Model checking of high-level object oriented specifications: the **L*f*P** experience[*]

Frédéric Gilliers†, François Bréant⋆, Denis Poitrenaud‡ & Fabrice Kordon‡

† SAGEM SA
Etablissement d'Eraggny, Avenue du Gros Chêne
95610 Eragny
B.P. 51 - 95612 Cergy Pontoise Cedex, France
`frederic.gillier@sagem.com`
⋆LaBRI, équipe vérification et test de systèmes informatiques
Université de Bordeaux 1
Domaine Universitaire, 351, cours de la Libération,
33405 Talence Cedex, FRANCE
`francois.breant@lip6.fr`
‡ Laboratoire d'Informatique de Paris 6/SRC
Université Pierre & Marie Curie
4, place Jussieu
F-75252 Paris CEDEX 05, France
`denis.poitrenaud@lip6.fr, fabrice.kordon@lip6.fr`

1 Introduction

The rapid advance of distributed technology has lead to systems stretching limits in terms of complexity and manageability [10]. This problem is crucial for reliable distributed systems which are required to have a deterministic behavior. To solve these development problems, it is of interest to consider development model based development approach [4].

Such an approach distinguishes two strong components [8]:

- a model on which any type of validation or verification techniques may be applied,
- the programs that implement this model and which are generated from it.

Such approaches become widely accepted under various names. As an example, MDA [12] (Model Driven Architecture) may be considered as a similar approach.

A distributed application is made of two orthogonal aspects: the control aspect, and the computational aspect. The control aspect manages the global state of the application whereas the computational aspect covers the domain specific computational components. Model-Based development is of particular interest for distributed systems for two main reasons. First, they are very difficult to develop since they are very undeterministic ; thus, some apparently minor choices may have dramatic influences on the system behavior. Second, control aspects (the difficult part to build) strongly interact with both the execution environment and computational aspects, these interactions have to be carefully studied to avoid unexpected behaviors.

In that context, formal methods are of particular interest since they allow to prove the system using model checking techniques for example. However, [11] demonstrated that a major problem for the use of formal methods is the large education required by engineers to use them. The idea is then to encapsulate them using "more common" languages for which no particular (or less) training is required. As an example, a similar approach is used in BLAST citeblast that allows the use of C programs as inputs for model checking.

We have designed **L*f*P** (language for Prototyping) [13], a formal notation dedicated to the specification, verification and code generation of distributed systems. **L*f*P** aims at providing a high level notation that fits

[*]The work presented in this paper is being performed withing the MORSE project. MORSE is a French government founded research project (RNTL) with industrial partners (Sagem, Aonix) and academic partners (LIP6 - Univ. P. & M. Curie, LaBRI - Univ. Bordeaux I).

the needs for describing the control part of a distributed system in a way that makes it usable for engineers. A first experience for verification from **L*f*P** specifications is described in [9] and [14]; it is based on a Petri net generated from the **L*f*P** specification for verification purposes.

This paper presents *direct model checking* on **L*f*P** specifications; which means that model checking is performed on the **L*f*P** specification instead of the corresponding Petri net like in [9, 14]. The symbolic model checker presented here is based on DDD (Data Decision Diagrams) that are an extension of BDD for discrete types [3]. We explain how we represent an **L*f*P** state using DDDs and the mechanisms we use to perform the state exploration. The key issues related to the formal verification of **L*f*P** specifications are related to the dynamic aspects of this language such as processes creation, RPC mechanisms, addressing, etc.

Section 2, briefly presents the overall methodology. We then present **L*f*P** in section 3 and finally detail in section 4 why DDD are well suited to represent **L*f*P** programs, our coding technique of **L*f*P** programs as well as issues and problems raised in this study.

2 Object Oriented Methodology

We propose a methodology to handle the specific issues of distributed applications. Its goal is to help the designer to achieve the development of a distributed application.

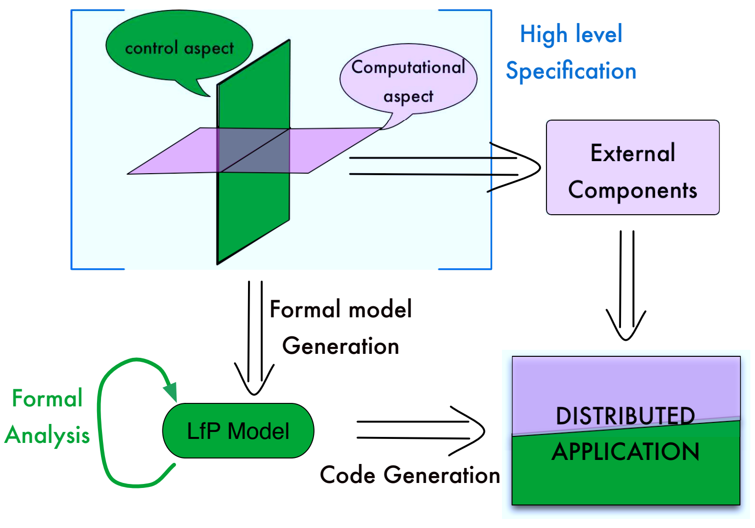


Figure 1: Development methodology associated to **L*f*P**

As shown on figure 1, our methodology relies on the separation between the control aspect and the data computation aspect in distributed systems. The control aspect handles the distribution of the application over the network, and the interaction protocols between the components of an application. The data computation aspect handles the domain specific calculus required to produce the results.

Our methodology starts with a high level description of the application written in UML. This specification should outline the interaction between the computational aspect and the control aspect of the system. We focus on the modelling of the control aspect since it is related to the specific issues identified in the development of distributed applications. Therefore, this control aspect of the specification is translated into **L*f*P** a formal language specifically designed to model distributed applications control aspect.

The interactions between the control aspect and the computational aspect are modeled using constructions of the **L*f*P** language very similar to private types defined in the Ada language [6].

The formal model obtained from the specification can then be formally checked against its requirements. State properties can be stated in the UML description in OCL, but properties which involve series of actions must be stated in temporal logic directly on the **L*f*P** specification. Formal verification of **L*f*P** specifications is the heart of this paper and will be fully described in section 4.

Once the model meets all its requirements, we provide a code generator for the **L*f*P** language. Automatic code generation translates the **L*f*P** semantics to provide an effective implementation of the specification and ensures the correctness of the implementation. It uses the description of the interactions between the control and the computational aspects to link the generated code to the external components. The underlying mechanism of code generation from an **L*f*P** model have been discussed in [5].

3 The L*f*P language

This section will present the **L*f*P** language through a simple client / server example. We show that **L*f*P** provides the appropriate abstractions to model interactions between components of an application.

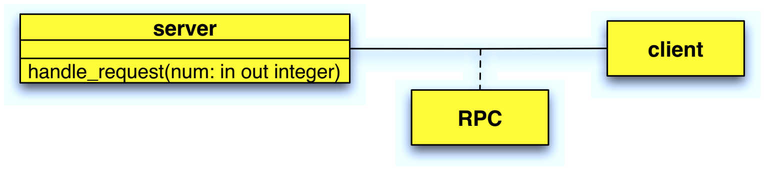


Figure 2: Simplified class diagram of the client server example

Let us first introduce our example with the simplified UML class diagram of figure 2 that shows the main model components. The client calls method `handle_request` on the server through a RPC. The server then returns a value through the `in out` parameter of this method. We will now present the **L*f*P** model corresponding to this system. First we will focus on the static description, then on the dynamic behavior of the components.

