)).Virtual)$

$\#\sigma = \#((Program(\tau)).Body)$

$\forall i : 1\,..\,\#\sigma \bullet$
  $\mathrm{dom}((\sigma(i)).\delta) = (((Program(\tau)).Body)(i)).Var \wedge$
  $\mathrm{dom}((\sigma(i)).\phi) = (((Program(\tau)).Body)(i)).Priv$

$\forall i : 1\,..\,\#\sigma \bullet$
  $(\ (\sigma(i)).\rho \neq \mathrm{empty}\ ) \wedge$
  $(\ \exists_1\ ar : AR \mid ar = \mathrm{top}(\ (\sigma(i)).\rho\ ) \wedge ar.Cur \in \mathrm{dom}(Program(\tau)).Method \bullet$
    $(\ ar.Cur = init\ ) \vee$
    $(\ \exists_1\ M : METHOD \mid M = ((Program(\tau)).Method)(\ ar.Cur\ ) \bullet$
      $(\forall k : BodyNum_{Class} \mid k \in \mathrm{dom}\, M.Alloc \bullet \exists n : 0\,..\,\mathrm{size}(\ (\sigma(k)).\rho\ ) \bullet$
        $(\mathrm{top}(\ \mathrm{pop}^n(\ (\sigma(k)).\rho\ )\ )).Cur = ar.Cur) \wedge$
      $i \in \mathrm{dom}\, M.Alloc \wedge$
      $\mathrm{dom}\, ar.Env = ((M.Body)((M.Alloc)(i))).Var\ )\ )$

$\bigcup\{\ sx : Var_{Glob} \mid sx \in \mathrm{dom}\, \psi \bullet \psi(\ sx\ )\ \} = \bigcup\{\ i : 1\,..\,\#\sigma \bullet (\sigma(i)).\delta\ \}$

5.3.1 Initial object and its existence

Back in section 4.5.4, we introduced a *Main* component in the *UNIT* schema, which is the name of the *starting class* of the program. Running a unit creates an initial object of the class named by the corresponding *Main* component. This first object is characterized by the following *initial* local configuration, *InitConfig*$_{Local}$.

$$
\begin{array}{|l}
\hline
\mathit{InitConfig}_{\mathit{Local}} \\
\mathit{Config}_{\mathit{Local}} \\
n? : \mathbf{N} \\
\hline
\alpha = \langle n? \rangle \\
\psi = (\lambda\, sx : \mathit{Var}_{\mathit{Glob}} \mid sx \in \mathrm{dom}((\mathit{Program}(\mathit{Main})).\mathit{Virtual}) \bullet \\
\quad (\lambda\, fx : \mathit{Var}_{\mathit{Part}} \mid fx \in \mathrm{ran}(((\mathit{Program}(\mathit{Main})).\mathit{Virtual})(sx)) \bullet \mathit{nil} \;) \;) \\
\#\sigma = \#((\mathit{Program}(\mathit{Main})).\mathit{Body}) \\
\forall\, i : 1 \,..\, \#\sigma \bullet \\
\quad (\sigma(i)).s = (((\mathit{Program}(\mathit{Main})).\mathit{Body})(i)).s \wedge \\
\quad (\sigma(i)).\delta = (\lambda\, fx : \mathit{Var}_{\mathit{Part}} \mid fx \in (((\mathit{Program}(\mathit{Main})).\mathit{Body})(i)).\mathit{Var} \bullet \mathit{nil} \;) \wedge \\
\quad (\sigma(i)).\phi = (\lambda\, x : \mathit{Var}_{\mathit{Priv}} \mid x \in (((\mathit{Program}(\mathit{Main})).\mathit{Body})(i)).\mathit{Priv} \bullet \mathit{nil} \;) \wedge \\
\quad (\exists\, ar : \mathit{AR} \mid ar.\mathit{Cur} = \mathit{init} \wedge ar.\mathit{Env} = \emptyset \bullet \\
\qquad (\sigma(i)).\rho = \mathrm{push}(\ \mathrm{empty}, ar\ )) \wedge \\
\quad (\sigma(i)).\eta = 0 \\
\tau = \mathit{Main} \\
\hline
\end{array}
$$

The input parameter, $n?$, is a system-provided number needed to ensure that object names are unique. Each time a program is run, a new integer is given as the name of its corresponding initial object. All global variables declared by the *main* class – whose name is given by the *Main* component– are initialized to *nil*. The initial local configuration has as many partial states as the number of fragments possessed by the body of the main class. Each partial state is initialized as expected. Notice that private stores are initialized with all their variables associated to the *nil* value. This initialization is possible because parameters are ruled out by the *UNIT* schema for the main class.

Up to this point, we have by-passed all the *proof obligations*. A proof obligation is a theorem about a given specification that claims that it is feasible. Of course, that claim has to be proved. It is easy to write a specification like:

$$
\begin{array}{|l}
\hline
A \\
n : \mathbf{N}_1 \\
\vdots \\
\hline
\ldots A\text{'s invariant} \ldots \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
B \\
A \\
\hline
n < 0 \\
\vdots \\
\hline
\end{array}
$$

Obviously, the theorem $\exists\, A \bullet B$ is false. As schemas get larger, proofs tend to be harder[4]. Be reassured –or shocked whichever attitude better reflects your beliefs– we

[4]That is a good enough reason to keep schemas as simple as possible relying on their composability to build complex specifications.

will only state one such theorem[5]. The $InitConfig_{Local}$ schema is provided as an example and the associated proof follows.

An initial local configuration is required to be a local configuration, however this requirement may be unfeasible. Indeed, it may not be possible to find values that satisfy both the local configuration schema and the constraints specified in the $InitConfig_{Local}$ schema. We have recognized a proof obligation. We are obligated to prove that there exists a set of values complying the $Config_{Local}$ schema and an integer that satisfy $InitConfig_{Local}$'s invariant. This is expressed by the theorem:

$$\exists\, Config_{Local};\ n? : \mathbb{N} \bullet InitConfig_{Local}$$

Proof

The proof is quite simple: we take the values given in the $InitConfig_{Local}$ schema for the restricted components of the $Config_{Local}$ schema, and we prove that they satisfy $Config_{Local}$'s invariant.

Given the components introduced by $Config_{Local}$ and any $n? \in \mathbb{N}$, let α, ψ, σ and τ have the values satisfying $InitConfig_{Local}$'s invariant. We prove that $Config_{Local}$'s invariant holds by showing that its constituent predicates are tautologies. The proofs of some predicates are omitted for they are trivial:

- $\tau \in \mathrm{dom}\, Program$,

$$\begin{array}{ll} \tau = Main & [InitConfig_{Local}\text{'s invariant}] \\ Main \in \mathrm{dom}\, Program & [UNIT\text{'s invariant}] \end{array}$$

- $\mathrm{dom}\,\psi = \mathrm{dom}((Program(\tau)).Virtual)$,

$$\begin{array}{ll} & \mathrm{dom}\,\psi \\ & = \mathrm{dom}(\ \lambda\, sx : Var_{Glob} \mid sx \in \mathrm{dom}((Program(Main)).Virtual) \bullet \\ & \qquad (\ \lambda\, fx : Var_{Part} \mid fx \in \mathrm{ran}(((Program(Main)).Virtual)(sx)) \bullet nil\ )\ ) \\ & = \mathrm{dom}((Program(Main)).Virtual) & [\text{dom definition}] \\ & = \mathrm{dom}((Program(\tau)).Virtual) & [\tau = Main] \end{array}$$

- $\bigcup\{\ sx : Var_{Glob} \mid sx \in \mathrm{dom}\,\psi \bullet \psi(\ sx\ )\ \} = \bigcup\{\ i : 1\,..\,\#\sigma \bullet (\sigma(i)).\delta\ \}$

a) $\bigcup\{\ sx : Var_{Glob} \mid sx \in \mathrm{dom}\,\psi \bullet \psi(\ sx\ )\ \} \subseteq \bigcup\{\ i : 1\,..\,\#\sigma \bullet (\sigma(i)).\delta\ \}$

$$\begin{array}{ll} & \forall\, sx : Var_{Glob} \mid sx \in \mathrm{dom}\,\psi \bullet \\ & \quad \psi(\ sx\ ) = (\ \lambda\, fx : Var_{Part} \mid fx \in \mathrm{ran}(((Program(Main)).Virtual)(\ sx\ )) \bullet nil\ ) \\ & \qquad\qquad [InitConfig_{Local}\text{'s invariant}] \\ & \quad \forall\, fx : Var_{Part} \mid fx \in \mathrm{ran}(((Program(Main)).Virtual)(\ sx\ )) \bullet \\ & \qquad \exists_1\, j : 1\,..\,\#((Program(Main)).Body) \bullet \\ & \qquad\quad fx \in ((Program(Main)).(Body(\ j\ ))).Var \wedge & [CLASS\text{'s invariant}] \end{array}$$

[5] If PolyGoth were to be widely distributed, it would be worth it to establish all the proof obligations and to effectively prove them. Since the present work is only a full-blown specification exercise, we do not consider it indispensable.

$$(fx \mapsto nil) \in (\sigma(j)).\delta \qquad [InitConfig_{Local}\text{'s invariant}]$$
$$\psi(\ sx\ ) \in \bigcup\{\ i : 1\,..\,\#\sigma \bullet (\sigma(i)).\delta\ \}$$

b) $\bigcup\{\ sx : Var_{Glob} \mid sx \in \mathrm{dom}\,\psi \bullet \psi(\ sx\ )\ \} \supseteq \bigcup\{\ i : 1\,..\,\#\sigma \bullet (\sigma(i)).\delta\ \}$

$\forall i : 1\,..\,\#\sigma \bullet$
$(\sigma(i)).\delta = (\ \lambda fx : Var_{Part} \mid fx \in (((Program(Main)).Body)(i)).Var \bullet nil\ )$
$[InitConfig_{Local}\text{'s invariant}]$
$\forall fx : Var_{Part} \mid fx \in \mathrm{dom}((\sigma(i)).\delta) \bullet$
$\exists_1 sx : Var_{Glob} \mid sx \in \mathrm{dom}((Program(Main)).Virtual) \bullet$
$fx \in \mathrm{ran}(((Program(Main)).Virtual)(\ sx\ ))$ $[CLASS\text{'s invariant}]$
$sx \in \mathrm{dom}\,\psi \wedge (fx \mapsto nil) \in \psi(\ sx\ )$ $[InitConfig_{Local}\text{'s invariant}]$
$(\sigma(i)).\delta \in \bigcup\{\ sx : Var_{Glob} \mid sx \in \mathrm{dom}\,\psi \bullet \psi(\ sx\ )\ \}$

5.3.2 Integrating the previous level

A transition at the object level is not seen anymore as a partial state change but rather as a local configuration transformation. All these transformations share a particularity: they leave unchanged the name, class, program and starting point components of the local configuration schema.

$\Xi\ Config_{Local}$

$Config_{Local}$
$Config_{Local}'$

$\alpha' = \alpha \wedge \tau = \tau'$

$Program' = Program \wedge Main' = Main$

In the sequel of this section, we exhibit some representative schemas specifying transitions that involve an increasing number of fragments. We start with a single fragment transition. Descriptions of two and more fragments transitions follow. Lastly, a summarizing schema, similar to the one used in section 5.2.5, is presented.

The *SingleMove* schema introduces all the components necessary to the description of the transitions that involve a single fragment.

$SingleMove$

$\Xi\ Config_{Local}$
$\Delta\ State_{\partial}$
$i : BodyNum_{Class}$

$i \in \mathrm{dom}\,\sigma$
$\sigma(\ i\ ) = \theta State_{\partial}$

A change of local configuration is induced by a transition of any of its partial states, say its i^{th}. If this transition complies to any of those specified by the $Axioms_{Basic}$ schema then, the resulting configuration is obtained as follows:

- the only changed partial state is precisely the i^{th}, and its values are those specified by the corresponding state transition:

$$\sigma' = \sigma \oplus \{\ i \mapsto \theta State_\partial'\ \};$$

- store's coherence may have been violated; it has to be restablished. The global store is selectively updated for those global variables that have any partial variable placed on the i^{th} fragment:

$$\psi' = \psi \oplus (\ \lambda\, sx : Var_{Glob} \mid sx \in \mathrm{dom}\,\psi \bullet \psi(\ sx\ ) \bowtie \delta'\ )$$

[Z] The functional update, $\bowtie$, is a restriction of the functional overriding, $\oplus$. Let f and g be two functions with identical signatures. The domain of function $f \bowtie g$ is the restriction of $f \oplus g$ to the domain of f. Its **Z** definition, reproduced from the appendix, is:

$$\begin{array}{|l} [X, Y] \\ \hline \_ \bowtie \_ : (\ X \nrightarrow Y\ ) \times (\ X \nrightarrow Y\ ) \rightarrow (\ X \nrightarrow Y\ ) \\ \hline \forall f, g : X \nrightarrow Y \bullet \\ \quad f \bowtie g = \mathrm{dom}\, f \lhd (f \oplus g) \\ \hline \end{array}$$

where $\lhd$ is the domain restriction function. $\diamond$

Specifications of the previous level are integrated into the current level by the *Local-Transition* schema below:

LocalTransition

SingleMove

$Axioms_{Basic}$

$\sigma' = \sigma \oplus \{\ i \mapsto \theta State_\partial'\ \}$

$\psi' = \psi \oplus (\ \lambda\, sx : Var_{Glob} \mid sx \in \mathrm{dom}\,\psi \bullet \psi(\ sx\ ) \bowtie \delta'\ )$

A proof obligation for this schema is that $Config_{Local}$'s invariant has to hold after a successful local transition.

As a second example of a one fragment transition, we provide the description of the *self* expression evaluation.

Self

SingleMove

$s = exp(\ self\ )$

$s' = exp(name(obj(\ \alpha\ )))$

$\Xi\ State_\partial$

$\sigma' = \sigma \oplus \{\ i \mapsto \theta State_\partial'\ \}$

$\psi' = \psi$

It can be seen that only the statement component of the concerned partial state is transformed –recall that $\Xi\, State_\partial$ specifies the unchanged ones– and that the result is the expected one.

As a last example of this kind of transitions we present a global variable assignment.

GlobVarAssign

$$
\begin{array}{l}
SingleMove \\
sx : Var_{Glob} \\
\vec{\gamma} : \mathrm{seq}_1\ OBJ \\
\hline
sx \in \mathrm{dom}\ \psi \\
\#\vec{\gamma} = \#(((Program(\tau)).Virtual)(sx)) \\
s = massign(\ sx\ , name(seqobj(\ \vec{\gamma}\ ))\ ) \\
\psi' = \psi \oplus \\
\quad \{\ sx \mapsto \{\ j : 1\,..\,\#\vec{\gamma} \bullet (((Program(\tau)).Virtual)(sx))(\ j\ ) \mapsto \vec{\gamma}(\ j\ )\ \}\ \}
\end{array}
$$

Again, sx and $\vec{\gamma}$ are auxiliary variables. The former denotes the variable being assigned, and the latter the assigned value. $\vec{\gamma}$ is required to be a composed value with as many components as the number of partial variables sx is declared to be splitted on. The global store, ψ, is updated in a straightforward way: its mapping for sx is replaced by a partial store that maps the partial variables sx is declared to be made of, onto the values which compose the assigned value $\vec{\gamma}$; variables and values having the same sequence index are paired.

All the fragments are selectively updated in their partial store component and the current statement becomes the expression denoting the assigned value:

GlobVarAssign

$$
\begin{array}{l}
GlobVarAssign \\
\hline
s' = exp(name(seqobj(\ \vec{\gamma}\ ))) \\
\delta' = \delta \bowtie \psi'(\ sx\ ) \\
\phi' = \phi \wedge \rho' = \rho \wedge \eta' = \eta \\
\sigma'(\ i\ ) = \theta State_\partial{}' \\
\forall j : 1\,..\,\#\sigma \mid j \neq i \bullet \\
\quad (\sigma'(\ j\ )).s = (\sigma(\ j\ )).s \wedge \\
\quad (\sigma'(\ j\ )).\delta = (\sigma(\ j\ )).\delta \bowtie \psi'(\ sx\ ) \wedge \\
\quad (\sigma'(\ j\ )).\phi = (\sigma(\ j\ )).\phi \wedge \\
\quad (\sigma'(\ j\ )).\rho = (\sigma(\ j\ )).\rho \wedge \\
\quad (\sigma'(\ j\ )).\eta = (\sigma(\ j\ )).\eta
\end{array}
$$

Z Actually, the above schema is incorrect since the global identifier *GlobVarAssign* is declared twice. We feel this minor escape from the strict language definition is justified since it enables partial presentations of schemas without introducing new names. ◇

5.3.3 Remote assignment

We now describe the *remote assignment* statement. Partial variables are visible all over the class, even though they are declared by exactly one class block. Intuitively, a

partial variable should be declared by the block that makes the largest use of it, but its access is not exclusive. Other *brother* blocks –i.e. blocks of the same class– may read or write every fragment of a global variable. The associated transitions involve changes for two distinct partial states, say the i^{th} and the j^{th}. The *DoubleMove* schema introduces the involved partial states and it indicates the induced configuration change.

DoubleMove

$$\Xi\ Config_{Local}$$
$$\Delta\ State_{\partial}$$
$$\Delta\ State_{\partial 1}$$
$$i, j : BodyNum_{Class}$$

$$\{\ i, j\ \} \subseteq \mathrm{dom}\,\sigma$$
$$\sigma(\ i\ ) = \theta State_{\partial}$$
$$\sigma(\ j\ ) = \theta State_{\partial 1}$$
$$\sigma' = \sigma \oplus \{\ i \mapsto \theta State_{\partial}', j \mapsto \theta State_{\partial 1}'\ \}$$

Z Recall that the $\Delta\ State_{\partial 1}$ notation introduces the two schemas $State_{\partial 1}$ and $State_{\partial 1}'$ whose components are decorated with $_1$ and $'_1$ respectively. ◇

The *PartAssign* schema describes the (remote) partial variable assignment: variable fx –declared by the j^{th} block– is assigned the value γ by the i^{th} fragment.

PartAssign

$$DoubleMove$$
$$fx : Var_{Part}$$
$$\gamma : OBJ$$

$$fx \in \mathrm{dom}\,\delta_1$$
$$s = fassign(\ fx, name(\ \gamma\ )\ )$$

The i^{th} fragment is only changed in its statement component; it becomes the expression denoting the assigned value, γ. The partial store component of the j^{th} fragment is updated. The other components are left as they were before the transition:

PartAssign

$$PartAssign$$

$$s' = exp(name(\ \gamma\ ))$$
$$\Xi\ State_{\partial}$$

$$s_1' = s_1$$
$$\delta_1' = \delta_1 \oplus \{\ fx \mapsto \gamma\ \}$$
$$\phi_1' = \phi_1 \wedge \rho_1' = \rho_1 \wedge \eta_1' = \eta_1$$

The resulting j^{th} fragment's partial store, δ_1', carries the updated value. It is used to actualize the global store:

PartAssign

$$PartAssign$$

$$\psi' = \psi \bowtie (\ \lambda\, sx : Var_{Glob} \bullet \psi(\ sx\ ) \bowtie \delta_1'\ )$$

5.3.4 Coordinated Call

A method call –either simple or coordinated– can be decomposed in three distinct stages: the creation and initialization of the corresponding execution context, the execution of the method's body and the results restitution together with the initial context[6]. The execution of the method's body is not different from any other statement. Therefore, we focus on the other two stages. Two transitions describe the first stage; *InCall* for a simple, and *InCoCall* for a coordinated call. The *InCoRet* transition suffices to specify the third stage of both the simple and the coordinated call. Each one of these transitions are exhibited through a sequence of three schemas: the first one introduces the needed variables and specifies the required pre-conditions, the second shows the transition's effect on the statement components of the involved fragments, and the last presents the transition's effect on the concerned stacks. In the remaining of this section, we explain the schemas corresponding to the coordinated call and to its return.

Various fragments participate in a coordinated call, thus multiple partial states are changed in this configuration transition. However, only their statement and stack components are involved; the others stay still. Besides, the global store component of the configuration is also unchanged. The *Multimove* schema encompasses this remark:

MultiMove

$$\Xi\ Config_{Local}$$

$$\forall j : 1 \,..\, \#\sigma \bullet$$
$$(\sigma'(\ j\ )).\delta = (\sigma(\ j\ )).\delta \wedge$$
$$(\sigma'(\ j\ )).\phi = (\sigma(\ j\ )).\phi \wedge (\sigma'(\ j\ )).\eta = (\sigma(\ j\ )).\eta$$
$$\psi' = \psi$$

Pre-conditions

Some auxiliary variables are needed to express the pre-conditions of a coordinated call.

InCoCallPrecond

$$MultiMove$$
$$ready : \mathbb{F}_1\ BodyNum_{Class}$$
$$mf : Name_{Method}$$
$$Actual : \text{seq}(\ Var_{Loc} \times Exp\ )$$
$$\partial actual : \text{seq}_1(\ \text{seq}(\ Var_{Loc} \times Exp\ )\ )$$
$$M : METHOD$$

The finite set *ready* gathers the indexes of those fragments that participate in the coordinated call. The called (method) name is held in mf and M is used as a shorthand for its corresponding method. *Actual* is used to collect the actual parameters provided in $\partial actual$, the non-empty sequence of partial contributions.

The method called has to be defined in the current program and it cannot be the initialization-reserved one, *init*.

[6] Of course, our view is biased!

InCoCallPrecond

InCoCallPrecond

$mf \neq init \wedge mf \in \mathrm{dom}((Program(\tau)).Method)$
$M = ((Program(\tau)).Method)(\ mf\ )$

The indexes of the participating fragments are determined by the *Alloc* component of the calling method. All the concerned fragments carry the current method name, say *mg*, on top of their stack, and they must have reached their respective coordinated call expressions.

InCoCallPrecond

InCoCallPrecond

$\exists\, mg : Name_{Method} \mid mg \in \mathrm{dom}((Program(\tau)).Method) \bullet$
$\quad ready = \mathrm{dom}((((Program(\tau)).Method)(\ mg\ )).Alloc) \wedge$
$\quad (\forall\, i : BodyNum_{Class} \mid i \in ready \bullet (\mathrm{top}(\ (\sigma(\ i\ )).\rho\ )).Cur = mg)$
$\forall\, j : 1\,..\,\#\sigma \mid j \in ready \bullet$
$\quad (\sigma(\ j\ )).s = exp(cocall(\ mf, \partial actual(\ j\ )\ ))$

Each calling fragment delivers exactly one (possibly empty) sequence of actual parameter contributions.

$$\#\partial actual = \#ready$$

A unique local variable names each method's formal parameter (see section 4.5.2); hence, a parameter contribution contains both the expression whose value is the actual parameter, and the name of the formal parameter it corresponds to. A parameter contribution, p, is specified as a couple: its first projection (noted $\pi_1(\ p\ )$) is a local variable and its second (noted $\pi_2(\ p\ )$) is an expression. Collisions of parameter names must be avoided:

InCoCallPrecond

InCoCallPrecond

$\#\partial actual = \#ready$
$\forall\, j, j' : 1\,..\,\#\partial actual \bullet$
$\quad \forall\, k : 1\,..\,\#(\partial actual(\ j\ ));\ k' : 1\,..\,\#(\partial actual(\ j'\ )) \bullet$
$\quad\quad \pi_1(\ (\partial actual(\ j\ ))(k)\ ) = \pi_1(\ (\partial actual(\ j'\ ))(k')\ ) \Rightarrow (\ j = j' \wedge k = k'\ )$

Actual is built by distributed concatenation –noted $\frown/-$ of all the contributions. It is used to express that formal parameters are rightly covered, i.e. there is an exact match between formal and actual parameters. Another requirement is implicitly written: actual parameters must be fully evaluated. It is specified with the existence of an object γ for each parameter such that its expression projection is the direct naming of γ.

