Dynamic Verification of SystemC with Statistical Model Checking

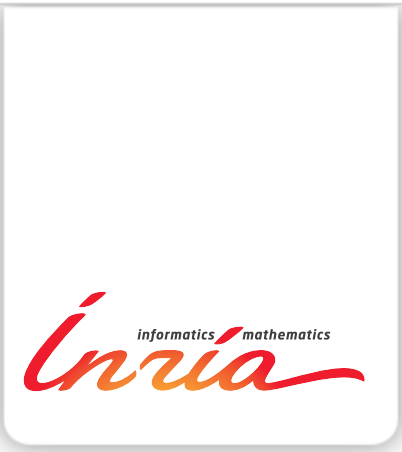
van Chan Ngo, Axel Legay, Jean Quilbeuf

Dynamic Verification of SystemC with Statistical Model Checking

Van Chan Ngo, Axel Legay, Jean Quilbeuf

RESEARCH REPORT
N° 8644
October 2014
Project-Team ESTASYS

ISSN 0249-6399 ISRN INRIA/RR--8644--FR+ENG



Dynamic Verification of SystemC with Statistical Model Checking

Van Chan Ngo, Axel Legay, Jean Quilbeuf

Project-Team ESTASYS

Research Report n° 8644 — October 2014 — 25 pages

Abstract: Many embedded and real-time systems have a inherent probabilistic behaviour (sensors data, unreliable hardware,...). In that context, it is crucial to evaluate system properties such as "*the probability that a particular hardware fails*". Such properties can be evaluated by using probabilistic model checking. However, this technique fails on models representing realistic embedded and real-time systems because of the state space explosion. To overcome this problem, we propose a verification framework based on *Statistical Model Checking*. Our framework is able to evaluate probabilistic and temporal properties on large systems modelled in SystemC, a standard system-level modelling language. It is fully implemented as an extension of the Plasma-lab statistical model checker. We illustrate our approach on a multi-lift system case study.

Key-words: SystemC, Statistical Model Checking, Quantitative Verification, Temporal Properties, Formal Verification

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Dynamic Verification of SystemC with Statistical Model Checking

Résumé : Beaucoup de systèmes embarqués et temps réel ont un comportement probabiliste inhérente (données de capteurs, matériels peu fiables, ...). Dans ce contexte, il est crucial pour évaluer les propriétés du système telles que " la probabilité' qu'un matériel particulier ne fonctionne pas". Ces propriétés peuvent être évaluées en utilisant le probabilistic model checking. Cependant, cette technique échoue sur les modèles représentant les systèmes embarqués et temps réel réalistes en raison de l'espace explosion de l'état. Pour surmonter ce problème, nous proposons un cadre de vérification sur la base *Statistical Model Checking*. Notre cadre est en mesure d'évaluer les propriétés probabilistes et temporelles sur les grands systèmes modélisés dans SystemC, un langage de modélisation au niveau du système standard. Il est pleinement mis en oeuvre comme une extension du modèle statistique Plasma-lab checker. Nous illustrons notre approche sur une étude de cas du système multi-ascenseur.

Mots-clés : SystemC, Statistical Model Checking, Quantitative Verification, Temporal Properties, Formal Verification

Contents

1 Introduction

SystemC (IEEE Standard 1666-2005) is a system-level design framework that can model both hardware and software components. It becomes increasingly prominent in domain of embedded systems (e.g., System-on-Chips [6]). Complex electronic systems and software control units can be combined into a single model, to simulate and observe the behavior of the system. These systems have both *temporal* and *probabilistic* behavior, for example, in critical real-time embedded systems, will have strict timing requirements and probabilistic effects such as the use of randomisation, sensor data, or component failures. Obviously, the verification of system correctness must take into account the both temporal and probabilistic properties, "*the probability of a particular event occuring within 1 second*" for example.

We consider the following examples which can be modeled as set of thread processes in SystemC and present some properties that cannot be verified with the simulation kernel.