3.1 Static structure of the model

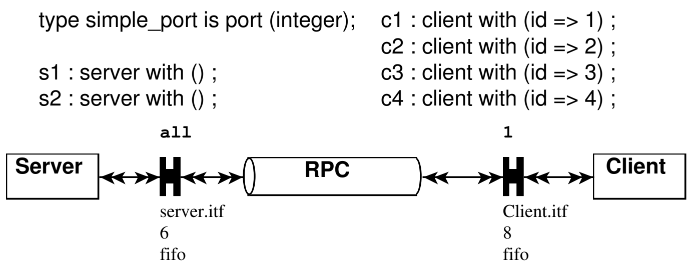

type simple_port is port (integer);
s1 : server with () ;
s2 : server with () ;
c1 : client with (id => 1) ;
c2 : client with (id => 2) ;
c3 : client with (id => 3) ;
c4 : client with (id => 4) ;
all
1
Server
RPC
Client
server.itf
6
fifo
Client.itf
8
fifo

Figure 3: Architecture diagram of the client / server example

The static structure of an **L*f*P** model is described with an *architecture diagram*. Figure 3 shows the architecture diagram corresponding to the class diagram of figure 2. The main elements of the class diagram appear on the architecture diagram:

- interaction classes are translated into **L*f*P** *media* which are components that define the low level interaction protocol between the application components;
- classes of the model are translated into **L*f*P** *classes* which implements the control aspects in application components.

In order to link the components and define the message queues of the application, the architecure diagram introduces *binders* to link the classes and media of the model. They formalize the message transmission between the components of the model. They are referenced in the components with variables of type `port` identified in the binder's binding attribute. These variables are of type `simple_port` defined in the diagram's definitions.

The architecture diagram also defines the static instances of the model: two instances of `server` and four instances of `client` are created on application start up. Each instance of the client has one of its attribute initialized with a value that identifies it.

On this specific architecture diagram, the binder that relates RPC to clients has multiplicity 1, there is one binder instance for each instance of class client. The binder that relates RPC to Server has mutiplicity all which means that the binder is shared by all the instances of the class. A message in this binder may be read and handled by any server.

3.2 Dynamic behavior of model components

The dynamic behavior of the components is defined by their behavioral diagrams. This is an automaton that defines the actions performed by the **L*f*P** component in response to an event.

3.2.1 The RPC media

This media is in charge to implement the "Remote Procedure Call" protocol between the client and the server, it is shown on figure 4. This means that the media must send the message provided by the client to the server, read the return message from the server and send it to the client.

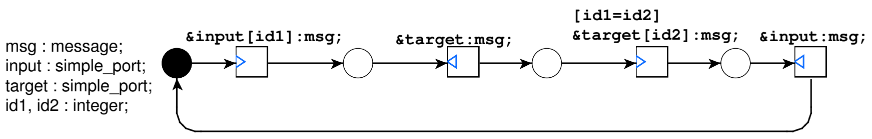


Figure 4: Behavioral diagram of media RPC

The media is related to client by input and to the server by target; both ports must be initialized by the component that creates the instance of RPC.

The first transition of the media reads a message on its input port and stores the discriminant in id1. This discriminant should identify the component that sent the message. Then the message puts the message in the target port.

The media then reads a message from the target port and only accepts it if its discriminant is equal to id1, that is if it was sent by the component that has sent the request. Finally the return message is sent back to the client, the media jumps to its initial state and is ready to handle a new message.

3.2.2 The client class

This class implements the client side of the system and is displayed on figure 5. This class first creates an instance of the RPC media to handle its communication with the server. Then it starts a loop that calls method handle_request. This means that a message requesting the execution of the method is put in the port itf, with a discriminant that contains the identifier of the client instance. The transition that contains this instruction is a call transition which also waits for the method's return message which updates the value of parameter i.

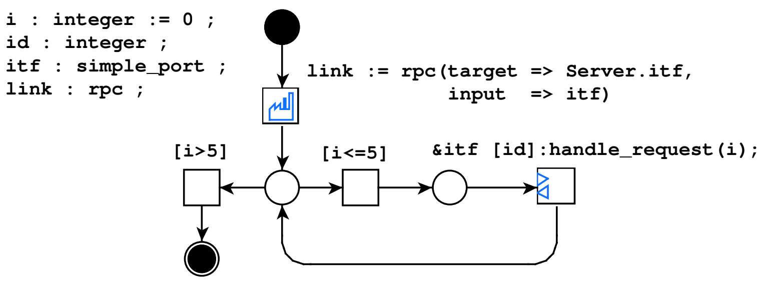
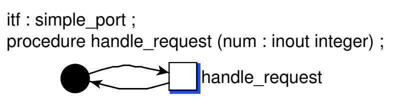


Figure 5: Behavioral diagram of class client

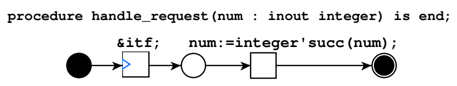
When the value of i becomes greater than five, the loops ends and the client finishes its execution.

3.2.3 The `server` class

The class `server` handles the request. Its main diagram is displayed on figure 6(a). When instanciated, a server keeps waiting for the activation of method `handle_request`. The behavior of this method is displayed on figure 6(b). It is activated by the arrival of an activation message on port `itf` which is the shared binder between the `RPC` media and the server class on the architecture diagram of figure 3.



(a) main diagram of the class



(b) behavioral diagram of method `handle_request`

Figure 6: Behavioral diagrams of class `server`

The method simply increases the value of its actual parameter and returns. Since the parameter mode is `inout`, the new value is sent through a return message to the caller. This message is implicitly sent on the method's activation port when the method returns. Since the discriminant of the return message is not specified, the discriminant of the activation message is used. In this case, it means that the discriminant of the return message contains the identifier of the client that sent the request.

4 Formal Verification

The verification of embedded distributed applications expressed in **L*f*P** covers a large number of properties. The verification process reduces to computing the reachability set of the program, which is the set of all possible states, and then evaluate assertions on the obtained set.

A data structure capable of representing large number of states must enable efficient operations such as equality test, set-theoretic operations, **L*f*P** specific operations, as well as a compact representation in memory.

We illustrate with a simple example a verification approach methodology based on the use of DDD (Data Decision Diagrams) for the symbolic computation of reachable states.

The first section introduces the DDD.

Section 2,3,4 successively describe the steps used for producing a verification program from an **L*f*P** specification:

- deriving an adequate model for verification purpose,
- computation of the reachable states,
- evaluation of assertions on the reachability set.

These steps are illustrated with the verification of the simple client/server application.

4.1 The Data Decision Diagrams (DDD)

The purpose of this paper is not to provide a complete definition of the DDD structure. Theoretical aspects of DDD are addressed in [3].