InCoCallPrecond

InCoCallPrecond

$$Actual = \widehat{\ } / \partial actual$$
$$\#Actual = \#(M.Param)$$
$$\forall j : 1 \,..\, \#Actual \bullet \exists_1 k : 1 \,..\, \#(M.Param) \bullet \exists \gamma : OBJ \bullet$$
$$Actual(\ j\ ) = (M.Param)(\ k\ ) \mapsto name(\ \gamma\ )$$

Control effect

We now describe the control effects of the coordinated call. Recall that a fragment participating to the call is named a *p-caller* –read it partial caller– and one that hosts a block of the called method is referred to as a *p-callee*. The statement component of each fragment is determined by one of the following situations:

- the fragment is both a p-caller and a p-callee:

  $$j \in (\mathrm{dom}(M.Alloc)) \cap ready$$

  then, the statement is the corresponding component of the called method block which is placed on that fragment. The result yielded by the evaluation of that statement is sent to the p-callers of the object itself.

  $$(\sigma'(\ j\ )).s =$$
  $$send(\ ((M.Body)(\ (M.Alloc)(j)\ )).s, (\alpha, ready)\ )$$

- the fragment is only a p-caller:

  $$j \in ready \setminus \mathrm{dom}(M.Alloc)$$

  then, the statement becomes a wait expression. It is later used to recognize the waiting state of the fragment.

  $$(\sigma'(\ j\ )).s = exp(wait(\ \alpha, mf\ ))$$

- the fragment is only a p-callee:

  $$j \in (\mathrm{dom}(M.Alloc)) \setminus ready$$

  then, the fragment's current statement execution is delayed until its contribution to the results of the called method is sent back to the p-callers.

  $$(\sigma'(\ j\ )).s =$$
  $$seqcomp(\ send(\ ((M.Body)(\ (M.Alloc)(j)\ )).s, (\alpha, ready)\ ), (\sigma(\ j\ )).s\ )$$

- the fragment is neither a p-caller nor a p-callee:

  $$j \notin (\mathrm{dom}(M.Alloc)) \cup ready$$

  then, the statement is unchanged

  $$(\sigma'(\ j\ )).s = (\sigma(\ j\ )).s$$

A schema could be written for each of the situations just described, but a logical implication suffices to group them:

InCoCallControlEffect

InCoCallPrecond

$$
\begin{array}{l}
\forall j : 1 \,..\, \#\sigma \bullet \\
\quad (\; j \in (\mathrm{dom}(M.Alloc)) \cap ready \Rightarrow \\
\qquad (\sigma'(\; j \;)).s = \\
\qquad\quad send(\; ((M.Body)(\; (M.Alloc)(j) \;)).s, (\alpha, ready) \;) \;) \wedge \\
\quad (\; j \in ready \setminus \mathrm{dom}(M.Alloc) \Rightarrow \\
\qquad (\sigma'(\; j \;)).s = exp(wait(\; \alpha, mf \;)) \;) \wedge \\
\quad (\; j \in (\mathrm{dom}(M.Alloc)) \setminus ready \Rightarrow \\
\qquad (\sigma'(\; j \;)).s = \\
\qquad\quad seqcomp(\; send(\; ((M.Body)(\; (M.Alloc)(j) \;)).s, (\alpha, ready) \;), (\sigma(\; j \;)).s \;) \;) \wedge \\
\quad (\; j \notin (\mathrm{dom}(M.Alloc)) \cup ready \Rightarrow \\
\qquad (\sigma'(\; j \;)).s = (\sigma(\; j \;)).s \;)
\end{array}
$$

Stack effect

As far as stacks are concerned, two situations arise in a coordinated call:

- the fragment is a p-callee: $j \in \mathrm{dom}(M.Alloc)$. Then, a new activation record, properly initialized, is pushed onto its stack.
- the fragment is not a p-callee: $j \notin \mathrm{dom}(M.Alloc)$. Then, the stack is unchanged.

InCoCallStackEffect

InCoCallPrecond

$$
\begin{array}{l}
\forall j : 1 \,..\, \#\sigma \mid j \in \mathrm{dom}(M.Alloc) \bullet \exists_1 \, ar : AR \mid \\
\quad ar.Cur = mf \wedge \\
\quad ar.Env = \\
\qquad (\; \lambda \, u : Var_{Loc} \mid u \in (((M.Body)(\; (M.Alloc)(j) \;)).Var) \bullet nil \;) \bowtie \\
\qquad\quad \{ \; \gamma : OBJ; \; k : 1 \,..\, \#Actual \mid \pi_2(Actual(\; k \;)) = name(\; \gamma \;) \bullet \\
\qquad\qquad \pi_1(Actual(\; k \;)) \mapsto \gamma \; \} \bullet \\
\quad (\sigma'(\; j \;)).\rho = \mathrm{push}(\; (\sigma(\; j \;)).\rho, ar \;) \\
\forall j : 1 \,..\, \#\sigma \mid j \notin \mathrm{dom}(M.Alloc) \bullet \\
\quad (\sigma'(\; j \;)).\rho = (\sigma(\; j \;)).\rho
\end{array}
$$

The overall transition is simply described with the conjunction of both effects:

$$InCoCall \mathrel{\widehat{=}} InCoCallControlEffect \wedge InCoCallStackEffect$$

Return

The call returns when all the involved fragments are ready to send their contribution to the result. This situation is attained when they are all executing the called method, and their statement component is a send of a fully evaluated expression to the waiting p-callers of the object itself:

$$\begin{array}{l}\forall j : 1\,..\,\#\sigma \mid j \in ready \bullet \\ \quad (\mathrm{top}(\ (\sigma(\ j\ )).\rho\ )).Cur = mf \wedge \\ \quad (\sigma(\ j\ )).s = send(\ exp(name(\ \vec{\gamma}(\ j\ )\ )), (\alpha, waiting)\ )\end{array}$$

Besides, those fragments waiting for the result –and not computing it– have to be ready to receive it:

$$\begin{array}{l}\forall j : 1\,..\,\#\sigma \mid j \in waiting \setminus ready \bullet \\ \quad (\sigma(\ j\ )).s = exp(wait(\ \alpha, mf\ ))\end{array}$$

Return pre-conditions form the following schema:

InCoRetPrecond

$$\begin{array}{l}MultiMove \\ ready, waiting : \mathbb{F}_1\ BodyNum_{Class} \\ \vec{\gamma} : \mathrm{seq}_1\ OBJ \\ mf : Name_{Method} \\ \hline mf \in \mathrm{dom}((Program(\tau)).Method) \\ ready = \mathrm{dom}(((Program(\tau)).Method)(\ mf\ )).Alloc \\ \#\vec{\gamma} = \#ready \\ \forall j : 1\,..\,\#\sigma \mid j \in ready \bullet \\ \quad (\mathrm{top}(\ (\sigma(\ j\ )).\rho\ )).Cur = mf \wedge \\ \quad (\sigma(\ j\ )).s = send(\ exp(name(\ \vec{\gamma}(\ j\ )\ )), (\alpha, waiting)\ ) \\ \forall j : 1\,..\,\#\sigma \mid j \in waiting \setminus ready \bullet \\ \quad (\sigma(\ j\ )).s = exp(wait(\ \alpha, mf\ ))\end{array}$$

The result is a composed object, $\vec{\gamma}$, made of all the fragment contributions. The expression denoting its direct naming is obtained by all the p-callers. All those p-callees that are not p-callers end up with the expression denoting their own contribution to the result. This value will probably be discarded later on with a *ValueDiscard* local transition (see A.3.1) in order to carry on evaluating a –previously left– pending statement.

InCoRetControlEffect

$$\begin{array}{l}InCoRetPrecond \\ \hline \forall j : 1\,..\,\#\sigma \bullet \\ \quad (\ j \in waiting \setminus ready \Rightarrow \\ \qquad (\sigma'(\ j\ )).s = exp(name(seqobj(\ \vec{\gamma}\ )))\ ) \wedge \\ \quad (\ j \in ready \setminus waiting \Rightarrow \\ \qquad (\sigma'(\ j\ )).s = exp(name(\vec{\gamma}(\ j\ ))\ ) \wedge \\ \quad (\ j \notin waiting \cup ready \Rightarrow \\ \qquad (\sigma'(\ j\ )).s = (\sigma(\ j\ )).s\ )\end{array}$$

The stacks of the p-callees are popped while the others remain untouched:

InCoRetStackEffect

InCoRetPrecond

$$\forall j : 1 \,..\, \#\sigma \bullet$$
$$(\ j \notin \mathit{ready} \Rightarrow (\sigma'(\ j\ )).\rho = (\sigma(\ j\ )).\rho\ ) \wedge$$
$$(\ j \in \mathit{ready} \Rightarrow (\sigma'(\ j\ )).\rho = \mathrm{pop}(\ (\sigma(\ j\ )).\rho\ )\ )$$

The overall transition is simply described with the conjunction of both effects:

$$\mathit{InCoRet} \mathrel{\widehat{=}} \mathit{InCoRetControlEffect} \wedge \mathit{InCoRetStackEffect}$$

Notice that, exception made of parameter description and calling syntax differences, the simple call schemas shown in A.3.2 are restrictions of the coordinated call ones. Leaving aside parameters and syntax differences, the simple call schema could easily be written as follows:

InCall

InCoCall

$$\exists i : 1 \,..\, \sigma \bullet$$
$$\mathit{ready} = \{\ i\ \}$$

No exceptions have to be made to recognize that the return from a simple call is a restriction of a coordinated call return:

InRet

InCoRet

$$\#\mathit{waiting} = 1$$

5.3.5 Summarizing

The schema calculus is used to aggregate the descriptions of all the transitions an object is capable of on its own. Since they involve two local configurations, the one before and the one after the transition, we name this schema *TwoConfigurationsTransition.*

$$
\begin{array}{|l}
\textit{TwoConfigurationsTransition} \\
\hline
\Xi\ \textit{Config}_{Local} \\
\hline
(\exists\, i : \textit{BodyNum}_{Class} \bullet \\
\quad (\exists \Delta\ \textit{State}_{\partial} \bullet \\
\qquad \textit{LocalTransition} \lor \\
\qquad \textit{Self} \lor \\
\qquad (\exists\, sx : \textit{Var}_{Glob};\ \vec{\gamma} : \mathrm{seq}\ \textit{OBJ} \bullet \textit{GlobVarExp} \lor \textit{GlobVarAssign}) \lor \\
\qquad (\exists \Delta\ \textit{State}_{\partial 1};\ j : \textit{BodyNum}_{Class} \bullet \\
\qquad\quad (\exists\, fx : \textit{Var}_{Part};\ \gamma : \textit{OBJ} \bullet \textit{PartVarExp} \lor \textit{PartAssign})))) \lor \\
(\exists\, mf : \textit{Name}_{Method};\ M : \textit{METHOD} \bullet \\
\quad (\exists\, i : \textit{BodyNum}_{Class};\ \textit{Actual} : \mathrm{seq}\ \textit{Exp} \bullet \textit{InCall}) \lor \\
\quad (\exists\, ready : \mathbb{F}_1\ \textit{BodyNum}_{Class};\ \textit{Actual} : \mathrm{seq}(\ \textit{Var}_{Loc} \times \textit{Exp}\ ); \\
\qquad \partial actual : \mathrm{seq}(\ \mathrm{seq}(\ \textit{Var}_{Loc} \times \textit{Exp}\ )\ ) \bullet \textit{InCoCall}) \lor \\
\quad (\exists\, ready, waiting : \mathbb{F}_1\ \textit{BodyNum}_{Class};\ \vec{\gamma} : \mathrm{seq}_1\ \textit{OBJ} \bullet \textit{InCoRet})) \\
\hline
\end{array}
$$

This larger mathematical entity provides a concise description of all the object level transitions. We have depicted the behaviour of objects in their individuality. We are ready to begin with the description of their interactions.

5.4 System level

In order to describe object interactions, we introduce the set of (global) configurations, *Config*. It is the set of finite sets of local configurations:

$$\textit{Config} == \mathbb{F}\ \textit{Config}_{Local}$$

Intuitively, a configuration is a snapshot of a system of fragmented objects. Each local configuration in a global one stands for an active object of the system being described.

Our objects may interact either directly by sending messages to each other, or indirectly by creating other objects. In both cases, the global configuration they belong to is transformed. The transition system is ultimately defined to capture in a binary relation all the global configuration changes our language is able to produce. We delay the definition of the transition system until section 6. Before that, we show one example of each kind of object interaction: a coordinated (external) method call and its return, and the creation of a new class instance.

5.4.1 Communications

An external coordinated method call can be thought of as an extension of the (internal) coordinated call presented in section 5.3.4. The extension lies in that p-callers and p-callees belong to two different objects: the source and the destination objects. P-callers are only changed in their statement components; p-callees stack components are altered as well.

TwoObjects

MultiMove
MultiMove$_1$

$\forall j : 1 .. \#\sigma \bullet$
$(\sigma'(\ j\ )).\rho = (\sigma(\ j\ )).\rho$

In the above schema, the bindings $\theta Config_{Local}$ and $\theta Config_{Local}'$ are associated to the source object, and $\theta Config_{Local_1}$ and $\theta Config_{Local_1}'$ to the destination. The schemas specifying the call are presented in the order adopted in the previous section: pre-conditions, control effect and stack effect.

Pre-conditions

The *ExtCoCallPrecond* schema introduces the same auxiliary variables as the *InCo-CallPrecond* one (see section 5.3.4).

ExtCoCallPrecond

TwoObjects
$ready : \mathbb{F}_1\ BodyNum_{Class}$
$mf : Name_{Method}$
$Actual : \text{seq}(\ Var_{Loc} \times Exp\ )$
$\partial actual : \text{seq}_1(\ \text{seq}(\ Var_{Loc} \times Exp\ )\ )$
$M : METHOD$

In fact, it is not surprising to find almost the same restrictions in the predicate part of both schemas; the differences stem from the distinction made by the present schema in the identities of the source and the destination objects. The sent method name, *mf*, has to be defined in the environment of the destination object ($Config_{Local_1}$), while the *ready* set is built out of source information ($Config_{Local}$). The destination object is explicitly named by all the p-callers, and its name is precisely α_1. Parameters are dealt with as if it were an internal coordinated call.

ExtCoCallPrecond

ExtCoCallPrecond

$$
\begin{array}{l}
mf \neq init \wedge mf \in \mathrm{dom}((Program_1(\tau_1)).Method) \\
M = ((Program_1(\tau_1)).Method)(\ mf\ ) \\
\exists\, mg : Name_{Method} \mid mg \in \mathrm{dom}((Program(\tau)).Method) \bullet \\
\quad ready = \mathrm{dom}((((Program(\tau)).Method)(\ mg\ )).Alloc) \wedge \\
\quad (\forall\, i : BodyNum_{Class} \mid i \in ready \bullet (\mathrm{top}(\ (\sigma(\ i\ )).\rho\ )).Cur = mg) \\
\forall j : 1 \,..\, \#\sigma \mid j \in ready \bullet \\
\quad (\sigma(\ j\ )).s = exp(cometh(\ name(\ obj(\alpha_1)\ ), mf, \partial actual(\ j\ )\ )) \\
\#\partial actual = \#ready \\
\forall j, j' : 1 \,..\, \#\partial actual \bullet \\
\quad \forall k : 1 \,..\, \#(\partial actual(\ j\ ));\ k' : 1 \,..\, \#(\partial actual(\ j'\ )) \bullet \\
\qquad \pi_1(\ (\partial actual(\ j\ ))(k)\ ) = \pi_1(\ (\partial actual(\ j'\ ))(k')\ ) \Rightarrow \\
\qquad\quad (\ j = j' \wedge k = k'\ ) \\
Actual = \frown / \partial actual \\
\#Actual = \#(M.Param) \\
\forall j : 1 \,..\, \#Actual \bullet \exists_1 k : 1 \,..\, \#(M.Param) \bullet \exists \gamma : OBJ \bullet \\
\quad Actual(\ j\ ) = (M.Param)(\ k\ ) \mapsto name(\ \gamma\ )
\end{array}
$$

However, an extra condition is missing. We have to specify that the message will be handled when all the concerned fragments are found idle. A fragment is idle whenever it has no work to do. That situation can be recognized by inspecting its statement component: an expression denoting the direct naming of any object reveals that the work is done. Therefore, the extra constraint is incorporated as follows:

ExtCoCallPrecond

ExtCoCallPrecond

$$
\begin{array}{l}
\forall k : 1 \,..\, \#\sigma_1 \mid k \in \mathrm{dom}(M.Alloc) \bullet \\
\quad \exists \gamma : OBJ \bullet (\sigma_1(\ k\ )).s = exp(name(\ \gamma\ ))
\end{array}
$$

Control effect

The statement components in both objects are altered only if the fragment they belong to is involved. For the destination object, the placement of the method blocks defines the p-callees. They have to evaluate the statement of their ascribed method block and then, send back the results to the p-callers of object α.

$$
\begin{array}{l}
\forall j : 1 \,..\, \#\sigma_1 \bullet \\
\quad (\ j \in (\mathrm{dom}(M.Alloc)) \Rightarrow \\
\qquad (\sigma_1'(\ j\ )).s = \\
\qquad\quad send(\ ((M.Body)(\ (M.Alloc)(j)\ )).s, (\alpha, ready)\ )\ )
\end{array}
$$

For the source object, the *ready* set characterizes the p-callers. They are bound to wait for the results of method *mf* from object α_1:

$$
\begin{array}{l}
\forall j : 1 \,..\, \#\sigma \bullet \\
\quad (\ j \in ready \Rightarrow \\
\qquad (\sigma'(\ j\ )).s = exp(wait(\ \alpha_1, mf\ ))\ )
\end{array}
$$

All other fragments keep on with their respective executions:

ExtCoCallControlEffect

ExtCoCallPrecond

$$
\begin{array}{l}
\forall j : 1 \,..\, \#\sigma_1 \bullet \\
\quad (\, j \in (\mathrm{dom}(M.Alloc)) \Rightarrow \\
\qquad (\sigma_1'(\, j \,)).s = \\
\qquad\quad send(\, ((M.Body)(\, (M.Alloc)(j) \,)).s, (\alpha, ready) \,) \,) \wedge \\
\quad (\, j \notin \mathrm{dom}(M.Alloc) \Rightarrow \\
\qquad (\sigma_1'(\, j \,)).s = (\sigma_1(\, j \,)).s \,) \\
\forall j : 1 \,..\, \#\sigma \bullet \\
\quad (\, j \in ready \Rightarrow \\
\qquad (\sigma'(\, j \,)).s = exp(wait(\, \alpha_1, mf \,)) \,) \wedge \\
\quad (\, j \notin ready \Rightarrow \\
\qquad (\sigma'(\, j \,)).s = (\sigma(\, j \,)).s \,)
\end{array}
$$

Stack effect

The stack components in the destination object are modified only if their fragments are p-callees. A new activation record is pushed onto the stack of each one of these fragments. The activation record is initialized with the called method name in its *Cur* component, and a local store in its *Env* one. The local store initially maps the declared variables to the *nil* value, then it is selectively updated with the actual parameter values.

ExtCoCallStackEffect

ExtCoCallPrecond

$$
\begin{array}{l}
\forall j : 1 \,..\, \#\sigma_1 \mid j \in \mathrm{dom}(M.Alloc) \bullet \exists_1 \, ar : AR \mid \\
\quad ar.Cur = mf \wedge \\
\quad ar.Env = \\
\qquad (\, \lambda \, u : Var_{Loc} \mid u \in (((M.Body)(\, (M.Alloc)(j) \,)).Var) \bullet nil \,) \bowtie \\
\qquad\quad \{ \, \gamma : OBJ; \; k : 1 \,..\, \#Actual \mid \pi_2(Actual(\, k \,)) = name(\, \gamma \,) \bullet \\
\qquad\qquad \pi_1(Actual(\, k \,)) \mapsto \gamma \, \} \bullet \\
\quad (\sigma_1'(\, j \,)).\rho = \mathrm{push}(\, (\sigma_1(\, j \,)).\rho, ar \,) \\
\forall j : 1 \,..\, \#\sigma_1 \mid j \notin \mathrm{dom}(M.Alloc) \bullet \\
\quad (\sigma_1'(\, j \,)).\rho = (\sigma_1(\, j \,)).\rho
\end{array}
$$

The overall transition is simply described with the conjunction of both effects:

$$ExtCoCall \triangleq ExtCoCallControlEffect \wedge ExtCoCallStackEffect$$

As the return schema is also used to specify the postlude of a class instance creation, its presentation is postponed.

5.4.2 Genesis

Schemas for the creation –simple or coordinated– of a class instance are tailored differently. As for previous transitions, object creation is described by a schema trilogy.

Transition effects are however grouped differently. The three schemas describe the pre-conditions, the effects on the created object and those on the creator.

Pre-conditions

As usual, the first schema introduces the necessary variables and it specifies the transition pre-conditions.

NewCallPrecond

$\Xi\ Config_{Local}$
$Config_{Local\,1}$
$C : CLASS$
$Actual : \text{seq}\ Exp$
$i : BodyNum_{Class}$

The creator is introduced with the $\Xi\ Config_{Local}$ schema while $Config_{Local\,1}$ presents the created object. The other variables are auxiliary. The statement component of the i^{th} fragment of the creator is the expression denoting a class instance creation.

NewCallPrecond

NewCallPrecond

$i \in \text{dom}\ \sigma$
$(\sigma(\ i\ )).s = exp(new(\ \tau_1, Actual\ ))$

The class name –not naively chosen to be τ_1– has to be defined in the current program. Variable C is used as a shorthand for the associated class.

NewCallPrecond

NewCallPrecond

$\tau_1 \in \text{dom}\ Program$
$C = Program(\ \tau_1\ )$

Actual parameters have to match formal ones, and the expressions denoting their values must be fully evaluated.

NewCallPrecond

NewCallPrecond

$\#Actual = \#(C.Param)$
$\forall k : 1\,..\,\#Actual \bullet \exists \gamma : OBJ \bullet Actual(\ k\ ) = name(\ \gamma\ )$

Effects on created

The created object is given a unique name. The name is the concatenation of the creator's name with the (involved) fragment number and its count component. This count component is needed in order to obtain a unique name out of purely local –to the transition– informations.

NewCallCreated

NewCallPrecond

$$\alpha_1 = \alpha \frown \langle i, (\sigma(\ i\ )).\eta \rangle$$

The global store is properly initialized, and the number of partial states is exactly the number of blocks the class is declared to be made of.

NewCallCreated

NewCallCreated

$$\psi_1 = (\ \lambda\, sx : Var_{Glob} \mid sx \in \mathrm{dom}(C.Virtual) \bullet$$
$$(\ \lambda\, fx : Var_{Part} \mid fx \in \mathrm{ran}((C.Virtual)(\ sx\ )) \bullet nil\ )\ )$$
$$\#\sigma_1 = \#(C.Body)$$

Each fragment has to execute its corresponding block statement. This set of statements constitutes what it known to be the *init* multiprocedure of the object. When the *init* execution is terminated[7], the name of the object itself is sent back to the creator's fragment that started the creation. All the other components of the partial states are initialized in a self explanatory manner. The environment is the creator's one[8].