Consider a multi-lift systems which is controlled by a software system. A multi-lift system consists of a hierarchy of components, e.g., the system contains multiple lifts, floors, users, etc.; a lift contains a panel of buttons, a door and a lift controller; a lift controller may contain multiple control units. Ideally, the behavior system need to be formally verified to satisfy desirable properties. For instance, one of the properties is:

If a user has requested to travel in certain direction, a lift should not pass by (i.e., traveling in the same direction without letting the user in).

However, this property is not satisfied. Typically, once a user presses a button on the external panel at certain floor, the controller assigns the request to the "nearest" lift. If the nearest lift is not the first reaching the floor in the same traveling direction, the property is violated. One counterexample that could be returned by a standard model checker is that the lift is held by some user for a long time so that other lifts pass by the floor in the same direction first. One solution is re-assign all external requests every time a lift travels to a different floor. Due to the complexity, many existing lift systems do not support re-assigning requests. The question is then:

What is the probability of violating the property, with typical randomized arrival of user requests from different floors or from the button panels inside the lifts?

Quantitative verification, in general, consists of a formal model capturing the system's behavior and an algorithm to analyse formally specified quantitative properties. The algorithm for computing the measures in quatitative properties depends on the class of sytems being considered and the temporal logic used for specifying the properties. Many algorithms with the corresponding mature tools have been discovered based on model checking technique that compute the probability by a numerical approach [26, 4, 7, 19, 12]. Timed automata with mature verification tools such as UPPAAL [16] are used to verify real-time systems. For a variety of probabilistic systems, the most popular modeling formalism is Markov chain or Markov decision processes (MDPs), for which probabilistic model checking tools such as PRISM [13] and MRMC [15] have been used.

Numerical model checking-based quantitative verification has many challenges, although it is widely used and has been successfully applied to the verification of a range of timed and probabilistic systems. One of the main challenges is that the complexity of the algorithms in terms of execution time and memory space. Therefore, scaling the verification to large systems is challenge duce to the traditional problem of model checking, *state explosion*. An alternative way to verify quantitative properties is simulation-based approach [1]. A monitoring procedure is used to produce a finite set of system's executions, and deduce whether the desired property is satisfied by this set or not. A *hypothesis testing* method is employed to provide a statistical evidence for the satisfaction or vilation of the property. Clearly, comparing to the numerical

approach, a simulation-based solution does not provide an exact answer. However, one can bound the probability of making error and the confindence level. Simulation-based approaches do not make a formal model of the system-under-verification (SUV), thus they are known to be far less execution time and memory space than numerical approaches. For some real-life systems, they are the only one option [28].

In this work, we propose a framework to verify properties of systems modeling in SystemC with both real-time and probabilistic characteristics. The framework contains two main components: a *monitor* that observes executions of the SUV based on the verifying properties, and a statistical model checker implementing a set of hypothesis testing algorithms. We use the similar techniques proposed by Tabakov et al. [24] to automatically generate the monitor corresponding to the user-defined verifying properties and the SUV. The statistical model checker is implemented as a plugin of the checker Plasma Lab [2]. Users express desired properties in *Bounded Linear Temporal Logic* (BLTL), our framework allows users to adapt BLTL for SystemC by adding the extended primitives to form atomic propositions, and user-defined time resolutions. First, these primitives help users exposing the values of data members of a SystemC module (e.g., the *protected* and *private* attributes), matching of a specific method and its execution, for intance, the start and end of execution of a SystemC module's method, the values of passing parameters and return values of a specific function, statements being executed in the user-code, or the time point a specific event notifies. Second, in general, for SystemC, time resolution which indicates when a state in the execution of the SUV should be sampled is the boundary of clock cycles. In our framework, users can define their own time resolutions, for example, the boundary of *delta-cycles*, the time at which a particular event is notified.

2 The SystemC Language

SystemC is a system-level design framework that can model both hardware and software components. Complex electronic systems and control units can be combined into a single model, to simulate and observe the behavior. In 2005 SystemC became standard as IEEE 1666-2005.