Data Decision Diagrams (DDDs) are *concise* data structures for representing *finite sets of assignment sequences* of the form $(e_1 := x_1; e_2 := x_2; \cdots; e_n := x_n)$ where e_i are variables and x_i are values. When an ordering on the variables is fixed and the variables are boolean, DDDs coincides with the well-know *Binary Decision Diagrams* [1, 2]. If an ordering on the variables is the only assumption, DDDs are the specialized version of the *Multi-valued Decision Diagrams* representing characteristic function of sets. For Data Decision Diagram, we assume no variable ordering and, even more, the same variable may occur many times in an assignment sequence, allowing the representation of dynamic structures: for a stack variable a, the sequence of assignments $(a := x_1; a := x_2; \cdots; a := x_n)$ may represent the stack content $x_1 x_2 \cdots x_n$.

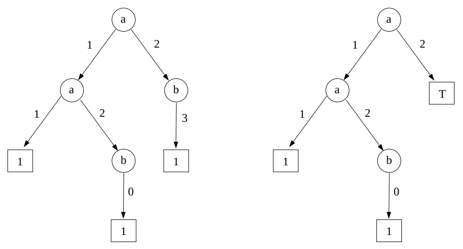


Figure 7: Two Data Decision Diagrams.

Traditionally, decision diagrams are often encoded as decision trees. Internal nodes are labeled with variables, arcs with values (of the adequate type) and leaves with either `0` or `1`. Figure 7, left-hand side, shows the decision tree for the set $S = \{(a := 1; a := 1), (a := 1; a := 2; b := 0), (a := 2; b := 3)\}$ of assignment sequences. As usual, `1`-leaves stand for accepting terminators and `0`-leaves for non-accepting terminators. Since there is no assumption on the cardinality of the variable domains, we consider `0` as the default value. Therefore `0`-leaves are not depicted in figure 7.

Unfortunately, any finite set of assignment sequences cannot be represented. Thus, we introduce a new kind of leaf label: $\top$ for *undefined*. Intuitively, $\top$ represents any finite set of assignment sequences. Figure 7, right-hand side, gives an approximation of the set $S \cup \{(a := 2; a := 3)\}$. Indeed, an ambiguity is introduced since after the assignment $a := 2$, two assignments have to be represented: $a := 3$ and $b := 3$. These two assignments affect two distinct variables so they can not be represented, as two distinct arcs outgoing from the same node cannot be labeled with the same value (in other words, non-determinism is not authorized in the decision tree).

However, since our verification approach only considers well formed assignments sequences with no compatibility issues regarding operations, we will consider $\top$ leafs as the result of an error in the state set computation. We now give an overview of DDDs, for a more formal and detailed presentation including theoretical aspects regarding $\top$, we refer the reader to [3].

4.1.1 Syntax and semantics of DDDs

In the following, E denotes a set of *variables*, and for any e in E, $\mathrm{Dom}(e)$ represents the *domain* of e.

Definition 1 (Data Decision Diagram) *The set $\mathbb{D}$ of DDDs is inductively defined by $d \in \mathbb{D}$ if:*

- $d \in \{0, 1, \top\}$ *or*
- $d = (e, \alpha)$ *with:*
  - $e \in E$
  - $\alpha : \mathrm{Dom}(e) \to \mathbb{D}$, *such that* $\{x \in \mathrm{Dom}(e) \mid \alpha(x) \neq 0\}$ *is finite.*

We denote $e \xrightarrow{z} d$, *the DDD* (e, α) *with* $\alpha(z) = d$ *and* $\alpha(x) = 0$ *for all* $x \neq z$.

Intuitively, a DDD can be seen as a tree. DDDs `0`, `1` and $\top$ are leaves, and a DDD of the form (e, α) is a tree whose root is labeled with variable e, and with an outgoing arc labeled with x to a subtree $\alpha(x)$ foreach value $x \in \mathrm{Dom}(e)$. From a practical point of view, as non-accepting branches (i.e. branches ending with a `0`-leaf) are not encoded, the “finite support” condition for α ensures that DDDs can be implemented (even when variables range over infinite domains).

The *meaning* $[\![d]\!]$ of a DDD d is a set of finite sets of assignment sequences. In particular, $[\![\top]\!]$ is the (infinite) set of all finite sets of asssignment sequences. When $\top$ does not appear in a DDD, the DDD represents a unique finite set of assignment sequences (i.e. its meaning is a singleton). Hence, such a DDD yields an exact (non approximate) representation and it is called *well-defined*.

The unique set in the meaning of a well-defined DDD d is the set of assignment sequences corresponding to accepting branches (i.e. branches ending with a `1`-leaf) in the tree representation of d. In particular, we have $[\![0]\!] = \{\emptyset\}$ and $[\![1]\!] = \{\{()\}\}$ (where $()$ is the empty sequence of assignments).

Equivalence checking for DDDs is crucial when DDDs are used to represent sets of states. Fortunately, DDDs admit *canonical forms* so that equivalence checking for DDDs in canonical form reduces to (syntactic) equality.

Intuitively, from the tree representation point of view, the canonical form of a DDD is obtained by replacing with 0 all sub-trees that have only 0-leaves and by sharing all subtrees which are equivalent. Two DDDs in canonical form are equivalent if and only if they are equal. Moreover, every DDD is equivalent to a DDD in canonical form.

In the following, *we only consider DDDs that are in canonical form*.

4.1.2 Operations on DDDs

First, we generalize the usual set-theoretic operations – *sum* (union), *product* (intersection) and *difference* – to finite sets of assignment sequences expressed in terms of DDDs. The crucial point of this generalization is that all DDDs are not well-defined and furthermore that the result of an operation on two well-defined DDDs is not necessarily well-defined. The *sum* $+$, the *product* $*$ and the *difference* $\setminus$ of two DDDs are inductively defined in the following tables. In these tables, for any $\diamond \in \{+,*,\setminus\}$, $\alpha_1 \diamond \alpha_2$ stands for the mapping in $\mathrm{Dom}(e_1) \rightarrow \mathbb{D}$ defined by $(\alpha_1 \diamond \alpha_2)(x) = \alpha_1(x) \diamond \alpha_2(x)$ for all $x \in \mathrm{Dom}(e_1)$.

$+$	0	1	$\top$	(e_2,α_2)
0	0	1	$\top$	(e_2,α_2)
1	1	1	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$
(e_1,α_1)	(e_1,α_1)	$\top$	$\top$	$(e_1,\alpha_1+\alpha_2)$ if $e_1 = e_2$ $\top$ if $e_1 \neq e_2$

$*$	0	1	$\top$	(e_2,α_2)
$0 \vee (e_1,\alpha_1) \equiv 0$	0	0	0	0
1	0	1	$\top$	0
$\top$	0	$\top$	$\top$	$\top$
(e_1,α_1)	0	0	$\top$	$(e_1,\alpha_1 * \alpha_2)$ if $e_1 = e_2$ 0 if $e_1 \neq e_2$

$\setminus$	0	1	$\top$	(e_2,α_2)
0	0	0	0	0
1	1	0	$\top$	1
$\top$	$\top$	$\top$	$\top$	$\top$
(e_1,α_1)	(e_1,α_1)	(e_1,α_1)	$\top$	$(e_1,\alpha_1 \setminus \alpha_2)$ if $e_1 = e_2$ (e_1,α_1) if $e_1 \neq e_2$

These set-theoretic operations on DDDs actually produce the best possible approximation of the result. More precisely, if d and d' are two DDDs, then the sum $d + d'$ (resp. the product $d * d'$, the difference $d \setminus d'$) is the “best defined” DDD whose meaning contains the set $\{S \cup S' \,|\, S \in [\![d]\!] \text{ and } S' \in [\![d']\!]\}$ (resp. the set $\{S \cap S' \,|\, S \in [\![d]\!], S' \in [\![d']\!]\}$, the set $\{S \setminus S' \,|\, S \in [\![d]\!] \text{ and } S' \in [\![d']\!]\}$).