NewCallCreated

NewCallCreated

$$\forall j : 1 \ldots \#(C.Body) \bullet$$
$$(\sigma_1(j)).s =$$
$$seqcomp(\ ((C.Body)(j)).s, send(\ exp(self), (\ \alpha, \{i\}\ )\ )\ ) \wedge$$
$$(\sigma_1(j)).\delta = (\ \lambda\, fx : Var_{Part} \mid fx \in ((C.Body)(j)).Var \bullet nil\ ) \wedge$$
$$(\sigma_1(j)).\phi =$$
$$(\ \lambda\, x : Var_{Priv} \mid x \in ((C.Body)(j)).Priv \bullet nil\ ) \bowtie$$
$$\{\ \gamma : OBJ;\ k : 1 \ldots \#Actual \mid Actual(\ k\ ) = name(\ \gamma\ ) \bullet$$
$$(C.Param)(\ k\ ) \mapsto \gamma\ \} \wedge$$
$$(\exists\, ar : AR \mid ar.Cur = init \wedge ar.Env = \varnothing \bullet$$
$$(\sigma_1(j)).\rho = \mathrm{push}(\ \mathrm{empty}, ar\ )) \wedge$$
$$(\sigma_1(j)).\eta = 0$$
$$Program_1 = Program$$
$$Main_1 = Main$$

Effects on creator

The creator is only altered in the fragment that requested the creation. Its statement component becomes the expression denoting its waiting for the results from the execution of *init* by the (still unknown) object α_1. Its count component is simply incremented

[7] It is easy to see how to modify this semantics in order to switch from synchronous to asynchronous creation of class instances.

[8] It would be interesting to provide the created object with new environments or restricted versions of the current one. Our presentation emphasizes the fact that objects carry "their vision of the world" (in their *Program* component). Further investigations are needed to assess the possibilities of such an approach.

to reflect the fact that one more object creation has been requested.

NewCallCreator

NewCallPrecond

$$\psi' = \psi$$
$$\forall j : 1 \,..\, \#\sigma \mid j \neq i \bullet \sigma'(\ j\ ) = \sigma(\ j\ )$$
$$(\sigma'(\ i\ )).s = exp(wait(\ \alpha_1, init\ ))$$
$$(\sigma'(\ i\ )).\delta = (\sigma(\ i\ )).\delta$$
$$(\sigma'(\ i\ )).\phi = (\sigma(\ i\ )).\phi$$
$$(\sigma'(\ i\ )).\rho = (\sigma(\ i\ )).\rho$$
$$(\sigma'(\ i\ )).\eta = (\sigma(\ i\ )).\eta + 1$$

The whole transition is described by the conjunction of both effects:

$$NewCall \triangleq NewCallCreated \wedge NewCallCreator$$

5.4.3 Return

The return from an external call is quite similar to the already presented internal one (see section 5.3.4). The introduced variables are the following:

ExtCoRetPrecond

TwoObjects
$$ready, waiting : \mathbb{F}_1\ BodyNum_{Class}$$
$$\vec{\gamma} : \mathrm{seq}_1\ OBJ$$
$$mf : Name_{Method}$$

$$mf \in \mathrm{dom}((Program_1(\tau_1)).Method)$$
$$ready = \mathrm{dom}(((Program_1(\tau_1)).Method)(\ mf\ )).Alloc$$
$$\#\vec{\gamma} = \#ready$$

The set *ready* holds the indexes of those fragments that are computing the method named *mf*; *waiting* is the indexes set of the waiting fragments in object α. As in the internal case, the result is a composed object, $\vec{\gamma}$, made of all the fragments contributions.

Pre-conditions are also very similar to the ones introduced for the return from an internal coordinated call: p-callees must be ready and p-callers must be waiting.

ExtCoRetPrecond

ExtCoRetPrecond

$$\forall j : 1 \,..\, \#\sigma_1 \mid j \in ready \bullet$$
$$(\mathrm{top}(\ (\sigma_1(\ j\ )).\rho\ )).Cur = mf \wedge$$
$$(\sigma_1(\ j\ )).s = send(\ exp(name(\ \vec{\gamma}(\ j\ )\ )), (\alpha, waiting)\ )$$
$$\forall j : 1 \,..\, \#\sigma \mid j \in waiting \bullet$$
$$(\sigma(\ j\ )).s = exp(wait(\ \alpha_1, mf\ ))$$

The sending object is only altered in the statement component of those fragments that elaborated the result. Their statement becomes (the expression denoting) their contribution of the result:

ExtCoRetControlEffect

ExtCoRetPrecond

$$
\begin{array}{l}
\forall j : 1 \,..\, \#\sigma_1 \bullet \\
\quad (\ j \in \mathit{ready} \Rightarrow \\
\qquad (\sigma_1'(\ j\ )).s = \mathit{exp}(\mathit{name}(\ \vec{\gamma}(\ j\ )\ ))\ ) \wedge \\
\quad (\ j \notin \mathit{ready} \Rightarrow \\
\qquad (\sigma_1'(\ j\ )).s = (\sigma_1(\ j\ )).s\ )
\end{array}
$$

Similarly, the receiving object is modified only in the statement component of those waiting fragments. Their statement becomes (the expression that denotes) the result. However, a distinction is made between the result of any method and that of the special *init* one. The *init* method delivers a peculiar result: a composed object made out of identical object names. The composed value is not very useful in naming the object. A single object name is needed; (the expression denoting) it is required for addressing any message to the object. As we can be sure that at least the first contribution is always present, we deliver (the expression denoting) that object name as the result of an *init* method[9].

ExtCoRetControlEffect

ExtCoRetPrecond

$$
\begin{array}{l}
\forall j : 1 \,..\, \#\sigma \bullet \\
\quad (\ (j \in \mathit{waiting} \wedge \mathit{mf} \neq \mathit{init}) \Rightarrow \\
\qquad (\sigma'(\ j\ )).s = \mathit{exp}(\mathit{name}(\mathit{seqobj}(\ \vec{\gamma}\ )))\ ) \wedge \\
\quad (\ (j \in \mathit{waiting} \wedge \mathit{mf} = \mathit{init}) \Rightarrow \\
\qquad (\sigma'(\ j\ )).s = \mathit{exp}(\mathit{name}(\ \vec{\gamma}(\ 1\ )\ ))\ ) \wedge \\
\quad (\ j \notin \mathit{waiting} \Rightarrow \\
\qquad (\sigma'(\ j\ )).s = (\sigma(\ j\ )).s\ )
\end{array}
$$

The stack effect concerns solely the sending object. Once again, a distinction is made between any method and *init*. The initial method lacks of local variables. Instead, it uses private and partial ones. Therefore, the corresponding activation record allocates an empty local store in its *Env* component. There is no need to pop out that record.

ExtCoRetStackEffect

ExtCoRetPrecond

$$
\begin{array}{l}
\forall j : 1 \,..\, \#\sigma_1 \bullet \\
\quad (\ (j \notin \mathit{ready} \vee \mathit{mf} = \mathit{init}) \Rightarrow (\sigma_1'(\ j\ )).\rho = (\sigma_1(\ j\ )).\rho\ ) \wedge \\
\quad (\ (j \in \mathit{ready} \wedge \mathit{mf} \neq \mathit{init}) \Rightarrow (\sigma_1'(\ j\ )).\rho = \mathrm{pop}(\ (\sigma_1(\ j\ )).\rho\ )\ )
\end{array}
$$

[9]This differenciation in the results raises the following questions: why should the result of a multiprocedure always be the composed object yielded by the evaluation of its body's statements? could we provide a *filter* (such as an arithmetic mean, a majority value or "exactly one out of all" value) as an extra parameter in the multiprocedure definition, generalizing the idea exposed in [Cooper, 1984]? Notice that deterministic filters are already achievable with a trivial multiprocedure composition. Nevertheless, a more sophisticated return mechanism could be devised and requires further research.

The whole transition is specified as usual by the conjunction of both effects:

$$ExtCoRet \mathrel{\widehat=} ExtCoRetControlEffect \land ExtCoRetStackEffect$$

5.4.4 Summarizing

The schema calculus is used to aggregate the descriptions of the transitions that involve more than a single object. Two schemas are used: *ThreeConfigurationsTransition* for creation related transitions and *FourConfigurationsTransition* for communications.

ThreeConfigurationsTransition

$$\Xi\, Config_{Local}$$
$$Config_{Local_1}$$

$$\exists\, C : CLASS \bullet$$
$$(\exists\, i : BodyNum_{Class};\ Actual : \mathrm{seq}\, Exp \bullet NewCall) \lor$$
$$(\exists\, ready : \mathbb{F}_1\, BodyNum_{Class};\ Actual : \mathrm{seq}(\ Var_{Priv} \times Exp\ );$$
$$\partial actual : \mathrm{seq}(\ \mathrm{seq}(\ Var_{Priv} \times Exp\ )\ ) \bullet CoNewCall)$$

FourConfigurationsTransition

$$\Xi\, Config_{Local}$$
$$\Xi\, Config_{Local_1}$$

$$\exists\, mf : Name_{Method};\ M : METHOD \bullet$$
$$(\exists\, i : BodyNum_{Class};\ Actual : \mathrm{seq}\, Exp \bullet ExtCall) \lor$$
$$(\exists\, ready : \mathbb{F}_1\, BodyNum_{Class};\ Actual : \mathrm{seq}(\ Var_{Loc} \times Exp\ );$$
$$\partial actual : \mathrm{seq}(\ \mathrm{seq}(\ Var_{Loc} \times Exp\ )\ ) \bullet ExtCoCall) \lor$$
$$(\exists\, ready, waiting : \mathbb{F}_1\, BodyNum_{Class};\ \vec{\gamma} : \mathrm{seq}_1\, OBJ \bullet ExtCoRet)$$

6 Putting all together

In the previous sections, we have specified sets of transitions over local configurations. However, we have not shown the formal deductive system needed to prove them. Such a *transition system* consists of *axioms* and *rules*. Axioms characterize the basic transitions, e.g. those we have specified so forth. Rules indicate how to deduce new transitions from proven ones. Axioms and rules together determine the set of all the transitions that are provable in the system. This set is known as a *transition relation*. Before we can specify our binary (transition) relation, noted $\overset{||trans}{\longrightarrow}$ ($\subseteq Config \times Config$), we have to exhibit the rules that depict computation progress.

6.1 Computation progress rules

We have been requiring statements and expressions to be evaluated, but we have not presented any actual way of doing so. The following sections compensate for this deficiency.

6.1.1 Parameters evaluation

Although no order is essential to the evaluation of parameters when no side-effects can occur, parameters evaluation is described as being performed from left to right[10]. Recall that actual parameters for a simple call are specified as sequences of expressions. Two such sequences are introduced together with two expressions in order to specify the evaluation order:

*LeftToRightParEval*1

$$
\begin{array}{l}
e, e' : Exp \\
Actual, Actual' : \mathrm{seq}\ Exp \\
\hline
\exists_1 j : 1 \mathinner{..} \#Actual \bullet \\
\quad Actual(\ j\ ) = e \wedge \\
\quad (\forall k : 1 \mathinner{..} \#Actual \mid k < j \bullet \\
\qquad \exists \gamma : OBJ \bullet Actual(\ k\ ) = name(\ \gamma\ )) \wedge \\
\quad Actual' = Actual \oplus \{\ j \mapsto e'\ \}
\end{array}
$$

The intuitive meaning is the following: e being the first not fully evaluated expression in *Actual*, from left to right, *Actual'* is identical to *Actual* except that expression e is replaced by e'.

Actual parameters for a coordinated call are specified as a sequence of couples (variable-expression); that is the main difference between the above schema and the one below which describes the evaluation of parameters for a coordinated call. The variable set is left generic so that the same schema may describe both class and method parameters.

*LeftToRightParEval*2[*VAR*]

$$
\begin{array}{l}
e, e' : Exp \\
Actual, Actual' : \mathrm{seq}(\ VAR \times Exp\ ) \\
\hline
\exists_1 j : 1 \mathinner{..} \#Actual \bullet \\
\quad (\exists v : VAR \bullet Actual(\ j\ ) = (\ v, e\ )) \wedge \\
\quad (\forall k : 1 \mathinner{..} \#Actual \mid k < j \bullet \\
\qquad \exists \gamma : OBJ;\ v : VAR \bullet Actual(\ k\ ) = (\ v, name(\ \gamma\ )\ )) \wedge \\
\quad Actual' = Actual \oplus \{\ j \mapsto (\ \pi_1(\ Actual(\ j\ )\ ), e'\ )\ \}
\end{array}
$$

6.1.2 Computation progress in expressions

Progress in the evaluation of an expression is given by rules of the form "if e becomes e', then $f(e)$ becomes $f(e')$", where f ranges over the set of syntactic constructs that involve an expression. All these rules –each f entails a rule– share the premise "e becomes e'". This premise is actually more complicated. It assumes that a local configuration transition occurs due to one of its fragments statement component being e and becoming e'.

[10]This choice is motivated by the implementation described in [Lecler, 1989].

ExpressionCommonPremise

$\Xi\ Config_{Local}$
$i : BodyNum_{Class}$
$e, e' : Exp$

$i \in \mathrm{dom}\ \sigma$

$(\sigma(\ i\ )).s = exp(\ e\ ) \land (\sigma'(\ i\ )).s = exp(\ e'\ )$

The consequence in each rule –"$f(e)$ becomes $f(e')$"– is also more complicated, but in a similar way. The schemas describing a rule consequence share the declaration of another local configuration transition, $\Xi\ Config_{Local1}$. Local configurations before and after the transition are pairwise identical, except for their responsible fragment statement component (the i^{th}). We capture this information in the schema below.

ExpEffectPrelude

ExpressionCommonPremise
$\Xi\ Config_{Local1}$

$\alpha_1 = \alpha$

$\psi_1 = \psi \land \psi_1' = \psi'$

$\forall j : 1\,..\,\#\sigma \mid j \neq i \bullet$
$\quad \sigma_1(\ j\ ) = \sigma(\ j\ ) \land \sigma_1'(\ j\ ) = \sigma'(\ j\ )$

$(\sigma_1(\ i\ )).\delta = (\sigma(\ i\ )).\delta \land (\sigma_1'(\ i\ )).\delta = (\sigma'(\ i\ )).\delta$
$(\sigma_1(\ i\ )).\phi = (\sigma(\ i\ )).\phi \land (\sigma_1'(\ i\ )).\phi = (\sigma'(\ i\ )).\phi$
$(\sigma_1(\ i\ )).\rho = (\sigma(\ i\ )).\rho \land (\sigma_1'(\ i\ )).\rho = (\sigma'(\ i\ )).\rho$
$(\sigma_1(\ i\ )).\eta = (\sigma(\ i\ )).\eta \land (\sigma_1'(\ i\ )).\eta = (\sigma'(\ i\ )).\eta$

$\tau_1 = \tau$

$Program_1 = Program \land Main_1 = Main$

As the reader may have guessed, the responsible fragment statement components of local configurations with the 1 subscript are meant to range over the set of syntactic constructs that involve an expression. For instance, the rule consequence for parameters evaluation of an external method call is written as follows:

*ExpEvalPar*1

ExpEffectPrelude

$\exists\ Actual, Actual' : \mathrm{seq}\ Exp \mid LeftToRightParEval1 \bullet$
$\quad (\exists\ mf : Name_{Method};\ \gamma : OBJ \bullet$
$\quad\quad (\sigma_1(\ i\ )).s = exp(meth(\ name(\ \gamma\ ), mf, Actual\ )) \land$
$\quad\quad (\sigma_1'(\ i\ )).s = exp(meth(\ name(\ \gamma\ ), mf, Actual'\ )))$

z The values for e and e' although constrained by the *LeftToRightParEval*1 schema, are those provided within the current environment, i.e. relative to *ExpEffectPrelude*'s declaration. ◇

Notice that the expression naming the destination object is (implicitly) required to be fully evaluated. Another rule tells how to evaluate it:

ExpEvalDest

ExpEffectPrelude

$$\exists\, mf : Name_{Method};\ Actual : \mathrm{seq}\ Exp \bullet$$
$$(\sigma_1(\ i\ )).s = exp(meth(\ e, mf, Actual\ )) \wedge$$
$$(\sigma_1'(\ i\ )).s = exp(meth(\ e', mf, Actual\ ))$$

An expression being assigned is evaluated in accordance with the rule:

ExpEvalAssign

ExpEffectPrelude

$$\exists\, fx : Var_{Part} \bullet$$
$$(\sigma_1(\ i\ )).s = fassign(\ fx, e\ ) \wedge (\sigma_1'(\ i\ )).s = fassign(\ fx, e'\ )$$

As a last example, a boolean expression discriminating an *if* statement is evaluated with the rule:

ExpEvalBool

ExpEffectPrelude

$$\exists\, s_1, s_2 : Stmt \bullet$$
$$(\sigma_1(\ i\ )).s = if(\ e, s_1, s_2\ ) \wedge (\sigma_1'(\ i\ )).s = if(\ e', s_1, s_2\ )$$

6.1.3 Computation progress in statements

Statements evaluation is formalized in a very similar manner. The common premise is written as follows:

StatementCommonPremise

$$\Xi\ Config_{Local}$$
$$i : BodyNum_{Class}$$
$$s, s' : Stmt$$

$$i \in \mathrm{dom}\ \sigma$$
$$(\sigma(\ i\ )).s = s \wedge (\sigma'(\ i\ )).s = s'$$

After having specified the *StmtEffectPrelude* (written in the appendix), we can easily give the rule for the sequential composition,

StmtEvalSeqComp

StmtEffectPrelude

$$\exists\, s_1 : Stmt \bullet$$
$$(\sigma_1(\ i\ )).s = seqcomp(\ s, s_1\ ) \wedge (\sigma_1'(\ i\ )).s = seqcomp(\ s', s_1\ )$$

and the one for the send statement:

StmtEvalSend

$$
\begin{array}{|l}
\hline
\textit{StmtEffectPrelude} \\
\hline
\exists\, receiver : CallId \bullet \\
\quad (\sigma_1(\ i\ )).s = send(\ s, receiver\ ) \land (\sigma_1'(\ i\ )).s = send(\ s', receiver\ ) \\
\hline
\end{array}
$$

6.1.4 Summarizing

As we have done in previous sections, we can summarize the rules we have just specified in a single schema:

ComputationProgress

$$
\begin{array}{|l}
\hline
\Xi\, Config_{Local} \\
\Xi\, Config_{Local_1} \\
\hline
(\ \exists\, i : BodyNum_{Class};\ e, e' : Exp \mid ExpEffectPrelude \bullet \\
\quad ExpEvalPar1 \lor ExpEvalPar2 \lor ExpEvalPar3 \lor \\
\quad ExpEvalDest \lor ExpEvalAssign \lor ExpEvalBool\ ) \\
\lor \\
(\ \exists\, i : BodyNum_{Class};\ s, s' : Stmt \mid StmtEffectPrelude \bullet \\
\quad StmtEvalSeqComp \lor StmtEvalSend\ ) \\
\hline
\end{array}
$$

6.2 The transition system at last

Now that we have presented the axioms and the rules that characterize our transition relation, we can write down the schema that specifies it. In what follows, the auxiliar infix function $\sqcup$ is used to exhibit particular local configurations out of a global configuration. Let X, Y and W be three sets; $W = X \sqcup Y$ means that all the elements of Y belong to W, but none to X (see A.1 for the exact definition).

Let C, C' and X be configurations. We define $\overset{ll|trans}{\longrightarrow}$ as follows:

- $C \overset{ll|trans}{\longrightarrow} C'$ if C' reflects the transformation of one of C's local configurations, and the transformation is one of the *two configurations transitions*:

$$
\begin{array}{l}
\exists\, TwoConfigurationsTransition \bullet \\
\quad C = X \sqcup \{\ \theta Config_{Local}\ \} \land \\
\quad C' = X \sqcup \{\ \theta Config_{Local}'\ \}
\end{array}
$$

- $C \overset{ll|trans}{\longrightarrow} C'$ if they are related by one of the *three configurations transitions*:

$$
\begin{array}{l}
\exists\, ThreeConfigurationsTransition \bullet \\
\quad C = X \sqcup \{\ \theta Config_{Local}\ \} \land \\
\quad C' = X \sqcup \{\ \theta Config_{Local}', \theta Config_{Local_1}\ \}
\end{array}
$$

- $C \overset{ll|trans}{\longrightarrow} C'$ if they are related by one of the *four configurations transitions*:

$$
\begin{array}{l}
\exists\, FourConfigurationsTransition \bullet \\
\quad C = X \sqcup \{\ \theta Config_{Local}, \theta Config_{Local_1}\ \} \land \\
\quad C' = X \sqcup \{\ \theta Config_{Local}', \theta Config_{Local_1}'\ \}
\end{array}
$$

- $C \overset{||trans}{\longrightarrow} C'$ if they can be deduced from a proved transition, i.e. a pair of configurations already in $\overset{||trans}{\longrightarrow}$, to which they relate by one of the *computation progress* rules:

$$
\begin{array}{l}
\exists\, ComputationProgress \bullet \\
\quad X \sqcup \{\, \theta Config_{Local} \,\} \overset{||trans}{\longrightarrow} X \sqcup \{\, \theta Config_{Local}{}' \,\} \land \\
\quad C = X \sqcup \{\, \theta Config_{Local_1} \,\} \land \\
\quad C' = X \sqcup \{\, \theta Config_{Local_1}' \,\}
\end{array}
$$

- no other configuration pair is in $\overset{||trans}{\longrightarrow}$.

The transition system is specified in **Z** with the following generic schema.

$$
\begin{array}{l}
\_ \overset{||trans}{\longrightarrow} \_ : Config \leftrightarrow Config \\
\hline
\forall\, C, C' : Config \bullet \\
\quad C \overset{||trans}{\longrightarrow} C' \Leftrightarrow \\
\qquad (\ \exists\, TwoConfigurationsTransition;\ X : Config \bullet \\
\qquad\quad C = X \sqcup \{\, \theta Config_{Local} \,\} \land \\
\qquad\quad C' = X \sqcup \{\, \theta Config_{Local}{}' \,\}\ ) \\
\qquad \lor \\
\qquad (\ \exists\, ThreeConfigurationsTransition;\ X : Config \bullet \\
\qquad\quad C = X \sqcup \{\, \theta Config_{Local} \,\} \land \\
\qquad\quad C' = X \sqcup \{\, \theta Config_{Local}{}', \theta Config_{Local_1} \,\}\ ) \\
\qquad \lor \\
\qquad (\ \exists\, FourConfigurationsTransition;\ X : Config \bullet \\
\qquad\quad C = X \sqcup \{\, \theta Config_{Local}, \theta Config_{Local_1} \,\} \land \\
\qquad\quad C' = X \sqcup \{\, \theta Config_{Local}{}', \theta Config_{Local_1}' \,\}\ ) \\
\qquad \lor \\
\qquad (\ \exists\, ComputationProgress;\ X : Config \bullet \\
\qquad\quad X \sqcup \{\, \theta Config_{Local} \,\} \overset{||trans}{\longrightarrow} X \sqcup \{\, \theta Config_{Local}{}' \,\} \land \\
\qquad\quad C = X \sqcup \{\, \theta Config_{Local_1} \,\} \land \\
\qquad\quad C' = X \sqcup \{\, \theta Config_{Local_1}' \,\}\ )
\end{array}
$$

Notice that the above transition system adopts an interleaved interpretation of parallelism: atomic transitions happen one after another in a non-fixed arbitrary order. The next section enriches our system with a couple of rules that allow concurrent actions within a single transition.

6.3 Parallelism rules

There are two different levels of parallelism in POLYGOTH: the fragment and the object levels. We present a rule for each kind.