The design process can be parallel with SystemC since it allows blocks implemented at different abstraction levels to run together in the same model. Communication between modules is specified using well-defined interfaces, which allows two modules that conform to the same interface to be swapped seamlessly. Therefore, designers can try alternative approaches early in the design process, before committing to a particular architecture.

SystemC is a library of C^{++} classes that means every SystemC model can be compiled with standard C^{++} compiler and linked with SystemC library to produce an executable specification. SystemC also provides an event-driven mechanisms for simulating parallel execution of the model's processes. The kernel borrows the *delta-cycle* concept from hardware design languages.

2.1 Language Features

2.1.1 Time Model

In SystemC, integer values are used as discrete time model. The smallest quantum of time that can be represented is called *time resolution* meaning that any time value smaller than the time resolution will be rounded off. The available time resolutions are femtosecond, picosecond, nanosecond, microsecond, millisecond, and second. SystemC provides functions to set time resolution and declare a time object, for example, the following statements set the time resolution to 10 picosecond and create a time object `t1` representing 20 picoseconds:

```
1 sc_set_time_resolution(10,SC_PS);
2 sc_time t1(20,SC_PS);
```

2.1.2 Modules

A SystemC model is composed of *modules*, which define the behavior of the modeled systems. Module data is inaccessible by the other modules of the system unless it is exposed explicitly. Thus, modules can be developed independently and can be reused. The skeleton of a module is given in Listing 1:

Listing 1 : Skeleton code of SystemC module

```
1 SC_MODULE(Name) {
2    // prots, processes, internal data, etc
3
4    SC_CTOR(Name) {
5       // Body of constructor,
6       // Process declaration,
7       // Sensitivities, etc.
8    }
9 };
```

In general, a module contains:

- ports which are used to communicate with the environment;
- processes that represent the functionality of the module;

- local data and channels to represent the states of module and communication between processes; and
- hierarchically, other modules.

The `SC_MODULE` marco defines a class named `Name` and the `SC_CTOR` defines its constructor, which maps designeated methods to *processes* and declares *event sensitives*. A module can be *instantiated* which is similar to the instantiation of class. However, to instantiate a module the user is required to suply a name to the instance. For example, to declare an instance of module `Name` named "xy", we state:

```
1 Name xy(``xy'');
```

2.1.3 Interfaces, Ports, and Channels

In hardware modeling languages, the hardware signal is used as the medium for communication and synchronization between processes. The communication and synchronization are abstracted in SystemC as *interfaces*, *ports*, and *channels* to provide the flexibility. Channels hold and transmit data, and an interface is a "window" into a channel that describes the set of operations of the channel. Ports are proxy objects that facilitate access to channels through interfaces.

An interface which is derived from the abstract base class `sc_interface` consists of a set of operations by specifying their *signatures*. We consider a simple interface used with the hardware signal: `sc_signal_in_if<T>` which is derived directly from `sc_interface` and is parameterized by data-type `T`. It provides a virtual method `read()` that returns a constant reference to `T`.

A module uses its ports to connect to and communicate with its environment via a channel's interface. Ports can be considered as the pins of a hardware component. A channel has to implement a port's interface to connect to the port. Any specialized port is derived from the port base class `sc_port`.

```
1 // N is number of channels that can be connected to the port
2 sc_port<if<ty>,N> p;
```

Here we declare a port `p`, which can access the number of N channels through the interface `if` with type `ty`. SystemC provides the following predefined ports, called *signal ports*: `sc_in`, `sc_out`, and `sc_inout` for input, output, and input-output ports. For example:

```
1 sc_in<int> a;
2 sc_out<int> b;
```

Here we define an input and an ouput ports named `a` and `b`, respectively, all of data type `int`.