The concatenation operator defined below corresponds to the concatenation of language theory.

$$d \cdot d' = \begin{cases} 0 & \text{if } d = 0 \vee d' = 0 \\ d' & \text{if } d = 1 \\ \top & \text{if } d = \top \wedge d' \neq 0 \\ (e, \alpha \cdot d') & \text{if } d = (e,\alpha) \end{cases}$$

Notice that any DDD may be defined using constants 0, 1, $\top$, the elementary concatenation $e \xrightarrow{x} d$ and operator $+$, as shown in the following example.

Example 1 *Let d_A be the DDD represented in left-hand side of Fig. 7, and d_B the right-hand side one.*

$$\begin{aligned} d_A &= a \xrightarrow{1} \left(a \xrightarrow{1} 1 + a \xrightarrow{2} b \xrightarrow{0} 1 \right) + a \xrightarrow{2} b \xrightarrow{3} 1 \\ d_B &= a \xrightarrow{1} \left(a \xrightarrow{1} 1 + a \xrightarrow{2} b \xrightarrow{0} 1 \right) + a \xrightarrow{2} \top \end{aligned}$$

Let us now detail some computations:

$$\begin{array}{rcl}
d_A + a\xrightarrow{2}a\xrightarrow{3}1 & = & a\xrightarrow{1}\left(a\xrightarrow{1}1 + a\xrightarrow{2}b\xrightarrow{0}1\right) + a\xrightarrow{2}\left(b\xrightarrow{3}1 + a\xrightarrow{3}1\right) \\
 & = & a\xrightarrow{1}\left(a\xrightarrow{1}1 + a\xrightarrow{2}b\xrightarrow{0}1\right) + a\xrightarrow{2}\top \\
 & = & d_B \\
\left(a\xrightarrow{1}1 * a\xrightarrow{2}1\right) * \top & = & 0 * \top = 0 \neq a\xrightarrow{1}1 * \left(a\xrightarrow{2}1 * \top\right) = a\xrightarrow{1}1 * \top = \top \\
d_A \setminus d_B & = & a\xrightarrow{2}\left(b\xrightarrow{3}1 \setminus \top\right) = a\xrightarrow{2}\top \\
d_B \cdot c\xrightarrow{4}1 & = & a\xrightarrow{1}\left(a\xrightarrow{1}c\xrightarrow{4}1 + a\xrightarrow{2}b\xrightarrow{0}c\xrightarrow{4}1\right) + a\xrightarrow{2}\top
\end{array}$$

4.1.3 Homomorphisms on DDDs

In order to iteratively compute the reachability set of an **L***f***P** program, we need to translate **L***f***P** instructions into DDD operations. These complex operations on DDDs are described by homomorphisms. Basically, an homomorphism is any mapping $\Phi : \mathbb{D} \to \mathbb{D}$ such that $\Phi(0) = 0$ and such that $\Phi(d_1) + \Phi(d_2)$ is better defined than $\Phi(d_1 + d_2)$ for every $d_1, d_2 \in \mathbb{D}$. The sum and the composition of two homomorphisms are homomorphisms.

So far, we have at one's disposal the homomorphism $d * \mathrm{Id}$ which allows to select the sequences belonging to the given DDD d; on the other hand we may also remove these given sequences, thanks to the homomorphism $\mathrm{Id} \setminus d$. The two other interesting homomorphisms $\mathrm{Id} \cdot d$ and $d \cdot \mathrm{Id}$ permit to concatenate sequences on the left or on the right side. For instance, a widely used left concatenation consists in adding a variable assignment $e_1 = x_1$ that is denoted $e_1\xrightarrow{x_1}\mathrm{Id}$. Of course, we may combine these homomorphisms using the sum and the composition.

However, the expressive power of this homomorphism family is limited; for instance we cannot express a mapping which modifies the assignment of a given variable. A first step to allow user-defined homomorphism Φ is to give the value of $\Phi(1)$ and of $\Phi(e\xrightarrow{x}d)$ for any $e\xrightarrow{x}d$. The key idea is to define $\Phi(e, \alpha)$ as $\sum_{x \in \mathrm{Dom}(e)} \Phi(e\xrightarrow{x}\alpha(x))$ and $\Phi(\top) = \top$. A sufficient condition for Φ being an homomorphism is that the mappings $\Phi(e, x)$ defined as $\Phi(e, x)(d) = \Phi(e\xrightarrow{x}d)$ are themselves homomorphisms. For instance, $inc(e, x) = e\xrightarrow{x+1}Id$ and $inc(1) = 1$ defines the homomorphism which increments the value of the first variable. A second step introduces induction in the description of the homomorphism. For instance, one may generalize the increment operation to the homomorphism $inc(e_1)$, which increments the value of the given variable e_1. A possible approach is to set $inc(e_1)(e, x) = e\xrightarrow{x+1}Id$ whenever $e = e_1$ and otherwise $inc(e_1)(e, x) = e\xrightarrow{x}inc(e_1)$. Indeed, if the first variable is e_1, then the homomorphism increments the values of the variable, otherwise the homomorphism is inductively applied to the next variables.

The formal definition of inductive homomorphisms can be found in [3]. The two following examples illustrate the usefulness of these homomorphisms to design new operators on DDD. The first example formalizes the increment operation. The second example is a swap operation between two variables. It gives a good idea of the techniques used to design homomorphisms for some variants of Petri net analysis.

Example 2 *This is the formal description of increment operation:*

$$\begin{array}{rcl}
inc(e_1)(e, x) & = & \begin{cases} e \xrightarrow{x+1} Id & \text{if } e = e_1 \\ e \xrightarrow{x} inc(e_1) & \text{otherwise} \end{cases} \\
inc(e_1)(1) & = & 1
\end{array}$$

Let us now detail the application of inc over a simple DDD:

$$\begin{array}{rcl}
inc(b)(a\xrightarrow{1}b\xrightarrow{2}c\xrightarrow{3}d\xrightarrow{4}1) & = & a\xrightarrow{1}inc(b)(b\xrightarrow{2}c\xrightarrow{3}d\xrightarrow{4}1) \\
 & = & a\xrightarrow{1}b\xrightarrow{3}Id(c\xrightarrow{3}d\xrightarrow{4}1) \\
 & = & a\xrightarrow{1}b\xrightarrow{3}c\xrightarrow{3}d\xrightarrow{4}1
\end{array}$$

