Introduction of Mobility for Distributed Numerical Simulation Frameworks : a Formal Study

Grégory Haïk

▶ **To cite this version:**

Grégory Haïk. Introduction of Mobility for Distributed Numerical Simulation Frameworks : a Formal Study. [Research Report] lip6.2003.008, LIP6. 2004. hal-02545665

HAL Id: hal-02545665
https://hal.science/hal-02545665v1

Submitted on 17 Apr 2020

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

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

Introduction of Mobility for Distributed Numerical Simulation Frameworks : a Formal Study

Grégory Haïk*
LIP6 Research Report 2003–008
Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie
4, place Jussieu 75252 Paris Cedex 05, FRANCE

Abstract

This paper presents a formal study of automatic partitioning – or *introduction of mobility* – in the domain of Distributed Numerical Simulation Frameworks. Applied to this domain, mobility introduction consists in the following : in order to ease efficiency constraints on the design of numerical data interfaces, slices of programs are remotely executed on the hosts were data is located, so that a number of remote interactions are transformed into the same number of local interactions. This technique enables the designer of data interfaces to always provide the finest grain of inter-component interaction – which means a high level of component reusability – without sacrifying efficiency of the distributed application. This paper presents a formal model of mobility introduction for distributed systems and demonstrates a soundness thoerem of such a transformation.

Résumé

Ce papier présente une étude formelle du partitionnement automatique – ou *introduction de mobilité* – dans le domaine des Canevas de Simulation Numérique Répartie. Appliquée à ce domaine, l'introduction de mobilité repose sur le principe suivant : pour alléger les contraintes d'efficacité qui pèsent sur la conception des interfaces de données numériques, des tranches de programmes sont exécutées à distance sur les machines qui hébergent les données accédées, de sorte que les interactions distantes soient transformées en interactions locales. Cette technique permet au concepteur d'interfaces de toujours offrir la plus fine granularité d'interaction inter-composant – et donc un degré élevé de réutilisabilité des composants – sans sacrifier les performances de l'application répartie. Ce papier présente un modèle formel de l'introduction de mobilité dans les systèmes répartis et démontre un théorème de correction pour cette transformation.

Keywords : Numerical simulation, software engineering, integration frameworks, distributed numerical simulation, interface design, locality, mobility, automatic partitioning, automatic distribution, program analysis, program transformation.

Mots-clés : Simulation numérique, génie logiciel, canevas d'intégration, simulation numérique répartie, conception d'interface, localité, mobilité, partitionement automatique, répartition automatique, analyse de programme, transformation de programme.

*Gregory.Haik@lip6.fr

Contents

1 Introduction

Numerical simulation is an application domain where integration frameworks are expected to solve many practical problems. Indeed, numerical simulation raises difficult software engineering challenges and, as a result, simulation programmers are, still today, forced to adapt, modify and rewrite their programs even when inputs, usage scenario, or required outputs are very slightly changed. This practice is not only costly in terms of human work, it is also error prone. Moreover, adaptation of numerical simulation programs is hardened by the amount of manipulated data, the number of program lines involved, the required computation times, and the very long lifetime of numerical simulation programs. These challenges advocate for numerical simulation frameworks in which parts of the integration process is automatized.

Indeed, over the last decades, many research projects have addressed the issue of assisting engineers in programming and maintaining numerical programs : starting with linear algebra software libraries (such as LAPACK [15] and alike), researchers in software engineering of numerical simulation have promoted a rationalization of numerical programs development by using *programming* frameworks such as, for instance, POOMA [18] and Overture [2]. More recently, a need has arisen for *integration* frameworks of existing numerical programs : in such specialized integration frameworks, reusable software components encapsulate numerical solvers, mesh generators, 3D CAD systems and databases, using internal glue code. Managing such components in the consistent and uniform environment provided by the framework increases productivity of numerical applications programmers.

Integration frameworks are also expected to address the issues linked to distribution of resources. Indeed, large scale numerical simulations tend to use supercomputers, workstations clusters, large data servers, and powerful 3D graphics consoles. Thus, we envision *Distributed Numerical Simulation Integration Frameworks* (DNSIF) to be responsible, in addition to the requirements above, for remote communications between components and deployment of the componential application.

Today, very few numerical simulation frameworks efficiently support distribution, although reseachers from academics and industry has promoted several projects in that field, such as Salome RNTL project [19], Simulog's E-sim Factory [20], Paris and Sinus projects at INRIA [7, 14]. Indeed, such DNSIF projects are facing a difficult issue : while performances are critical in numerical simulation, conversely reusability of components is critical for the integration framework to remain meaningful; and unfortunately, these two constraints are often antagonistic. This paper explores a solution for addressing this antagonism in the specific field of *numerical data interfaces*.

Efficiency constraints require data interfaces to be specialized for specific usage, and conversely the best reusability is achieved when data is accessible at the finest grain, so that clients can choose their own way how to use that data. Although this problem also arises in a centralized setting, it is even more acute in a distributed environment. Indeed, refining the grain of data interfaces means, when data is accessed remotely, that the total amount of network messages increases. And because networks, operating systems, communication libraries, and the integration framework itself have their own latencies, refining the grain of remote interactions leads to lower global efficiency.

In this paper, we present an experimentation that consists in optimizing a program accessing data through a fine-grained interface (providing a good level of reusability) by introducing mobility primitives into the program. Mobility is used as a mean to increase locality between data holders and their client programs [5]. Improved locality leads to smaller amounts of network messages, thus reducing the latency overhead coming from the fine grain of the interfaces, as illustrated in figure 1. This way, we expect to reconcile reusable interfaces required by a meaningful

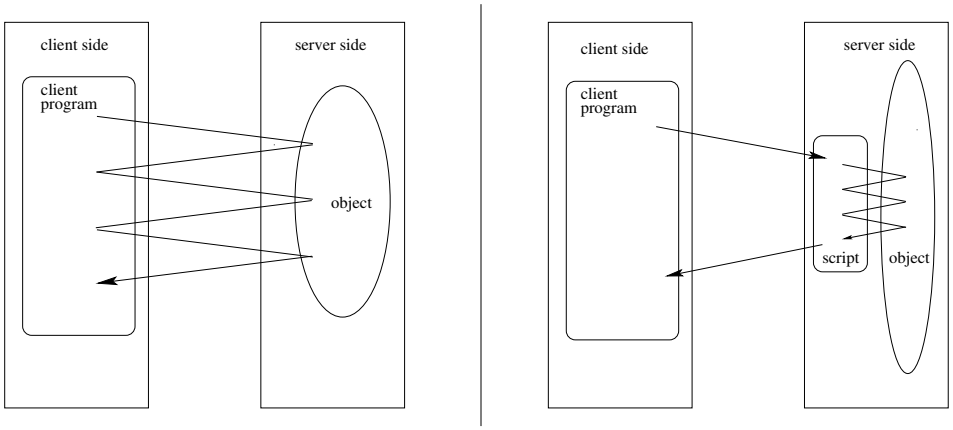

Figure 1: Moving client code to server host

integration framework with the global efficiency needed in numerical simulation.

Let us consider a program, part of a DNSF[1], that is client of remote data components. The program also accesses local resources such as the user console, the local file-system, OS-provided primitives (local timers, local host name, and alike)... The proposed optimization consists in moving slices of the program onto the computers hosting the remote data accessed in the slice to be made mobile. When a slice is moved, it has to bring with it associated data. Thus, we have to select wisely which slices are going to be moved, and which ones are going to remain on the client's side. Moreover, because the program also accesses local resources, we have to ensure that the transformation does not affect the global behavior of the program. In this paper, we only address the latter issue.

Intuitively, (i) mobility introduction is sound if (ii) moved statements do not access local resources. The next sections demonstrate this intuition. We first define what is a location independent statement, we make explicit the meaning of "mobility introduction" and "soundness", and finally demonstrates the link between the propositions (i) and (ii).

This report is structured as follows : first, we informally introduce in section 2 the goal and the method of the approach, and we review related work in section 3. Then, in section 4, we present the two languages for which our study is designed. In section 5, we define the notion of equivalence between program behaviors, and we link all the definitions of the formal model by a soundness theorem in section 6.

2 Informal Overview

In this section, we first show a short example that illustrates the principles of our experiment. Then, we informally describe the two toy languages we use, and finally we present the theorem of sound mobility introduction and the theoretical framework in which it is demonstrated.

2.1 Principles of the experiment

Figure 2 shows a simple program that computes and prints the sums of each columns of a remote matrix. It is made of two loops : the external one (variable i) is ranging

[1]Distributed Numerical Simulation Programming or Integration Framework

```
1 program foo {
2
3  declare int i;
4  declare int j;
5  declare Matrix remoteMatrix;
6
7  remoteMatrix = (Matrix)get("myMatrix");
8
9  i = 0;
10 while (j != remoteMatrix.length_x()) {
11   declare int s;
12   s = 0;
13   j = 0;
14   while (j != remoteMatrix.length_y()) {
15     s = s + remoteMatrix.get(i,j);
16     j = j + 1;
17     }
18   i = i + 1;
19   println(s);
20   };
21 };
```

Figure 2: A program printing sums of each column

over columns, whereas the internal one (variable j) over lines. For each i, the sum s is computed and finally printed.

The boxed statements, on lines 14-17, only contain simple arithmetic and boolean primitives (non-equality test, sums), and refer to a single remote object, namely `remoteMatrix`. Moreover, it is not linked to any local-dependent primitive such as `printInt`. Thus, these statements are not specifically bound to the computer they are executed on. They could be moved to another computer without modifying the global behavior of the program. If the target computer is chosen wisely, the performances of the program would be dramatically improved. Indeed, if these statements are executed on the computer hosting `remoteMatrix`, then a number of remote interactions are transformed in the same number of local interactions, which are - under certain circumstances - an order of magnitude faster.

Figure 3 illustrate the transformation from program `foo` to an equivalent program `foomob` where mobility primitives have been introduced. The statements boxed on figure 2 are sent (lines 18–23) to the host running the `remoteMatrix` object by a code mobility primitive (`remoteMatrix <- {...}`). This way, the internal loop only does local interactions with the matrix. In next sections, this is called *Code Mobility Introduction* (CMI). But since the associations between variables `x`, `y`, `s` `remoteMatrix` and their values were located on the client host (assuming that they are located were they have been declared), we also need to move the variables and their values so that the moved slice accesses the variables locally. This is done by introducing statements such as `migrate(h, v)` : the association between the variable `v` and its value is migrated to the designated host `h`. In next sections, this is called *Data Mobility Introduction* (DMI). Free variables of a moved statement are migrated before its remote execution (lines 14–17). They are migrated back to their original location after the remote execution (lines 24–28).

The remaining sections of this paper addresses the conditions under which such a transformation is valid.

2.2 The Need for Toy Languages

The formal model is not based on a well known programming language. We rather use two toy languages, that both have a very small syntax and a well defined operational semantics. We have chosen to study simple imperative languages instead of λ-calculus based toy languages because we want the formal model to be blatantly

```
1 program foomob {
2
3  declare int i;
4  declare int j;
5  declare Matrix remoteMatrix;
6  declare Host clientHost;
7
8  remoteMatrix = (Matrix)get("myMatrix");
9  i = 0;
10 while (j != remoteMatrix.length_x()) {
11   declare int s;
12   s = 0;
13   j = 0;
14   migrate(remoteMartix, i);
15   migrate(remoteMatrix, j);
16   migrate(remoteMatrix, s);
17   migrate(remoteMatrix, remoteMatrix);
18   remoteMatrix <- {
19   while (j != remoteMatrix.length_y()) {
20     s = s + remoteMatrix.get(i,j);
21     j = j + 1;
22     }
23   };
24   clientHost = getLocalHost();
25   migrate(clientHost, i);
26   migrate(clientHost, j);
27   migrate(clientHost, s);
28   migrate(clientHost, remoteMatrix);
29   i = i + 1;
30   printlnInt(s);
31   };
32 };
```

Figure 3: Equivalent program with mobility primitives

usable for analyzing the kind of languages used in the domain of numerical simulation (like Fortran, C/C++, python, Java), where fonctionnal programming is just unpopular. Moreover there is no need – in our experiment – for *autonomous, communicating* mobile agents such as Emerald [12], Obliq [3], Javanaise [10], Nomadic PICT [25], IBM Aglets [1], ObjectSpace Voyager [24] and alike : our model simply relies on remote code execution. Therefore, regarding the formal study, we have designed a specific model instead of extending existing mobility-based calculi such as the Join-calculus [9] or mobile ambients [4], that are far too much expressive for our needs. Thus, we defined Silfa (Simple Imperative Language For Analysis) and SilfaM (standing for Silfa + Mobility)[2]. The syntax used in the example above is very close to Silfa's. The only difference is that in Silfa, all the composite expressions are noted `prim(e1, e2,...)` or `obj.meth(e1, e2, ...)`. Thus, `i + 1` is noted in Silfa `plus(i,1)`, which reduces the number of reduction rules in Silfa's grammar.