6.3.1 Fragment level parallelism

In order to allow different fragments to proceed in parallel, some precautions must be taken. Otherwise, we could easily end up with a myriad of inconsistent states of the host object. Indeed, fragments may meddle in each others' stores (see the global assignment and the remote one in 5.3.2). Fragment's non interference can be uncovered with a simple commutativity test. Consider three local configurations representing the same object, $Config_{Local}$, $Config_{Local}'$ and $Config_{Local_1}$. Suppose that $Config_{Local}'$ and $Config_{Local_1}$ are the final states of two distinct fragment transitions of $Config_{Local}$ taken in isolation. If the two transitions do not interfere, then they can occur in any order and still produce the same results. Here, the result is a local configuration and its relevant components are the sequence of partial states, σ, and the global store, ϕ. Since both are functions, the values yielded by a serial occurrence of the two transitions may be computed by updating the values produced by the first with those resulting from the second. Hence, if the update operation is commutative on the sequence of partial states and on the global store, the concerned transitions do not interfere:

FragmentNonInterference

$\Xi\ Config_{Local}$
$Config_{Local_1}$

$\alpha = \alpha_1 \wedge \tau = \tau_1$

$\sigma' \bowtie \sigma_1 = \sigma_1 \bowtie \sigma'$

$\psi' \bowtie \psi_1 = \psi_1 \bowtie \psi'$

We now suppose that both transitions occur in parallel. Then, the resulting local configuration, say $Config_{Local}'_1$, would be specified in terms of both transition final states –had they occurred serially– as follows:

TwoInOne

$Config_{Local}'$
$\Xi\ Config_{Local_1}$

$\sigma_1' = \sigma' \bowtie \sigma_1$
$\psi_1' = \psi' \bowtie \psi_1$

The non interference of transitions –as we have presented it– is not a strong enough property to consider their parallel occurrence desirable. Think of the possible communications each transition may involve. It is easy to imagine a scenario where an (external) object is ready to accept a message sent by either one of the fragments but is certainly not able to handle them both in parallel. We have to restrict the way other local configurations are altered by potentially parallel transitions. We could ban the parallel occurrence of all transitions entailing non-local effects, but that would be a too drastic expidient. If we simply require all objects bearing the same name to be identical, the trick is done![11]. If this property holds across two global configurations, we say that they *comply*:

[11] Actually, a more sofisticated requirement is needed in order to include all the reasonably parallel transitions. Indeed, in the just referred scenario, two messages from the same object can be handled

$$\_ \text{ complies } \_ : \mathit{Config} \leftrightarrow \mathit{Config}$$

$$\forall X, X' : \mathit{Config} \bullet$$
$$X \text{ complies } X' \Leftrightarrow$$
$$(\forall c, c' : \mathit{Config}_{Local} \mid c \in X \wedge c' \in X' \bullet c.\alpha = c'.\alpha \Leftrightarrow c = c')$$

The following property is self explanatory:

ParallelismInside

FragmentNonInterference
TwoInOne

$$\forall X, X', X_1 : \mathit{Config} \bullet$$
$$(\; X \sqcup \{ \theta \mathit{Config}_{Local} \} \overset{||trans}{\longrightarrow} X' \sqcup \{ \theta \mathit{Config}_{Local}{}' \} \wedge$$
$$X \sqcup \{ \theta \mathit{Config}_{Local} \} \overset{||trans}{\longrightarrow} X_1 \sqcup \{ \theta \mathit{Config}_{Local_1} \} \wedge$$
$$X' \text{ complies } X_1 \;) \Rightarrow$$
$$(\; X \sqcup \{ \theta \mathit{Config}_{Local} \} \overset{||trans}{\longrightarrow} (X' \cup X_1) \sqcup \{ \theta \mathit{Config}_{Local}{}'_1 \} \vee$$
$$X \sqcup \{ \theta \mathit{Config}_{Local} \} \overset{||trans}{\longrightarrow} X' \sqcup \{ \theta \mathit{Config}_{Local}{}' \} \;)$$

Notice that we do not demand a *maximal* parallelism. We concede the serialized occurrence of a potentially parallel pair of transitions. The parallelism we have just characterized falls in the spectrum of fine to medium grained, depending on the size of object global data and on the thickness of its fragmentation. Next section exhibits a parallelism in the medium to coarse grained spectrum.

6.3.2 Object level parallelism

As trivial as it may seem, whenever two *independent* subsets of a configuration can participate in two separate transitions, both may occur in parallel. Two configurations are said to be independent if they share no local configuration. Indeed, if no local configuration is shared by two configurations, they have no way to communicate and thereby, no possible interference arises from a parallel occurrence of their transitions. Recall that a configuration comprises all objects, *active*, *garbage* and *permanent* data[12]. We specify this coarser parallelism as follows:

in parallel if the acceptance of both does not entail a fragment interference. We can require any pair of objects bearing the same name to be in non interfering states, and update them in a single object as the result of the parallel execution of the transitions that made them diverge. In this paper, we maintain the simpler requirement in order to avoid a somehow futile complexity.

[12]We say an object is active if at least one of its fragments is not idle; if all of its fragments are idle, it is garbage if no other object stores its name, otherwise, it is permanent.

ParallelismOutside

$$X, X', Y, Y' : Config$$

$$\begin{array}{l} (\ X \xrightarrow{\|trans} X' \wedge \\ \quad Y \xrightarrow{\|trans} Y' \wedge \\ \quad X \cap Y = \varnothing\ ) \Rightarrow \\ (\ X \cup Y \xrightarrow{\|trans} X' \cup Y' \vee \\ \quad X \cup Y \xrightarrow{\|trans} X \cup Y'\ ) \end{array}$$

Notice that again we do not demand a maximal parallelism, but it is still very easily expressible.

6.4 Enhanced transition system

We define an enhanced transition relation, noted $\xrightarrow{\|trans}$, that incorporates the parallelism rules just presented. Let C and C' be two configurations in what follows.

- $C \xrightarrow{\|trans} C'$ if the transition is provable in the previous transition system:

$$C \xrightarrow{\|trans} C'$$

- $C \xrightarrow{\|trans} C'$ if two provable transitions may be derived from an object in C that bear (internal) parallelism, and C' is the configuration that results from their parallel occurrence:

$$\begin{array}{l} \exists\, FragmentNonInterference;\ TwoInOne; \\ \quad X, X', X_1 : Config \bullet \\ \quad C = X \sqcup \{\ \theta Config_{Local}\ \} \wedge \\ \quad C \xrightarrow{\|trans} X' \sqcup \{\ \theta Config_{Local}{}'\ \} \wedge \\ \quad C \xrightarrow{\|trans} X_1 \sqcup \{\ \theta Config_{Local_1}\ \} \wedge \\ \quad C' = (X' \cup X_1) \sqcup \{\ \theta Config_{Local_1}'\ \} \end{array}$$

- $C \xrightarrow{\|trans} C'$ if two independent subsets of C root two provable transitions, and C' is the union of the resulting subsets:

$$\begin{array}{l} \exists\, X, X', Y, Y' : Config \bullet \\ \quad C = X \sqcup Y \wedge \\ \quad X \xrightarrow{\|trans} X' \wedge \\ \quad Y \xrightarrow{\|trans} Y' \wedge \\ \quad C' = X' \cup Y' \end{array}$$

- no other configuration pair is in $\xrightarrow{\|trans}$.

The transition system is specified in **Z** with the following generic schema:

$$
\begin{array}{|l}
\hline
\_ \overset{\|trans}{\longrightarrow} \_ : Config \leftrightarrow Config \\
\hline
\forall C, C' : Config \bullet \\
\quad C \overset{\|trans}{\longrightarrow} C' \Leftrightarrow \\
\qquad (\ C \overset{\|trans}{\longrightarrow} C'\ ) \\
\qquad \vee \\
\qquad (\ \exists\, FragmentNonInterference;\ TwoInOne; \\
\qquad\quad X, X', X_1 : Config \bullet \\
\qquad\quad C = X \sqcup \{\ \theta Config_{Local}\ \} \wedge \\
\qquad\quad C \overset{\|trans}{\longrightarrow} X' \sqcup \{\ \theta Config_{Local}{}'\ \} \wedge \\
\qquad\quad C \overset{\|trans}{\longrightarrow} X_1 \sqcup \{\ \theta Config_{Local_1}\ \} \wedge \\
\qquad\quad C' = (X' \cup X_1) \sqcup \{\ \theta Config'_{Local_1}\ \}\ ) \\
\qquad \vee \\
\qquad (\ \exists X, X', Y, Y' : Config \bullet \\
\qquad\quad C = X \sqcup Y \wedge \\
\qquad\quad X \overset{\|trans}{\longrightarrow} X' \wedge \\
\qquad\quad Y \overset{\|trans}{\longrightarrow} Y' \wedge \\
\qquad\quad C' = X' \cup Y'\ ) \\
\hline
\end{array}
$$

6.5 Meaning of a unit

Now, having described the transition relation $\overset{\|trans}{\longrightarrow}$, we specify what we want to consider as the meaning of a unit. We introduce the set of (finite and infinite) sequences of configurations. We name those sequences *traces* and their set *Trace*:

$$Trace == \mathbb{N} \nrightarrow Config$$

The meaning of a unit, u, is the set of traces that satisfy the following properties:

- the same initial (local) configuration is included in their first configuration. Furthermore, the initial (local) configuration is the one whose *Program* and *Main* components have u's values:

$$
\begin{array}{l}
\exists_1\, Config_{Local};\ n? : \mathbb{N} \bullet \\
\quad \forall t : Trace \mid t \in Meaning(\ u\ ) \bullet \\
\qquad (\ InitConfig_{Local} \wedge Program = u.Program \wedge Main = u.Main \wedge \\
\qquad \{\ \theta Config_{Local}\ \} \subseteq t(\ 0\ )\ )
\end{array}
$$

  Note that we do not restrict our semantics to self-contained units: units may be defined within an existing (running) system, and classes can contain references –through direct naming expressions– to objects created by other units.

- any two successive configurations of a trace are related by a provable transition:

$$
\begin{array}{l}
(\ \forall k : \mathbb{N}_1 \mid k \in \operatorname{dom} t \bullet \\
\quad t(\ k-1\ ) \overset{\|trans}{\longrightarrow} t(\ k\ )\ )
\end{array}
$$

- every finite trace is complete:

$$\begin{array}{l}(\ (\exists n : \mathbb{N}_1 \bullet n = \#(\mathrm{dom}\, t)) \Rightarrow \\ \quad \neg \ (\ \exists C : \mathit{Config} \bullet t(\ \#(\mathrm{dom}\, t) - 1 \ ) \stackrel{\|_{trans}}{\longrightarrow} C \ ) \ )\end{array}$$

- the whole trace satisfies the fairness condition. The fairness condition –intuitively– requires that there is no trace in which transitions infinitely often allowed never occur. More formally, it is impossible to find an object o that can produce a provable transition from C to C', and an infinite set, *Ready*, of indexes of the trace such that for every index in the set, the corresponding configuration of the trace is a superset of C –i.e. the transition induced by o is allowed to occur– and the next configuration in the trace includes object o, i.e. the o-led transition does not occur.

$$\begin{array}{l}\neg \ (\ \exists o : \mathit{Config}_{Local}; \ C, C' : \mathit{Config}; \ \mathit{Ready} : \mathbb{P}\, \mathbb{N} \bullet \\ \quad o \in C \wedge C \stackrel{\|_{trans}}{\longrightarrow} C' \wedge o \notin C' \wedge \\ \quad \mathit{Ready} \subseteq \mathrm{dom}\, t \wedge \mathit{Ready} \neq \varnothing \wedge \\ \quad (\ \forall i : \mathbb{N} \mid i \in \mathit{Ready} \bullet \\ \qquad (\exists j : \mathbb{N} \mid j > i \bullet j \in \mathit{Ready}) \wedge \\ \qquad C \subseteq t(\ i \ ) \wedge o \in t(\ i + 1 \ ) \ ) \ )\end{array}$$

The following generic schema specifies our meaning function:

$$\begin{array}{|l}\hline \mathit{Meaning} : \mathit{UNIT} \rightarrow \mathbb{P}\ \mathit{Trace} \\ \hline \forall u : \mathit{UNIT} \bullet \exists_1 \mathit{Config}_{Local}; \ n? : \mathbb{N} \bullet \\ \quad \forall t : \mathit{Trace} \mid t \in \mathit{Meaning}(\ u \ ) \bullet \\ \qquad (\ \mathit{InitConfig}_{Local} \wedge \mathit{Program} = u.\mathit{Program} \wedge \mathit{Main} = u.\mathit{Main} \wedge \\ \qquad \{ \ \theta \mathit{Config}_{Local} \ \} \subseteq t(\ 0 \ ) \ ) \wedge \\ \qquad (\ \forall k : \mathbb{N}_1 \mid k \in \mathrm{dom}\, t \bullet \\ \qquad t(\ k - 1 \ ) \stackrel{\|_{trans}}{\longrightarrow} t(\ k \ ) \ ) \wedge \\ \qquad (\ (\exists n : \mathbb{N}_1 \bullet n = \#(\mathrm{dom}\, t)) \Rightarrow \\ \qquad \neg \ (\ \exists C : \mathit{Config} \bullet t(\ \#(\mathrm{dom}\, t) - 1 \ ) \stackrel{\|_{trans}}{\longrightarrow} C \ ) \ ) \wedge \\ \qquad \neg \ (\ \exists o : \mathit{Config}_{Local}; \ C, C' : \mathit{Config}; \ \mathit{Ready} : \mathbb{P}\, \mathbb{N} \bullet \\ \qquad o \in C \wedge C \stackrel{\|_{trans}}{\longrightarrow} C' \wedge o \notin C' \wedge \\ \qquad \mathit{Ready} \subseteq \mathrm{dom}\, t \wedge \mathit{Ready} \neq \varnothing \wedge \\ \qquad (\ \forall i : \mathbb{N} \mid i \in \mathit{Ready} \bullet \\ \qquad (\exists j : \mathbb{N} \mid j > i \bullet j \in \mathit{Ready}) \wedge \\ \qquad C \subseteq t(\ i \ ) \wedge o \in t(\ i + 1 \ ) \ ) \ ) \\ \hline\end{array}$$

This definition concludes our work. Some final remarks are due and an early appraisal of what has been done is in order.

7 Comments and related work

The transition system we have obtained is quite large and complex. Even though a simpler semantics would have been more elegant, "you can't have your cake and eat it". On one hand, the transition relation describes a realistic programming language which is, in itself, quite complex. A fragmented object is essentially more complicated than a plain one. On the other hand, very few things are hidden in our specification. Getting deep into details has a huge information overhead. The operational approach entails the definition of concrete structures and the burden of their associated management. When those structures are an abstraction of the objects one is interested in, that burden is precisely what the language is relieving one from, so their management has to be neatly specified at least once. Putting it in another way, the transition relation of an operational semantics "à la Plotkin" is the well understood (and formalized) part of the definition: it is a binary relation among configurations and it defines the engine of an abstract machine. The not so well formalized part is usually the machine itself: its states and their available operations. We think that languages like **Z** are well suited for this purpose. This section provides an early assessment of the merits of such a specification. We start with the multiprocedure concept.

7.1 About multiprocedures

A brief inspection of the *METHOD* schema (section 4) reveals that a multiprocedure declaration is a slight generalisation of the procedure one: the body is simply expanded to be a collection of commands that will be executed in parallel. This simple step clearly stresses the underlying assumptions: instead of a plain sequential computation, a parallel activity is being abstracted into a procedure. The *Alloc* component models the (static) nesting of a multiprocedure within a class body. This is one aspect that was difficult to explain before the specification was written. As we shall comment in what follows, this component plays a central role in the description of the communication and synchronisation related aspects of multiprocedures. It should be noted that a general static nesting of multiprocedures is not dealt with in this semantics. For the time being, we cannot foresee a straight generalisation of procedure nesting. Conflicts arise between what we could call a *depth dimension* introduced by the traditional (static) nesting and a *width dimension* due to the parallel multiprocedure body. Scoping and visibility rules are not trivially found to ensure a well disciplined use of time (depth) and space (width) localities. These notions are closely related to inheritance in a fragmented object-oriented language like POLYGOTH. They are quite interesting and deserve further research.

The internal simple call schema, *InCall* (section A.3.2), captures the multiprocedure activation. Some difficulties emerged here. We had to choose between a model that would assume an unbounded degree of parallelism and one that would bound it. In the first case, blocks (from various methods) allocated on the same fragment would be active simultaneously. This approach entails an uncontroled nondeterminism that calls for synchronisation and mutual exclusion mechanisms. Besides, it is somehow antagonistic to the locality professed by multiprocedures. In the second case, only one active block per fragment is allowed. This proposition alleviates the problem just posed and emphasizes that a scheduling strategy is needed as part of the object machine, but it

is not a sound solution. Indeed, as the following scenario shows, unwanted interferences still may occur.

PartialExample

$Config_{Local}$

$$\psi = \{\ sx \mapsto \{\ y \mapsto int(1)\ \}\ \}$$
$$(\sigma(1)).s = exp(\ call(\ mf, \langle\rangle\ )\ )$$
$$(\sigma(2)).s = fassign(\ y, meth(\ fvar(y), add, \langle name(\ int(1)\ )\rangle\ )\ )$$
$$\tau = C$$
$$(((Program(C)).Method)(mf)).Param = \langle\rangle$$
$$(((Program(C)).Method)(mf)).Alloc = \{\ 2 \mapsto 1\ \}$$
$$(((((Program(C)).Method)(mf)).Body)(1)).s =$$
$$\quad fassign(\ y, meth(\ fvar(y), add, \langle name(\ int(2)\ )\rangle\ )\ )$$

The following theorems are provable with the formal transition system:

$$\exists\, C : Config_{Local} \bullet$$
$$\{\ PartialExample\ \} \overset{\|trans^*}{\longrightarrow} \{\ C\ \} \wedge ((C.\psi)(sx))(y) = int(4)$$
$$\exists\, C : Config_{Local} \bullet$$
$$\{\ PartialExample\ \} \overset{\|trans^*}{\longrightarrow} \{\ C\ \} \wedge ((C.\psi)(sx))(y) = int(3)$$
$$\exists\, C : Config_{Local} \bullet$$
$$\{\ PartialExample\ \} \overset{\|trans^*}{\longrightarrow} \{\ C\ \} \wedge ((C.\psi)(sx))(y) = int(2)$$

In the same initial conditions, three different results may be obtained. A generalisation of the known ([Andrews and Schneider, 1983]) concepts and notations for concurrent programming is needed to control nondeterminism, unless such internal calls are only allowed when the called method is *local* to the calling block. Allowing simple calls within multiprocedures could be seen as a design inconsistency whith respect to the coordinated call motivations (see section 2.1.3). Indeed, the coordinated call has the nice property of preserving a one onto one caller-callee relationship. Clearly, this simple call breaks the multiprocedure abstraction as far as composability is concerned. Furthermore, it grants individuality to a multiprocedure component which seems to revert the original intentions. The nondeterminism problem just exposed holds even if simple calls are disallowed and coordinated calls are mandatory for multiprocedures, although it is pushed back at the *object interface* level, that is to say, to the control mechanism used to handle incoming external calls.

As the external simple call schema (*ExtCall* in section A.3.3) shows, the only requirement for an incoming call to be served is that the concerned fragments be idle. One could think that such a discipline ensures that two methods allocated on disjoint subsets of fragments may safely –meaning without interference– be executed in parallel. However, it can be seen from the *PartialExample* schema that either a simple or a coordinated internal call would break the rule: take the statements on fragments number 1 and 2 to be the bodies of two supposedly non-interfering methods. A control mechanism is needed in order to regulate external communications. Several such mechanisms have been proposed [Kafura and Lee, 1989], but inheritance interference is not

yet solved satisfactorily. It is desirable to find a mechanism that would provide both internal and external synchronisation means in a uniform way. Moreover, it could even cope with asynchronism in calls with *future* results –they would be unevaluated or lazily evaluated objects– and in the object creation mechanism, if those two alternatives were chosen. The design of such a mechanism is an open research problem.

Several observations can be made about the coordinated call schema, *InCoCall* (section 5.3.4) & *ExtCoCall* (section 5.4.1):

- we insist on the fact that the simple call is (formally shown to be) a particular coordinated call where the *Ready* set is a singleton, exception made of the actual parameter syntax.

- the apparent complexity of the schema that describes the coordinated call preconditions faints as soon as the expressiveness of the construct is understood: this general call synchronizes a group of activities, collects parameters and then distributes them, and multicasts the results. The group of p-callers is determined with the *Alloc* component of the multiprocedure. Multiprocedures can be compared to SCRIPTS [Francez et al., 1986]. A *script* defines a communication pattern among a set of *roles*, where a *role* is a formal process parameter of a script. In order to participate in a particular communication, processes *enroll* themselves in an instance of the corresponding script. A script role corresponds to a multiprocedure block, and the enrolment is achieved through the coordinated call. However, roles are not directly ascribed: they are hidden in the multiprocedure abstraction together with the communication pattern. Notice that any set of processes can participate in a script (if they know it and agree on the role distribution). Contrastingly, a set of fragments can participate in a coordinated multiprocedure call if, and only if, they are all *enrolled* in the same multiprocedure. The importance of the *Alloc* component appears now sharply: it determines the communication patterns a fragment may get *enrolled* in. It should be noted that a more dynamic definition of the *Alloc* component would yield a more flexible and open communication topology.

- the need for a strong synchronisation amongst cooperating processes in order to program multiprocessors has been recognized. Some low-level primitives have also been proposed. For instance, the *barrier* notation in [Jordan et al., 1989] that forces processes to wait for each other. One arbitrary process is then allowed to execute the sequential section of code within the barrier markers. All other processes jump over the marked section of code and wait until the single process completes the sequencial code. No special annotation in the code are needed to obtain the same behaviour with multiprocedures: write a multiprocedure whose blocks are the `DOALL` or `CASE` sections of code; then write a single-blocked multiprocedure whose body is the section of code within the barrier marks; finally, insert sets of coordinated calls to the single-blocked multiprocedure wherever the barrier was. It is not really fair to compare a programming language concept with a FORTRAN macro instruction, but we hope the comparison will eradicate any confusions.

- Formal to actual parameter matching is checked at run-time because there is no fixed contribution defined at compile-time. A static analysis of programs

could identify always-matching calls and anticipate their parameter checking. The freedom in furnishing parameters to a coordinated call should be appreciated. It suggests also another definition of a coordinated call that would be based on the availability of the parameters rather than on the participation of all the p-callers. This derivative has the drawback of introducing another source of nondeterminism (different matchings could be possible), and it is subject to the same design inconsistency critic that was formulated for the simple call. It is however an interesting alternative since it alleviates the strong requirement that all the multiprocedure blocks issue the call. Further study will tell.

- the stack effect schema shows that recursive coordinated calls are handled. A distributed solution, originally proposed in [Francez et al., 1986], of the Towers of Hanoi makes use of this novel parallel recursion. A parallel recursive sorting with partial calls is reported in [Herman and Trilling, 1990], although with a somehow different approach than ours. Our present research explores how fitted this recursion is to parallel algorithms.

- the return mechanism is exactly the same for coordinated calls as for simple calls. As it was written in a footnote, further research is needed to find out if there is a generalisation of the return mechanism in this new setting. We could think of what L. Trilling [Trilling and Quiniou, 1988] calls the *anti-procedure*: some sort of procedure dual. *Coreturns* could be defined to exercise more control on what happens when a coordinated call returns. The *coreturn* would be a coordinated call dual for anti-multiprocedures. Of course, these are only initial thoughts. Much work remains to be done.

As a final remark, multiprocedures can be contrasted with the *composite procedures* of DINO [Rosing et al., 1988]. DINO is a language for writing numerical programs on distributed memory multiprocessors. A DINO *environment* represents a virtual processor and is very similar to our fragment notion. These environments can be structured as arrays; our corresponding structure is a fragmented object whose fragments are identical[13]. An environment may declare distributed data, in which case a predefined distribution function is indicated. In the presence of an array of environments, *composite procedures* may be defined. They are multiprocedures whose blocks are all identical and whose *Alloc* component is a bijection. However, no coordinated call equivalent is proposed in DINO. No provisions are made for composite procedures residing in environments of different kind. Synchronisation is achieved through reading and writing distributed variables either in a synchronous or an asynchronous way. Remote and local accesses are explicitly specified. Data distribution seems to be the salient aspect of DINO. Though, composite procedures are a particular instance of multiprocedures.