Example 3 *The homomorphism $swap(e_1, e_2)$ swaps the values of variables e_1 and e_2. It is designed using three other kinds of homomorphisms: $rename(e_1)$, $down(e_1, x_1)$, $up(e_1, x_1)$. The homomorphism $rename(e_1)$ renames the first variable into e_1; $down(e_1, x_1)$ sets the variable e_1 to x_1 and copies the old assignment of e_1 in the first position; $up(e_1, x_1)$ puts in the second position the assignment $e_1 = x_1$.*

$$
\begin{array}{lcl}
swap(e_1,e_2)(e,x) & = & \begin{cases} rename(e_1)\circ down(e_2,x) & if\, e=e_1 \\ rename(e_2)\circ down(e_1,x) & if\, e=e_2 \\ e \xrightarrow{x} swap(e_1,e_2) & otherwise \end{cases} \\
swap(e_1,e_2)(1) & = & \top \\
 & & \\
rename(e_1)(e,x) & = & e_1 \xrightarrow{x} Id \\
rename(e_1)(1) & = & \top \\
 & & \\
down(e_1,x_1)(e,x) & = & \begin{cases} e \xrightarrow{x} e \xrightarrow{x_1} Id & if\, e=e_1 \\ up(e,x)\circ down(e_1,x_1) & otherwise \end{cases} \\
down(e_1,x_1)(1) & = & \top \\
 & & \\
up(e_1,x_1)(e,x) & = & e \xrightarrow{x} e_1 \xrightarrow{x_1} Id \\
up(e_1,x_1)(1) & = & \top
\end{array}
$$

Let us now detail the application of swap over a simple DDD which enlights the role of the inductive homomorphisms:

$$
\begin{array}{lcl}
swap(b,d)(a\xrightarrow{1}b\xrightarrow{2}c\xrightarrow{3}d\xrightarrow{4}1) & = & a\xrightarrow{1}swap(b,d)(b\xrightarrow{2}c\xrightarrow{3}d\xrightarrow{4}1) \\
 & = & a\xrightarrow{1}rename(b)\circ down(d,2)(c\xrightarrow{3}d\xrightarrow{4}1) \\
 & = & a\xrightarrow{1}rename(b)\circ up(c,3)\circ down(d,2)(d\xrightarrow{4}1) \\
 & = & a\xrightarrow{1}rename(b)\circ up(c,3)(d\xrightarrow{4}d\xrightarrow{2}1) \\
 & = & a\xrightarrow{1}rename(b)(d\xrightarrow{4}c\xrightarrow{3}d\xrightarrow{2}1) \\
 & = & a\xrightarrow{1}b\xrightarrow{4}c\xrightarrow{3}d\xrightarrow{2}1
\end{array}
$$

One may remark that $swap(b,e)(a\xrightarrow{1}b\xrightarrow{2}c\xrightarrow{3}d\xrightarrow{4}1) = a\xrightarrow{1}\top$.

Basically, the DDD data structure provides a compact encoding and a usage of memory similar to the BDD (Binary Decision Diagram). The operations calls and their results are stored in a operation cache. Data are stored in a unicity table based on a hash table. A canonical representation of the states allows for efficient comparison of diagrams. Set-theoretic operations are defined and families of inductive homomorphisms representing the operations of the **L***f***P** language can be user defined. The DDD structure meets both the requirement of efficiency and compacity.

4.2 Deriving an adequate model for verification purpose

The direct computation of reachable states from the initial model may be impossible in many cases. The combinatory explosion can have different origins: inherent complexity of the problem, level of detail of the model, size of the model, concurrency.

One main difficulty is to evaluate the complexity of the model and identify modifications in order to enable reachability set computation. Different models may have to be derived in accordance to the properties to be verified.

We distinguish two ways of making changes to the model: quantitative and qualitative modifications. The only purpose of these modification is to focus on the search of potential problems.

4.2.1 Quantitative alteration

The quantitatives modifications often lead to a redimensionning of data. Such modifications are easy to achieve if the size of components are clearly visible in the design. The choice is guided by the effect of the change on the complexity and by considering the symetries of the system. It may also permit to study different phases in the execution of the model, for instance initialisation, termination.

Example 4 *For instance, the simple model presented in this article depicts a client/server application composed of 4 clients communicating with 2 servers. Each client invokes 5 times a service that increment of 1 a parameter value. The service is supported by both servers, any available server can execute the service.*

If we call F(M, S, C) the number of possibles sequences of M messages by C clients to S server, a preliminary complexity analysis gives:

$$\begin{array}{ll}
F(M,1,C) & = F(M,1,(C-1)) * (((C-1)*M+1)^{M}) \\
 & > F(M,1,(C-1)) * (((C-1)*M)^{M}) \\
 & > F(M,1,(C-2)) * (((C-2)*M)^{M}) * (((C-1)*M)^{M}) \\
 & > (((C-1)! * M)^{M})^{(c-1)} \\
F(M,S,C) & > S^{F(M,1,C)}
\end{array}$$

In the case of our example, we reach a number of sequences F(6, 2, 4)= 2ˆ1.03 e+28 that doesn't even considers the additional complexity from the model. It becomes necessary to modify the model in order to compute the reachable states. We have been able to verify properties by reducing the size of the message sequence in various ways.

However, resizing the application for verification purpose may not be an acceptable solution. We currently study other approaches using abstract representations of set of states by means of DDD, [15].

4.2.2 Qualitative alteration

The qualitative changes include:

- simplification of the model that preserves information of the properties to be verified,
- addition of working hypothesis often linked to the domain of application and restricting the set of reachable states.

The simplification of the model often leads to the abstraction of some mechanisms. For instance, the merging of two transitions when conditions allow it, will reduce the interleaving and thus, will reduce the size of the reachability set. Such simplification may be acceptable when the validity of properties remain unaffected.

A working hypothesis defines what is a consistent state and what is a discardable state. Applying a working hypothesis on a set of states filters all inconsistent states with respect to the hypothesis.

Example 5 *For instance, additional hypothesis have been added to the initial model of a train system [7]. This model was describing the interactions and the behavior of a set of trains. The global state of the system was composed of the state of each train. The model was quite complex (>100 transitions) and its parallelism complicated the computation of the reachability set when dealing with a complex scenario. As time was not explicitly part of the model, valid states computed by firing sequences in the model would never exist in reality.*

The hypothesis that has been added had the effect of discarding all global states that consider the positions of trains at different dates. The application of this hypothesis had an immediate effect on state computation and allowed the verification of complex situations.

4.3 Computation of the reachable states

The computation of the reachable states consists in encoding the state of an **L***f***P** program by means of DDD and implementing the homomorphisms that compute the new states corresponding to the firing of transitions in the model.

4.3.1 Coding of a state

The state of an **L***f***P** specification is encoded into a DDD. Variables of the system are variables of the DDD and their values are attached to the arcs. The coding of the state represents a mapping of variables that can only be accessed sequentially by the inductive homomorphisms. Thus, the implementation of homomorphisms strongly depends on the coding of the state. Three important issues have to be solved in the encoding of an **L***f***P** state by means of DDD as done in [15]. First, a canonical form must be defined and conserved during the state computation. Second, the particular encoding of an **L***f***P** state must support several dynamic aspects. And last, the encoding of the state must guarantee the compatibility with theoretic-set operators.

Type declaration of v	DDD
Array v;	$v \xrightarrow{elt1} v \xrightarrow{elt2} ...$
Set v;	$v \xrightarrow{size} v \xrightarrow{elt1} v \xrightarrow{elt2} ...$
VectorMultiSet v;	$v \xrightarrow{nbvect} v \xrightarrow{vect1size1} v \xrightarrow{vect1elt1} v \xrightarrow{vect1elt2} ...$

Table 2: Data types representation by means of DDD.