Silfa is designed for interaction with CORBA objects. Thus, it can be seen as a very simple model of CorbaScript [13]. Silfa is limited to **synchronous method calls**. As a minimalistic language, its **values** are restricted to booleans, integers, floats, strings, and distributed object references. Silfa is **typed**, and does not provide type inference. Silfa is **imperative** : computation is achieved by successive store transformations, where a store maps variables to values. Silfa provides a set of **predefined primitives** (procedures and functions), including boolean, arithmetic, I/O, and naming operations[3], but does not support user-defined sub-programs. This is subject to future work (cf. section 7). Finally, programmers can define **blocks** for restricting the lexical scope of variables and parenthizing sequences of statements.

SilfaM adds to Silfa a code mobility primitive `(`*obj*`)<-`*s* and a data mobility primitive `Migrate(`*x*`,` *obj*`)`. The code mobility primitive sends a statement/slice *s* (pos-

[2] *Silfa* is also the phonetic reverse slang of *facile*, French word for *easy*.
[3] The Silfa library provides a simplified access to the COS Naming.

sibly including sub-statements) to a remote machine. The destination host is designated by a reference to a distributed object *obj* hosted by this machine. Similarly, the data mobility primitive moves the association between a variable x and its value to the host designated by the distributed object reference *obj*.

As stated before, the two toy languages have a well defined semantics. In this model, programs are bound to hosts where they are executed, but they manage a global store. Each host manages a local store, associating a local address to a value. The global store gives, for each global address, a pair composed of a local address and the host where the value is referenced. Moreover, a global addressing function maps variable names to global address. This way, any program can access the value of a variable through (i) the global addressing function and (ii) the global store. Then, from a formal point of view, the program in figure 3 would be workable even without the `migrate` operations, since values of `i`, `j`, `s`, and `remoteMatrix` are accessible from anywhere[4]. Still, the goal of the transformation is to avoid as much remote interactions as possible, and the `migrate` primitive illustrates a good implementation of the language (cf. section 7), so we introduced the data mobility primitive into SilfaM.

Silfa and SilfaM computation model is based, like in any sequential imperative language, on successive transformations of the global store. Here, in addition, programs produce a global trace that captures the locations where operations (primitives and remote object method calls) are actually performed. These traces have no functional meaning : they are only a semantics based record for defining equivalence between program behaviors and finally proving the soundness of the mobility introduction theorem.

Formal definitions of the environments, states, access functions and operational semantics are given in section 4.2.

2.3 Mobility introduction and Location-Dependence

Intuitively, mobility introduction is sound (i) if mobilized statements do not access local resources (ii). The next sections demonstrate this intuition. We first define what is a location independent statement, we make explicit the meanings of "mobility introduction" and "soundness", and finally demonstrates the link between the propositions (i) and (ii).

2.3.1 Location-independent statements

Location-independent statements of a program are defined as follows : we suppose that we statically know, in the set of all primitives, which ones actually accesses local resources. From this, we recursively define the local-independent statements by looking into each statement of the program and searching for location-dependent primitives. Formal definitions of location-independent primitives (def. 5.1) and location-independent statements (def. 8) are given in section 5.1.

2.3.2 Mobility introduction

Mobility introduction is divided into *Code Mobility Introduction* (CMI) and *Data Mobility Introduction* (DMI).

CMI consists in transforming a Silfa statement into a SilfaM statement that remotely executes it. For instance, an image of (`x := x + 1`) by CMI is :

[4]Line 17 in figure 3 means that the association bewteen the variable `remoteMatrix`, and its value – which is an object reference – is migrated to the host running `remoteMatrix`.

```
remoteObj <- {x := x + 1}
```

From the remote host running `remoteObj`, the inner expression is computed by accessing the value of `x` in the global store, which is finally updated to the new value of `x`. As long as `(+)` is an operation that has the same effect wherever it is executed, this transformation is sound.

DMI consists in transforming a Silfa statement into a SilfaM statement that first migrates a variable-value association to a remote host and then executes it. For instance, an image of (`x := 1`) by DMI is :

```
migrate(remoteObj, x); x := 1
```

2.3.3 CMI Soundness Theorem

The CMI soundness theorem asserts that applying CMI to local-independent statements of a program does not affect the global behavior of the program. This is shown by comparing the global behaviors of the original program (in Silfa) and its image SilfaM program by CMI, and showing that these two *global behaviors* are *equivalent*. In other words, we show that the CMI image of a Silfa program is a refinement of the original program.

We actually do not directly define global behavior. Instead, we define an equivalence on global traces (that capture among other things the inputs and outputs of the program) and on global states (representing values of user-defined variables), and the CMI soundness theorem links these equivalences with Silfa and SilfaM semantics.

Two global traces are said to be equivalent if (i) the same inputs have been given to both programs, (ii) the same outputs have been produced, and (iii) the location-dependant primitives where called on the same host, regardless of where the location-independent primitives have been called. This degree of liberty constitutes the only difference between this equivalence and equality. See section 5.3 for the formal definition.

Two global states are said to be equivalent if every variable is associated with the same value in both states. The difference with equality between states is that in two equivalent states, variable-value associations can be located on different hosts. See section 5.4 for the formal definition. The CMI soundness theorem (theorem 8) states the following :

For any object variable obj, granted that a Silfa statement s is location-independent, the SilfaM semantics of $(obj) \leftarrow s$ is equivalent to the Silfa semantics of s.

Therefore, it is semantically safe for a Silfa compiler to transform any statement of the input program, granted that the statement is location-independent - which is a syntactic check.

We do not prove directly the CMI soundness theorem. Instead, we prove a strong lemma (cf. lemma 7 in section 6.2.2), that directly leads to the CMI soundness theorem. This lemma states that if a statement is location-independent, then executing it on one host or another maintains the equivalence between pairs of states and traces. In other words, the host were a location independent statement is executed has no impact on the behavior of the program. From this lemma, applying the definition of code mobility primitive semantics leads to the CMI soundness theorem. The lemma is proven by induction over the statement syntactic structure. Let us review interesting cases of the induction :

- **Primitive call**
  This is the most interesting case regarding our problem. The argument is that if

a primitive call is location-independent, then so is its sub-expression; moreover the called primitive is a location-independent primitive. The first consequence guarantees that the induction hypothesis is applicable, and the second leads to equivalence between the two new events being entered in the traces.

- **Variable read**
  The two input states are equivalent, which means by definition that all variables are associated with the same value. Thus, both variable read operations lead to this value. Moreover traces are unchanged by a variable read expression.
- **Variable update**
  Here, we use a lemma that guarantees stability of equivalence between states when updating a variable with the same value from two different hosts.
- **Loop**
  Compared to all other cases, this one is a little bit more difficult, because we do not know how many times the inner statement is going to be executed. Thus, we build a sequence of states and a sequence of traces corresponding to the result of each execution of the inner statement. Then, we prove that stability of equivalences between states and traces are guaranteed from one execution of the inner loop to the next one (with an evaluation of the loop test in between). This recursively leads to the stability between the initial pair of states and traces to the last pair.

2.3.4 DMI Soundness Theorem

Informally, the DMI soundness theorem states that introducing a data mobility primitive anywhere in the program does not change the global behavior of the program. According to our model, this theorem is very easy to prove, it is a almost direct consequence of the model's definitions : we only rely on a simple lemma (lemma 3) asserting that performing a migrate operation on the state leads to an equivalent state.

Besides the next section, which reviews related works, the remaining of this paper provides formal definitions of the notions above and prove our two theorems.

3 Related Works

In a similar manner than *automatic parallelization* transparently analyzes and transforms sequential programs in order to discover opportunities for introducing parallelism [8], *automatic partitioning* (or *automatic distribution*) tends to analyze and transform centralized programs in order to discover opportunities for introducing distribution. During the last few years, some research has been conducted in this domain, as reviewed below.

JavaParty [16] is an extension of Java that automatically transforms regular Java classes into remotely accessible ones. It also provides migration of these classes' instances. Users specify which objects are to be made remote/mobile by tagging their classes with a new modifier (keyword `remote`). When an object is migrated, it accesses Java API on the host where it is executed : outputs and – more generally – location-dependent primitives are not examined and, regarding to our definition of soundness, JavaParty's object migration is unsound.

Doorastha [6] is quite similar to JavaParty, but differs from it on several issues : Java

syntax is not modified, and users insert pragmas in Java comments. Thus, compatibility with genuine Java compilers is preserved. In Doorastha's object migration model, calls to `System.out` are forwarded to the original JVM's console, which denotes consideration for the problem we are addressing. Still, in Doorastha, there is no tracking of *every* location-dependent primitives of the Java API, which leads to inconsistencies.

Pangaea [21] is a distribution system for Java applications that works with both JavaParty and Doorastha as back-ends. It is based on a static analysis of Java programs that computes an approximation of the runtime object graph. The Pangaea user specifies, through a graphical user interface, which objects are tied to which hosts. From this specification, the system computes a good placement of every other objects among the anticipated runtime population, by minimizing the number of repeating remote calls. Regarding to the soundness of mobility introduction, we do not consider that Pangaea is correct, because it relies on JavaPary or Doorastha.

J-Orchestra [23] and **Addistant** [22] are two similar automatic partitioning systems for Java bytecode. J-Orchestra distributes Java classes among the network (with the help of the user, like in Pangaea), using a runtime profiler for making placement decisions. In J-Orchestra, classes that contain platform-specific code in native format are considered anchored to their host : they can not be made mobile. A semi-automatic process ensures that no such class will eventually be ran on the wrong machine. Regarding to our notion of soundness, J-Orchestra is the only partitioning system that provides a sound mechanism for distributing code.

Coign [11] is a partitioning system for applications made of COM components. It combines typical usage scenarios, application and network profilers in order to make placement decisions, by scrutinizing inter-component communications. As Coign is designed for client-server distribution, it constrains GUI calls to remain on client side, while data storage calls are stuck to server side. This denotes consideration for the problem we are addressing. Still, this check is not (according to our model) general enough to be considered sound : for instance, we believe that Coign should also prevent to distribute components issuing calls the local DNS system (localhost, gethostbyname, etc).

In addition to the differences reviewed above between our work and related research, one should notice the three main contributions of our approach :

- First, our grain of mobility introduction is atomic : SilfaM primitives can make mobile every single statement of the original program, while systems reviewed above can only distribute COM components or Java objects. Our fine-grain mobility introduction enables to take advantage of automatic distribution for slices of code for which previous techniques would have been constrained by the including component/object.
- Second, we provide a formal framework (partially based on Queinnec's and De Roure's work on first-class environments [17]) that asserts the conditions of a sound automatic distribution. Other approaches focussed on real-world languages such as Java sources, Java bytecode or binaries. Thus, the validity of the program transformations reviewed could not easily be formally proven[5], and we even consider that the majority of them are unsound. We believe these two approaches – formal model of a proven transformation and unproven prototypes for real-world languages – are mutually profitable.

[5]Because of the technicalities involved in managing real-world languages in a formal manner.

- Finally, there is an important difference in the goal of related research and ours : previous works have focussed on the distribution of a *stand-alone*, centralized program that is to be executed on a network of computers. Distribution is seen as a motivation in itself, coming from the suboptimal usage of computer resources of laboratories and companies or from the fact that a particular application should be divided between a client side and a server side. On the contrary, we do not consider stand-alone programs to be candidates for transformation : we study programs that interact with other computers by RPC-like techniques. Here, distribution is not seen as a goal in itself, but rather as a mean to minimize the physical distance between a set of distributed resources and their client code. This is why Silfa and SilfaM are provided with a remote method call syntactic construct.

4 The Silfa and SilfaM Languages

This section describes our two toy languages. We first present the syntaxes, then we formally define their operational semantics.

4.1 Silfa and SilfaM Syntaxes

Silfa and SilfaM syntaxes are shown in figure 4.

The syntax used in the example of section 2 is very close to Silfa's. The only difference is that in Silfa, all the composite expressions are noted `prim(e1, e2, ...)` or `obj.meth(e1, e2, ...)`. Thus, `i + 1` is noted in Silfa `plus(i,1)`, which reduces the number of reduction rules in Silfa's grammar.