A channel is an implementation of an interface by providing concrete definitions of all of the interface's operations. Thus, different channels may implement the same interface in different ways. On other hand, a channel can implement more than one interface. Channels provode means fo communication between modules and between processes within a module. The following are several classes of channels in SystemC:

- A primitive channel does not contain any hierarchy or process and is derived from the base class `sc_prim_channel`. SystemC contains several built-in channels: `sc_signal`, `sc_mutex`, `sc_semaphore`, and `sc_fifo`.
- A hierarchical channel can have a structure, contain processes and access directly other channels. All hiercarchical channels are derived from the base class `sc_channel` that is just a redefinition of the class `sc_module`. Thus from a language point of view a hierarchical channel is nothing but a module.

2.1.4 Processes

Processes which provide the mechanism for simulating concurrent behavior are basic units of functionality. A process must be contained in a module and declared to be a process in the module's constructor. There are two kinds of processes: *method process* (with macro `SC_METHOD`) and *thread process* (with macro `SC_THREAD`).

When triggered, a method process always executes its body until the return. That means it only returns the control to the kernel when it is at the end of its body. A thread process, on the other hand, may have its execution suspended by calling the library function `wait()` or any of its variants. All local variables and the point of suspension are saved. When the execution is resummed, it will continue from that point, rather than from the beginning of the process. Thus, unlike method processes, a thread process implicitly keeps its state of execution. This feature makes thread process more expressive than method process, for example, by means of `wait` statements multicycle behavior may be easily described by thread process, but would require more effort with method process.

2.1.5 Events

An event is an object C^{++} of class `sc_event`, that determines whether and when a process's execution should be triggered or resumed. By default, SystemC defines for each `sc_signal` an associated event `value_changed_event()`, that is notified whenever the value of the signal is written or modified. The effect of the notification (by calling `e.notify()`) of `e` causes all processes that are sensitive to it (or use `wait(e)`) to be triggered or resumed. The notification can have different effects, depending on its argument:

- `notify()` without arguments makes *immediate notification* and puts all processes that are sensitive to the event to the pool of runnable processes before the return of the function call `notify()`.
- `notify()` with arguments as zero time unit (e.g., `SC_ZERO_TIME`) delays the effect of the event notification until all currently triggered processes have finished executing. The simulation clock does not advance during this delay. It is called *delta-delayed* notification.
- `notify()` with arguments as non-zero time units delays the effect of the notification by the number of time units. The argument value is added to the simulation clock, and the event is put in a queue. It is called *time-delayed* notification.

All event notifications are pending, they can be *canceled*, which removes any pending effect of the event. A process can wait for an event in some bounded of time. For example, `wait(2,SC_SEC,e)` resumes the execution after 2 seconds of simulation time if `e` is notified earlier.

2.1.6 Sensitivity

A module can be sensitive to events which is declared via the `«` operator as follows:

```
1 // special attribute of module named sensitive
2 sensitive << ‘‘event1’’ << ‘‘event2’’;
```

If the list of events remains the same throughout simulation, it is called a *static sensitivity list*. Otherwise, it is a *dynamic sensitivity list*. That is, during simulation a thread process may suspend itself and designate a specific event `e` as its current waiting event. Then, only this event can resume the execution of the process (the static sensitivity list is ignored). A process can wait for an event, composite events, or for a time:

```
wait(e);
wait(e1 & e2 & e3);
wait(e1 | e2 | e3);
wait(10,SC_NS);
```

2.2 Simulation Kernel

The SystemC simulation kernel is an event-driven simulation. The structural information is represented by the modules and ports. Only one thread is dispatched by the scheduler to run at a time point, and the scheduler is non-preemptive, that is, the running process returns control to the kernel only when it finishes executing or explicitly suspends itself by calling `wait()`.

Like hardware modeling languages, the SystemC scheduler supports the notion of delta-cycles [18]. A delta-cycle lasts for an infinitesimal amount of time and is used to impose a partial order of simultaneous actions which interprets zero-delay semantics. Thus, the simulation time is not advanced when the scheduler processes a delta-cycle. During a delta-cycle, the scheduler executes actions in two phases: the *evaluate* and the *update* phases. The simulation semantics of the SystemC scheduler is presented as follows:

1. *Initialize*. During the initialization, each process is executed once unless it is turned off by calling `dont_initialize()`, or until a synchronization point (i.e., a `wait`) is reached. The order in which these processes are executed is unspecified.