Canonicity is a fundamental requirement that allows reachable state computation. A canonical representation of the state allows to reduce the comparison between DDD to a pointer comparison (O(1)). Canonicity is achieved by imposing construction constraints on all structures. The operations that manipulate the state or any structure within the state produce states that respect the constraints. For instance, an absolute order between elements of a set must be defined. Operations on a sets must respect the order.

The coding of the state requires a coding of the data structures and associated operations supported by the input language that respect the canonical form. In the case of **L*f*P**, we need to implement the following features:

- multi-sets of vectors,
- sets and multi-sets of scalars,
- instances of classes and media,
- variables at all scope (global, instance, local and class),
- arrays,
- stacks.

Dynamicity causes the handling of states with varying size while respecting the constraints of canonicity and operation compatibility. **L*f*P** provides dynamic features that are integrated in the state encoding:

- creation and destruction of instances,
- structures such as sets and multi-sets, FIFOs,
- method of function calls (stack).

By definition of the DDD structure, only one arc with a given value can be the output of a node. Thus, dynamicity can cause compatibility problems when adding DDD.

Example 6 *The following example recalls a typical case of incompatibility. The addition of the 2 DDD produces an invalid result where the node 'a' get 2 arcs valued '$V1$'.*

$a \xrightarrow{v1} a \xrightarrow{v3} c \xrightarrow{v4} ... + a \xrightarrow{v1} c \xrightarrow{v2} e \xrightarrow{v5} ... = a \xrightarrow{v1} \top$

We used prefixing techniques to construct DDD that are always compatible. For instance, a set is identified by one variable 'v'. A DDD representing the set 'v' can be seen as an assignment sequence using the same variable. Prefixing a set with a variable containing the size of the set will enforce the compatibility between DDD containing sets.

Typically, arrays and structures are coded using one variable. The semantic of a value (size, value, ...) depends on its position. The inductive homomorphisms use the position of the value to decode the structures.

The table 2 summarizes the encoding of some common structures.

Now that we can code all basic structures by means of DDD, we can show the encoding of the whole state. At the top of the hierarchy the different components of an **L*f*P** program appear in the following order:

1. global variables,
2. global binders (binder all),
3. instances,
4. end of state (special marker).

Each component is a DDD that represents a structure or a single variable like, for example, a special marker. The different fragments represented here are then assembled using the concatenation operator.

A canonical representation requires a particular ordering of object instances: the instances are grouped by class and an order must be defined in such a way that order of the instance remains the same whatever is the sequence of insertions. All object instances have the same structure shown below:

1. begin instance (special marker),
2. instance marker,
3. local media,
4. local binders,
5. instance variables,
6. program counter (*PC*),
7. stack (empty if not within a call),
8. end of instance marker.

The structure of the local media is simple:

1. media id (variable *me*),
2. message storage (multiset of vectors),
3. program counter (*PC*),

A block pushed on the stack is defined for each method. This block represents local data used by the method call. The corresponding DDD has the following shape:

1. parameters,
2. local variables,
3. return state.

Example 7 *The following DDD is an example of a global state of the client/server application with 1 client and 1 server. The variables prefixed with 'Glb' are the global variables, then comes the global binders and then the instances. In some cases, the use of symbols instead of values has been used to improve the legibility. Variables of the media in the client instance are easily recognized with the prefix 'RPC'.*

$\mathit{Glb.tmp1} \xrightarrow{0} \mathit{Glb.tmp2} \xrightarrow{0} \mathit{Glb.tmp3} \xrightarrow{0} \mathit{Glb.tmp4} \xrightarrow{0}$

$\mathit{binder\_all\_in} \xrightarrow{0} \mathit{binder\_all\_out} \xrightarrow{0}$

$\mathit{\_\_BeginOfInstance} \xrightarrow{\mathit{server.mk}}$

$\mathit{server.mk} \xrightarrow{0} \mathit{server.iv.\_discr} \xrightarrow{0} \mathit{server.pc} \xrightarrow{\mathit{\_none\_begin}}$

$\mathit{\_\_EndOfInstance} \xrightarrow{0}$

$\mathit{\_\_BeginOfInstance} \xrightarrow{\mathit{CLIENT.mk}}$

$\mathit{CLIENT.mk} \xrightarrow{0}$

$\mathit{RPC.me} \xrightarrow{0} \mathit{RPC.loc\_set} \xrightarrow{0} \mathit{RPC.loc.discr} \xrightarrow{1} \mathit{RPC.loc.id1} \xrightarrow{0} \mathit{RPC.loc.id2} \xrightarrow{0}$

$\mathit{RPC.pc} \xrightarrow{\mathit{\_none\_begin}}$

$\mathit{CLIENT.locbinder} \xrightarrow{0} \mathit{CLIENT.locbinder\_out} \xrightarrow{0}$

$\mathit{CLIENT.iv.i} \xrightarrow{0} \mathit{CLIENT.iv.id} \xrightarrow{1}$

$\mathit{CLIENT.pc} \xrightarrow{\mathit{\_none\_begin}}$

$\mathit{\_\_EndOfInstance} \xrightarrow{0}$

$\mathit{\_\_EndOfState} \xrightarrow{0} \mathit{1}$

4.3.2 A set of basic Homomorphisms for L*f*P

A set of basic homomorphisms has been implemented to realize simple operations of the **L*f*P** language. The composition operation ($\circ$) allows to combine the operations in order to design the often complex homomorphism that represents the firing of a transition. A basic homomorphism takes as input a set of states and produces a set of intermediate states.

We summarize in the table 4 the homomorphisms that deal with the functionning of the state machines and the evaluation of boolean guards. Table 6 describes the homomorphisms that deal with communications and media.

Typically, an homomorphism associated to a transition starts by checking if all boolean conditions are satisfied, using the *MarkFireable* homomorphism. *MarkFireable* creates a new state where one instance that satisfies the condition is marked. If multiple instances qualify, then a state will be created for each one of them. All the other homomorphisms use the mark to identify the instance to be processed.

When the computation of the new states obtained by the firing of a transition is finished, the homomorphism *ResetMark* concludes the firing by reseting all markers from the new states.

State Machine Homomorphisms	***Parameters***	***Description***
MarkFireable	Class, ExprG, ExprL	Mark exactly one instance of an object of **Class** if its state satisfies the global boolean expression **ExprG** and the local boolean expression **ExprL**. Returns an empty set of states if no instance qualifies.
AssignVar	Class, Var, Expr	Assigns to the variable **Var** the result of the evaluation of the expression **Expr**. Variables in the expression and the assigned variable are assumed to be in the scope of the marked instance of class **Class**.
ResetMark	Class	Reset the marker of a marked instance of **Class**. This homomorphism concludes any transition firing.
Goto	Class, State	Assign the program counter of the marked instance of **Class** with the new automaton state **State**.
ResetGlbTmp		Initialize all temporaries to 0, in order to avoid duplicated states.
VarAdd	Class, Var, Inc, Modulo	Increments the variable **Var** in the scope of the marked instance of **Class** with **Inc** and **Modulo** parameters.
ProcessKill	Class	Destroy (remove from the state) the marked instance of **Class**.
InsertInstance	Class	Insert an instance of **Class** in the state.

Table 4: Homomorphisms related to state machines.