Silfa is designed for interaction with CORBA objects. Thus, it can be seen as a very simple model of CorbaScript [13]. Silfa is limited to **synchronous method calls**. As a minimalistic language, its **values** are restricted to booleans, integers, floats, strings, and distributed object references. Silfa is **typed**, and does not provide type inference. Silfa is **imperative** : computation is achieved by successive store transformations, where a store maps variables to values. Silfa provides a set of **predefined primitives** (procedures and functions), including boolean, arithmetic, I/O, and naming operations[6], but does not support user-defined sub-programs. This is subject to future work. Finally, programmers can define **blocks** for restricting the lexical scope of variables and parenthesizing sequences of statements.

SilfaM adds to Silfa a code mobility primitive `(`*obj*`)<-`*s* and a data mobility primitive `Migrate(`*x*`,` *obj*`)`. The code mobility primitive sends a statement/slice s (possibly including sub-statements) to a remote machine. The destination host is designated by a reference to a distributed object *obj* hosted by this machine. Similarly, the data mobility primitive moves the association between a variable x and its value to the host designated by the distributed object reference *obj*.

4.2 Operational Semantics

The operational semantics gives the meanings of Silfa and SilfaM programs. It is expressed as a set of functions, whose signatures are given in figure 5.

4.2.1 Domains

Formally, a state $\Sigma \in State$ associates a global address $\alpha \in Addr$ to a value : from α, one can retrieve, *via* a function $f \in Localization$ embedded in Σ, the machine

[6]The Silfa library provides a simplified access to the COS Naming.

$Program \rightarrow$ `program` $ProgId\ Block$ $Block \rightarrow$ `{` $Declarations\ Statement$ `}` $Declarations \rightarrow \varepsilon \mid Declaration$ `;` $Declarations$ $Declaration \rightarrow$ `declare` $Type\ VarId$ $Statements \rightarrow \varepsilon \mid Statement$ `;` $Statements$ $Statement \rightarrow$ `skip` $\mid RemoteCall \mid PrimitiveCall$ $\mid VarId$ `:=` $Expression \mid Block$ $\mid$ `if` $Expression$ `then` $Statement$ `else` $Statement$ $\mid$ `while (` $Expression$ `)` `do` $Statement$ $Expression \rightarrow VarId \mid Literal \mid PrimitiveCall$ $\mid RemoteCall \mid$ `(` $Type$ `)` $Expression$ $PrimitiveCall \rightarrow PrimId$ `(` $Expressions$ `)` $RemoteCall \rightarrow ObjId$ `.` $MethId$ `(` $Expressions$ `)` $Expressions \rightarrow \varepsilon \mid Expression$ $\mid Expressions$ `,` $Expression$ $Type \rightarrow$ `Boolean` $\mid$ `Int` $\mid$ `Float` $\mid$ `String` $\mid InterfaceId$ $*Id \rightarrow Identifier$
SilfaM grammar is augmented with the following rules : $Statement \rightarrow$ `(` $ObjId$ `)` `<-` $Statement$ $Statement \rightarrow$ `Migrate(` $VarId$ `,` $ObjId$`)`

Figure 4: Silfa and SilfaM Syntax

$h \in Host$ where the value is hosted and the local address $l \in Loc$ under which the value can be found. The second member of Σ ($g \in DistributedStore$) associates a local store $\sigma \in Store$ to each host. Thus, by combining the two members of Σ, one can retrieve the value associated in Σ with any address.

Silfa and SilfaM computation model is based, like in any sequential imperative language, on successive transformations of the global state ($\Sigma \in State$). Although this is sufficient for properly defining the behavior of program, we make programs to produce a global trace ($\Omega \in Trace$). This trace has no functional meaning : it is only a semantics based record for defining equivalence between program behaviors and finally proving the soundness of the mobility introduction theorem. A trace a sequence of external events ($\omega \in Event$). These events are either remote method calls (*Rcall*) or primitive calls (*Pcall*). Primitive calls are used to track (i) the locations where operations are actually performed and (ii) input/output values given to/produced by the program. See sections 4.2.4 and 4.2.5 for the meaning of the remaining functions.

4.2.2 Using Environments

Environments $\rho \in Env$ are sequences of maps between identifiers $id \in Identifier$ and global addresses $\alpha \in Addr$. Each element of an environment stores the declarations of a block. The function $Lookup(id, \rho)$, defined below, recursively inspects the environment ρ from the last one (deepest block) to the first one (program's block) in order to find the address α associated with id.

Definition 1 *Let* Lookup *be the function that maps $id \in Identifier$ to the relevant global address $\alpha \in Addr$ in the global environment ρ :*

$v \in Val$: Bool $\cup$ Int $\cup$ Float $\cup$ String $\cup$ ObjRef
$\alpha \in Addr : 0, 1, 2, ...$
$l \in Loc : 0, 1, 2, ...$
$h \in Host : 0, 1, 2, ...$
$\rho \in Env : (Id \rightarrow (Addr \cup \{\perp\}))^\star$
$\sigma \in Store : Loc \rightarrow Val$
$f \in Localization : Addr \rightarrow (Loc \times Host) \cup \{\perp\}$
$g \in DistributedStore : Host \rightarrow Store$
$\Sigma \in State : Localization \times DistributedStore$
$\omega \in Event : Pcall \oplus Rcall$
$Pcall : Prim \times Host \times Val$
$Rcall : Val \times Meth \times Val$
$\Omega \in Trace : (Event)^\star$
$\mathcal{P} : Program \times Host \rightarrow (State \times Trace)$
$\mathcal{D} : Declaration \times Env \rightarrow Env$
$\mathcal{S} : Statement \times Host \times Env \times State \times Trace \rightarrow State \times Trace$
$\mathcal{E} : Expression \times Host \times Env \times State \times Trace \rightarrow Val \times Trace$

Figure 5: Domains of Silfa and SilfaM Semantics

$$
\begin{array}{lcl}
Lookup & : & Identifier \times Env \rightarrow Addr \\
Lookup(id, []) & \triangleq & \perp \\
Lookup(id, \rho' : head) & \triangleq & \begin{cases} Lookup(id, \rho') & \text{if } head(id) = \perp \\ head(id) & \text{otherwise} \end{cases}
\end{array}
$$

Notation : $Lookup(id, \rho)$ is noted $\rho(id)$.

4.2.3 Using States

A state $\Sigma \in State$ is a structure that stores variable-value associations. Every association is actually stored in one of the local stores ($g \in DistributedStore$) of each host, and a localization function $f \in Localization$) is provided to retrieve the host and the local address of each global address. States Σ are manipulated by three functions : $Value$, $Update$, and $Migrate$. Well-formed states must have the property that from every address α is associated with its own couple (l, h) : the localization function is called $\perp$-injective. In this section, we define $\perp$-injectivity and the well-formed character of states. Then, we will define the tree access functions $Value$, $Update$ and $Migrate$, showing that the last two access functions conserve the well-formed property of states.

Definition 2 *A state $\Sigma = (f, g) \in State$ is* well-formed *iff its localization function f is $\perp$-injective, which is defined as follows :*

$$f(\alpha) = f(\beta) \neq \perp \implies \alpha = \beta$$

Function $Value(\alpha, \Sigma)$ combines the two members of the state Σ in order to get the value associated to the global address α. It is used in the semantics of expressions in order to evaluate variables - cf. rule **(var)** in figure 6.

Definition 3 *Let* Value *be the function that returns the value associated with address α in state Σ :*

$$
\begin{array}{lcl}
Value & : & Addr \times State \rightarrow Val \\
Value(\alpha, \Sigma) & \triangleq & \text{let } (f, g) = \Sigma \text{ in} \\
 & & \text{let } (l, h) = f(\alpha) \text{ in} \\
 & & g(h)(l)
\end{array}
$$

Let us define now a function *fresh* that returns a new location $l \in Loc$ that is unused in Σ for the host h :

Definition 4 *Let* $(fresh : State \times Host \rightarrow Loc)$ *be any function such that, for* $\Sigma = (f, g) \in State$ *and* $h \in Host$ *:*

$$\forall(\alpha \in Addr) \quad f(\alpha) \neq (fresh(\Sigma, h), h)$$

We suppose the existence of such a *fresh* function.

Function $Update(\alpha, v, \Sigma, h)$ is used by an assignment statement running on host h – cf. rule **(assign)** in figure 6 – for updating the address α to the value v in Σ. Function *Update* produces a new state with the appropriate bindings. The new state contains a new distributed store updated with a new local store function that binds α to v. If α has not been assigned yet, then a new local address l' is created on the host h where the assignment is executed. Note that the actual behavior is slightly different from the informal description given in section 2 : associations are stored on the host were the first assignment was performed, not where the variable has been declared.

Definition 5 *Let* Update *be the function that writes, from host* h *a value* v *into the state* Σ*, under address* α*, and returns the updated state :*

$$
\begin{array}{lcl}
Update & : & Addr \times Val \times State \times Host \rightarrow State \\
\\
Update(\alpha, v, \Sigma, h) & \triangleq & \begin{cases}
\text{let } l' = fresh(\Sigma, h) \text{ in} & \text{if } f(\alpha) = \bot \\
\quad (f[\alpha \mapsto (l', h)], g[h \mapsto g(h)[l' \mapsto v]]) & \\
\text{let } (loc, host) = f(\alpha) \text{ in} & \text{otherwise} \\
\quad (f, g[host \mapsto g(host)[loc \mapsto v]]) &
\end{cases} \\
\\
 & & \text{where } (f, g) = \Sigma
\end{array}
$$

Property 1 *Access function* Update *conserves the well-formed character of states : if* Σ *is well-formed, then so is* $\Sigma' = Update(\alpha, v, \Sigma, h)$*.*

Proof : Let $(f, g) = \Sigma$ and $(f', g') = \Sigma'$. According to the definition of $Update$, if $f(\alpha) \neq \bot$, then $f' = f$, so f' is $\bot$-injective. Now let us focus on the case where $f(\alpha) = \bot$.

From the definition of $Update$, $f' = f[\alpha \mapsto (l', h)]$ where $l' = fresh(\Sigma, h)$. Let $\beta, \gamma \in Addr$ such that $f'(\beta) = f'(\gamma) \neq \bot$. We show that $\beta = \gamma$:

- If $\beta = \alpha$ and $\gamma \neq \beta$, then $f'(\beta) = (l', h)$ and $f'(\gamma) = f(\gamma)$. Thus, $f(\gamma) = (l', h) = (fresh(\Sigma, h), h)$, which is not consitent with the definition of *fresh* (def. 4).
- Similarly, if $\gamma = \alpha$ and $\beta \neq \gamma$, then $f(\beta) = (fresh(\Sigma, h), h)$, which is not consitent with the definition of *fresh* as well.
- If $\beta \neq \alpha$ and $\gamma \neq \alpha$, then $f(\beta) = f(\gamma)$. Moreover, $f'(\beta) \neq \bot$ by hypothesis, so $f(\beta) \neq \bot$. Consequently, $\bot$-injectivity of f gives $\beta = \gamma$.

Finally, f' is $\bot$-injective regardless of $f(\alpha)$. As a consequence, Σ' is well-formed. □

Function $Migrate(\alpha, h, \Sigma)$ is used by SilfaM programs when they execute a data mobility primitive. It moves the binding between α and its value in Σ to the host h, and returns the modified state. It creates :

- a new local address l' on destination host h;
- a new localization function f' similar to f besides the association between α and (l', h);
- a new local store σ', similar to the one associated to h in Σ besides σ' now binds the newly created local address l' to the value of α in Σ;
- a new distributed store g', similar to the one of Σ besides it now uses σ'.

Function $Migrate$ finally returns the new state (f', g').

Definition 6 *Let* Migrate *be the function that moves the binding between α and its value in Σ to the host h, and returns the modified state.*

$$
\begin{array}{lcl}
Migrate & : & Addr \times Host \times State \rightarrow State \\
 & & \\
Migrate(\alpha, h, \Sigma) & \triangleq & \text{let } (f, g) = \Sigma \text{ in} \\
 & & \text{let } l' = fresh(\Sigma, h) \text{ in} \\
 & & \text{let } f' = f[\alpha \mapsto (l', h)] \text{ in} \\
 & & \text{let } \sigma' = g(h)[l' \mapsto Value(\alpha, \Sigma)] \text{ in} \\
 & & \text{let } g' = g[h \mapsto \sigma'] \text{ in} \\
 & & (f', g')
\end{array}
$$

Property 2 *Function* Mirgate *conserves the well-formed character of states : if Σ is well-formed, then so is $\Sigma' = Migrate(\alpha, h, \Sigma)$.*

Proof : Let $(f, g) = \Sigma$ and $(f', g') = \Sigma'$. According to the definition of $Migrate$, $f' = f[\alpha \mapsto (l', h)]$ where $l' = fresh(\Sigma, h)$. Let $\beta, \gamma \in Addr$ such that $f'(\beta) = f'(\gamma) \neq \perp$. We show that $\beta = \gamma$:

- If $\beta = \alpha$ and $\gamma \neq \beta$, then $f'(\beta) = (l', h)$ and $f'(\gamma) = f(\gamma)$. Thus, $f(\gamma) = (l', h) = (fresh(\Sigma, h), h)$, which is not consitent with the definition of *fresh* (def. 4).
- Similarly, if $\gamma = \alpha$ and $\beta \neq \gamma$, then $f(\beta) = (fresh(\Sigma, h), h)$, which is not consitent with the definition of *fresh* as well.
- If $\beta \neq \alpha$ and $\gamma \neq \alpha$, then $f(\beta) = f(\gamma)$. Moreover, $f'(\beta) \neq \perp$ by hypothesis, so $f(\beta) \neq \perp$. Consequently, $\perp$-injectivity of f gives $\beta = \gamma$.