2. *Evaluate*. The kernel starts a delta-cycle and run all processes that are ready to run one at a time. In this same phase a process can be made ready to run by an event notification.

3. *Update*. Execute any pending calls to `update()` resulting from calls to `request_update()` in the evaluate phase. Note that a primitive channel uses `request_update()` to have the kernel call its `update()` function after the execution of processes.

4. The kernel enters the delta notifcation phase where notified events trigger their dependent processes. Note that immediate notifications may make new processes runable during step 2. If so the kernel loops back to step 2 and starts another evaluation phase and a new delta-cycle. It does not advance simulation time.

5. If there are no more runable processes, the kernel advances simulation time to the earliest pending timed notification. All processes sensitive to this event are triggered and the kernel loops back to step 2 and starts a new delta-cycle. This process is finised when all processes have terminated or the specified simulation time is passed.

The simulation semantics can be represented by the pseudo code in Listing 2.

Listing 2 : Simulation Semantics of SystemC

```
PC // All primitive channels
P // All processes
R ← ∅ // Set of runnable processes
D ← ∅ // Set of pending delta notifications
U ← ∅ // Set of update requests
T ← ∅ // Set of pending timed notifications
// Start elaboration: collect all update requests in U
for all chan ∈ U do
   run chan.update()
end for
for all p ∈ P do
```

```
12    if p is initialized and p is not clocked thread then
13       R ← R ∪ p // Make p runnable
14    end if
15 end for
16 for all p ∈ P do
17    if p is triggered by an event in D then
18       R ← R ∪ p
19    end if
20 end for // End of initialization phase
21
22 repeat
23    while R ≠ ∅ do // New delta-cycle begins
24       for all r ∈ R do // Evaluation phase
25          R ← R \ r
26          run r until it calls wait() or returns
27       end for
28       for all chan ∈ U do // Update phase
29          run chan.update()
30       end for
31       for all p ∈ P dp // Delta notification phase
32          if p is triggered by an event in D then
33             R ← R ∪ p // Make p runnable
34          end if
35       end for // End of delta-cycle
36    end while
37
38    if T ≠ ∅ then
39       Advance the simulation clock to the earliest timed delay t
40       T ← T \ t
41       for all p ∈ P do // Timed notification phase
42          if t triggers p then
43             R ← R ∪ p // Make p runnable
44          end if
45       end for
46    end if
47 until end of simulation
```

2.3 Example: Producer-consumer Model

This example shows the communication between modules throught a shared channel. The model consists of two modules `Producer` and `Consumer` that communicate via a fixed length FIFO. It demonstrates how construct these modules and the communication channel using `sc_interface` and `sc_channel` based classes. In this example, we will build a simple channel for character writing and reading. Listing 3 shows a character interface definition. We can see that our interfaces have to derived from the based class `sc_interface` and have only pure virtual methods.

2.3.1 Interfaces

Listing 3 : Communication Interfaces

```
1 #include <systemc.h>
2
3 class fifo_write_if : virtual public sc_interface {
4 public:
5    virtual void fifo_write(char) = 0;
6    virtual void fifo_reset() = 0;
7 };
8
```

```
9  class fifo_read_if : virtual public sc_interface {
10 public:
11    virtual void fifo_read(char&) = 0;
12    virtual int fifo_num_available() = 0;
13 };
```

2.3.2 Modules

The `Producer` writes a character to the FIFO with the probability of 50% and the `Consumer` reads a character from the FIFO with the same probability for every milisecond. We model the probability with the randomization function of standard C^{++} library. The following listings implement the specification of the `Producer` and the `Consumer`.