Communication Homomorphism	***Parameters***	***Description***
SendCallMeth	Class, Binder, Target, Meth, NbParam, Param List	Generate a message containing a remote call of procedure **Meth** by an instance of **Class** to **Target** and store it in **Binder**. Parameters **Param List** are in the scope of the initiating instance. The binder can be either multi set, FIFO, local or global.
SendCallVMeth	same as SendCallMeth, Var	Same as **SendCallMeth**, except that the method is the value of a variable **Var** in the scope of the emitting instance.
ProcessReturn	Class, Binder, NbParam, Param. List	Process a return message concluding a remote procedure call (RPC) initiated by **Class** after reception of the message in **Binder**. Assign values stored in the message to the list of variables passed in the parameter list **Param. List**.
ReceiveMethCall	Class, Binder, Locals, NextState, ReturnState	Process the reception of an RPC message emitted by an instance of **Class** and stored in **Binder**. Save the context on the local stack of the receiving instance, push local variables **Locals** and **ReturnState**, set program counter value to **NextState**.
Select	Class, Binder, Branch Descr.	Process a branch (state with multiple output arcs) with boolean and message guards in an instance of **Class** . Messages are read from **Binder**. A descriptor **Branch Descr** specifies the precondition and postcondition starting each branch.
Binder2Media	Binder	Transfer messages from a **binder** to a media. The message contains the id of the destination. **Binder** can be multi set or FIFO, local or global.
PurgeMedia		Empty the Media
Media2Binder	Binder	Transfer messages from a Media to a Binder. **Binder** can be multi set, FIFO, local or global.

Table 6: Homomorphisms related to Communications.

4.3.3 Coding of a transition

In most of the cases, each transition is associated to an homomorphism. For performance reason, an homomorphism is associated to states with multiple output branches and processes all transitions at the head of each branch.

A transition homomorphism can be seen as the composition of a precondition evaluation homomorphism and a postcondition homomorphism.

The following fragment of code shows an example of a C++ function returning the composition of the two homomorphisms that constitutes the firing of the transition. The operator $\circ$ is implemented with the C++ operator &. Operands of & appears in the inverse order of their application.

The parameter *gs* is an instance of the class descriptor of the application. It provides the capabilities to generate the DDD structure corresponding to the initial state of the system and ways to refer to variables within the DDD structure.

```
GHom FIRE_SERVER_handle_request_succ(GenAppState &gs)
{
return
   POST_SERVER_handle_request_succ(gs)&
   TEST_SERVER_handle_request_succ(gs);
}
```

According to the **L***f***P** specification, the second transition of method *handle_request* increments the value of *num* and terminates the method call (cf fig. 6(b)). For implementation purpose, name are given to object. The transition is called *SERVER_handle_request_succ* and the state precondition of the transition is called *SERVER_handle_request_start*.

The following code implements the precondition and postcondition homomorphisms for this transition.

The *TEST_SERVER_handle_request_succ* homomorphism creates states with one marked instance of class *Server* that satisfies the transition precondition.

```
GHom TEST_SERVER_handle_request_succ(GenAppState &gs)
{
return
  MarkFireable<GenCSServer>(
          Cst(1), // no condition on global variables, always true
          (Var(GenCSServer::PC) ==   // precondition is only testing the state
           Cst(AllStates::SERVER_handle_request_start)));
}
```

The *POST_SERVER_handle_request_succ* homomorphism realizes the action associated to the transition. The composition starts by incrementing the local variable num by the constant 1.

Then a return message from RPC is generated. This message carries a discriminant value *gs.Server._discr* that was set when the RPC was initiated. It contains the identifier of the initiator. The identifier of the message is *AllStates* :: *SERVER__none_ret_handle_request* specifies the type of the message. The message contains only one parameter : *num*.

Note that the construction of *gs* follows the same hierarchy as the original **L***f***P** program. For example *gs.Server.handle_request.num* refers to the local variable of method *handle_request*. The value of the C++ variable contains the identifier of the corresponding variable in the DDD.

The *Pop* homomorphism restores the context at the end of the method call. The *AssignVar* homomorphism reset the value of a local variable in order to limitate the generation of states. The *ResetMark* homomorphism concludes the firing of the transition by reseting the instance marker.

```
GHom POST_SERVER_handle_request_succ(GenAppState &gs)
{
    ResetMark<GenCSServer>()&  // reset the mark concluding
    AssignVar<GenCSServer>(gs.Server._discr, Cst(0))& // reset this local variable
    Pop(gs.Server)&   // exit from the method call
    scSendCallMeth<GenCSServer>(true, gs.BinderAllId_out,
                                gs, gs.Server._discr,
                                AllStates::SERVER__none__ret_handle_request,
                                1, gs.Server.handle_request.num)&
    Add<GenCSServer>(gs.Server.handle_request.num, 1, -1);
}
```

4.3.4 Reachable states computation

The simple computation of the reachable state space is an iterative process where all transitions are applied during one iteration.

```
Begin
DDD CURRENT
Set of transitions TSET
Create the initial state CURRENT

DDD ACC=NULL
Repeat
    DDD OLD_CURRENT=CURRENT
    For each transition T in TSET:
        CURRENT=CURRENT+T(CURRENT)
        BOOL GotNew=(ACC+CURRENT!=ACC)
        ACC=ACC+CURRENT
        CURRENT=CURRENT-OLDCURRENT
Until (!GotNew)
Return ACC
End
```

A preliminary filtering allows to eliminate transitions that have no chance to be fired. In our case this preliminary filtering is done by testing the simple condition on the program counter of each instance. A static structure generated by the verification program gives the list of potentially firable transitions for each value of the program counter of any instance. The list of potentially firable transition is updated on the fly when new states are computed. This improvement made a noticable difference in performance.

A number of algorithms can be applied for the computation of the reachable states. Their study is beyond the scope of this paper.

4.4 Verification of properties

The verification process handles generic properties of distributed systems such as liveness, search for deadlocks, coverage, and specific properties tied to the system such as assertions on variables of the state. In any case, the verification of a property consists in expressing it by means of an homomorphism that will be applied to the set of reachable states.

4.4.1 Searching for deadlocks

If *T1,.. Tn* are the transitions of the system and *Hom_T1,.. Hom_Tn* the corresponding homomorphisms. Let *Hom_Sum* be the sum *Hom_Sum=Hom_T1 + Hom_T2 + .. Hom_Tn*. *Hom_Sum* will return the *null* homomorphism if it applies to a blocking state.

We reproduce the output generated by the verification program in the computation of blocking states of the system with 2 servers, 2 clients sending 2 messages. To simplify the reading, the DDD containing the state of the system is displayed by putting between parenthesis the value associated to a variable. Comments have been added to improve legibility. In the following, only one instance of each class is shown to save space.

```
[ TERMINALS =
// global variables:
 Glb.tmp1(0) Glb.tmp2(0) Glb.tmp3(0) Glb.tmp4(0)

//global binders:
 binder_all_in(0)  binder_all_out(0)

// instances
 __BeginOfInstance(server.mk)
// marker
 server.mk(0)
//instance variables & program counter
 server.iv._discr(0)  pc=server.pc(_none_begin)
 __EndOfInstance(0)

... // others instance of server
```

```
 __BeginOfInstance(CLIENT.mk)
 CLIENT.mk(0)

//media of client, instance variables
 RPC.me(0) RPC.loc_set(0) RPC.loc.discr(1) RPC.loc.id1(1) RPC.loc.id2(1)
 pc=RPC.pc(_none_begin)

// client local binders
 CLIENT.locbinder(0)  CLIENT.locbinder_out(0)

//client instance variables
 CLIENT.iv.i(2) CLIENT.iv.id(1)

 pc=CLIENT.pc(_none_end)
 __EndOfInstance(0)

... // others instance of client

__EndOfState(0)]
```

Only one state has been found with the following characteristics:

- no communication are pending
- all servers are ready to process a request
- all clients are terminated

So the only blocking state is a terminaison state as it is defined in the model. This model has no invalid deadlocks.