Finally, f' is $\perp$-injective, and as a result, Σ' is well-formed. □

4.2.4 Silfa Semantics

Silfa semantics is detailed in figure 6. Semantics of programs, declarations, statements and expressions are respectively given by function $\mathcal{P}$, $\mathcal{D}$, $\mathcal{S}$, $\mathcal{E}$. Let us explain the rule **(prog)** :

$$\mathcal{P}[\![\texttt{program}\ progId\ block]\!]\ h \triangleq \mathcal{S}[\![block]\!]\ h\ \rho\ \Sigma\ \Omega$$

The host where the execution of the Silfa program takes place is designated by h. This rule means that the semantics of the whole program is equal to the semantics of its internal block in a null global environment $\rho = \varepsilon$, an empty global state $\Sigma = (\lambda\alpha.\perp), \lambda h.\lambda l.\perp)$ and a null trace of events $\Omega = \varepsilon$. Note that the empty global state is well-formed. Indeed, $\lambda\alpha.\perp$ is $\perp$-injective. Moreover, as long as the two functions that are used to modify states conserve the well-formed character of states, every intermediate states computed by any program are also well-formed. Thus, in the remaining of this paper, we do not refere explicitly to the well-formed character of states : every state is supposed to be well-formed.

Now let us examine rule **(assign)** :

$\mathcal{P}[\![$ `program` *progId block* $]\!]\ h \triangleq \mathcal{S}[\![block]\!]\ h\ \rho\ \Sigma\ \Omega$ where $\rho = \varepsilon; \Sigma = (\lambda\alpha.\bot), \lambda h.\lambda l.\bot); \Omega = \varepsilon$	**(prog)**
$\mathcal{D}[\![$`declare` *type* $x]\!]\rho \triangleq \rho[x \mapsto \alpha]$, α fresh	**(declare)**
$\mathcal{D}[\![d_1$ `;` $d_2]\!]\rho \triangleq \mathcal{D}[\![d_1]\!] \circ \mathcal{D}[\![d_2]\!]\rho$	**(decls)**
$\mathcal{S}[\![\{d\ s\}]\!]\ h\ \rho\ \Sigma\ \Omega \triangleq \mathcal{S}[\![s]\!]\ h\ (\rho : \mathcal{D}[\![d]\!]\rho)\ \Sigma\ \Omega$	**(block)**
$\mathcal{S}[\![$`skip`$]\!]\ h\ \rho\ \Sigma\ \Omega \triangleq (\Sigma, \Omega)$	**(skip)**
$\mathcal{S}[\![x$ `:=`$e]\!]\ h\ \rho\ \Sigma\ \Omega \triangleq (Update(\rho(x), v, \Sigma, h), \Omega')$ where $\mathcal{E}[\![e]\!]\ h\ \rho\ \Sigma\ \Omega = (v, \Omega')$	**(assign)**
$\mathcal{S}[\![p(e)]\!]\ h\ \rho\ \Sigma\ \Omega \triangleq (\Sigma, \Omega' : Pcall(p, h, v))$ where $\mathcal{E}[\![e]\!]\ h\ \rho\ \Sigma\ \Omega = (v, \Omega')$	**(s-pcall)**
$\mathcal{S}[\![e_1.m(e_2)]\!]\ h\ \rho\ \Sigma\ \Omega \triangleq (\Sigma, \Omega_2 : Rcall(v_1, m, v_2))$ where $\mathcal{E}[\![e_1]\!]\ h\ \rho\ \Sigma\ \Omega = (v_1, \Omega_1)$ and $\mathcal{E}[\![e_2]\!]\ h\ \rho\ \Sigma\ \Omega_1 = (v_2, \Omega_2)$	**(s-rcall)**
$\mathcal{S}[\![s_1$ `;` $s_2]\!]\ h\ \rho\ \Sigma\ \Omega \triangleq\ \mathcal{S}[\![s_2]\!]\ h\ \rho\ \Sigma'\ \Omega'$ where $\mathcal{S}[\![s_1]\!]\ h\ \rho\ \Sigma\ \Omega = (\Sigma', \Omega')$	**(seq)**
$\mathcal{S}[\![$`if` e `then` s_1 `else` $s_2]\!]\ h\ \rho\ \Sigma\ \Omega \triangleq$ if $test$ then $\mathcal{S}[\![s_1]\!]\ h\ \rho\ \Sigma\ \Omega'$ else $\mathcal{S}[\![s_2]\!]\ h\ \rho\ \Sigma\ \Omega'$ where $\mathcal{E}[\![e]\!]\ h\ \rho\ \Sigma\ \Omega = (test, \Omega')$	**(if)**
$\mathcal{S}[\![$`while` e `do` $s]\!]\ h\ \rho\ \Sigma\ \Omega \triangleq$ if $test$ then $\mathcal{S}[\![s;$ `while` e `do` $s]\!]\ h\ \rho\ \Sigma\ \Omega'$ else (Σ, Ω') where $\mathcal{E}[\![e]\!]\ h\ \rho\ \Sigma\ \Omega = (test, \Omega')$	**(while)**
$\mathcal{E}[\![p(e)]\!]\ h\ \rho\ \Sigma\ \Omega \triangleq (p(v), \Omega' : Pcall(p, h, v))$ where $\mathcal{E}[\![e]\!]\ h\ \rho\ \Sigma\ \Omega = (v, \Omega')$	**(e-pcall)**
$\mathcal{E}[\![e_1.m(e_2)]\!]\ h\ \rho\ \Sigma\ \Omega \triangleq (o.m(v), \Omega_2 : Rcall(o, m, v))$ where $\mathcal{E}[\![e_1]\!]\ h\ \rho\ \Sigma\ \Omega = (o, \Omega_1)$ and $\mathcal{E}[\![e_2]\!]\ h\ \rho\ \Sigma\ \Omega_1 = (v, \Omega_2)$	**(e-rcall)**
$\mathcal{E}[\![x]\!]\ h\ \rho\ \Sigma\ \Omega \triangleq (Value(\rho(x), \Sigma), \Omega)$	**(var)**
$\mathcal{E}[\![v]\!]\ h\ \rho\ \Sigma\ \Omega \triangleq (v, \Omega)$	**(lit)**
SilfaM semantics is augmented with the following rules :	
$\mathcal{S}[\![(obj) \leftarrow s]\!]\ h\ \rho\ \Sigma\ \Omega \triangleq \mathcal{S}[\![s]\!]\ host(Value(\rho(obj), \Sigma))\ \rho\ \Sigma\ \Omega$	**(code-mob)**
$\mathcal{S}[\![$`migrate(`x`,` obj`)`$]\!]\ h\ \rho\ \Sigma\ \Omega \triangleq (Migrate(\rho(x), host(Value(\rho(obj), \Sigma)), \Sigma), \Omega)$	**(data-mob)**

Figure 6: Silfa and SilfaM Semantics

$$\mathcal{S}[\![x\texttt{:=}e]\!]\ h\ \rho\ \Sigma\ \Omega \triangleq (Update(\rho(x), v, \Sigma, h), \Omega')$$
$$\text{where } \mathcal{E}[\![e]\!]\ h\ \rho\ \Sigma\ \Omega = (v, \Omega')$$

It means that before executing the assignment, one should first evaluate the inner expression e. We assume that evaluation of e gives, in the conditions (h, ρ, Σ, Ω) in which the assignment is executed, a value v and a new trace Ω' containing the eventual new events performed by e. Then, the assignment updates the global state by associating `x` to the computed value v, and gives the new trace Ω'.

4.2.5 SilfaM Semantics

SilfaM semantics is defined on figure 6. For a code mobility primitive – rule **(code-mob)** –, the inner statement is executed on the host computed from the obj. Environment rho,, state Σ and trace Ω are kept unchanged. The host where obj is running is computed as follows : $\rho(obj)$ is the global adress of the object reference. The concrete object reference value is obtained by access function $Value$. From this object reference, we get its host by an auxiliary function, namely $host : ObjRef \rightarrow Host$. The inner satement s is executed on $host(Value(\rho(obj), \Sigma))$:

$$\mathcal{S}[\![(obj) \leftarrow s]\!]\ h\ \rho\ \Sigma\ \Omega \triangleq \mathcal{S}[\![s]\!]\ host(Value(\rho(obj), \Sigma))\ \rho\ \Sigma\ \Omega$$

For a data mobility primitive – rule **(data-mob)–**, the state is updated with the *Migrate* function described in section 4.2.3. No further statement is to be computed (data mobility primitive is a leaf statement, unlike code mobility primitive). Global trace Ω is kept unchanged, and the new global state is the result of migrating the global address of x, which is $\rho(x)$, to the destination host, which is $host(Value(\rho(obj), \Sigma))$ like in **(code-mob)** :

$$\mathcal{S}[\![\texttt{migrate(}x\texttt{,}obj\texttt{)}]\!]\ h\ \rho\ \Sigma\ \Omega \triangleq (Migrate(\rho(x), host(Value(\rho(obj), \Sigma)), \Sigma), \Omega)$$

5 Equivalence of Semantics

Intuitively, mobility introduction is sound (i) if mobilized statements do not access local resources (ii). The next sections demonstrate this intuition. We first define what is a location independent statement, we make explicit the meanings of "mobility introduction", "soundness", and finally demonstrate the link between the propositions (i) and (ii).

The CMI soundness theorem, presented in section 6, asserts that applying CMI to local-independent statements of a program does not affect the global behavior of the program. This is shown by comparing the global behaviors of the original program (in Silfa) and its image SilfaM program by CMI, and showing that these two *global behaviors* are *equivalent*. In other words, we show that the CMI image of a Silfa program is a refinement of the original program.

We actually do not directly define global behavior. Instead, we define an equivalence on global traces (that capture among other things the inputs and outputs of the program) and on global states (representing values of user-defined variables), and the CMI soundness theorem links these equivalences with Silfa and SilfaM semantics.

5.1 Location-Independence

Location-independent statements of a program are defined as follows : we suppose that we statically know, in the set of all primitives, which ones actually access local resources. This set is called LIP, for Location-Independent Primitives (such as boolean and arithmetic operations), separating them from location-dependant primitives such as I/O operations, system calls like reading the system clock, reading the local host name, etc...

Definition 7 *Let $LIP \subset Prim$ be the location-independent primitives.*

Example : $(+) \in LIP$ $(=) \in LIP$ $printInt \notin LIP$.

From this set LIP, we recursively define the *location-independent statements and expressions* by looking into each statement of the program and searching for location-dependent primitives. If a statement or an expression a is location-independent, then we say that the predicate $LI(a)$ is true.

Definition 8 *A syntactic construct $a \in Statement \cup Expression$ is* location-independent, *noted $LI(a)$, if and only if one of the following holds :*

$a \in VarId \qquad a \in Literal$
$a = prim(e_1, ..., e_n)$ with $prim \in LIP \wedge LI(e_1) \wedge ... \wedge LI(e_n)$
$a = obj.meth(e_1, ..., e_n)$ with $LI(e_1) \wedge ... \wedge LI(e_n)$
$a =$ `if` e `then` s_1 `else` s_2 with $LI(e) \wedge LI(s_1) \wedge LI(s_2)$
... and so on for other statements...

5.2 Equivalence of Events

Two global traces are said to be equivalent if (i) the same inputs have been given to both programs, (ii) the same outputs have been produced, and (iii) the location-dependant primitives where called on the same host, regardless of where the location-independent primitives have been called. This degree of liberty constitutes the only difference between this equivalence and equality.
We first define an equivalence relation between events, then we propagate this definition on traces, that are sequences of events.

Definition 9 $\omega_1, \omega_2 \in Event$ *are* equivalent *(noted* $\omega_1 \equiv \omega_2$*) if, and only if :*

$$(\omega_1 = \omega_2) \vee \exists(p, v, h, h') \begin{cases} \omega_1 = Pcall(p, h, v) \\ \omega_2 = Pcall(p, h', v) \\ p \in LIP \end{cases}$$

Remark : We suppose, in the definition above, that we are provided with an equality relation between values, including distributed object references. Here, we consider equality over the referenced objects : two distributed object references are equal if they point to the same object.