Listing 4 : File producer.h

```
1  #ifndef PRODUCER_H
2  #define PRODUCER_H
3
4  #include <systemc.h>
5  #include <tlm.h>
6  #include "fifo.cpp"
7
8  SC_MODULE(Producer) {
9     // Definitions of ports
10    sc_port<fifo_write_if> out; // output port
11
12    // Definition of processes
13    void producer_process();
14
15    // Constructor
16    SC_CTOR(Producer) {
17       // Process registration, sensitiving lists, etc.
18       SC_THREAD(producer_process); // thread process
19
20       // No sensitiving list
21    }
22 };
23
24 #endif
```

Listing 5 : File producer.cpp

```
1  #include "producer.h"
2
3  void Producer::producer_process() {
4     const char* str = "xxxxxxxxx";
5     const char* p = str;
6
7     while (true) {
8           if (rand() % 2) {
9              out->fifo_write(*p);
10          p++;
11          if (!*p) {
12             p = str;
13          }
14       }
15
16       wait(1,SC_MS); // waits for 1 milisecond
17    }
18 }
```

Listing 6 : File consumer.h

```
1 #ifndef CONSUMER_H
2 #define CONSUMER_H
3
4 #include <systemc.h>
5 #include <tlm.h>
6 #include "fifo.cpp"
7
8 SC_MODULE(Consumer) {
9 public:
10    // Definitions of ports
11    sc_port<fifo_read_if> in; // input port
12
13    // Definition of processes
14    void consumer_process();
15
16    // Constructor
17    SC_CTOR(Consumer) {
18       // Process registration, sensitiving lists, etc.
19       SC_THREAD(consumer_process); // thread process
20
21       // No sensitiving list
22    }
23 };
24 #endif
```

Listing 7 : File consumer.cpp

```
1 #include "consumer.h"
2
3 void Consumer::consumer_process() {
4    char c;
5    while (true) {
6          if (rand() % 2) {
7             in->fifo_read(c);
8             std::cout << c << "(" << sc_time_stamp() << ")";
9       }
10
11       wait(1,SC_MS); // waits for 1 milisecond
12    }
13 }
```

The producer has one thread which runs infinitely to write a character into the FIFO that it connects to via the output port using the interface `fifo_write_if` (line 10 in `producer.h`). The producer's process suspends itself explicitly by calling `wait()` (line 16 in `producer.cpp`). The implementation of the consumer is in the similar way.

2.3.3 Channel

Now we implement the FIFO that is derived from `fifo_write_if` and `fifo_read_if` as in Listing 8.

Listing 8 : File fifo.cpp

```
1 #include <systemc.h>
2 #include "fifo_if.h"
3
4 class fifo : public sc_channel, public fifo_write_if, public fifo_read_if {
```

```
5  private:
6     enum e {max = 10}; // capacity of the fifo
7     char data[max];
8     int num_elements, first;
9     sc_event write_event, read_event;
10
11 public:
12    fifo(sc_module_name name) : sc_channel(name), num_elements(0), first(0) {}
13
14    void fifo_write(char c) {
15       if (num_elements == max) {
16          wait(read_event);
17       }
18
19       data[(first + num_elements) % max] = c;
20       ++num_elements;
21       write_event.notify();
22    }
23
24    void fifo_read(char &c) {
25       if (num_elements ==  0) {
26          wait(write_event);
27       }
28
29       c = data[first];
30       --num_elements;
31       first = (first + 1) % max;
32       read_event.notify();
33    }
34
35    void fifo_reset() {
36       num_elements = 0;
37       first = 0;
38    }
39
40    int fifo_num_available() {
41       return num_elements;
42    }
43 };
44
45 #endif
```

Since the FIFO is bounded capacity meaning that it may be full when the producer tries to write a character, or it may be empty when the consumer attempts to read a character. Thus, the implementation of the FIFO must handle this situation using the *blocking* `read()` and `write()` operations. The channel `fifo` has the local variabels to store the available characters, the positions of the next character to read and to write.

The `read()` operation first checks the number of available characters. If the fifo is empty, the operation suspends execution with a call to `wait(write_event)` (line 26) and the execution is resumed only when the `write_event` is notified. As soon as a character is read from the fifo, the event `read_event` will be notified (line 32).