4.4.2 Coverage test

A routine showing a compact view of the state space shows all possible values assigned to all variables composing the state. A look at the program counter of all instances shows that all the states have been explored. Additional states may be added to the model to deal with transitions sharing the same input and output state. This test can also be useful to:

- estimate or determine the size of the communication buffers (binders),
- define the domain of variables

In this example we only show the results for one instance of each class.

```
[ COVERAGE =

Glb.tmp1(0) Glb.tmp2(0) Glb.tmp3(0) Glb.tmp4(0)

binder_all_in(0 1 2 ) binder_all_out(0 1 2 )
__BeginOfInstance(536870922 )
server.mk(0 )
server.iv._discr(0 1 2 )
server.pc(_none_begin handle_request_begin handle_request_start)
__EndOfInstance(0 )

...

__BeginOfInstance(536870931 )
CLIENT.mk(0 )
RPC.me(0 ) RPC.loc.discr(0 1 ) RPC.loc.id1(0 1 ) RPC.loc.id2(0 1 )
RPC.pc(_none_begin _none_p1 _none_p2 _none_p3 )

CLIENT.locbinder(0 1 ) CLIENT.locbinder_out(0 1 )

CLIENT.iv.i(0 1 2 ) CLIENT.iv.id(1 )
CLIENT.pc(_none_begin _none_start _none_process _none__wait_handler_request _none_end ),
__EndOfInstance(0 )

...

__EndOfState(0 )
]
```

4.4.3 Assertions on variables

An assertion on a variable can be easily translated into an homomorphism that evaluates the boolean expression corresponding to the negation of the assertion. This homomorphism is applied on the set of reachable states and should return the empty set, if the original assertion is satisfied.

Example 8 *The computation of states of the system with 2 servers, 2 clients sending 2 messages has been performed.*

We now would like to show that the set of states that satisfy the negation of the expression ($(Client.PC = client\_none\_end) \Rightarrow (client.i = 2)$) is empty.

The computation of such a set can be easily done by building the homomorphism that will check the existence of such states. We directly reuse the MarkFireable homomorphism that allows for checking boolean expressions on states.

```
GHom CheckAssertTerm(GenAppState &gs){
return
  MarkFireable<GenCSClient>(
          Cst(1), // no condition on global variables, always true

          (Var(GenCSClient::PC) == Cst(AllStates::CLIENT__none_end))&&
          (Var(gs.Client.i)!!=2)
          );
}
```

5 conclusion and future work

In this paper, we have presented how we could handle the model checking of **L*f*P**, a high-level specification language. The main problem raised in this study resides in dynamic aspects proposed in **L*f*P**: creation of processes, RPC mechanisms, addressing, etc.

DDD were successfully used for the analysis of Petri nets [3] and were also experimented successfully for **L*f*P**. To achieve model checking, a basic set of homomorphisms has been defined and implemented to enable the computation of the reachability set of an **L*f*P** specification. An execution environment supporting basic debugging capabilities and the computation of the reachable set of an **L*f*P** model has been implemented. We have started developping a code generator that translates an XML file containing the **L*f*P** program and generates C++ files that will be linked to the execution environment.

The efficiency of the DDD operations and the compacity of the structure allowed the computation of large reachability sets [7] of complex **L*f*P** models. We summarized the main technical difficulties in the representation of a **L*f*P** program by means of DDD: canonical representation of the state, support for dynamicity, diversity and complexity of the mechanisms to implement.

We also address complexity issues that may be solved by modification of the model. The derivation of models for specific verification purposes is currently mostly supported by the user expertise. We believe that an abstract representation of the state may help solving these issues [15].

Compiler optimization techniques will be developped to optimize the construction of homomorphisms. A language for the specification of properties needs to be developped as well as associated translation tools to generate the homomorphisms.

The development of such tools will provide a wide access to the efficiency of the DDD structure and will free the user from the tedious manual implementation of homomorphisms.

References

[1] B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, 1978.

[2] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, Orlando, Florida, June 1990. ACM/IEEE, IEEE Computer Society Press.

[3] J. M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P. A. Wacrenier. Data decision diagram for Petri nets analysis. In *Proc. of ICATPN 2002*, volume 2360 of *LNCS*, pages 101–120. Springer Verlag, june 2002.

[4] F. Gilliers, F. Kordon, and D. Regep. Proposal for a Model Based Development of Distributed Embedded Systems. In *2002 Monterey Workshop : Radical Innovations of Software and Systems Engineering in the Future*. Springer Verlag, 2002.

[5] F. Gilliers, F. Kordon, and J-P. Velu. Generation of distributed programs in their target execution environment. In *Proceedings of the 15th International Workshop on Rapid System Prototyping*, pages 90–97. IEEE Computer Society, 2004.

[6] ISO. *Information Technology – Programming Languages – Ada*. ISO, February 1995. ISO/IEC/ANSI 8652:1995.

[7] F. Kordon and M. Lemoine, editors. *Formal Methods for Embedded Distributed Systems: How to Master the Complexity*. Kluwer Academic, 2004. ISBN:1-4020-7997-4.

[8] F Kordon and Luqi. An Introduction to Rapid System Prototyping. *IEEE Trans. Softw. Eng.*, 28(9):817–821, 2002.

[9] F. Kordon, I. Mounier, E. Paviot-Adet, and E. Regep. Formal verification of embedded distributed systems in a prototyping approach. In *Monterey Workshop 2001: on Engineering Automation for Software Intensive System Integration*, June 2001.

[10] N. Leveson. Software engineering: Stretching the limits of complexity. *Communications of the ACM*, 40(2):129–131, 1997.

[11] Luqi and J. Goguen. Formal methods: Promises and problems. *IEEE Software*, 14(1):73–85, January / February 1997.

[12] OMG. Model Driven Architecture (MDA), Document number ormsc/2001-07-01. Technical report, OMG, 2001.

[13] D. Regep and F. Kordon. LfP: A Specification Language for Rapid Prototyping of Concurrent Systems. In *Proceedings of the 12th International Workshop on Rapid System Prototyping*, pages 90–97. IEEE Computer Society, 2001.

[14] D. Regep, Y. Thierry-Mieg, and F. Kordon. Modélisation et vérification de systèmes répartis: une approche intégrée avec LfP. In *Proceedings of AFADL'03*, January 2003.

[15] Y. Thierry-Mieg, J-M. Ilié, and D. Poitrenaud. A symbolic symbolic state space representation. In *Proceedings of the 24th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'04)*, Madrid, Spain, September 2004. Springer Verlag. To appear.