5.3 Equivalence of Traces

The definition below propagates the equivalence of events to an equivalence of traces.

Definition 10 $\Omega_1, \Omega_2 \in Trace$ *are* equivalent *(noted* $\Omega_1 \equiv \Omega_2$*) if, and only if one of the following holds :*

(i) $\quad \Omega_1 = \Omega_2 = \varepsilon$

(ii) $\quad \exists(\omega_1, \omega_2, \Omega'_1, \Omega'_2) \begin{cases} \omega_1 \equiv \omega_2 \\ \Omega'_1 \equiv \Omega'_2 \end{cases}$ with $\Omega_1 = \Omega'_1 : \omega_1$ and $\Omega_2 = \Omega'_2 : \omega_2$

Notation : $(v, \Omega) \equiv (v', \Omega')$ iff $(v = v') \wedge (\Omega \equiv \Omega')$.

5.4 Equivalence of States

Two global states are said to be equivalent if every variable is associated with the same value in both states. The difference with equality between states is that in two equivalent states, variable-value associations can be located on different hosts.

Definition 11 $\Sigma_1, \Sigma_2 \in State$ *are* equivalent *(noted* $\Sigma_1 \equiv \Sigma_2$*) if, and only if :*

$$\forall(\alpha) Value(\alpha, \Sigma_1) = Value(\alpha, \Sigma_2)$$

Notation : $(\Sigma, \Omega) \equiv (\Sigma', \Omega')$ iff $(\Sigma \equiv \Sigma') \wedge (\Omega \equiv \Omega')$

We will now show two technical lemmas asserting that migrating a variable-value association lead to an equivalent state, and that updating an address with the same value from two different hosts conserves the state equivalence.

The following lemma guarantees that when a *Migrate* function is applied to a state, then the value bound to the address subject to migration – as well as all other addresses – is kept unchanged in the returned state. This lemma just checks that definitions above are consistent one with each-other. Its proof consists in unrolling the definitions 3 and 6 above. This lemma is used in the proof if DMI soundness theorem (cf. section 6.1).

Lemma 3 *Let* $\alpha \in Addr, h \in Host, \Sigma \in State$ *and* $\Sigma' = Migrate(\alpha, h, \Sigma)$. *Then* $\Sigma' \equiv \Sigma$. *In other words :*

$$\forall(\beta) \quad Value(\beta, \Sigma') = Value(\beta, \Sigma)$$

Proof :

$$\begin{array}{rcll}
Value(\beta,\Sigma') & = & \text{let } (f,g) = \Sigma' \text{ in} & \text{(def. 3)} \\
 & & \text{let } (l_1,h_1) = f(\alpha) \text{ in} & \\
 & & g(h_1)(l_1) & \\
 & = & \text{let } (f,g) = Migrate(\alpha,h,\Sigma) \text{ in} & \text{(hyp.)} \\
 & & \text{let } (l_1,h_1) = f(\beta) \text{ in} & \\
 & & g(h_1)(l_1) & \\
 & = & \text{let } (f',g') = \Sigma \text{ in} & \text{(def. 6)} \\
 & & \text{let } l' = fresh(\Sigma,h) \text{ in} & \\
 & & \text{let } f'' = f'[\alpha \mapsto (l',h)] \text{ in} & \\
 & & \text{let } \sigma' = g'(h)[l' \mapsto Value(\alpha,\Sigma)] \text{ in} & \\
 & & \text{let } g'' = g'[h \mapsto \sigma'] \text{ in} & \\
 & & \text{let } (f,g) = (f'',g'') \text{ in} & \\
 & & \text{let } (l_1,h_1) = f(\beta) \text{ in} & \\
 & & g(h_1)(l_1) & \\
 & = & \text{let } (f',g') = \Sigma \text{ in} & \text{(subst. } f,g\text{)} \\
 & & \text{let } l' = fresh(\Sigma,h) \text{ in} & \\
 & & \text{let } f'' = f'[\alpha \mapsto (l',h)] \text{ in} & \\
 & & \text{let } \sigma' = g'(h)[l' \mapsto Value(\alpha,\Sigma)] \text{ in} & \\
 & & \text{let } g'' = g'[h \mapsto \sigma'] \text{ in} & \\
 & & \text{let } (l_1,h_1) = f''(\beta) \text{ in} & \\
 & & g''(h_1)(l_1) & \\
 & = & \text{let } (f',g') = \Sigma \text{ in} & \text{(subst. } f'',g''\text{)} \\
 & & \text{let } l' = fresh(\Sigma,h) \text{ in} & \\
 & & \text{let } \sigma' = g'(h)[l' \mapsto Value(\alpha,\Sigma)] \text{ in} & \\
 & & \text{let } (l_1,h_1) = f'[\alpha \mapsto (l',h)](\beta) \text{ in} & \\
 & & g'[h \mapsto \sigma'](h_1)(l_1) &
\end{array}$$

Case $\alpha \neq \beta$

$$\begin{array}{rcll}
Value(\beta,\Sigma') & = & \text{let } (f',g') = \Sigma \text{ in} & \text{(extension [])} \\
 & & \text{let } l' = fresh(\Sigma,h) \text{ in} & \\
 & & \text{let } \sigma' = g'(h)[l' \mapsto Value(\alpha,\Sigma)] \text{ in} & \\
 & & \text{let } (l_1,h_1) = f'(\beta) \text{ in} & \\
 & & g'[h \mapsto \sigma'](h_1)(l_1) &
\end{array}$$

Sub-case $\alpha \neq \beta, h \neq h_1$

$$
\begin{array}{lcll}
Value(\beta, \Sigma') & = & \text{let } (f', g') = \Sigma \text{ in} & \text{(extension [])} \\
 & & \text{let } l' = fresh(\Sigma, h) \text{ in} & \\
 & & \text{let } \sigma' = g'(h)[l' \mapsto Value(\alpha, \Sigma)] \text{ in} & \\
 & & \text{let } (l_1, h_1) = f'(\beta) \text{ in} & \\
 & & g'(h_1)(l_1) & \\
 & = & \text{let } (f', g') = \Sigma \text{ in} & \text{(unused } l', \sigma') \\
 & & \text{let } (l_1, h_1) = f'(\beta) \text{ in} & \\
 & & g'(h_1)(l_1) & \\
 & = & Value(\beta, \Sigma) & \text{(def. 3)}
\end{array}
$$

Sub-case $\alpha \neq \beta, h = h_1$

$$
\begin{array}{lcll}
Value(\beta, \Sigma') & = & \text{let } (f', g') = \Sigma \text{ in} & (\beta\text{-red.}) \\
 & & \text{let } l' = fresh(\Sigma, h) \text{ in} & \\
 & & \text{let } \sigma' = g'(h)[l' \mapsto Value(\alpha, \Sigma)] \text{ in} & \\
 & & \text{let } (l_1, h_1) = f'(\beta) \text{ in} & \\
 & & \sigma'(l_1) & \\
 & = & \text{let } (f', g') = \Sigma \text{ in} & \text{(subst. } \sigma') \\
 & & \text{let } l' = fresh(\Sigma, h) \text{ in} & \\
 & & \text{let } (l_1, h_1) = f'(\beta) \text{ in} & \\
 & & g'(h)[l' \mapsto Value(\alpha, \Sigma)](l_1) &
\end{array}
$$

$l' = l_1$ is not possible because it would mean $f'(\beta) = (fresh(\Sigma, h), h)$, which is a contradiction with the definition of fresh (def. 4). So $l' \neq l_1$. Therefore,

$$
\begin{array}{lcll}
Value(\beta, \Sigma') & = & \text{let } (f', g') = \Sigma \text{ in} & \text{(extension [])} \\
 & & \text{let } l' = fresh(\Sigma, h) \text{ in} & \\
 & & \text{let } (l_1, h_1) = f'(\beta) \text{ in} & \\
 & & g'(h)(l_1) & \\
 & = & \text{let } (f', g') = \Sigma \text{ in} & \text{(unused } l') \\
 & & \text{let } (l_1, h_1) = f'(\beta) \text{ in} & \\
 & & g'(h)(l_1) & \\
 & = & Value(\beta, \Sigma) & \text{(def. 3)}
\end{array}
$$

Case $\alpha = \beta$

$$
\begin{array}{lcll}
Value(\beta, \Sigma') & = & \text{let } (f', g') = \Sigma \text{ in} & (\beta\text{-red.}) \\
 & & \text{let } l' = fresh(\Sigma, h) \text{ in} & \\
 & & \text{let } \sigma' = g'(h)[l' \mapsto Value(\alpha, \Sigma)] \text{ in} & \\
 & & \text{let } (l_1, h_1) = (l', h) \text{ in} & \\
 & & g'[h \mapsto \sigma'](h_1)(l_1) & \\
 & = & \text{let } (f', g') = \Sigma \text{ in} & \text{(subst. } l_1, h_1) \\
 & & \text{let } l' = fresh(\Sigma, h) \text{ in} & \\
 & & \text{let } \sigma' = g'(h)[l' \mapsto Value(\alpha, \Sigma)] \text{ in} & \\
 & & g'[h \mapsto \sigma'](h)(l') & \\
 & = & \text{let } (f', g') = \Sigma \text{ in} & (\beta\text{-red.}) \\
 & & \text{let } l' = fresh(\Sigma, h) \text{ in} & \\
 & & \text{let } \sigma' = g'(h)[l' \mapsto Value(\alpha, \Sigma)] \text{ in} & \\
 & & \sigma'(l') & \\
 & = & \text{let } (f', g') = \Sigma \text{ in} & \text{(subst. } \sigma') \\
 & & \text{let } l' = fresh(\Sigma, h) \text{ in} & \\
 & & g'(h)[l' \mapsto Value(\alpha, \Sigma)](l') & \\
 & = & Value(\alpha, \Sigma) & (\beta\text{-red.}) \\
 & = & Value(\beta, \Sigma) \quad \square & \text{(case hypothesis)}
\end{array}
$$

Lemma 4 *Updating an address from two different hosts with the same value conserves the state equivalence :*

$$\Sigma \equiv \Sigma' \implies Update(\alpha, v, \Sigma, h) \equiv Update(\alpha, v, \Sigma', h')$$

Proof : Let $\Sigma_1 = Update(\alpha, v, \Sigma, h)$ and $\Sigma_2 = Update(\alpha, v, \Sigma', h')$. Let $(f, g) = \Sigma$ and $(f', g') = \Sigma'$. Let $\beta \in Addr$. We will show that in any case $Value(\beta, \Sigma_1) = Value(\beta, \Sigma_2)$. The outline of the proof is that if β is the updated address α, then its value is the updated value (in Σ_1 as well as in Σ_2), and otherwise $Value(\beta, \Sigma_1) = Value(\beta, \Sigma)$ and $Value(\beta, \Sigma_2) = Value(\beta, \Sigma')$ – which are equal because $\Sigma \equiv \Sigma'$.

Case $\beta = \alpha$, $f(\alpha) = \bot$

According to definitions of $Value$ (def. 3) and $Update$ (def. 5), $Value(\beta, \Sigma_1) = Value(\alpha, \Sigma_1) = g[h \mapsto g(h)[l' \mapsto v]](h_n)(l_n)$, where $l' = fresh(\Sigma, h)$ and $(l_n, h_n) = f[\alpha \mapsto (l', h)](\alpha) = (l', h)$. Thus, $Value(\beta, \Sigma_1) = g(h)[h \mapsto g(h)[l' \mapsto v]](h)(l') = g(h)[l' \mapsto v](l') = v$.

Case $\beta = \alpha$, $f(\alpha) = (loc, host)$

$Value(\beta, \Sigma_1) = Value(\alpha, \Sigma_1) = g[host \mapsto g(host)[loc \mapsto v]](host)(loc) = g(host)[loc \mapsto v](loc) = v$.

The two previous cases lead to the conclusion that if $\beta = \alpha$, then $Value(\beta, \Sigma_1) = v$. Now let us check that the same holds for Σ_2 :

Case $\beta = \alpha$, $f'(\alpha) = \bot$

$Value(\beta, \Sigma_2) = Value(\alpha, \Sigma_2) = g'[h' \mapsto g'(h')[l'' \mapsto v]](h'_n)(l'_n)$, where $l'' = fresh(\Sigma', h')$ and $(l'_n, h'_n) = f'[\alpha \mapsto (l'', h')](\alpha) = (l'', h')$. Thus, $Value(\beta, \Sigma_2) = g'(h')[h' \mapsto g'(h')[l'' \mapsto v]](h')(l'') = g'(h')[l'' \mapsto v](l'') = v$.