7.2 About fragmentation

The clarification of the fragmentation mechanism is undoubtedly a contribution of this specification. The role played by the *virtual* declaration in the concrete syntax of POLYGOTH, which basically amounts to the *Virtual* component of the *CLASS* schema

[13]As it is remarked in the next section, most POLYGOTH examples use objects with identical fragments. Therefore, a special array-like syntactic sugar is provided in the concrete syntax of POLYGOTH.

(section 4), was thought to be an innocuous global view of the object fragmented state. However, the global view is not at all innocuous. Allowing such a view calls for the use of global variables, the existence of global assignment and remote partial assignment, and the need for a global storage coherence. The coherence requirement is specified in the $Config_{Local}$ schema, where it is written that both local and global *views* of the storage should be identical. A careful reading of the two assignment schemas should reveal the annoyance (and the implied mechanisms and costs) of such a feature. Furthermore, remote modifications are expressible through the use of multiprocedures. A powerful reason is needed in order to justify a global view, other than that given by multiprocedures, of the fragmented state. Otherwise, why should the programmer bother specifying fragmentation for?

The fragmented object abstraction introduces two layers of parallelism and communications: internal, that is to say within the object fragments, and external, that is to say amongst objects. As the specification has shown, these layers are closely intertwined. Indeed, a synchronous *new* operation makes internal parallelism the sole source of external parallelism. The later is achieved through parallel external calls.

Fragments may be thought of as *virtual processors* upon which data is distributed. It is not surprising to find that many POLYGOTH programs make use of very regular data structures; they bear easy to exploit parallelism. Parallelism driven by data distribution is not a new idea. Even though they are used in a different approach, several distribution notations for regular data structures can be found in languages like DINO [Rosing et al., 1988] or systems like the ones reported in [Callahan and Kennedy, 1988]. Fragments accomodate irregular structures as well as data persistence. Further research is needed in order to establish if permanent data should also be distributed in a permanent way. Other alternatives include a *dynamic* fragmentation of object states that would be method-bound. Still another alternative under investigation turns fragmentation the other way around. Collections of objects (possibly from different classes) would be tied through the nesting in each component of a multiprocedure block. This macro-object would provide group methods (those multiprocedures that integrate the various objects into a group) as well as individual methods through a unique interface.

7.3 About Z

We were able to write an informal presentation of the POLYGOTH kernel only *after* the formal specification was completed. This simple fact shows by itself the insight that can be obtained through a specification effort. Readers may feel that schemas and related mathematical notations are only paraphrasing the text, however it happened the other way around: all of the text was written after the specification paragraphs. Readers may also think that Z renders descriptions harder and longer. Indeed, Z enforces that everything is said. Specifications are therefore verbose. Executability concerns can also be held responsible for this: our specification is very close to an executable form. Nevertheless, the main source of complexity in our specification is grounded elsewhere and is more profoundly rooted in the POLYGOTH language than in the right choice of a mathematical description technique. A simpler description would have been derived had we abstracted POLYGOTH's distribution aspects, but how could we drop such an essential characteristic? Our specification is therefore placed at an abstraction level where fragmentation and distribution issues can be described. Our configurations are

complex, but we think that exhibiting fragmentation justifies the additional complexity. The main advantage of using **Z** lies in the subsequent developments of this work. Changes in the language can be analyzed by formal transformations of its current specification. Implementations can be formally derived, different strategies can be identified and choices can be well motivated and documented. Lastly, a well defined notation is better suited as a knowledge transmission vehicle than a suigeneris one. **Z** schemas were precious in obtaining a highly hierarchical description of the semantics. The structure of the specification mirrors the three different conceptual levels present in the language: internal working of fragments (section 5.2), object morphology and internal actions (section 5.3) and system behaviour of a collection of objects (section 5.4). Explaining text should not be neglected when a specification is written. A clear explanation is worth a hundred formulas, even though we believe that explanations get clearer when the subjects being explained have been formally described. We had to restructure several schemas because they were not easy to explain, and in fact, they were awkwardly defined. When the obligation proof was established (section 5.3.1), two errors (omissions) were discovered. We are quite sure that other inadequacies would be revealed by the undone proofs. This would only confirm our weakness in dealing with complex systems and strengthen our belief that formal methods are needed even if their use seems to complicate things. Tools are needed. The type checker was of great help. A theorem prover would be highly appreciated.

Other specification techniques have been used in the definition of similar languages. In [Breu and Zucca, 1989], an algebraic specification, written in ASL [Wirsing, 1986], of a concurrent object-oriented language with inheritance is proposed. The description given in that paper seems cumbersome and difficult to read. Parallelism is dealt with in a less elegant way than ours: a rule is provided for each pair of external labelled transitions. However, the inheritance specification is nicely exposed. As it is suggested in the referred paper, the specification effort could lead to an object-oriented specification language. Indeed, a canonical representation of objects and classes in **Z** could be found and its semi-mechanical translation into POLYGOTH units could follow. Similar aims are pursued in other works, notably in [Hall, 1989].

As it was indicated in the introduction, the technique employed to define POLYGOTH's semantics is very similar to the one reported in [America et al., 1986]. Besides obvious differences stemming from the language being described, it can be noticed that our work provides a truly distributed semantics since there is no global state in our (global) configurations: local configurations are specified instead. This locality is suggested as a modification needed in order to introduce parallelism rules in the referred report. A more important difference lies in the use of **Z** to specify states, configurations and transitions, leading the way towards exploring the possibility of automatic implementation of parallel languages by means of an interpreter.

8 Conclusions

The main concepts and notations introduced in the POLYGOTH programming language have been formally defined. The multiprocedure generalizes the procedure concept in a parallel framework and proves to be a powerful communication abstraction. Multiprocedures bear distribution in their very nature. They are the procedural abstraction

of a parallel program. The coordinated call –a natural extension of the procedure call for multiprocedures– brings full composability to parallel programs in a procedural way. Besides, this new calling mechanism enables the expression of a parallel recursion that has not yet been explored, but seems to be a promising and elegant alternative approach to writing parallel algorithms. The introduction of multiprocedures in an object-oriented framework led to the definition of a fragmentation mechanism that addresses both parallelism and (logical) distribution in a unified manner. A fragmented object is a distributed entity whose components are tied by the multiprocedures provided as its methods. A small, but representative, subset of POLYGOTH has been considered in order to exhibit the operational behaviour of its most interesting constructs. A transition system "à la Plotkin" was defined. Individual transitions as well as configurations were completely specified with the **Z** language. This work was originally motivated by the need for a formal POLYGOTH definition. The definition was necessary both for presentation purposes and for a language review. A simplified semantics, not reported here, was written without using any particular notation. It became clear that if any (re)use was expected from this undertaking, a more workable, hence formal, notation had to be used. The state-based approach of **Z** seemed to fit in very well. As a result, this paper can be read in three different (and complementary) ways: as the operational semantics of a class-based language that is a test bed for the multiprocedure concept and for a fragmentation mechanism, as a large **Z** specification exercise and as a working material to assess the merits of using a well defined notation to specify a computing system.

The development of a denotational semantics for POLYGOTH is surely challenging since fragmented objects would be assigned a (mathematical) meaning. However, we will delay such an attempt until the reviewing of the language is completed. Then, the work reported in [America et al., 1989] should provide a good starting point for the construction of our semantical domains.

9 Acknowledgements

I wish to express my gratitude to Valérie Issarny for the interest she took in the development of this work and for the wisdom of her advices. I am also in debt with Daniel Le Métayer for his careful reading of the formal specification and his accurate remarks. Jean Pierre Banâtre has had the kindness to read several times this somehow tedious paper. He is also responsible of my acquaintance with multiprocedures. I thank him for that and for his patient insistence on my writing the final period. Lastly, I acknowledge Philippe Darondeau for the time he generously spent reading this paper and for the interesting critics he formulated. I shall consider his suggestions in subsequent developments of this work.

References

[America, 1987] America, P. (1987). POOL-T: a parallel object-oriented language. In Yonezawa, A. and Tokoro, M., editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press.

[America et al., 1986] America, P., de Bakker, J., Kok, J., and Rutten, J. (1986). Operational semantics of a parallel object-oriented language. In 13^{th} *Symposium on Principles of Programming Languages*, pages 194–208, St. Petersburg, Florida.

[America et al., 1989] America, P., de Bakker, J., Kok, J., and Rutten, J. (1989). Denotational semantics of a parallel object-oriented language. *Information and Computation*, 83(2):194–208.

[Andrews and Schneider, 1983] Andrews, G. and Schneider, F. (1983). Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43.

[Athas and Seitz, 1988] Athas, W. and Seitz, C. (1988). Multicomputers: Message-passing concurrent computers. *Computer*, 21(8):9–24.

[Bal et al., 1989] Bal, H., Steiner, J., and Tanenbaum, A. (1989). Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322.

[Banâtre, 1980] Banâtre, J.P. (1980). Contribution à l'étude de méthodes et d'outils de construction de programmes parallèles et fiables. Thèse d'Etat, Université de Rennes-I.

[Banâtre et al., 1988] Banâtre, J.P., Banâtre, M., and Muller, G. (1988). Main aspects of the GOTHIC distributed system. In *European Teleinformatics Conference on Research into Networks and Distributed Applications*, pages 747–760, Vienna, Austria.

[Banâtre et al., 1986] Banâtre, J.P., Banâtre, M., and Ployette, F. (1986). The concept of multi-function: A general structuring tool for distributed operating systems. In 6^{th} *Conference on Distributed Computing Systems*, pages 478–485, Cambridge, Mass.

[Banâtre and Benveniste, 1989] Banâtre, J.P. and Benveniste, M. (1989). Multiprocedures: Generalized procedures for concurrent programming. In 3^{rd} *Workshop on Large Grain Parallelism*, Software Engineering Institute, Carnegie Mellon University.

[Black et al., 1986] Black, A., Hutchinson, N., Jul, E., and Levy, H. (1986). Object structure in the EMERALD system. *ACM SIGPLAN Notices*, 21(11):78–86. ACM Conference on Object-Oriented Programming Systems, Languages and Applications.

[Breu and Zucca, 1989] Breu, R. and Zucca, E. (1989). An algebraic compositional semantics of an object oriented notation with concurrency. In *Proceedings of the* 9^{th} *Conference on Foundation of Software Technology and Theoretical Computer Science*, number 405, pages 131–142, Bangadore, India. Springer-Verlag.

[Callahan and Kennedy, 1988] Callahan, D. and Kennedy, K. (1988). Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, (2):151–169.

[Cook, 1978] Cook, S. (1978). Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90.

[Cooper, 1984] Cooper, E. (1984). Replicated procedure call. In 3^{rd} *ACM Symposium on Principles of Distributed Computing*, pages 220–232.

[Francez et al., 1986] Francez, N., Hailpern, B., and Taubenfeld, G. (1986). Script: A communication abstraction mechanism and its verification. *Science of Computer Programming*, (6):35–88.

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. (1983). *Smalltalk-80: the language and its implementation.* Addison-Wesley Publishing Company, Inc., Readings, Mass.

[Hall, 1989] Hall, J. (1989). Using **Z** as a specification calculus for object-oriented systems. To appear in the VDM90 Symposium proceedings.

[Hennessy and Plotkin, 1979] Hennessy, M. and Plotkin, G. (1979). Full abstraction for a simple parallel programming language. In 8^{th} *Conference on Mathematical Foundations of Computer Science*, number 74, pages 108–120. Springer-Verlag.

[Herman and Trilling, 1990] Herman, D. and Trilling, L. (1990). Programming a parallel sort using distributed procedure calls. Rapport de recherche, IRISA. In preparation.

[Hoare, 1974] Hoare, C.A.R. (1974). Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557.

[Issarny, 1989] Issarny, V. (1989). Le traitement d'exceptions: Aspects théoriques et pratiques. Rapport de recherche 1118, INRIA.

[Jones, 1986] Jones, C. (1986). *Systematic Software Development using VDM.* Prentice-Hall International.

[Jordan et al., 1989] Jordan, H., Benten, M., Alaghband, G., and Jakob, R. (1989). The Force: A highly portable parallel programming language. In *International Conference on Parallel Processing*, pages II–112–II–117, Penn State University, Pennsylvania.

[Kafura and Lee, 1989] Kafura, D.G.. and Lee, K.H.. (1989). Inheritance in actor based concurrent object-oriented languages. *The Computer Journal*, 32(4):297–303.

[Lecler, 1989] Lecler, P. (1989). Une approche de la programmation des systèmes distribués fondée sur la fragmentation des données et des calculs, et sa mise en œuvre dans le système GOTHIC. Thèse, Université de Rennes-I.

[Lieberman, 1987] Lieberman, H. (1987). Concurrent object-oriented programming in ACT 1. In Yonezawa, A. and Tokoro, M., editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press.

[Nierstrasz, 1987] Nierstrasz, O. (1987). Active objects in Hybrid. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 243–253, Orlando, Florida.

[Plotkin, 1981] Plotkin, G. (1981). A structural approach to operational semantics. Department of computer science research report DAIMI FN-19, Aarhus University, Aarhus C, Denmark.

[Ployette et al., 1987] Ployette, F., Le Certen, P., and Lecler, P. (1987). POLYGOTH: le langage de GOTHIC. LSP project internal research report, IRISA.

[Rosing et al., 1988] Rosing, M., Schnabel, R., and Weaver, R. (1988). Dino: Summary and examples. In *ACM 3rd Conference on Hypercubes Concurrent Computers and Applications*, pages 472–481.

[Schaffert et al., 1986] Schaffert, G., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. (1986). An introduction to Trellis/Owl. *ACM SIGPLAN Notices*, 21(11):9–16. ACM Conference on Object-Oriented Programming Systems, Languages and Applications.

[Spivey, 1988] Spivey, J. (1988). The *f*UZZ manual. Computing Science Consultancy, Garsington, Oxford.

[Spivey, 1989] Spivey, J. (1989). *The Z Notation: A Reference Manual.* Prentice-Hall International.

[Trilling and Quiniou, 1988] Trilling, L. and Quiniou, R. (1988). Collective predicates expressing control of OR-parallelism in PROLOG. In O'Shea, T. and Sgurev, V., editors, *Artificial Intelligence III: Methodology, Systems and Applications*, pages 159–167. North-Holland.

[Wirsing, 1986] Wirsing, M. (1986). Structured algebraic specifications. *Theoretical Computer Science*, (43):123–250.

A The complete Z formal specification

A.1 Auxiliary definitions

$\mathrm{stack}\, X == \mathrm{seq}\, X$

$\mathrm{stack}_1\, X == \mathrm{seq}_1\, X,$

$[X]$

$$
\begin{array}{l}
\mathrm{empty} : \mathrm{stack}\, X \\
\mathrm{push} : (\ \mathrm{stack}\, X \times X\ ) \rightarrow \mathrm{stack}\, X \\
\mathrm{pop} : \mathrm{stack}_1\, X \rightarrow \mathrm{stack}\, X \\
\mathrm{top} : \mathrm{stack}_1\, X \rightarrow X \\
\mathrm{size} : \mathrm{stack}\, X \rightarrow \mathbb{N} \\
\hline
\forall s : \mathrm{stack}\, X;\ t : \mathrm{stack}_1\, X;\ x : X \bullet \\
\quad \mathrm{empty} = \langle\rangle \wedge \\
\quad \mathrm{push}(s, x) = \langle x \rangle \frown s \wedge \\
\quad \mathrm{pop}(t) = (\ \lambda\, n : 1 \mathinner{..} \#t - 1 \bullet t(\ n + 1\ )\ ) \wedge \\
\quad \mathrm{top}(t) = t(\ 1\ ) \wedge \\
\quad \mathrm{size}(s) = \#s
\end{array}
$$

$[X, Y]$

$$
\begin{array}{l}
\_ \bowtie \_ : (\ X \nrightarrow Y\ ) \times (\ X \nrightarrow Y\ ) \rightarrow (\ X \nrightarrow Y\ ) \\
\hline
\forall f, g : X \nrightarrow Y \bullet \\
\quad f \bowtie g = \mathrm{dom}\, f \lhd (f \oplus g)
\end{array}
$$

LawsOfUpdate$[X, Y]$

$$
\begin{array}{l}
f, g, h : X \nrightarrow Y \\
S : \mathbb{P}\, X \\
T : \mathbb{P}\, Y \\
\hline
f \bowtie f = f \\
\varnothing \bowtie f = \varnothing \\
f \bowtie \varnothing = f \\
\mathrm{dom}\, f \cap \mathrm{dom}\, g = \varnothing \Rightarrow f \bowtie g = f \\
\forall x : X \bullet \\
\quad (x \in \mathrm{dom}\, f \cap \mathrm{dom}\, g \Rightarrow (f \bowtie g)(\ x\ ) = g(\ x\ )) \wedge \\
\quad (x \in (\mathrm{dom}\, f) \setminus (\mathrm{dom}\, g) \Rightarrow (f \bowtie g)(\ x\ ) = f(\ x\ )) \\
\mathrm{dom}(f \bowtie g) = \mathrm{dom}\, f \\
\mathrm{ran}(f \bowtie g) = (\mathrm{ran}(\mathrm{dom}\, f \lhd g)) \cup (\mathrm{ran}(\mathrm{dom}\, g \lhd f)) \\
S \lhd (f \bowtie g) = (S \lhd f) \bowtie (S \lhd g) \\
f \bowtie (g \oplus h) = (f \bowtie g) \oplus (f \bowtie h) \\
(f \bowtie g = f \oplus g) \Leftrightarrow (\mathrm{dom}\, g \subseteq \mathrm{dom}\, f)
\end{array}
$$

$$
\begin{array}{|l}
\hline
[X, Y] \\
\pi_1 : X \times Y \rightarrow X \\
\pi_2 : X \times Y \rightarrow Y \\
\hline
\forall x : X;\ y : Y \bullet \\
\quad \pi_1(\ x, y\ ) = x \land \\
\quad \pi_2(\ x, y\ ) = y \\
\forall p : X \times Y \bullet \\
\quad \pi_1(\ p\ ) \mapsto \pi_2(\ p\ ) = p \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
[X] \\
\_ \sqcup \_ : \mathbb{P}\, X \times \mathbb{P}\, X \rightarrow \mathbb{P}\, X \\
\hline
\forall W, S, T : \mathbb{P}\, X \bullet \\
\quad S \sqcup T = W \Leftrightarrow \\
\qquad S \cup T = W \land S \cap T = \varnothing \\
\hline
\end{array}
$$

A.2 Syntax

$$[Var_{Glob}, Var_{Part}, Var_{Priv}, Var_{Loc}]$$

$$[Name_{Method}, Name_{Class}]$$

$$NObj == \mathrm{seq}\, \mathbb{N}$$

$$BodyNum_{Meth} == \mathbb{N}_1$$
$$BodyNum_{Class} == \mathbb{N}_1$$

$$CallId == NObj \times \mathbb{F}_1\, BodyNum_{Class}$$

$$
\begin{aligned}
OBJ ::= {} & nil \mid tt \mid ff \\
\mid {} & int\ \langle\!\langle \mathbb{Z} \rangle\!\rangle \\
\mid {} & obj\ \langle\!\langle NObj \rangle\!\rangle \\
\mid {} & seqobj\ \langle\!\langle\ \mathrm{seq}_1\, OBJ\ \rangle\!\rangle \\
Exp ::= {} & svar \langle\!\langle\ Var_{Glob}\ \rangle\!\rangle \mid fvar \langle\!\langle\ Var_{Part}\ \rangle\!\rangle \\
\mid {} & pvar \langle\!\langle\ Var_{Priv}\ \rangle\!\rangle \mid lvar \langle\!\langle\ Var_{Loc}\ \rangle\!\rangle \\
\mid {} & call \langle\!\langle\ Name_{Method} \times \mathrm{seq}\, Exp\ \rangle\!\rangle \\
\mid {} & cocall \langle\!\langle\ Name_{Method} \times \mathrm{seq}(\ Var_{Loc} \times Exp)\ \rangle\!\rangle \\
\mid {} & meth \langle\!\langle\ Exp \times Name_{Method} \times \mathrm{seq}\, Exp\ \rangle\!\rangle \\
\mid {} & cometh \langle\!\langle\ Exp \times Name_{Method} \times \mathrm{seq}(\ Var_{Loc} \times Exp)\ \rangle\!\rangle \\
\mid {} & new \langle\!\langle\ Name_{Class} \times \mathrm{seq}\, Exp\ \rangle\!\rangle \\
\mid {} & conew \langle\!\langle\ Name_{Class} \times \mathrm{seq}(\ Var_{Priv} \times Exp)\ \rangle\!\rangle \\
\mid {} & name \langle\!\langle\ OBJ\ \rangle\!\rangle \\
\mid {} & self \\
\mid {} & wait \langle\!\langle\ NObj \times Name_{Method}\ \rangle\!\rangle
\end{aligned}
$$

$$
\begin{array}{rl}
Stmt ::= & fassign \langle\langle Var_{Part} \times Exp \rangle\rangle \\
| & passign \langle\langle Var_{Priv} \times Exp \rangle\rangle \\
| & lassign \langle\langle Var_{Loc} \times Exp \rangle\rangle \\
| & massign \langle\langle Var_{Glob} \times Exp \rangle\rangle \\
| & exp \langle\langle Exp \rangle\rangle \\
| & seqcomp \langle\langle Stmt \times Stmt \rangle\rangle \\
| & if \langle\langle Exp \times Stmt \times Stmt \rangle\rangle \\
| & do \langle\langle Exp \times Stmt \rangle\rangle \\
| & send \langle\langle Stmt \times CallId \rangle\rangle
\end{array}
$$

BLOCK[VAR]

$Var : \mathbb{F}\ VAR$
$s : Stmt$

Fragment

$BLOCK[Var_{Part}]$
$Priv : \mathbb{F}\ Var_{Priv}$

Block

$BLOCK[Var_{Loc}]$

METHOD

$Param : \mathrm{iseq}\ Var_{Loc}$
$Alloc : BodyNum_{Class} \twoheadrightarrow BodyNum_{Meth}$
$Body : \mathrm{seq}_1\ Block$

$\mathrm{ran}\ Alloc = \mathrm{dom}\ Body$

$\forall i : 1 \mathinner{..} \#Param \bullet \exists j : 1 \mathinner{..} \#Body \bullet$
$\quad Param(\ i\ ) \in (Body(\ j\ )).Var$

$init : Name_{Method}$

CLASS

$$
\begin{array}{l}
Param : \text{iseq } Var_{Priv} \\
Virtual : Var_{Glob} \pfun \text{seq}_1\ Var_{Part} \\
Method : Name_{Method} \pfun METHOD \\
Body : \text{seq}_1\ Fragment \\
\hline
\forall i : 1 \mathinner{..} \#Body \bullet \\
\quad \forall fx : Var_{Part} \mid fx \in (Body(\ i\ )).Var \bullet \\
\quad\quad \exists sx : Var_{Glob} \mid sx \in \text{dom } Virtual \bullet fx \in \text{ran}(Virtual(\ sx\ )) \\
\forall sx, sy : Var_{Glob} \mid sx \in \text{dom } Virtual \land sy \in \text{dom } Virtual \bullet \\
\quad \forall i : 1 \mathinner{..} \#(Virtual(\ sx\ )); j : 1 \mathinner{..} \#(Virtual(\ sy\ )) \bullet \\
\quad\quad (Virtual(\ sx\ ))(\ i\ ) = (Virtual(\ sy\ ))(\ j\ ) \Rightarrow (sx = sy \land i = j) \\
\forall sx : Var_{Glob} \mid sx \in \text{dom } Virtual \bullet \\
\quad \forall i : 1 \mathinner{..} \#(Virtual(\ sx\ )) \bullet \exists j : 1 \mathinner{..} \#Body \bullet \\
\quad\quad (Virtual(\ sx\ ))(\ i\ ) \in (Body(\ j\ )).Var \\
\forall i, j : 1 \mathinner{..} \#Body \mid i \neq j \bullet \\
\quad (Body(\ i\ )).Var \cap (Body(\ j\ )).Var = \varnothing \\
\forall i : 1 \mathinner{..} \#Param \bullet \exists j : 1 \mathinner{..} \#Body \bullet \\
\quad Param(\ i\ ) \in (Body(\ j\ )).Priv
\end{array}
$$