Case $\beta = \alpha$, $f'(\alpha) = (loc', host')$

$Value(\beta, \Sigma_2) = Value(\alpha, \Sigma_2) = g'[host' \mapsto g'(host')[loc' \mapsto v]](host')(loc') = g(host')[loc' \mapsto v](loc') = v$.

The two previous cases lead to the conclusion that if $\beta = \alpha$, then $Value(\beta, \Sigma_2) = v$. Thus, if $\beta = \alpha$, then $Value(\beta, \Sigma_1) = Value(\beta, \Sigma_2)$. Now let us check that equality also holds if $\beta \neq \alpha$.

Case $\beta \neq \alpha$, $f(\alpha) = \bot$

$Value(\beta, \Sigma_1) = g[h \mapsto g(h)[l' \mapsto v]](h_\beta)(l_\beta)$, where $l' = fresh(\Sigma, h)$ and $(l_\beta, h_\beta) = f[\alpha \mapsto (l', h)](\beta) = f(\beta)$. If $h_\beta = h$, then $Value(\beta, \Sigma_1) = g(h_\beta)[l' \mapsto v](l_\beta)$. But l' can not be equal to l_β because otherwise we would have $f(\beta) = (fresh(\Sigma, h), h)$, which is in contradiction with definition of $fresh$ (def. 4). Thus, $l_\beta \neq l'$. As a consequence, if $h_\beta = h$, then $Value(\beta, \Sigma_1) = g(h_\beta)(l_\beta) = Value(\beta, \Sigma)$ according to the definition of $Value$ (def. 3). Otherwise (if $h_\beta \neq h$), $Value(\beta, \Sigma_1) = g(h_\beta)(l_\beta) = Value(\beta, \Sigma)$.

To sum up, if $\beta \neq \alpha$ and $f(\alpha) = \bot$, then $Value(\beta, \Sigma_1) = Value(\beta, \Sigma)$.

Case $\beta \neq \alpha$, $f(\alpha) = (l_\alpha, h_\alpha)$

$Value(\beta, \Sigma_1) = g[h_\alpha \mapsto g(h_\alpha)[l_\alpha \mapsto v]](h_\beta)(l_\beta)$, where $(l_\beta, h_\beta) = f(\beta)$. If $h_\beta = h_\alpha$, then $Value(\beta, \Sigma_1) = g(h_\beta)[l_\alpha \mapsto v](l_\beta)$. But as long as Σ is well-formed, then f is $\bot$-injective and as a result $l_\beta \neq l_\alpha$. Consequently, $Value(\beta, \Sigma_1) = g(h_\beta)(l_\beta) = Value(\beta, \Sigma)$. Otherwise (if $h_\beta \neq h_\alpha$), then $Value(\beta, \Sigma_1) = g(h_\beta)(l_\beta) = Value(\beta, \Sigma)$.

The two previous cases lead to the conclusion that if $\beta \neq \alpha$, then $Value(\beta, \Sigma_1) = Value(\beta, \Sigma)$. Similarly, we show that – under the same condition – $Value(\beta, \Sigma_2) = Value(\beta, \Sigma')$. Moreover, $\Sigma \equiv \Sigma'$, so $Value(\beta, \Sigma) = Value(\beta, \Sigma')$. Consequently, if $\beta \neq \alpha$, then $Value(\beta, \Sigma_1) = Value(\beta, \Sigma_2)$.

Finally, for every β, $Value(\beta, \Sigma_1) = Value(\beta, \Sigma_2)$, which means by definition that $\Sigma_1 \equiv \Sigma_2$. $\square$

6 Soundness of Mobility Introduction

Mobility introduction is divided into *code mobility introduction* (CMI) and *data mobility introduction* (DMI).
CMI consists in transforming a Silfa statement into a SilfaM statement that remotely executes it. For instance, an image of (`x := 1`) by CMI is (`remoteObj <- {x := 1}`).
DMI consists in transforming a Silfa statement into a SilfaM statement that first migrates a variable-value association to a remote host and then executes it. For instance, an image of (`x := 1`) by DMI is (`migrate(remoteObj, x); x := 1`).

6.1 Data Mobility Introduction Soundness Theorem

The following theorem states that for any statement s, for any variable x, for any variable obj bound to an object reference, the semantics of `migrate(`obj, x`);`s is equivalent to the semantics of s.

Theorem 5 *Let s be a Silfa statement. Let $h \in Host$. Let $obj, x \in Var$. Let ρ, Σ be such that $Value(\rho(obj), \Sigma) \in Obj$. Let $\Omega \in Trace$. Then :*

$$\mathcal{S}[\![\texttt{migrate(}obj, x\texttt{);}\ s]\!]\ h\ \rho\ \Sigma\ \Omega \equiv \mathcal{S}[\![s]\!]\ h\ \rho\ \Sigma\ \Omega$$

Proof : Let $\alpha = \rho(x)$ and $h' = host(Value(\rho(obj), \Sigma))$. According to the definition of the SilfaM semantics of data mobility **(data-mob)**, we have :

$$\mathcal{S}[\![\texttt{migrate(}obj, x\texttt{)}\ ]\!]\ h\ \rho\ \Sigma\ \Omega = (Migrate(\alpha, h', \Sigma), \Omega)$$

Thus, from SilfaM semantics of sequence **(seq)** :

$$\mathcal{S}[\![\texttt{migrate(}obj, x\texttt{);}\ s]\!]\ h\ \rho\ \Sigma\ \Omega = \mathcal{S}[\![\mathrm{s}]\!]\ h\ \rho\ (Migrate(\alpha, h', \Sigma)\ \Omega$$

From lemma 3, for every $\beta \in Addr$, $Value(\beta, Migrate(\alpha, h', \Sigma)) = Value(\beta, \Sigma)$, which means (by definition 11) that $Migrate(\alpha, h', \Sigma) \equiv \Sigma$. Thus :

$$\mathcal{S}[\![\texttt{migrate(}obj, x\texttt{);}\ s]\!]\ h\ \rho\ \Sigma\ \Omega \equiv \mathcal{S}[\![s]\!]\ h\ \rho\ \Sigma\ \Omega \quad \square$$

6.2 Code Mobility Introduction Soundness Theorem

The CMI soundness theorem (theorem 8) states the following : for any object variable obj, granted that a Silfa statement s, is location-independent, the SilfaM semantics of $(obj) \leftarrow s$ is equivalent to the Silfa semantics of s. Therefore, it is semantically safe for a Silfa compiler to transform any statement of the input program, granted that the statement is location-independent - which is a syntactic check.

We do not prove directly the CMI soundness theorem. Instead, we prove a strong lemma, that directly leads to the CMI soundness theorem. This lemma (lemma 7) states that if a statement is location-independent, then executing it on one host or another maintains the equivalence between pairs of states and traces. In other words,

the host were a location independent statement is executed has no impact on the behavior of the program. From this lemma, applying the definition of code mobility primitive semantics leads to the CMI soundness theorem.

In this section, we first present a lemma guaranteeing that evaluating a location-independent expression on one host or another is equivalent (section 6.2.1), then we use it for proving the same result applied to statements (section 6.2.2), and finally we demonstrate the CMI soundness theorem (section 6.2.3).

6.2.1 Moving Expressions

The following lemma asserts that evaluating the same expression on two different hosts, but in the context of equivalent states and equivalent traces, yields to the same value and equivalent traces.

Lemma 6 *If $LI(e)$, then $\forall(h, h', \rho, \Sigma, \Sigma', \Omega, \Omega')$ with $(\Sigma, \Omega) \equiv (\Sigma', \Omega')$, $\exists(v_1, v_2, \Omega_1, \Omega_2)$ such that the three following propositions hold :*

$$\mathcal{E}[\![e]\!]\ h\ \rho\ \Sigma\ \Omega = (v_1, \Omega_1)$$
$$\mathcal{E}[\![e]\!]\ h'\ \rho\ \Sigma'\ \Omega' = (v_2, \Omega_2)$$
$$(v_1, \Omega_1) \equiv (v_2, \Omega_2)$$

Proof : The lemma is proven by induction over the syntactic structure of e.

Case $e = p(e')$

This is the most interesting case regarding our problem (a similar case is present in the proof of lemma 7). The argument is that if e is location-independent, then the called primitive p is a location-independent primitive : $p \in LIP$, which leads to the equivalence between Ω_1 and Ω_2.

(i) $\quad \mathcal{E}[\![p(e')]\!]\ h\ \rho\ \Sigma\ \Omega' = (p(v), \Omega' : Pcall(p, h, v))$
(ii) $\quad \mathcal{E}[\![e']\!]\ h\ \rho\ \Sigma\ \Omega = (v, \Omega')$

(iii) $\quad \mathcal{E}[\![p(e')]\!]\ h'\ \rho\ \Sigma\ \Omega'' = (p(v'), \Omega'' : Pcall(p, h', v'))$
(iv) $\quad \mathcal{E}[\![e']\!]\ h'\ \rho\ \Sigma\ \Omega' = (v', \Omega'')$

As long as $LI(e)$, we have also $LI(e_1)$. Furthermore, $\Omega \equiv \Omega'$. Thus, we can apply the induction hypothesis to (ii) and (iv), leading to $(v, \Omega') \equiv (v', \Omega'')$. Consequently, $v_1 = p(v) = p(v') = v_2$.
Moreover, since $LI(e)$, $p \in LIP$ by definition 7 of LIP. Therefore, $\Omega_1 = \Omega' : Pcall(p, h, v) \equiv \Omega'' : Pcall(p, h', v') = \Omega_2$ by definition 10 of equivalence between traces.

Case $e = x$

The two input states Σ_1 and Σ_2 are equivalent, which means by definition that x has the same value in both states.

(i) $\quad \mathcal{E}[\![x]\!]\ h\ \rho\ \Sigma\ \Omega = (Value(\rho(x), \Sigma), \Omega)$
(ii) $\quad \mathcal{E}[\![x]\!]\ h'\ \rho\ \Sigma\ \Omega' = (Value(\rho(x), \Sigma), \Omega')$

Thus $v_1 = v_2 = Value(\rho(x), \Sigma)$ and $\Omega_1 = \Omega \equiv \Omega' = \Omega_2$.

Case $e = e_1.m(e_2)$

(i) $\mathcal{E}[\![e_1.m(e_2)]\!]\ h\ \rho\ \Sigma\ \Omega = (v_b, \Omega_b : Rcall(v_a, m, v_b))$
(ii) $\mathcal{E}[\![e_1]\!]\ h\ \rho\ \Sigma\ \Omega = (v_a, \Omega_a)$
(iii) $\mathcal{E}[\![e_2]\!]\ h\ \rho\ \Sigma\ \Omega_a = (v_b, \Omega_b)$

(iv) $\mathcal{E}[\![e_1.m(e_2)]\!]\ h'\ \rho\ \Sigma'\ \Omega' = (v'_b, \Omega'_b : Rcall(v'_a, m, v'_b))$
(v) $\mathcal{E}[\![e_1]\!]\ h'\ \rho\ \Sigma'\ \Omega' = (v'_a, \Omega'_a)$
(vi) $\mathcal{E}[\![e_2]\!]\ h'\ \rho\ \Sigma'\ \Omega'_a = (v'_b, \Omega'_b)$

As long as e is location-independent, by definition 8 of predicate LI, its subexpression e_1 is also location-independent (and so is e_2). Moreover, $\Omega \equiv \Omega'$, so we can apply the induction hypothesis to (ii) and (v) :
(vii) $v_a = v'_a$
$(viii)$ $\Omega_a \equiv \Omega'_a$

Moreover, $(viii)$ and $LI(e_2)$ implies that we can apply the induction hypothesis to (iii) and (vi), leading to :
(ix) $v_b = v'_b$
(x) $\Omega_b \equiv \Omega'_b$

Consequently, as long as $v_1 = v_b$ and $v_2 = v'_b$, we have $v_1 = v_2$. Meanwhile, $(vii) \wedge (viii) \wedge (ix) \wedge (x)$ gives :
$\Omega_1 = \Omega_b : Rcall(v_a, m, v_b) \equiv \Omega'_b : Rcall(v'_a, m, v'_b) = \Omega_2$.

Case $e = v$

(i) $\mathcal{E}[\![v]\!]\ h\ \rho\ \Sigma\ \Omega = (v, \Omega)$
(ii) $\mathcal{E}[\![v]\!]\ h'\ \rho\ \Sigma'\ \Omega' = (v, \Omega')$

Thus $v_1 = v_2 = v$ and $\Omega_1 = \Omega \equiv \Omega' = \Omega_2$ $\square$

6.2.2 Moving Statements

The following lemma asserts that executing the same Silfa statement on two different hosts, but in the context of equivalent states and equivalent traces, yields to equivalent states and equivalent traces.

Lemma 7 *Let s be a location-independent Silfa statement.*

$$\text{If } (\Sigma, \Omega) \equiv (\Sigma', \Omega') \text{ then } \mathcal{S}[\![s]\!]\ h\ \rho\ \Sigma\ \Omega \equiv \mathcal{S}[\![s]\!]\ h'\ \rho\ \Sigma'\ \Omega'$$

This proposition can be represented by the following schema :

$$\begin{array}{ccc} (\Sigma, \Omega) & \xrightarrow{\mathcal{S}[\![s]\!]\ h\ \rho} & (\Sigma'_1, \Omega'_1) \\ ||| & & |||? \\ (\Sigma', \Omega') & \xrightarrow{\mathcal{S}[\![s]\!]\ h'\ \rho} & (\Sigma'_2, \Omega'_2) \end{array}$$

Proof : The lemma is proven by induction over s.

Case $s = s_1 ; s_2$

By the definition of Silfa semantics - rule **(seq)** :

(i) $\quad \mathcal{S}[\![s_1\texttt{;}\ s_2]\!]\ h\ \rho\ \Sigma\ \Omega = \mathcal{S}[\![s_2]\!]\ h\ \rho\ \Sigma'_1\ \Omega'_1$

(ii) $\quad \mathcal{S}[\![s_1]\!]\ h\ \rho\ \Sigma\ \Omega = (\Sigma'_1, \Omega'_1)$

(iii) $\quad \mathcal{S}[\![s_1\texttt{;}\ s_2]\!]\ h'\ \rho\ \Sigma'\ \Omega' = \mathcal{S}[\![s_2]\!]\ h'\ \rho\ \Sigma'_2\ \Omega'_2$

(iv) $\quad \mathcal{S}[\![s_1]\!]\ h'\ \rho\ \Sigma'\ \Omega' = (\Sigma'_2, \Omega'_2)$

By definition 8, $LI(s_1\texttt{;}\, s_2) \implies LI(s_1) \wedge LI(s_2)$. Thus, we can apply the induction hypothesis to (ii) and (iv) : $(\Sigma'_1, \Omega'_1) \equiv (\Sigma'_2, \Omega'_2)$. Consequently, by induction hypothesis applied to the right-side members of (i) and (iii), we have :

$$\mathcal{S}[\![s_1\texttt{;}\ s_2]\!]\ h\ \rho\ \Sigma\ \Omega \equiv \mathcal{S}[\![s_1\texttt{;}\ s_2]\!]\ h'\ \rho\ \Sigma'\ \Omega'$$

Case $s = \texttt{while}\ e\ \texttt{do}\ s'$

Computation of the loop `while` e `do` s' is split into the sequence of atomic computations of s'. We build the sequences of states $(\Sigma_1^n)_{n\geq 0}$ (resp. $(\Sigma_2^n)_{n\geq 0}$) and traces $(\Omega_1^n)_{n\geq 0}$ (resp. $(\Omega_2^n)_{n\geq 0}$) that link the successive computations of s' on host h (resp. h'). Afterwards, we will show that executing a single step conserves the equivalences between Σ_1^i and Σ_2^i on the first hand, and between Ω_1^i and Ω_2^i on the second hand :

$$\begin{array}{ccccccc}
(\Sigma, \Omega) & \xrightarrow{\cdots} & (\Sigma_1^i, \Omega_1^i) & \xrightarrow{\mathcal{S}[\![s']\!]\ h\ \rho} & (\Sigma_1^{i+1}, \Omega_1^{i+1}) & \xrightarrow{\cdots} & \mathcal{S}[\![\ \texttt{while}\ e\ \texttt{do}\ s']\!]\ h\ \rho\ \Sigma\ \Omega \\
\| & & \| & & \| & & \| \\
(\Sigma', \Omega') & \xrightarrow{\cdots} & (\Sigma_2^i, \Omega_2^i) & \xrightarrow{\mathcal{S}[\![s']\!]\ h'\ \rho} & (\Sigma_2^{i+1}, \Omega_2^{i+1}) & \xrightarrow{\cdots} & \mathcal{S}[\![\ \texttt{while}\ e\ \texttt{do}\ s']\!]\ h'\ \rho\ \Sigma'\ \Omega'
\end{array}$$

Definition of intermediate states and traces

First, let $(\Sigma_1^0, \Omega_1^0) \triangleq (\Sigma, \Omega)$ and $(\Sigma_2^0, \Omega_2^0) \triangleq (\Sigma', \Omega')$. We define $(\Sigma_1^{i+1}, \Omega_1^{i+1})$ as a function of (Σ_1^i, Ω_1^i) : by the rule **(while)** defining the semantics of loops, we have, for any arbitrary (Σ_1^i, Ω_1^i) :

$$\mathcal{S}[\![\texttt{while}\ e\ \texttt{do}\ s']\!]\ h\ \rho\ \Sigma_1^i\ \Omega_1^i = \begin{cases} \mathcal{S}[\![s'\texttt{;}\ \texttt{while}\ e\ \texttt{do}\ s']\!]\ h\ \rho\ \Sigma_1^i\ \Omega_{ie}^i & \text{if } test \\ (\Sigma_1^i, \Omega_{1e}^i) & \text{otherwise} \end{cases}$$

$$\text{where } \mathcal{E}[\![e]\!]\ h\ \rho\ \Sigma_1^i\ \Omega_1^i = (test, \Omega_{1e}^i)$$

Furthermore, by applying rule **(seq)**, we have :

$$\mathcal{S}[\![\texttt{while}\ e\ \texttt{do}\ s']\!]\ h\ \rho\ \Sigma_1^i\ \Omega_1^i = \begin{cases} \mathcal{S}[\![\texttt{while}\ e\ \texttt{do}\ s']\!]\ h\ \rho\ \Sigma_1^{i+1}\ \Omega_1^{i+1} & \text{if } test \\ \quad \text{where } \mathcal{S}[\![s']\!]\ h\ \rho\ \Sigma_1^i\ \Omega_{1e}^i = (\Sigma_1^{i+1}, \Omega_1^{i+1}) & \\ (\Sigma_1^i, \Omega_{1e}^i) & \text{otherwise} \end{cases}$$

The previous proposition defines Σ_1^{i+1} and Ω_1^{i+1} as functions of Σ_1^i and Ω_1^i. Similarly, we define Σ_2^{i+1} and Ω_2^{i+1} :

$$\mathcal{S}[\![\texttt{while}\ e\ \texttt{do}\ s']\!]\ h'\ \rho\ \Sigma_2^i\ \Omega_2^i = \begin{cases} \mathcal{S}[\![\texttt{while}\ e\ \texttt{do}\ s']\!]\ h'\ \rho\ \Sigma_2^{i+1}\ \Omega_2^{i+1} & \text{if } test' \\ \quad \text{where } \mathcal{S}[\![s']\!]\ h'\ \rho\ \Sigma_2^i\ \Omega_{2e}^i = (\Sigma_2^{i+1}, \Omega_2^{i+1}) & \\ (\Sigma_2^i, \Omega_{2e}^i) & \text{otherwise} \end{cases}$$

$$\text{where } \mathcal{E}[\![e]\!]h'\ \rho\ \Sigma_2^i\ \Omega_2^i = (test', \Omega_{2e}^i)$$

Finally, we define as constant the sequence $(\Sigma_1^n)_{n\geq k}$ and $(\Omega_1^n)_{n\geq k}$ (resp. for h') where k is smallest index such as the test is false.

If $\mathcal{E}[\![e]\!]\ h\ \rho\ \Sigma_1^i\ \Omega_1^i = (false, \Omega_{1e}^i)$
then $\forall(k \geq i)\ (\Sigma_1^{k+1}, \Omega_1^{k+1}) \triangleq (\Sigma_1^i, \Omega_{1e}^i) = \mathcal{S}[\![\texttt{while}\ e\ \texttt{do}\ s']\!]\ h\ \rho\ \Sigma\ \Omega$

If $\mathcal{E}[\![e]\!]\ h'\ \rho\ \Sigma_2^i\ \Omega_2^i = (false, \Omega_{2e}^i)$
then $\forall(k \geq i)\ (\Sigma_1^{k+1}, \Omega_1^{k+1}) \triangleq (\Sigma_1^i, \Omega_{1e}^i) = \mathcal{S}[\![\texttt{while}\ e\ \texttt{do}\ s']\!]\ h'\ \rho\ \Sigma'\ \Omega'$

Proving stability of equivalence

We show by a recurrence over n that :

$$\forall(n \geq 0)\quad (\Sigma_1^n, \Omega_1^n) \equiv (\Sigma_2^n, \Omega_2^n)$$

First, let us show that $(\Sigma_1^i, \Omega_1^i) \equiv (\Sigma_2^i, \Omega_2^i) \implies (\Sigma_1^{i+1}, \Omega_1^{i+1}) \equiv (\Sigma_2^{i+1}, \Omega_2^{i+1})$. Let us assume that $\Omega_1^i \equiv \Omega_2^i$. As stated before, :

$$\mathcal{E}[\![e]\!]\ h\ \rho\ \Sigma_1^i\ \Omega_1^i = (test, \Omega_{1e}^i)$$
$$\mathcal{E}[\![e]\!]\ h'\ \rho\ \Sigma_2^i\ \Omega_2^i = (test', \Omega_{2e}^i)$$

Moreover, by definition 8, $LI(e)$. Thus, by lemma 6, $test = test'$ and $\Omega_{1e}^i \equiv \Omega_{2e}^2$. Consequently, if $test$ and $test'$ are false, then $(\Sigma_1^{i+1}, \Omega_1^{i+1}) = (\Sigma_1^i, \Omega_{1e}^i) \equiv (\Sigma_2^i, \Omega_{2e}^i) = (\Sigma_2^{i+1}, \Omega_2^{i+1})$. Otherwise (if $test = true$), then :

$$\mathcal{S}[\![s']\!]\ h\ \rho\ \Sigma_1^i\ \Omega_{1e}^i = (\Sigma_1^{i+1}, \Omega_1^{i+1})$$
$$\mathcal{S}[\![s']\!]\ h'\ \rho\ \Sigma_2^i\ \Omega_{2e}^i = (\Sigma_2^{i+1}, \Omega_2^{i+1})$$

By induction hypothesis, as long as $LI(s) \implies LI(s')$, we have $(\Sigma_1^{i+1}, \Omega_1^{i+1}) \equiv (\Sigma_2^{i+1}, \Omega_2^{i+1})$. Finally :

$(\Sigma_1^i, \Omega_1^i) \equiv (\Sigma_2^i, \Omega_2^i) \implies (\Sigma_1^{i+1}, \Omega_1^{i+1}) \equiv (\Sigma_2^{i+1}, \Omega_2^{i+1})$
$(\Sigma_1^0, \Omega_1^0) \equiv (\Sigma_2^0, \Omega_2^0)$
$\implies \forall(n \geq 0)\quad (\Sigma_1^n, \Omega_1^n) \equiv (\Sigma_2^n, \Omega_2^n)$

Thus, for $n = k$:

$$\mathcal{S}[\![\texttt{while}\ e\ \texttt{do}\ s']\!]\ h\ \rho\ \Sigma\ \Omega = (\Sigma_1^k, \Omega_1^k) \equiv (\Sigma_2^k, \Omega_2^k) = \mathcal{S}[\![\texttt{while}\ e\ \texttt{do}\ s']\!]\ h'\ \rho\ \Sigma'\ \Omega'$$

Case $s = x := e$

$$\mathcal{S}[\![x := e]\!]\ h\ \rho\ \Sigma\ \Omega = (Update(x, v, \Sigma, h), \Omega_e)$$
$$\mathcal{E}[\![e]\!]\ h\ \rho\ \Sigma\ \Omega = (v, \Omega_e)$$
$$\mathcal{S}[\![x := e]\!]\ h'\ \rho\ \Sigma'\ \Omega' = (Update(x, v', \Sigma', h'), \Omega'_e)$$
$$\mathcal{E}[\![e]\!]\ h'\ \rho\ \Sigma'\ \Omega' = (v', \Omega'_e)$$

$LI(s) \implies LI(e)$. Consequently, by lemma 6, $(v, \Omega_e) \equiv (v', \Omega'_e)$. Thus, application of lemma 4 leads to the conclusion.