UNIT

$$
\begin{array}{l}
Program : Name_{Class} \pfun CLASS \\
Main : Name_{Class} \\
\hline
Main \in \text{dom } Program \\
(Program(\ Main\ )).Param = \langle\rangle \\
\forall \tau : Name_{Class} \mid \tau \in \text{dom } Program \bullet \\
\quad init \in \text{dom}((Program(\tau)).Method) \land \\
\quad (((Program(\tau)).Method)(\ init\ )).Alloc = \text{dom}((Program(\tau)).Body) \lhd \text{id } \mathbb{N}
\end{array}
$$

A.3 Semantics

$$Store_{Part} == Var_{Part} \pfun OBJ$$

$$Store_{Priv} == Var_{Priv} \pfun OBJ$$

$$Store_{Loc} == Var_{Loc} \pfun OBJ$$

$$Store_{Glob} == Var_{Glob} \pfun (\ Store_{Part}\ )$$

$$Count == \mathbb{N}$$

AR

$$
\begin{array}{l}
Cur : Name_{Method} \\
Env : Store_{Loc}
\end{array}
$$

A.3.1 Fragment level

$State_\partial$

$s : Stmt$
$\delta : Store_{Part}$
$\phi : Store_{Priv}$
$\rho :$ stack(AR)
$\eta : Count$

$\Delta\ State_\partial$

$State_\partial$
$State_\partial'$

$\Xi\ State_\partial$

$State_\partial$
$State_\partial'$

$\delta = \delta' \wedge \phi = \phi' \wedge \rho = \rho' \wedge \eta = \eta'$

$add, sub, div, mod : Name_{Method}$

$IntBinOp$

$\Xi\ State_\partial$
$i, j : \mathbb{Z}$
$Op : \{\ add, sub, div, mod\ \}$

$s = exp(meth(\ name(int(\ i\ )), Op, \langle name(int(\ j\ ))\rangle\ ))$

$Op = add \Rightarrow s' = exp(name(int(\ i + j\ )))$
$Op = sub \Rightarrow s' = exp(name(int(\ i - j\ )))$
$(Op = div \wedge j \neq 0) \Rightarrow s' = exp(name(int(\ i \operatorname{div} j\ )))$
$Op = mod \Rightarrow s' = exp(name(int(\ i \bmod j\ )))$

$equal, less, greater : Name_{Method}$

$IntBinRel$

$\Xi\ State_\partial$
$i, j : \mathbb{Z}$
$Rel : \{\ equal, less, greater\ \}$

$s = exp(meth(\ name(int(\ i\ )), Rel, \langle name(int(\ j\ ))\rangle\ ))$

$(Rel = equal \wedge i = j) \Rightarrow s' = exp(name(tt))$
$(Rel = equal \wedge i \neq j) \Rightarrow s' = exp(name(ff))$
$(Rel = less \wedge i < j) \Rightarrow s' = exp(name(tt))$
$(Rel = less \wedge i \geq j) \Rightarrow s' = exp(name(ff))$
$(Rel = greater \wedge i > j) \Rightarrow s' = exp(name(tt))$
$(Rel = greater \wedge i \leq j) \Rightarrow s' = exp(name(ff))$

$and, or : Name_{Method}$

BoolBinRel

$\Xi State_\partial$
$\beta, \gamma : \{ tt, ff \}$
$Op : \{ and, or \}$

$s = exp(meth(\ name(\beta), Op, \langle name(\gamma) \rangle\ ))$

$(Op = and \land \beta = \gamma = tt) \Rightarrow s' = exp(name(\ tt\ ))$
$(Op = and \land (\beta = ff \lor \gamma = ff)) \Rightarrow s' = exp(name(\ ff\ ))$
$(Op = or \land \beta = \gamma = tt) \Rightarrow s' = exp(name(\ ff\ ))$
$(Op = or \land (\beta = tt \lor \gamma = tt)) \Rightarrow s' = exp(name(\ tt\ ))$

$not : Name_{Method}$

BoolPreRel

$\Xi State_\partial$
$\beta : \{ tt, ff \}$

$s = exp(meth(\ name(\beta), not, \langle\rangle\ ))$

$\beta = tt \Rightarrow s' = exp(name(ff))$
$\beta = ff \Rightarrow s' = exp(name(tt))$

LocPartVarExp

$\Xi State_\partial$
$fx : Var_{Part}$

$fx \in \mathrm{dom}\, \delta$

$s = exp(fvar(\ fx\ ))$
$s' = exp(name(\delta(\ fx\ )))$

PrivVarExp

$\Xi State_\partial$
$x : Var_{Priv}$

$x \in \mathrm{dom}\, \phi$

$s = exp(pvar(\ x\ ))$
$s' = exp(name(\phi(\ x\ )))$

LocVarExp

$\Xi State_\partial$
$u : Var_{Loc}$

$u \in \mathrm{dom}(\mathrm{top}(\ \rho\ )).Env$

$s = exp(lvar(\ u\ ))$
$s' = exp(name(((\mathrm{top}(\ \rho\ )).Env)(\ u\ )))$

ValueDiscard

$\Xi\ State_\partial$
$s_1 : Stmt$
$\gamma : OBJ$

$s = seqcomp(\ exp(name(\ \gamma\ )), s_1\ )$
$s' = s_1$

If

$\Xi\ State_\partial$
$\beta : \{\ tt, ff\ \}$
$s_1, s_2 : Stmt$

$s = if(\ name(\ \beta\ ), s_1, s_2\ )$
$\beta = tt \Rightarrow s' = s_1$
$\beta = ff \Rightarrow s' = s_2$

Do

$\Xi\ State_\partial$
$s_1 : Stmt$
$\gamma : OBJ$

$s = do(\ name(\ \gamma\ ), s_1\ )$
$s' = if(\ name(\ \gamma\ ), seqcomp(\ s_1, s\ ), exp(name(nil))\ )$

LocPartAssign

$\Delta\ State_\partial$
$fx : Var_{Part}$
$\gamma : OBJ$

$fx \in \mathrm{dom}\,\delta$
$s = fassign(\ fx, name(\ \gamma\ )\ )$
$s' = exp(name(\ \gamma\ ))$
$\delta' = \delta \oplus \{\ fx \mapsto \gamma\ \}$
$\phi' = \phi \wedge \rho' = \rho \wedge \eta' = \eta$

PrivAssign

$\Delta\ State_\partial$
$x : Var_{Priv}$
$\gamma : OBJ$

$x \in \mathrm{dom}\,\phi$
$s = passign(\ x, name(\ \gamma\ )\ )$
$s' = exp(name(\ \gamma\ ))$
$\delta' = \delta$
$\phi' = \phi \oplus \{\ x \mapsto \gamma\ \}$
$\rho' = \rho \wedge \eta' = \eta$

LocAssign

$\Delta\ State_\partial$
$u : Var_{Loc}$
$ar : AR$
$\gamma : OBJ$

$u \in \mathrm{dom}(\mathrm{top}(\ \rho\ )).Env$
$s = lassign(\ u, name(\ \gamma\ )\ )$
$ar.Cur = (\mathrm{top}(\ \rho\ )).Cur$
$ar.Env = (\mathrm{top}(\ \rho\ )).Env \oplus \{\ u \mapsto \gamma\ \}$
$s' = exp(name(\ \gamma\ ))$
$\delta' = \delta \wedge \phi' = \phi$
$\rho' = \mathrm{push}(\ \mathrm{pop}(\ \rho\ ), ar\ )$
$\eta' = \eta$

Axioms$_{Basic}$

$\Delta\ State_\partial$

$(\exists_1 i, j : \mathbb{Z} \bullet$
$(\exists_1 Op : \{\ add, sub, div, mod\ \} \bullet IntBinOp) \vee$
$(\exists_1 Rel : \{\ equal, less, greater\ \} \bullet IntBinRel)\ ) \vee$
$(\exists_1 \beta : \{\ tt, ff\ \} \bullet$
$(\exists_1 \gamma : \{\ tt, ff\ \};\ Op : \{\ and, or\ \} \bullet BoolBinRel) \vee$
$BoolPreRel \vee$
$(\exists_1 s_1, s_2 : Stmt \bullet If)\ ) \vee$
$(\exists_1 \gamma : OBJ \bullet$
$(\exists_1 fx : Var_{Part} \bullet LocPartVarExp \vee LocPartAssign) \vee$
$(\exists_1 x : Var_{Priv} \bullet PrivVarExp \vee PrivAssign) \vee$
$(\exists_1 u : Var_{Loc};\ ar : AR \bullet LocVarExp \vee LocAssign) \vee$
$(\exists_1 s_1 : Stmt \bullet Do \vee ValueDiscard)\ )$

A.3.2 Object level

$Config_{Local}$

$\alpha : NObj$
$\psi : Store_{Glob}$
$\sigma : \mathrm{seq}_1\ State_{\vartheta}$
$\tau : Name_{Class}$
$UNIT$

$\tau \in \mathrm{dom}\ Program$

$\mathrm{dom}\,\psi = \mathrm{dom}((Program(\tau)).Virtual)$

$\#\sigma = \#((Program(\tau)).Body)$

$\forall i : 1 \,..\, \#\sigma \bullet$
$\quad \mathrm{dom}((\sigma(i)).\delta) = (((Program(\tau)).Body)(i)).Var \wedge$
$\quad \mathrm{dom}((\sigma(i)).\phi) = (((Program(\tau)).Body)(i)).Priv$

$\forall i : 1 \,..\, \#\sigma \bullet$
$\quad (\ (\sigma(i)).\rho \neq \mathrm{empty}\ ) \wedge$
$\quad (\ \exists_1\ ar : AR \mid ar = \mathrm{top}(\ (\sigma(i)).\rho\ ) \wedge ar.Cur \in \mathrm{dom}(Program(\tau)).Method \bullet$
$\qquad (\ ar.Cur = init\ ) \vee$
$\qquad (\ \exists_1\ M : METHOD \mid M = ((Program(\tau)).Method)(\ ar.Cur\ ) \bullet$
$\qquad\quad (\forall k : BodyNum_{Class} \mid k \in \mathrm{dom}\ M.Alloc \bullet \exists n : 0 \,..\, \mathrm{size}(\ (\sigma(k)).\rho\ ) \bullet$
$\qquad\qquad (\mathrm{top}(\ \mathrm{pop}^n(\ (\sigma(k)).\rho\ )\ )).Cur = ar.Cur) \wedge$
$\qquad\quad i \in \mathrm{dom}\ M.Alloc \wedge$
$\qquad\quad \mathrm{dom}\ ar.Env = ((M.Body)((M.Alloc)(i))).Var\ )\ )$

$\bigcup\{\ sx : Var_{Glob} \mid sx \in \mathrm{dom}\,\psi \bullet \psi(\ sx\ )\ \} = \bigcup\{\ i : 1 \,..\, \#\sigma \bullet (\sigma(i)).\delta\ \}$

$InitConfig_{Local}$

$Config_{Local}$
$n? : \mathbb{N}$

$\alpha = \langle n? \rangle$

$\psi = (\ \lambda\, sx : Var_{Glob} \mid sx \in \mathrm{dom}((Program(Main)).Virtual) \bullet$
$\quad (\ \lambda\, fx : Var_{Part} \mid fx \in \mathrm{ran}(((Program(Main)).Virtual)(sx)) \bullet nil\ )\ )$

$\#\sigma = \#((Program(Main)).Body)$

$\forall i : 1 \,..\, \#\sigma \bullet$
$\quad (\sigma(i)).s = (((Program(Main)).Body)(i)).s \wedge$
$\quad (\sigma(i)).\delta = (\ \lambda\, fx : Var_{Part} \mid fx \in (((Program(Main)).Body)(i)).Var \bullet nil\ ) \wedge$
$\quad (\sigma(i)).\phi = (\ \lambda\, x : Var_{Priv} \mid x \in (((Program(Main)).Body)(i)).Priv \bullet nil\ ) \wedge$
$\quad (\exists\, ar : AR \mid ar.Cur = init \wedge ar.Env = \varnothing \bullet$
$\qquad (\sigma(i)).\rho = \mathrm{push}(\ \mathrm{empty}, ar\ )) \wedge$
$\quad (\sigma(i)).\eta = 0$

$\tau = Main$

Ξ Config$_{Local}$

$Config_{Local}$
$Config_{Local}'$

$\alpha' = \alpha \land \tau = \tau'$

$Program' = Program \land Main' = Main$

SingleMove

$\Xi\ Config_{Local}$
$\Delta\ State_\partial$
$i : BodyNum_{Class}$

$i \in \mathrm{dom}\,\sigma$
$\sigma(\ i\ ) = \theta State_\partial$

LocalTransition

SingleMove

$Axioms_{Basic}$

$\sigma' = \sigma \oplus \{\ i \mapsto \theta State_\partial'\ \}$
$\psi' = \psi \oplus (\ \lambda\, sx : Var_{Glob} \mid sx \in \mathrm{dom}\,\psi \bullet \psi(\ sx\ ) \bowtie \delta'\ )$

Self

SingleMove

$s = exp(\ self\ )$

$s' = exp(name(obj(\ \alpha\ )))$
$\Xi\ State_\partial$

$\sigma' = \sigma \oplus \{\ i \mapsto \theta State_\partial'\ \}$
$\psi' = \psi$

GlobVarExp

SingleMove
$sx : Var_{Glob}$
$\vec{\gamma} : \mathrm{seq}_1\ OBJ$

$sx \in \mathrm{dom}\,\psi$
$\#\vec{\gamma} = \#(((Program(\tau)).Virtual)(sx))$
$\forall j : 1\,..\,\#\vec{\gamma} \bullet$
$\quad \vec{\gamma}(\ j\ ) = (\psi(\ sx\ ))(\ (((Program(\tau)).Virtual)(sx))(\ j\ )\ )$

$s = exp(svar(\ sx\ ))$

$s' = exp(name(seqobj(\ \vec{\gamma}\ )))$
$\Xi\ State_\partial$

$\sigma' = \sigma \oplus \{\ i \mapsto \theta State_\partial'\ \}$
$\psi' = \psi$

GlobVarAssign

$$
\begin{array}{l}
\textit{SingleMove} \\
sx : \textit{Var}_{\textit{Glob}} \\
\vec{\gamma} : \mathrm{seq}_1\ \textit{OBJ} \\
\hline
sx \in \mathrm{dom}\,\psi \\
\#\vec{\gamma} = \#(((\textit{Program}(\tau)).\textit{Virtual})(sx)) \\
s = \textit{massign}(\ sx\ , \textit{name}(\textit{seqobj}(\ \vec{\gamma}\ ))\ ) \\
s' = \textit{exp}(\textit{name}(\textit{seqobj}(\ \vec{\gamma}\ ))) \\
\delta' = \delta \bowtie \psi'(\ sx\ ) \\
\phi' = \phi \wedge \rho' = \rho \wedge \eta' = \eta \\
\sigma'(\ i\ ) = \theta\textit{State}_\partial{}' \\
\forall j : 1\,..\,\#\sigma \mid j \neq i \bullet \\
\quad (\sigma'(\ j\ )).s = (\sigma(\ j\ )).s \wedge \\
\quad (\sigma'(\ j\ )).\delta = (\sigma(\ j\ )).\delta \bowtie \psi'(\ sx\ ) \wedge \\
\quad (\sigma'(\ j\ )).\phi = (\sigma(\ j\ )).\phi \wedge \\
\quad (\sigma'(\ j\ )).\rho = (\sigma(\ j\ )).\rho \wedge \\
\quad (\sigma'(\ j\ )).\eta = (\sigma(\ j\ )).\eta \\
\psi' = \psi \oplus \\
\quad \{\ sx \mapsto \{\ j : 1\,..\,\#\vec{\gamma} \bullet (((\textit{Program}(\tau)).\textit{Virtual})(sx))(\ j\ ) \mapsto \vec{\gamma}(\ j\ )\ \}\ \}
\end{array}
$$

DoubleMove

$$
\begin{array}{l}
\Xi\ \textit{Config}_{\textit{Local}} \\
\Delta\ \textit{State}_\partial \\
\Delta\ \textit{State}_{\partial 1} \\
i, j : \textit{BodyNum}_{\textit{Class}} \\
\hline
\{\ i, j\ \} \subseteq \mathrm{dom}\,\sigma \\
\sigma(\ i\ ) = \theta\textit{State}_\partial \\
\sigma(\ j\ ) = \theta\textit{State}_{\partial 1} \\
\sigma' = \sigma \oplus \{\ i \mapsto \theta\textit{State}_\partial{}', j \mapsto \theta\textit{State}_{\partial 1}'\ \}
\end{array}
$$

PartVarExp

$$
\begin{array}{l}
\textit{DoubleMove} \\
fx : \textit{Var}_{\textit{Part}} \\
\hline
fx \in \mathrm{dom}\,\delta_1 \\
s = \textit{exp}(\textit{fvar}(\ fx\ )) \\
s' = \textit{exp}(\textit{name}(\ \delta_1(\ fx\ )\ )) \\
\Xi\ \textit{State}_\partial \\
s_1' = s_1 \\
\Xi\ \textit{State}_{\partial 1} \\
\psi' = \psi
\end{array}
$$

PartAssign

$$
\begin{array}{l}
DoubleMove \\
fx : Var_{Part} \\
\gamma : OBJ \\
\hline
fx \in \mathrm{dom}\, \delta_1 \\
s = fassign(\ fx, name(\ \gamma\ )\ ) \\
s' = exp(name(\ \gamma\ )) \\
\Xi\ State_{\partial} \\
s'_1 = s_1 \\
\delta'_1 = \delta_1 \oplus \{\ fx \mapsto \gamma\ \} \\
\phi'_1 = \phi_1 \wedge \rho'_1 = \rho_1 \wedge \eta'_1 = \eta_1 \\
\psi' = \psi \bowtie (\ \lambda\, sx : Var_{Glob} \bullet \psi(\ sx\ ) \bowtie \delta'_1\ )
\end{array}
$$

MultiMove

$$
\begin{array}{l}
\Xi\ Config_{Local} \\
\hline
\forall j : 1\,..\,\#\sigma \bullet \\
\quad (\sigma'(\ j\ )).\delta = (\sigma(\ j\ )).\delta \wedge \\
\quad (\sigma'(\ j\ )).\phi = (\sigma(\ j\ )).\phi \wedge (\sigma'(\ j\ )).\eta = (\sigma(\ j\ )).\eta \\
\psi' = \psi
\end{array}
$$

InCallPrecond

$$
\begin{array}{l}
MultiMove \\
mf : Name_{Method} \\
Actual : \mathrm{seq}\, Exp \\
M : METHOD \\
i : BodyNum_{Class} \\
\hline
i \in \mathrm{dom}\, \sigma \\
(\sigma(\ i\ )).s = exp(call(\ mf, Actual\ )) \\
mf \neq init \wedge mf \in \mathrm{dom}((Program(\tau)).Method) \\
M = ((Program(\tau)).Method)(\ mf\ ) \\
\#Actual = \#(M.Param) \\
\forall k : 1\,..\,\#Actual \bullet \exists \gamma : OBJ \bullet Actual(\ k\ ) = name(\ \gamma\ )
\end{array}
$$

InCallControlEffect

InCallPrecond

$$\forall j : 1 \,..\, \#\sigma \mid j \neq i \wedge j \in \mathrm{dom}(M.Alloc) \bullet$$
$$(\sigma'(\ j\ )).s =$$
$$seqcomp(\ send(\ ((M.Body)(\ (M.Alloc)(j)\ )).s, (\alpha, \{i\})\ ), (\sigma(\ j\ )).s\ )$$

$$\forall j : 1 \,..\, \#\sigma \mid j \neq i \wedge j \notin \mathrm{dom}(M.Alloc) \bullet$$
$$(\sigma'(\ j\ )).s = (\sigma(\ j\ )).s$$

$$i \in \mathrm{dom}(M.Alloc) \Rightarrow$$
$$(\sigma'(\ i\ )).s = send(\ (((M.Body)(\ (M.Alloc)(i)\ )).s), (\alpha, \{i\})\ )$$

$$i \notin \mathrm{dom}(M.Alloc) \Rightarrow$$
$$(\sigma'(\ i\ )).s = exp(wait(\ \alpha, mf\ ))$$

InCallStackEffect

InCallPrecond

$$\forall j : 1 \,..\, \#\sigma \mid j \in \mathrm{dom}(M.Alloc) \bullet \exists_1\ ar : AR \mid$$
$$ar.Cur = mf \wedge$$
$$ar.Env =$$
$$(\ \lambda\, u : Var_{Loc} \mid u \in (((M.Body)(\ (M.Alloc)(j)\ )).Var) \bullet nil\ ) \bowtie$$
$$\{\ \gamma : OBJ;\ k : 1 \,..\, \#Actual \mid Actual(\ k\ ) = name(\ \gamma\ ) \bullet$$
$$(M.Param)(\ k\ ) \mapsto \gamma\ \} \bullet$$
$$(\sigma'(\ j\ )).\rho = \mathrm{push}(\ (\sigma(\ j\ )).\rho, ar\ )$$

$$\forall j : 1 \,..\, \#\sigma \mid j \notin \mathrm{dom}(M.Alloc) \bullet$$
$$(\sigma'(\ j\ )).\rho = (\sigma(\ j\ )).\rho$$

$$InCall \mathrel{\widehat{=}} InCallControlEffect \wedge InCallStackEffect$$

InCoCallPrecond

$$
\begin{array}{l}
MultiMove \\
ready : \mathbb{F}_1\, BodyNum_{Class} \\
mf : Name_{Method} \\
Actual : \mathrm{seq}(\ Var_{Loc} \times Exp\ ) \\
\partial actual : \mathrm{seq}_1(\ \mathrm{seq}(\ Var_{Loc} \times Exp\ )\ ) \\
M : METHOD \\
\hline
mf \neq init \wedge mf \in \mathrm{dom}((Program(\tau)).Method) \\
M = ((Program(\tau)).Method)(\ mf\ ) \\
\exists\, mg : Name_{Method} \mid mg \in \mathrm{dom}((Program(\tau)).Method) \bullet \\
\quad ready = \mathrm{dom}((((Program(\tau)).Method)(\ mg\ )).Alloc) \wedge \\
\quad (\forall\, i : BodyNum_{Class} \mid i \in ready \bullet (\mathrm{top}(\ (\sigma(\ i\ )).\rho\ )).Cur = mg) \\
\forall\, j : 1\,..\,\#\sigma \mid j \in ready \bullet \\
\quad (\sigma(\ j\ )).s = exp(cocall(\ mf, \partial actual(\ j\ )\ )) \\
\#\partial actual = \#ready \\
\forall\, j, j' : 1\,..\,\#\partial actual \bullet \\
\quad \forall\, k : 1\,..\,\#(\partial actual(\ j\ ));\ k' : 1\,..\,\#(\partial actual(\ j'\ )) \bullet \\
\qquad \pi_1(\ (\partial actual(\ j\ ))(k)\ ) = \pi_1(\ (\partial actual(\ j'\ ))(k')\ ) \Rightarrow (\ j = j' \wedge k = k'\ ) \\
Actual = \frown / \partial actual \\
\#Actual = \#(M.Param) \\
\forall\, j : 1\,..\,\#Actual \bullet \exists_1\, k : 1\,..\,\#(M.Param) \bullet \exists\, \gamma : OBJ \bullet \\
\quad Actual(\ j\ ) = (M.Param)(\ k\ ) \mapsto name(\ \gamma\ )
\end{array}
$$