Case $s = e_1.m(e_2)$

(i) $\mathcal{S}[\![e_1.m(e_2)]\!]\ h\ \rho\ \Sigma\ \Omega = (\Sigma, \Omega_2 : Rcall(v_1, m, v_2))$
(ii) $\mathcal{E}[\![e_1]\!]\ h\ \rho\ \Sigma\ \Omega = (v_1, \Omega_1)$
(iii) $\mathcal{E}[\![e_2]\!]\ h\ \rho\ \Sigma\ \Omega_1 = (v_2, \Omega_2)$

(iv) $\mathcal{S}[\![e_1.m(e_2)]\!]\ h'\ \rho\ \Sigma'\ \Omega' = (\Sigma', \Omega'_2 : Rcall(v'_1, m, v'_2))$
(v) $\mathcal{E}[\![e_1]\!]\ h'\ \rho\ \Sigma'\ \Omega' = (v'_1, \Omega'_1)$
(vi) $\mathcal{E}[\![e_2]\!]\ h'\ \rho\ \Sigma'\ \Omega'_1 = (v'_2, \Omega'_2)$

$LI(s) \implies LI(e_1) \wedge LI(e_2)$. Thus, lemma 6 on (ii) and (v) gives $(v_1, \Omega_1) \equiv (v_1, \Omega_1)$, and consequently lemma 6 gives for (iii) and (vi) : $(v_2, \Omega_2) \equiv (v_2, \Omega_2)$, which leads to the conclusion.

Case $s = \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2$

$$\mathcal{S}[\![\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]\ h\ \rho\ \Sigma\ \Omega = \begin{cases} \mathcal{S}[\![s_1]\!]\ h\ \rho\ \Sigma\ \Omega_1^e & \text{if } test \\ \mathcal{S}[\![s_2]\!]\ h\ \rho\ \Sigma\ \Omega_1^e & \text{otherwise} \end{cases}$$

$$\mathcal{S}[\![\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]\ h'\ \rho\ \Sigma'\ \Omega' = \begin{cases} \mathcal{S}[\![s_1]\!]\ h'\ \rho\ \Sigma'\ \Omega_2^e & \text{if } test' \\ \mathcal{S}[\![s_2]\!]\ h'\ \rho\ \Sigma'\ \Omega_2^e & \text{otherwise} \end{cases}$$

$$\mathcal{E}[\![e]\!]\ h\ \rho\ \Sigma\ \Omega = (test, \Omega_1^e)$$
$$\mathcal{E}[\![e]\!]\ h'\ \rho\ \Sigma'\ \Omega' = (test', \Omega_2^e)$$

Moreover, $LI(s) \implies LI(e)$. Thus, by application of lemma 6 to the two expressions, $\Omega_1^e \equiv \Omega_2^e$ and $test = test'$. Let us consider that $test = test' = true$. Consequently :
$\mathcal{S}[\![\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]\ h\ \rho\ \Sigma\ \Omega = \mathcal{S}[\![s_1]\!]\ h\ \rho\ \Sigma\ \Omega_1^e$
$\mathcal{S}[\![\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]\ h'\ \rho\ \Sigma'\ \Omega' = \mathcal{S}[\![s_1]\!]\ h'\ \rho\ \Sigma'\ \Omega_2^e$

However, $(\Sigma, \Omega_1^e) \equiv (\Sigma, \Omega_2^e)$. Thus, by ind. hyp. ($LI(s) \implies LI(s_1)$),

$$\mathcal{S}[\![\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]\ h\ \rho\ \Sigma\ \Omega \equiv \mathcal{S}[\![\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]\ h'\ \rho\ \Sigma'\ \Omega'$$

Idem for s_2 if $test = test' = false$

Case $s = p(e)$

$$\mathcal{S}[\![p(e)]\!]\ h\ \rho\ \Sigma\ \Omega = (\Sigma, \Omega_e : Pcall(p, h, v))$$
$$\mathcal{E}[\![e]\!]\ h\ \rho\ \Sigma\ \Omega = (v, \Omega_e)$$
$$\mathcal{S}[\![p(e)]\!]\ h'\ \rho\ \Sigma'\ \Omega' = (\Sigma', \Omega'_e : Pcall(p, h', v))$$
$$\mathcal{E}[\![e]\!]\ h'\ \rho\ \Sigma'\ \Omega' = (v', \Omega'_e)$$

By lemma 6, since $LI(s) \implies LI(e)$, we have $(v, \Omega_e) \equiv (v', \Omega'_e)$, which leads to the conclusion.

Case $s = \texttt{skip}$

$$\mathcal{S}[\![\texttt{skip}]\!]\ h\ \rho\ \Sigma\ \Omega = (\Sigma, \Omega) \equiv (\Sigma', \Omega') = \mathcal{S}[\![\texttt{skip}]\!]\ h'\ \rho\ \Sigma'\ \Omega'$$

Case $s = \{d\ s'\}$

$$\mathcal{S}[\![\{d\ s\}]\!]\ h\ \rho\ \Sigma\ \Omega = \mathcal{S}[\![s]\!]\ h\ (\rho : \mathcal{D}[\![d]\!]\rho)\ \Sigma\ \Omega$$
$$\mathcal{S}[\![\{d\ s\}]\!]\ h'\ \rho\ \Sigma'\ \Omega' = \mathcal{S}[\![s]\!]\ h'\ (\rho : \mathcal{D}[\![d]\!]\rho)\ \Sigma'\ \Omega'$$

Thus by ind. hyp. : $\mathcal{S}[\![\{d\ s\}]\!]\ h\ \rho\ \Sigma\ \Omega \equiv \mathcal{S}[\![\{d\ s\}]\!]\ h'\ \rho\ \Sigma'\ \Omega'$ $\square$

6.2.3 Introducing Code Mobility Primitive

The CMI soundness theorem, presented below, asserts that applying CMI to local-independent statements of a program does not affect the global behavior of the program. This is the main result of this paper. The theorem is a direct consequence of lemma 7.

Theorem 8 *Let s be a location-independent Silfa statement. Let $h \in Host$. Let $obj \in Var$. Let ρ, Σ be such that $Value(\rho(obj), \Sigma) \in Obj$. Let $\Omega \in Trace$. Then :*

$$\mathcal{S}[\![(obj) \leftarrow s]\!]\ h\ \rho\ \Sigma\ \Omega \equiv \mathcal{S}[\![s]\!]\ h\ \rho\ \Sigma\ \Omega$$

Proof : The theorem is a direct consequence of lemma 7. Indeed,

$$\begin{array}{lll} \mathcal{S}[\![(obj) \leftarrow s]\!]\ h\ \rho\ \Sigma\ \Omega & = \mathcal{S}[\![s]\!]\ host(Value(\rho(obj), \Sigma))\ \rho\ \Sigma\ \Omega & \text{(def. } \mathcal{S}\text{, } \textbf{code-mob}\text{)} \\ & \equiv \mathcal{S}[\![s]\!]\ h\ \rho\ \Sigma\ \Omega & \text{(lemma 7)} \quad \square \end{array}$$

7 Concluding Remarks

In this paper, we have presented a new approach – mobility introduction –, close to automatic partitioning, for optimizing systems based on Distributed Object Computing or, more generally, any kind of Remote Procedure Call techniques. In our model, one can make mobile every single statement of the original program, while previous research only considered mobility of more coarse-grained slices of code like COM components or Java objects. We have shown that such an approach can ease constraints on the design of Distributed Numerical Simulation Frameworks, and particularly on design of numerical data interfaces. Moreover, we have issued a simple syntactic criteria for mobility introduction to be applied without corrupting the global behavior of programs : knowing what primitives access to local resources is sufficient and necessary for an optimizing compiler to apply such a transformation consistently.

We have implemented a simple prototype that applies mobility introduction on programs written in the toy language Silfa. Programs are augmented with data and code mobility primitives, then the compiler generates executable Java code based on a simple remote execution library. The compiler statically analyzes input programs and determines what portions of code are location-independent. Then, an empirical criteria is applied to decide which ones are worth to be actually executed remotely : we mobilize the deepest loops that access a single remote object. Once this choice is made, we add data mobility primitives to the free variables of the loop and to its written variables, as it is described in section 2. Although this criteria is very simple, it already provides interesting results. When applied on the example presented in section 2 of this paper[7], the transformed program is 50% faster than the original one on a modern local network with about 0.1 ms of program-to-program latency, and up to 90% faster with a latency of 10 ms.

Future work

There are two features we want to add in a near future to our language : user-defined procedures and functions, and data arrays. Then, Silfa will look like a pointer-less Pascal, which is what numerical simulation programmers actually use. Introduction of user-defined procedures and functions require the compiler to compute the call graph of the program. This is not a difficult issue if there is no pointer to function nor higher-order functions. Introduction of arrays into the language will require that

[7] These tests have been done with a 100*100 matrix.

the decision for introducing code mobility would be constrained by the size of the data that would have to be moved. This is because with big amounts of data, the bandwidth (and not only the latency) of the network becomes an issue. Finally, we intend to consider runtime informations in the decision procedure, such as statistical data on previous timings, actual latency, actual size of data.

We intend to apply mobility introduction in the domain of Distributed Numerical Simulation Frameworks (DNSF), by implementing programs from this application domain and showing performance gains with fine-grained data interfaces. Still, we believe that DNSF is not the only application domain of mobility introduction. For instance, any system based on Distributed Object Computing or Remote Procedure Call techniques where performances of the communication framework are critical can benefit from our approach. Nowadays, in such systems, reusability is often sacrificed in order to preserve performances. With mobility introduction techniques, it would not be the case.

References

[1] IBM Corp. The IBM aglets workbench. http://www.trl.ibm.co.jp/aglets/, 1996.

[2] Federico Bassetti, David Brown, Kei Davis, William Henshaw, and Dan Quinlan. OVERTURE: An object-oriented framework for high-performance scientific computing. In *Proceedings of Supercomputing'98 (CD-ROM)*. ACM SIGARCH and IEEE, 1998.

[3] Luca Cardelli. A language with distributed scope. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.*, pages 286–297, New York, NY, 1995.

[4] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[5] David M. Chess, Colin G. Harrison, and Aaron Kershenbaum. Mobile Agents: Are they a good idea? In *Mobile Object Systems – Toward a Programmable Internet, LNCS 1222*, pages 25–47, Berlin, Germany, 1997. Springer-Verlag.

[6] M. Dahm. The doorastha system. *Technical report B-1-2000, Freie Universität Berlin*, 2000.

[7] Alexandre Denis, Christian Pérez, Thierry Priol, and André Ribes. Padico: A component-based software infrastructure for grid computing. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, Nice, France, April 2003. IEEE Computer Society.

[8] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[9] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421. Springer-Verlag, 1996.

[10] D. Hagimont and D. Louvegnies. Javanaise: distributed shared objects for Internet cooperative applications. In *Middleware'98*, The Lake District, England, 1998.

[11] Galen C. Hunt and Michael L. Scott. The coign automatic distributed partitioning system. In *Operating Systems Design and Implementation*, pages 187–200, 1999.

[12] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[13] P. Merle, C. Gransart, and J. Geib. How to make corba objects user-friendly with a generic object-oriented dynamic environment, 1996.

[14] T. Nguyen and C. Plumejeaud. An integration platform for metacomputing applications. In J.J. Dongarra P.M.A. Sloot, C.J. Kenneth Tan and A.G. Hoekstra, editors, *Lecture Notes in Computer Science – Computational Science - ICCS 2002: International Conference, Amsterdam, The Netherlands, April 21-24, 2002. Proceedings, Part I*, volume 2329 / 2002, ISSN: 0302-9743. Springer-Verlag Heidelberg, 2002.

[15] Linear Algebra Package. http://www.netlib.org/lapack/.

[16] Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, nov 1997.

[17] Christian Queinnec and David De Roure. Sharing code through first-class environments. In *Proceedings of ICFP'96 — ACM SIGPLAN International Conference on Functional Programming*, pages 251–261, Philadelphia (Pennsylvania, USA), 1996.

[18] John V. W. Reynders, Paul J. Hinker, Julian C. Cummings, Susan R. Atlas, Subhankar Banerjee, William F. Humphrey, Steve R. Karmesin, Katarzyna Keahey, Marikani Srikant, and Mary Dell Tholburn. POOMA: A Framework for Scientific Simulations of Paralllel Architectures. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming in C++*, pages 547–588. MIT Press, 1996.

[19] Salome. http://www.opencascade.org/salome/salome_platform.php.

[20] Simulog. http://www.simulog.fr/.

[21] André Spiegel. Automatic distribution in pangaea. *Proc. Workshop on Communications-Based Systems, CBS 2000*, April 2000.

[22] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano. A bytecode translator for distributed execution of "legacy" Java software. *Lecture Notes in Computer Science*, 2072:236–256, 2001.

[23] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning, 2002.

[24] ObjectSpace Inc. ObjectSpace Voyager. http://www.objectspace.com/voyager, 1997.

[25] Pawel Wojciechowski and Peter Sewell. Nomadic pict: Language and infrastructure design for mobile agents. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.