InCoCallControlEffect

$$
\begin{array}{l}
InCoCallPrecond \\
\hline
\forall\, j : 1\,..\,\#\sigma \bullet \\
\quad (\ j \in (\mathrm{dom}(M.Alloc)) \cap ready \Rightarrow \\
\qquad (\sigma'(\ j\ )).s = \\
\qquad\quad send(\ ((M.Body)(\ (M.Alloc)(j)\ )).s, (\alpha, ready)\ )\ ) \wedge \\
\quad (\ j \in ready \setminus \mathrm{dom}(M.Alloc) \Rightarrow \\
\qquad (\sigma'(\ j\ )).s = exp(wait(\ \alpha, mf\ ))\ ) \wedge \\
\quad (\ j \in (\mathrm{dom}(M.Alloc)) \setminus ready \Rightarrow \\
\qquad (\sigma'(\ j\ )).s = \\
\qquad\quad seqcomp(\ send(\ ((M.Body)(\ (M.Alloc)(j)\ )).s, (\alpha, ready)\ ), (\sigma(\ j\ )).s\ )\ ) \wedge \\
\quad (\ j \notin (\mathrm{dom}(M.Alloc)) \cup ready \Rightarrow \\
\qquad (\sigma'(\ j\ )).s = (\sigma(\ j\ )).s\ )
\end{array}
$$

InCoCallStackEffect

InCoCallPrecond

$\forall j : 1 \,..\, \#\sigma \mid j \in \mathrm{dom}(M.Alloc) \bullet \exists_1 ar : AR \mid$
$ar.Cur = mf \wedge$
$ar.Env =$
$(\lambda u : Var_{Loc} \mid u \in (((M.Body)(\ (M.Alloc)(j)\ )).Var) \bullet nil\ ) \bowtie$
$\{\ \gamma : OBJ;\ k : 1 \,..\, \#Actual \mid \pi_2(Actual(\ k\ )) = name(\ \gamma\ ) \bullet$
$\pi_1(Actual(\ k\ )) \mapsto \gamma\ \} \bullet$
$(\sigma'(\ j\ )).\rho = \mathrm{push}(\ (\sigma(\ j\ )).\rho, ar\ )$

$\forall j : 1 \,..\, \#\sigma \mid j \notin \mathrm{dom}(M.Alloc) \bullet$
$(\sigma'(\ j\ )).\rho = (\sigma(\ j\ )).\rho$

$InCoCall \mathrel{\widehat=} InCoCallControlEffect \wedge InCoCallStackEffect$

InCoRetPrecond

MultiMove
$ready, waiting : \mathbb{F}_1\ BodyNum_{Class}$
$\vec{\gamma} : \mathrm{seq}_1\ OBJ$
$mf : Name_{Method}$

$mf \in \mathrm{dom}((Program(\tau)).Method)$
$ready = \mathrm{dom}(((Program(\tau)).Method)(\ mf\ )).Alloc$
$\#\vec{\gamma} = \#ready$

$\forall j : 1 \,..\, \#\sigma \mid j \in ready \bullet$
$(\mathrm{top}(\ (\sigma(\ j\ )).\rho\ )).Cur = mf \wedge$
$(\sigma(\ j\ )).s = send(\ exp(name(\ \vec{\gamma}(\ j\ )\ )), (\alpha, waiting)\ )$

$\forall j : 1 \,..\, \#\sigma \mid j \in waiting \setminus ready \bullet$
$(\sigma(\ j\ )).s = exp(wait(\ \alpha, mf\ ))$

InCoRetControlEffect

InCoRetPrecond

$\forall j : 1 \,..\, \#\sigma \bullet$
$(\ j \in waiting \setminus ready \Rightarrow$
$(\sigma'(\ j\ )).s = exp(name(seqobj(\ \vec{\gamma}\ )))\ ) \wedge$
$(\ j \in ready \setminus waiting \Rightarrow$
$(\sigma'(\ j\ )).s = exp(name(\vec{\gamma}(\ j\ ))\ ) \wedge$
$(\ j \notin waiting \cup ready \Rightarrow$
$(\sigma'(\ j\ )).s = (\sigma(\ j\ )).s\ )$

InCoRetStackEffect

InCoRetPrecond

$\forall j : 1 \,..\, \#\sigma \bullet$
$(\ j \notin ready \Rightarrow (\sigma'(\ j\ )).\rho = (\sigma(\ j\ )).\rho\ ) \wedge$
$(\ j \in ready \Rightarrow (\sigma'(\ j\ )).\rho = \mathrm{pop}(\ (\sigma(\ j\ )).\rho\ )\ )$

$$InCoRet \mathrel{\widehat=} InCoRetControlEffect \land InCoRetStackEffect$$

InRet

$$InCoRet$$

$$\#waiting = 1$$

TwoConfigurationsTransition

$$\Xi\, Config_{Local}$$

$$(\exists\, i : BodyNum_{Class} \bullet$$
$$(\exists \Delta\, State_{\partial} \bullet$$
$$LocalTransition \lor$$
$$Self \lor$$
$$(\exists\, sx : Var_{Glob};\ \vec{\gamma} : \mathrm{seq}\, OBJ \bullet GlobVarExp \lor GlobVarAssign) \lor$$
$$(\exists \Delta\, State_{\partial 1};\ j : BodyNum_{Class} \bullet$$
$$(\exists\, fx : Var_{Part};\ \gamma : OBJ \bullet PartVarExp \lor PartAssign)))) \lor$$
$$(\exists\, mf : Name_{Method};\ M : METHOD \bullet$$
$$(\exists\, i : BodyNum_{Class};\ Actual : \mathrm{seq}\, Exp \bullet InCall) \lor$$
$$(\exists\, ready : \mathbb{F}_1\, BodyNum_{Class};\ Actual : \mathrm{seq}(\ Var_{Loc} \times Exp\ );$$
$$\partial actual : \mathrm{seq}(\ \mathrm{seq}(\ Var_{Loc} \times Exp\ )\ ) \bullet InCoCall) \lor$$
$$(\exists\, ready, waiting : \mathbb{F}_1\, BodyNum_{Class};\ \vec{\gamma} : \mathrm{seq}_1\, OBJ \bullet InCoRet))$$

A.3.3 System level

$$Config == \mathbb{F}\, Config_{Local}$$

TwoObjects

$$MultiMove$$
$$MultiMove_1$$

$$\forall j : 1\,..\,\#\sigma \bullet$$
$$(\sigma'(\ j\ )).\rho = (\sigma(\ j\ )).\rho$$

ExtCallPrecond

$$
\begin{array}{l}
TwoObjects \\
mf : Name_{Method} \\
Actual : \mathrm{seq}\ Exp \\
M : METHOD \\
i : BodyNum_{Class} \\
\hline
i \in \mathrm{dom}\,\sigma \\
(\sigma(\ i\ )).s = exp(\ meth(\ name(\ obj(\alpha_1)\ ), mf, Actual\ )\ ) \\
mf \neq init \wedge mf \in \mathrm{dom}((Program_1(\tau_1)).Method) \\
M = ((Program_1(\tau_1)).Method)(\ mf\ ) \\
\#Actual = \#(M.Param) \\
\forall k : 1\,..\,\#Actual \bullet \exists \gamma : OBJ \bullet Actual(\ k\ ) = name(\ \gamma\ ) \\
\forall k : 1\,..\,\#\sigma_1 \mid k \in \mathrm{dom}(M.Alloc) \bullet \\
\quad \exists \gamma : OBJ \bullet (\sigma_1(\ k\ )).s = exp(name(\ \gamma\ ))
\end{array}
$$

ExtCallControlEffect

$$
\begin{array}{l}
ExtCallPrecond \\
\hline
\forall j : 1\,..\,\#\sigma_1 \mid j \in \mathrm{dom}(M.Alloc) \bullet \\
\quad (\sigma_1'(\ j\ )).s = send(\ ((M.Body)(\ (M.Alloc)(j)\ )).s, (\alpha, \{i\})\ ) \\
\forall j : 1\,..\,\#\sigma_1 \mid j \notin \mathrm{dom}(M.Alloc) \bullet \\
\quad (\sigma_1'(\ j\ )).s = (\sigma_1(\ j\ )).s \\
\forall j : 1\,..\,\#\sigma \mid j \neq i \bullet \\
\quad (\sigma'(\ j\ )).s = (\sigma(\ j\ )).s \\
(\sigma'(\ i\ )).s = exp(wait(\ \alpha_1, mf\ ))
\end{array}
$$

ExtCallStackEffect

$$
\begin{array}{l}
ExtCallPrecond \\
\hline
\forall j : 1\,..\,\#\sigma_1 \mid j \in \mathrm{dom}(M.Alloc) \bullet \exists_1\ ar : AR \mid \\
\quad ar.Cur = mf \wedge \\
\quad ar.Env = \\
\qquad (\ \lambda\, u : Var_{Loc} \mid u \in (((M.Body)(\ (M.Alloc)(j)\ )).Var) \bullet nil\ ) \bowtie \\
\qquad\quad \{\ \gamma : OBJ;\ k : 1\,..\,\#Actual \mid Actual(\ k\ ) = name(\ \gamma\ ) \bullet \\
\qquad\qquad (M.Param)(\ k\ ) \mapsto \gamma\ \} \bullet \\
\quad (\sigma_1'(\ j\ )).\rho = \mathrm{push}(\ (\sigma_1(\ j\ )).\rho, ar\ ) \\
\forall j : 1\,..\,\#\sigma_1 \mid j \notin \mathrm{dom}(M.Alloc) \bullet \\
\quad (\sigma_1'(\ j\ )).\rho = (\sigma_1(\ j\ )).\rho
\end{array}
$$

$$ExtCall \mathrel{\widehat=} ExtCallControlEffect \wedge ExtCallStackEffect$$

ExtCoCallPrecond

$$
\begin{array}{l}
TwoObjects \\
ready : \mathbb{F}_1\ BodyNum_{Class} \\
mf : Name_{Method} \\
Actual : \mathrm{seq}(\ Var_{Loc} \times Exp\ ) \\
\partial actual : \mathrm{seq}_1(\ \mathrm{seq}(\ Var_{Loc} \times Exp\ )\ ) \\
M : METHOD \\
\hline
mf \neq init \wedge mf \in \mathrm{dom}((Program_1(\tau_1)).Method) \\
M = ((Program_1(\tau_1)).Method)(\ mf\ ) \\
\exists\, mg : Name_{Method} \mid mg \in \mathrm{dom}((Program(\tau)).Method) \bullet \\
\quad ready = \mathrm{dom}((((Program(\tau)).Method)(\ mg\ )).Alloc) \wedge \\
\quad (\forall\, i : BodyNum_{Class} \mid i \in ready \bullet (\mathrm{top}(\ (\sigma(\ i\ )).\rho\ )).Cur = mg) \\
\forall\, j : 1\,..\,\#\sigma \mid j \in ready \bullet \\
\quad (\sigma(\ j\ )).s = exp(cometh(\ name(\ obj(\alpha_1)\ ), mf, \partial actual(\ j\ )\ )) \\
\forall\, k : 1\,..\,\#\sigma_1 \mid k \in \mathrm{dom}(M.Alloc) \bullet \\
\quad \exists\, \gamma : OBJ \bullet (\sigma_1(\ k\ )).s = exp(name(\ \gamma\ )) \\
\#\partial actual = \#ready \\
\forall\, j, j' : 1\,..\,\#\partial actual \bullet \\
\quad \forall\, k : 1\,..\,\#(\partial actual(\ j\ ));\ k' : 1\,..\,\#(\partial actual(\ j'\ )) \bullet \\
\qquad \pi_1(\ (\partial actual(\ j\ ))(k)\ ) = \pi_1(\ (\partial actual(\ j'\ ))(k')\ ) \Rightarrow \\
\qquad\quad (\ j = j' \wedge k = k'\ ) \\
Actual = \frown\!/\, \partial actual \\
\#Actual = \#(M.Param) \\
\forall\, j : 1\,..\,\#Actual \bullet \exists_1\, k : 1\,..\,\#(M.Param) \bullet \exists\, \gamma : OBJ \bullet \\
\quad Actual(\ j\ ) = (M.Param)(\ k\ ) \mapsto name(\ \gamma\ )
\end{array}
$$

ExtCoCallControlEffect

$$
\begin{array}{l}
ExtCoCallPrecond \\
\hline
\forall\, j : 1\,..\,\#\sigma_1 \bullet \\
\quad (\ j \in (\mathrm{dom}(M.Alloc)) \Rightarrow \\
\qquad (\sigma_1'(\ j\ )).s = \\
\qquad\quad send(\ ((M.Body)(\ (M.Alloc)(j)\ )).s, (\alpha, ready)\ )\ ) \wedge \\
\quad (\ j \notin \mathrm{dom}(M.Alloc) \Rightarrow \\
\qquad (\sigma_1'(\ j\ )).s = (\sigma_1(\ j\ )).s\ ) \\
\forall\, j : 1\,..\,\#\sigma \bullet \\
\quad (\ j \in ready \Rightarrow \\
\qquad (\sigma'(\ j\ )).s = exp(wait(\ \alpha_1, mf\ ))\ ) \wedge \\
\quad (\ j \notin ready \Rightarrow \\
\qquad (\sigma'(\ j\ )).s = (\sigma(\ j\ )).s\ )
\end{array}
$$

ExtCoCallStackEffect

ExtCoCallPrecond

$$\forall j : 1 \mathinner{..} \#\sigma_1 \mid j \in \mathrm{dom}(M.Alloc) \bullet \exists_1 ar : AR \mid$$
$$ar.Cur = mf \wedge$$
$$ar.Env =$$
$$(\lambda u : Var_{Loc} \mid u \in (((M.Body)((M.Alloc)(j))).Var) \bullet nil) \bowtie$$
$$\{ \gamma : OBJ;\ k : 1 \mathinner{..} \#Actual \mid \pi_2(Actual(k)) = name(\gamma) \bullet$$
$$\pi_1(Actual(k)) \mapsto \gamma \} \bullet$$
$$(\sigma_1'(j)).\rho = \mathrm{push}((\sigma_1(j)).\rho, ar)$$
$$\forall j : 1 \mathinner{..} \#\sigma_1 \mid j \notin \mathrm{dom}(M.Alloc) \bullet$$
$$(\sigma_1'(j)).\rho = (\sigma_1(j)).\rho$$

$$ExtCoCall \mathrel{\widehat=} ExtCoCallControlEffect \wedge ExtCoCallStackEffect$$

ExtCoRetPrecond

TwoObjects
$ready, waiting : \mathbf{F}_1\ BodyNum_{Class}$
$\vec{\gamma} : \mathrm{seq}_1\ OBJ$
$mf : Name_{Method}$

$$mf \in \mathrm{dom}((Program_1(\tau_1)).Method)$$
$$ready = \mathrm{dom}(((Program_1(\tau_1)).Method)(mf)).Alloc$$
$$\#\vec{\gamma} = \#ready$$
$$\forall j : 1 \mathinner{..} \#\sigma_1 \mid j \in ready \bullet$$
$$(\mathrm{top}((\sigma_1(j)).\rho)).Cur = mf \wedge$$
$$(\sigma_1(j)).s = send(exp(name(\vec{\gamma}(j))), (\alpha, waiting))$$
$$\forall j : 1 \mathinner{..} \#\sigma \mid j \in waiting \bullet$$
$$(\sigma(j)).s = exp(wait(\alpha_1, mf))$$

ExtCoRetControlEffect

ExtCoRetPrecond

$$\forall j : 1 \mathinner{..} \#\sigma_1 \bullet$$
$$(j \in ready \Rightarrow$$
$$(\sigma_1'(j)).s = exp(name(\vec{\gamma}(j)))) \wedge$$
$$(j \notin ready \Rightarrow$$
$$(\sigma_1'(j)).s = (\sigma_1(j)).s)$$
$$\forall j : 1 \mathinner{..} \#\sigma \bullet$$
$$((j \in waiting \wedge mf \neq init) \Rightarrow$$
$$(\sigma'(j)).s = exp(name(seqobj(\vec{\gamma})))) \wedge$$
$$((j \in waiting \wedge mf = init) \Rightarrow$$
$$(\sigma'(j)).s = exp(name(\vec{\gamma}(1)))) \wedge$$
$$(j \notin waiting \Rightarrow$$
$$(\sigma'(j)).s = (\sigma(j)).s)$$

ExtCoRetStackEffect

ExtCoRetPrecond

$\forall j : 1 \,..\, \#\sigma_1 \bullet$
$\quad (\, (j \notin ready \lor mf = init) \Rightarrow (\sigma_1'(\, j \,)).\rho = (\sigma_1(\, j \,)).\rho \,) \land$
$\quad (\, (j \in ready \land mf \neq init) \Rightarrow (\sigma_1'(\, j \,)).\rho = \mathrm{pop}(\, (\sigma_1(\, j \,)).\rho \,) \,)$

$ExtCoRet \mathrel{\widehat=} ExtCoRetControlEffect \land ExtCoRetStackEffect$

NewCallPrecond

$\Xi\, Config_{Local}$
$Config_{Local_1}$
$C : CLASS$
$Actual : \mathrm{seq}\, Exp$
$i : BodyNum_{Class}$

$i \in \mathrm{dom}\, \sigma$
$(\sigma(\, i \,)).s = exp(new(\, \tau_1, Actual \,))$

$\tau_1 \in \mathrm{dom}\, Program$
$C = Program(\, \tau_1 \,)$

$\#Actual = \#(C.Param)$
$\forall k : 1 \,..\, \#Actual \bullet \exists \gamma : OBJ \bullet Actual(\, k \,) = name(\, \gamma \,)$

NewCallCreated

NewCallPrecond

$\alpha_1 = \alpha \frown \langle i, (\sigma(\, i \,)).\eta \rangle$

$\psi_1 = (\, \lambda sx : Var_{Glob} \mid sx \in \mathrm{dom}(C.Virtual) \bullet$
$\quad (\, \lambda fx : Var_{Part} \mid fx \in \mathrm{ran}((C.Virtual)(\, sx \,)) \bullet nil \,) \,)$

$\#\sigma_1 = \#(C.Body)$

$\forall j : 1 \,..\, \#\sigma_1 \bullet$
$\quad (\sigma_1(j)).s =$
$\qquad seqcomp(\, ((C.Body)(j)).s, send(\, exp(self), (\, \alpha, \{i\} \,) \,) \,) \land$
$\quad (\sigma_1(j)).\delta = (\, \lambda fx : Var_{Part} \mid fx \in ((C.Body)(j)).Var \bullet nil \,) \land$
$\quad (\sigma_1(j)).\phi =$
$\qquad (\, \lambda x : Var_{Priv} \mid x \in ((C.Body)(j)).Priv \bullet nil \,) \bowtie$
$\qquad\quad \{\, \gamma : OBJ;\ k : 1 \,..\, \#Actual \mid Actual(\, k \,) = name(\, \gamma \,) \bullet$
$\qquad\qquad (C.Param)(\, k \,) \mapsto \gamma \,\} \land$
$\quad (\exists\, ar : AR \mid ar.Cur = init \land ar.Env = \varnothing \bullet$
$\qquad (\sigma_1(j)).\rho = \mathrm{push}(\, \mathrm{empty}, ar \,)) \land$
$\quad (\sigma_1(j)).\eta = 0$

$Program_1 = Program$
$Main_1 = Main$

NewCallCreator

NewCallPrecond

$\psi' = \psi$

$\forall j : 1 \,..\, \#\sigma \mid j \neq i \bullet \sigma'(\ j\ ) = \sigma(\ j\ )$

$(\sigma'(\ i\ )).s = exp(wait(\ \alpha_1, init\ ))$
$(\sigma'(\ i\ )).\delta = (\sigma(\ i\ )).\delta$
$(\sigma'(\ i\ )).\phi = (\sigma(\ i\ )).\phi$
$(\sigma'(\ i\ )).\rho = (\sigma(\ i\ )).\rho$
$(\sigma'(\ i\ )).\eta = (\sigma(\ i\ )).\eta + 1$

$NewCall \mathrel{\widehat{=}} NewCallCreated \wedge NewCallCreator$

CoNewCallPrecond

$\Xi\ Config_{Local}$
$Config_{Local_1}$
$C : CLASS$
$ready : \mathbb{F}_1\ BodyNum_{Class}$
$Actual : \text{seq}(\ Var_{Priv} \times Exp\ )$
$\partial actual : \text{seq}_1(\ \text{seq}(\ Var_{Priv} \times Exp\ )\ )$

$\tau_1 \in \text{dom}\ Program$
$C = Program(\ \tau_1\ )$

$\exists\, mg : Name_{Method} \mid mg \in \text{dom}((Program(\tau)).Method) \bullet$
$\quad ready = \text{dom}((((Program(\tau)).Method)(\ mg\ )).Alloc) \wedge$
$\quad (\forall i : BodyNum_{Class} \mid i \in ready \bullet (\text{top}(\ (\sigma(\ i\ )).\rho\ )).Cur = mg)$

$\forall j : 1 \,..\, \#\sigma \mid j \in ready \bullet$
$\quad (\sigma(\ j\ )).s = exp(conew(\ \tau_1, \partial actual(\ j\ )\ ))$

$\#\partial actual = \#ready$
$\forall j, j' : 1 \,..\, \#\partial actual \bullet$
$\quad \forall k : 1 \,..\, \#(\partial actual(\ j\ ));\ k' : 1 \,..\, \#(\partial actual(\ j'\ )) \bullet$
$\qquad \pi_1(\ (\partial actual(\ j\ ))(k)\ ) = \pi_1(\ (\partial actual(\ j'\ ))(k')\ ) \Rightarrow$
$\qquad\quad (\ j = j' \wedge k = k'\ )$
$Actual = \frown / \partial actual$
$\#Actual = \#(C.Param)$
$\forall j : 1 \,..\, \#Actual \bullet \exists_1 k : 1 \,..\, \#(C.Param) \bullet \exists \gamma : OBJ \bullet$
$\quad Actual(\ j\ ) = (C.Param)(\ k\ ) \mapsto name(\ \gamma\ )$

CoNewCallCreated

CoNewCallPrecond

$\exists_1 k : BodyNum_{Class} \mid k \in ready \bullet \alpha_1 = \alpha \frown \langle k, (\sigma(\ k\ )).\eta \rangle$

$\psi_1 = (\ \lambda sx : Var_{Glob} \mid sx \in \mathrm{dom}(C.Virtual) \bullet$
$\quad (\ \lambda fx : Var_{Part} \mid fx \in \mathrm{ran}((C.Virtual)(\ sx\ )) \bullet nil\ )\ )$

$\#\sigma_1 = \#(C.Body)$

$\forall j : 1 .. \#(C.Body) \bullet$
$\quad (\sigma_1(j)).s =$
$\quad\quad seqcomp(\ ((C.Body)(j)).s, send(\ exp(self), (\ \alpha, ready\ )\ )\ ) \wedge$
$\quad (\sigma_1(j)).\delta = (\ \lambda fx : Var_{Part} \mid fx \in ((C.Body)(j)).Var \bullet nil\ ) \wedge$
$\quad (\sigma_1(j)).\phi =$
$\quad\quad (\ \lambda x : Var_{Priv} \mid x \in ((C.Body)(j)).Priv \bullet nil\ ) \bowtie$
$\quad\quad\quad \{\ \gamma : OBJ;\ k : 1 .. \#Actual \mid \pi_2(Actual(\ k\ )) = name(\ \gamma\ ) \bullet$
$\quad\quad\quad\quad \pi_1(Actual(\ k\ )) \mapsto \gamma\ \} \wedge$
$\quad (\exists ar : AR \mid ar.Cur = init \wedge ar.Env = \emptyset \bullet$
$\quad\quad (\sigma_1(j)).\rho = \mathrm{push}(\ \mathrm{empty}, ar\ )) \wedge$
$\quad (\sigma_1(j)).\eta = 0$

$Program_1 = Program$
$Main_1 = Main$

CoNewCallCreator

CoNewCallPrecond

$\psi' = \psi$

$\forall j : 1 .. \#\sigma \mid j \notin ready \bullet$
$\quad \sigma'(\ j\ ) = \sigma(\ j\ )$

$\forall j : 1 .. \#\sigma \mid j \in ready \bullet$
$\quad (\sigma'(\ j\ )).s = exp(wait(\ \alpha_1, init\ )) \wedge$
$\quad (\sigma'(\ j\ )).\delta = (\sigma(\ j\ )).\delta \wedge$
$\quad (\sigma'(\ j\ )).\phi = (\sigma(\ j\ )).\phi \wedge$
$\quad (\sigma'(\ j\ )).\rho = (\sigma(\ j\ )).\rho \wedge$
$\quad (\sigma'(\ j\ )).\eta = (\sigma(\ j\ )).\eta + 1$

$CoNewCall \mathrel{\widehat{=}} CoNewCallCreated \wedge CoNewCallCreator$

ThreeConfigurationsTransition

$\Xi\ Config_{Local}$
$Config_{Local\,1}$

$\exists C : CLASS \bullet$
$\quad (\exists i : BodyNum_{Class};\ Actual : \mathrm{seq}\ Exp \bullet NewCall) \vee$
$\quad (\exists ready : \mathbb{F}_1\ BodyNum_{Class};\ Actual : \mathrm{seq}(\ Var_{Priv} \times Exp\ );$
$\quad\quad \partial actual : \mathrm{seq}(\ \mathrm{seq}(\ Var_{Priv} \times Exp\ )\ ) \bullet CoNewCall)$

FourConfigurationsTransition

$\Xi\ Config_{Local}$
$\Xi\ Config_{Local}{}_1$

$\exists\, mf : Name_{Method};\ M : METHOD \bullet$
$(\exists\, i : BodyNum_{Class};\ Actual : \mathrm{seq}\, Exp \bullet ExtCall) \lor$
$(\exists\, ready : \mathbb{F}_1\, BodyNum_{Class};\ Actual : \mathrm{seq}(\ Var_{Loc} \times Exp\ );$
$\partial actual : \mathrm{seq}(\ \mathrm{seq}(\ Var_{Loc} \times Exp\ )\ ) \bullet ExtCoCall) \lor$
$(\exists\, ready, waiting : \mathbb{F}_1\, BodyNum_{Class};\ \vec{\gamma} : \mathrm{seq}_1\, OBJ \bullet ExtCoRet)$

A.4 Putting all together

A.4.1 Computation progress rules

Parameters evaluation

LeftToRightParEval1

$e, e' : Exp$
$Actual, Actual' : \mathrm{seq}\, Exp$

$\exists_1\, j : 1\,..\,\#Actual \bullet$
$Actual(\ j\ ) = e \land$
$(\forall\, k : 1\,..\,\#Actual \mid k < j \bullet$
$\exists\, \gamma : OBJ \bullet Actual(\ k\ ) = name(\ \gamma\ )) \land$
$Actual' = Actual \oplus \{\ j \mapsto e'\ \}$

LeftToRightParEval2[VAR]

$e, e' : Exp$
$Actual, Actual' : \mathrm{seq}(\ VAR \times Exp\ )$

$\exists_1\, j : 1\,..\,\#Actual \bullet$
$(\exists\, v : VAR \bullet Actual(\ j\ ) = (\ v, e\ )) \land$
$(\forall\, k : 1\,..\,\#Actual \mid k < j \bullet$
$\exists\, \gamma : OBJ;\ v : VAR \bullet Actual(\ k\ ) = (\ v, name(\ \gamma\ )\ )) \land$
$Actual' = Actual \oplus \{\ j \mapsto (\ \pi_1(\ Actual(\ j\ )\ ), e'\ )\ \}$

Computation progress in expressions

ExpressionCommonPremise

$\Xi\ Config_{Local}$
$i : BodyNum_{Class}$
$e, e' : Exp$

$i \in \mathrm{dom}\, \sigma$
$(\sigma(\ i\ )).s = exp(\ e\ ) \land (\sigma'(\ i\ )).s = exp(\ e'\ )$

ExpEffectPrelude

$$
\begin{array}{l}
ExpressionCommonPremise \\
\Xi\ Config_{Local_1} \\
\hline
\alpha_1 = \alpha \\
\psi_1 = \psi \land \psi'_1 = \psi' \\
\forall j : 1 \mathinner{..} \#\sigma \mid j \neq i \bullet \\
\quad \sigma_1(\ j\ ) = \sigma(\ j\ ) \land \sigma'_1(\ j\ ) = \sigma'(\ j\ ) \\
(\sigma_1(\ i\ )).\delta = (\sigma(\ i\ )).\delta \land (\sigma'_1(\ i\ )).\delta = (\sigma'(\ i\ )).\delta \\
(\sigma_1(\ i\ )).\phi = (\sigma(\ i\ )).\phi \land (\sigma'_1(\ i\ )).\phi = (\sigma'(\ i\ )).\phi \\
(\sigma_1(\ i\ )).\rho = (\sigma(\ i\ )).\rho \land (\sigma'_1(\ i\ )).\rho = (\sigma'(\ i\ )).\rho \\
(\sigma_1(\ i\ )).\eta = (\sigma(\ i\ )).\eta \land (\sigma'_1(\ i\ )).\eta = (\sigma'(\ i\ )).\eta \\
\tau_1 = \tau \\
Program_1 = Program \land Main_1 = Main
\end{array}
$$

ExpEvalPar1

$$
\begin{array}{l}
ExpEffectPrelude \\
\hline
\exists\ Actual, Actual' : \mathrm{seq}\ Exp \mid LeftToRightParEval1 \bullet \\
\quad (\exists\ mf : Name_{Method} \bullet \\
\qquad (\sigma_1(\ i\ )).s = exp(call(\ mf, Actual\ )) \land \\
\qquad (\sigma'_1(\ i\ )).s = exp(call(\ mf, Actual'\ ))) \lor \\
\quad (\exists\ mf : Name_{Method};\ \gamma : OBJ \bullet \\
\qquad (\sigma_1(\ i\ )).s = exp(meth(\ name(\ \gamma\ ), mf, Actual\ )) \land \\
\qquad (\sigma'_1(\ i\ )).s = exp(meth(\ name(\ \gamma\ ), mf, Actual'\ ))) \lor \\
\quad (\exists \omega : Name_{Class} \bullet \\
\qquad (\sigma_1(\ i\ )).s = exp(new(\ \omega, Actual\ )) \land \\
\qquad (\sigma'_1(\ i\ )).s = exp(new(\ \omega, Actual'\ )))
\end{array}
$$

ExpEvalPar2

$$
\begin{array}{l}
ExpEffectPrelude \\
\hline
\exists\ Actual, Actual' : \mathrm{seq}(\ Var_{Loc} \times Exp\ ) \mid LeftToRightParEval2[Var_{Loc}] \bullet \\
\quad (\exists\ mf : Name_{Method} \bullet \\
\qquad (\sigma_1(\ i\ )).s = exp(cocall(\ mf, Actual\ )) \land \\
\qquad (\sigma'_1(\ i\ )).s = exp(cocall(\ mf, Actual'\ ))) \lor \\
\quad (\exists\ mf : Name_{Method};\ \gamma : OBJ \bullet \\
\qquad (\sigma_1(\ i\ )).s = exp(cometh(\ name(\ \gamma\ ), mf, Actual\ )) \land \\
\qquad (\sigma'_1(\ i\ )).s = exp(cometh(\ name(\ \gamma\ ), mf, Actual'\ )))
\end{array}
$$

ExpEvalPar3

ExpEffectPrelude

$$\exists\, Actual, Actual' : \mathrm{seq}(\ Var_{Priv} \times Exp\ ) \mid LeftToRightParEval2[Var_{Priv}] \bullet$$
$$(\exists\, \omega : Name_{Class} \bullet$$
$$(\sigma_1(\ i\ )).s = exp(conew(\ \omega, Actual\ )) \wedge$$
$$(\sigma_1'(\ i\ )).s = exp(conew(\ \omega, Actual'\ )))$$

ExpEvalDest

ExpEffectPrelude

$$\exists\, mf : Name_{Method} \bullet$$
$$(\exists\, Actual : \mathrm{seq}\, Exp \bullet$$
$$(\sigma_1(\ i\ )).s = exp(meth(\ e, mf, Actual\ )) \wedge$$
$$(\sigma_1'(\ i\ )).s = exp(meth(\ e', mf, Actual\ ))) \vee$$
$$(\exists\, Actual : \mathrm{seq}(\ Var_{Loc} \times Exp\ ) \bullet$$
$$(\sigma_1(\ i\ )).s = exp(cometh(\ e, mf, Actual\ )) \wedge$$
$$(\sigma_1'(\ i\ )).s = exp(cometh(\ e', mf, Actual\ )))$$

ExpEvalAssign

ExpEffectPrelude

$$(\exists\, fx : Var_{Part} \bullet$$
$$(\sigma_1(\ i\ )).s = fassign(\ fx, e\ ) \wedge (\sigma_1'(\ i\ )).s = fassign(\ fx, e'\ )) \vee$$
$$(\exists\, x : Var_{Priv} \bullet$$
$$(\sigma_1(\ i\ )).s = passign(\ x, e\ ) \wedge (\sigma_1'(\ i\ )).s = passign(\ x, e'\ )) \vee$$
$$(\exists\, u : Var_{Loc} \bullet$$
$$(\sigma_1(\ i\ )).s = lassign(\ u, e\ ) \wedge (\sigma_1'(\ i\ )).s = lassign(\ u, e'\ )) \vee$$
$$(\exists\, sx : Var_{Glob} \bullet$$
$$(\sigma_1(\ i\ )).s = massign(\ sx, e\ ) \wedge (\sigma_1'(\ i\ )).s = massign(\ sx, e'\ ))$$

ExpEvalBool

ExpEffectPrelude

$$(\ \exists\, s_1, s_2 : Stmt \bullet$$
$$(\sigma_1(\ i\ )).s = if(\ e, s_1, s_2\ ) \wedge (\sigma_1'(\ i\ )).s = if(\ e', s_1, s_2\ )\ ) \vee$$
$$(\ \exists\, s : Stmt \bullet$$
$$(\sigma_1(\ i\ )).s = do(\ e, s\ ) \wedge (\sigma_1'(\ i\ )).s = do(\ e', s\ )\ )$$

Computation progress in statements

StatementCommonPremise

$\Xi\ Config_{Local}$
$i : BodyNum_{Class}$
$s, s' : Stmt$

$i \in \mathrm{dom}\, \sigma$

$(\sigma(\ i\ )).s = s \land (\sigma'(\ i\ )).s = s'$

StmtEffectPrelude

StatementCommonPremise
$\Xi\ Config_{Local_1}$

$\alpha_1 = \alpha$

$\psi_1 = \psi \land \psi'_1 = \psi'$

$\forall j : 1 \mathinner{..} \#\sigma \mid j \neq i \bullet$
$\quad \sigma_1(\ j\ ) = \sigma(\ j\ ) \land \sigma'_1(\ j\ ) = \sigma'(\ j\ )$

$(\sigma_1(\ i\ )).\delta = (\sigma(\ i\ )).\delta \land (\sigma'_1(\ i\ )).\delta = (\sigma'(\ i\ )).\delta$
$(\sigma_1(\ i\ )).\phi = (\sigma(\ i\ )).\phi \land (\sigma'_1(\ i\ )).\phi = (\sigma'(\ i\ )).\phi$
$(\sigma_1(\ i\ )).\rho = (\sigma(\ i\ )).\rho \land (\sigma'_1(\ i\ )).\rho = (\sigma'(\ i\ )).\rho$
$(\sigma_1(\ i\ )).\eta = (\sigma(\ i\ )).\eta \land (\sigma'_1(\ i\ )).\eta = (\sigma'(\ i\ )).\eta$

$\tau_1 = \tau$

$Program_1 = Program \land Main_1 = Main$

StmtEvalSeqComp

StmtEffectPrelude

$\exists s_1 : Stmt \bullet$
$\quad (\sigma_1(\ i\ )).s = seqcomp(\ s, s_1\ ) \land (\sigma'_1(\ i\ )).s = seqcomp(\ s', s_1\ )$

StmtEvalSend

StmtEffectPrelude

$\exists receiver : CallId \bullet$
$\quad (\sigma_1(\ i\ )).s = send(\ s, receiver\ ) \land (\sigma'_1(\ i\ )).s = send(\ s', receiver\ )$

A schema to summarize

ComputationProgress

$\Xi\ Config_{Local}$
$\Xi\ Config_{Local_1}$

$(\ \exists\, i : BodyNum_{Class};\ e, e' : Exp \mid ExpEffectPrelude \bullet$
$ExpEvalPar1 \lor ExpEvalPar2 \lor ExpEvalPar3 \lor$
$ExpEvalDest \lor ExpEvalAssign \lor ExpEvalBool\ )$
$\lor$
$(\ \exists\, i : BodyNum_{Class};\ s, s' : Stmt \mid StmtEffectPrelude \bullet$
$StmtEvalSeqComp \lor StmtEvalSend\ )$

A.4.2 The formal system at last

$_ \xrightarrow{!\|trans} _ : Config \leftrightarrow Config$

$\forall\, C, C' : Config \bullet$
$C \xrightarrow{!\|trans} C' \Leftrightarrow$
$(\ \exists\, TwoConfigurationsTransition;\ X : Config \bullet$
$C = X \sqcup \{\ \theta Config_{Local}\ \} \land$
$C' = X \sqcup \{\ \theta Config_{Local}{}'\ }\ )$
$\lor$
$(\ \exists\, ThreeConfigurationsTransition;\ X : Config \bullet$
$C = X \sqcup \{\ \theta Config_{Local}\ \} \land$
$C' = X \sqcup \{\ \theta Config_{Local}{}', \theta Config_{Local_1}\ }\ )$
$\lor$
$(\ \exists\, FourConfigurationsTransition;\ X : Config \bullet$
$C = X \sqcup \{\ \theta Config_{Local}, \theta Config_{Local_1}\ \} \land$
$C' = X \sqcup \{\ \theta Config_{Local}{}', \theta Config_{Local_1}{}'\ }\ )$
$\lor$
$(\ \exists\, ComputationProgress;\ X : Config \bullet$
$X \sqcup \{\ \theta Config_{Local}\ \} \xrightarrow{!\|trans} X \sqcup \{\ \theta Config_{Local}{}'\ \} \land$
$C = X \sqcup \{\ \theta Config_{Local_1}\ \} \land$
$C' = X \sqcup \{\ \theta Config_{Local_1}{}'\ }\ )$

A.4.3 Parallelism

Internal

FragmentNonInterference

$\Xi\ Config_{Local}$
$Config_{Local_1}$

$\alpha = \alpha_1 \land \tau = \tau_1$

$\sigma' \bowtie \sigma_1 = \sigma_1 \bowtie \sigma'$

$\psi' \bowtie \psi_1 = \psi_1 \bowtie \psi'$

TwoInOne

$$Config_{Local}'$$
$$\Xi\, Config_{Local\,1}$$

$$\sigma_1' = \sigma' \bowtie \sigma_1$$
$$\psi_1' = \psi' \bowtie \psi_1$$

$$\_ \text{ complies } \_ : Config \leftrightarrow Config$$

$$\forall X, X' : Config \bullet$$
$$X \text{ complies } X' \Leftrightarrow$$
$$(\forall c, c' : Config_{Local} \mid c \in X \land c' \in X' \bullet c.\alpha = c'.\alpha \Leftrightarrow c = c')$$

ParallelismInside

FragmentNonInterference
TwoInOne

$$\exists X, X', X_1 : Config \bullet$$
$$(\; X \sqcup \{ \theta Config_{Local} \} \overset{l|trans}{\longrightarrow} X' \sqcup \{ \theta Config_{Local}' \} \land$$
$$X \sqcup \{ \theta Config_{Local} \} \overset{l|trans}{\longrightarrow} X_1 \sqcup \{ \theta Config_{Local\,1} \} \land$$
$$X' \text{ complies } X_1 \;) \Rightarrow$$
$$(\; X \sqcup \{ \theta Config_{Local} \} \overset{l|trans}{\longrightarrow} (X' \cup X_1) \sqcup \{ \theta Config_{Local}'_1 \} \lor$$
$$X \sqcup \{ \theta Config_{Local} \} \overset{l|trans}{\longrightarrow} X' \sqcup \{ \theta Config_{Local}' \} \;)$$

External

ParallelismOutside

$$X, X', Y, Y' : Config$$

$$(\; X \overset{l|trans}{\longrightarrow} X' \land$$
$$Y \overset{l|trans}{\longrightarrow} Y' \land$$
$$X \cap Y = \varnothing \;) \Rightarrow$$
$$(\; X \cup Y \overset{l|trans}{\longrightarrow} X' \cup Y' \lor$$
$$X \cup Y \overset{l|trans}{\longrightarrow} X \cup Y' \;)$$

A.4.4 Enhanced transition system

$$
\begin{array}{l}
\_ \overset{\|_{trans}}{\longrightarrow} \_ : Config \leftrightarrow Config \\
\hline
\forall C, C' : Config \bullet \\
\quad C \overset{\|_{trans}}{\longrightarrow} C' \Leftrightarrow \\
\qquad (\ C \overset{\|_{trans}}{\longrightarrow} C'\ ) \\
\qquad \vee \\
\qquad (\ \exists\, FragmentNonInterference;\ TwoInOne; \\
\qquad\quad X, X', X_1 : Config \bullet \\
\qquad\quad C = X \sqcup \{\ \theta Config_{Local}\ \} \wedge \\
\qquad\quad C \overset{\|_{trans}}{\longrightarrow} X' \sqcup \{\ \theta Config_{Local}{}'\ \} \wedge \\
\qquad\quad C \overset{\|_{trans}}{\longrightarrow} X_1 \sqcup \{\ \theta Config_{Local_1}\ \} \wedge \\
\qquad\quad C' = (X' \cup X_1) \sqcup \{\ \theta Config'_{Local_1}\ \}\ ) \\
\qquad \vee \\
\qquad (\ \exists\, X, X', Y, Y' : Config \bullet \\
\qquad\quad C = X \sqcup Y \wedge \\
\qquad\quad X \overset{\|_{trans}}{\longrightarrow} X' \wedge \\
\qquad\quad Y \overset{\|_{trans}}{\longrightarrow} Y' \wedge \\
\qquad\quad C' = X' \cup Y'\ )
\end{array}
$$

A.4.5 Meaning of a unit

$$Trace == \mathbb{N} \nrightarrow Config$$

$$
\begin{array}{l}
Meaning : UNIT \rightarrow \mathbb{P}\ Trace \\
\hline
\forall u : UNIT \bullet \exists_1\ Config_{Local};\ n? : \mathbb{N} \bullet \\
\quad \forall t : Trace \mid t \in Meaning(\ u\ ) \bullet \\
\qquad (\ InitConfig_{Local} \wedge Program = u.Program \wedge Main = u.Main \wedge \\
\qquad\quad \{\ \theta Config_{Local}\ \} \subseteq t(\ 0\ )\ ) \wedge \\
\qquad (\ \forall k : \mathbb{N}_1 \mid k \in \operatorname{dom} t \bullet \\
\qquad\quad t(\ k-1\ ) \overset{\|_{trans}}{\longrightarrow} t(\ k\ )\ ) \wedge \\
\qquad (\ (\exists n : \mathbb{N}_1 \bullet n = \#(\operatorname{dom} t)) \Rightarrow \\
\qquad\quad \neg\ (\ \exists C : Config \bullet t(\ \#(\operatorname{dom} t) - 1\ ) \overset{\|_{trans}}{\longrightarrow} C\ )\ ) \wedge \\
\qquad \neg\ (\ \exists o : Config_{Local};\ C, C' : Config;\ Ready : \mathbb{P}\ \mathbb{N} \bullet \\
\qquad\quad o \in C \wedge C \overset{\|_{trans}}{\longrightarrow} C' \wedge o \notin C' \wedge \\
\qquad\quad Ready \subseteq \operatorname{dom} t \wedge Ready \neq \emptyset \wedge \\
\qquad\quad (\ \forall i : \mathbb{N} \mid i \in Ready \bullet \\
\qquad\qquad (\exists j : \mathbb{N} \mid j > i \bullet j \in Ready) \wedge \\
\qquad\qquad C \subseteq t(\ i\ ) \wedge o \in t(\ i+1\ )\ )\ )
\end{array}
$$

LISTE DES DERNIERES PUBLICATIONS INTERNES PARUES A L'IRISA

PI 522 **PROGRAMMING BY MULTISET TRANSFORMATION**
Jean-Pierre BANATRE, Daniel LE METAYER
Mars 1990, 26 Pages.

PI 523 **GOTHIC MEMORY MANAGEMENT : A MULTIPROCESSOR SHARED SINGLE LEVEL STORE**
Béatrice MICHEL
Mars 1990, 20 Pages.

PI 524 **ORDER NOTIONS AND ATOMIC MULTICAST IN DISTRIBUTED SYSTEMS : A SHORT SURVEY**
Michel RAYNAL
Mars 1990, 18 Pages.

PI 525 **MULTI-SCALE AUTOREGRESSIVE PROCESSES**
Michèle BASSEVILLE, Albert BENVENISTE
Mars 1990, 136 Pages.

PI 526 **TRANSFORMATIONS PYRAMIDALES D'IMAGES NUMERIQUES**
Nadia BAAZIZ, Claude LABIT
Mars 1990, 100 Pages.

PI 527 **LE LANGAGE SIGNAL : UN EXEMPLE EN SEGMENTATION AUTOMATIQUE DE LA PAROLE CONTINUE**
Claude LE MAIRE
Mars 1990, 112 Pages.

PI 528 **CONDITIONAL REWRITE RULES AS AN ALGEBRAIC SEMANTICS OF PROCESSES**
Eric BADOUEL
Mars 1990, 46 Pages.

PI 529 **RESEAUX SYSTOLIQUES SPECIFIQUES A BASE DU PROCESSEUR API15C**
Patrice FRISON, Eric GAUTRIN, Dominique LAVENIER, Jean-Luc SCHARBARG
Mars 1990, 26 Pages.

PI 530 **SEMI-GRANULES AND SCHIELDING FOR OFF-LINE SCHEDULING**
Bernard LE GOFF, Paul LE GUERNIC, Julian ARAOZ DURAND
Avril 1990, 46 Pages.

PI 531 **DATA-FLOW TO VON NEUMANN : THE SIGNAL APPROACH**
Paul LE GUERNIC, Thierry GAUTIER
Avril 1990, 22 Pages.

PI 532 **OPERATIONAL SEMANTICS OF A DISTRIBUTED OBJECT-ORIENTED LANGUAGE AND ITS Z FORMAL SPECIFICATION**
Marc BENVENISTE
Avril 1990, 100 Pages.

Imprimé en France
par
. l'Institut National de Recherche en Informatique et en Automatique .

ISSN 0249-6399