The `write()` operation is implemented similarly. It checks that whether the fifo is full or not. If the fifo is full, the operation suspends execution with a call to `wait(read_event)` (line 16) and the execution is resumed only when the `read_event` is notified. As soon as a character is writtend to the fifo, the event `write_event` will be notified (line 21).

2.3.4 Binding and Simulation

We now bind the modules of the producer and consumer with the fifo channel via the output and input ports. One can write the binding code in a separated module or inside the `sc_main()` function that is the entry of the executable SystemC model. The following listing presents an example of binding and simulation of the producer-consumer model.

Listing 9 : File main.cpp

```
#include <time.h>
#include "fifo.cpp"
#include "consumer.h"
#include "producer.h"

int sc_main(int argc, char *argv[]) {
  srand(time(NULL));  // set the "seed" for randomization
  fifo afifo("fifo"); // create a channel fifo

  Producer prod("producer");
  Consumer cons("consumer");

  prod.out(afifo); // the producer binding
  cons.in(afifo);  // the consumer binding

  sc_start (10, SC_MS);  // run the simulation for 10 miliseconds

  cout << endl << "Fished␣at␣time␣" << sc_time_stamp() << endl;

  return 0;
}
```

If we compile and run the file `main.cpp`, some of the possible outputs of the model are given as follows:

Listing 10 : Simulation Outputs

```
(x, 0) (x, 1)         (x, 3)                        (x, 8) (x, 9)
                             (x, 4)
(x, 0)        (x, 2)  (x, 3) (x, 4)                        (x, 9)
```

(x, i) means that the consumer reads a character "x" at the $(i + 1)th$ milisecond.

3 Statistical Model Checking

This section introduces the concept of SMC [20, 29, 17] which is an formal analysis technique for stochastic systems. We assume that $\mathcal{M}$ is a model of a stochastic system and φ is a bounded property, meaning that φ can be defined on a set of finite executions of $\mathcal{M}$. The statistical probabilistic model checking problem consists of deciding: (1) *Qualitative*: is the probability that $\mathcal{M}$ satisfies φ greater or equal to a threshold θ with a specific level of statistical confidence? and (2) *Quantitative*: what is the probability that $\mathcal{M}$ satisfies φ with a specific level of statistical confidence?. The former is denoted by $\mathcal{M} \models Pr_{\geq\theta}(\varphi)$.

The key idea of SMC is that each simulation of the system is associated with a random variable B_i, its outcome , denoted b_i, is 1 if the simulation satisfies φ and 0 otherwise. Then statistical methods are used in order to decide with the defined level of confidence whether the system satisfies the property or not. Many statistical methods are implemented including sequential hypothesis testing, Monte Carlo simulation, or 2-sided Chernoff bound in a set of existing tools [21, 27, 2] that have shown the advantages over well-known tools such at PRISM on several case studies.

Although SMC can only provide approximate results with a user-specified level of statistical confidence (as opposed to the exact results provided by standard probabilistic model checking method), it is compensated for by its better scability, resource consumption. And the fact that the models to be analyzed can often be known only approximately, thus an approximate result in analyzing desired properties within specific bounds is quite acceptable. Unlike probabilistic model checking method, SMC is recently used in various research domains such as verification of industrial softwares, system biology [14], and the medical area [9].

4 Related Work

Several verification techniques have proposed for SystemC, for example, SystemC Verification Working Group proposed the SystemC verification standard (SCV) [11] that provides APIs to verify functionality of programs. However, SCV does not mentions about temporal specifications. One can use SCV as a complementatry of our verification framework in terms of automatically generating inputs for the SUV.

One of the earliest attempt at checking temporal specification properties in a SystemC model is carried out by Braun et al. [3]. The temporal properties are expressed as Finite Linear Temporal Logic (FLTL) and the temporal resolution is defined by the boundary of the clock cycles. FLTL is linear temporal logic that interprets formulas over finite traces and supports temporal operators w.r.t clock cycles. Then they proposed two approaches to extend test-bench features to SystemC for checking temporal properties: the checking is implemented directly within the SystemC language as an add-on library as SystemC itself. The other approach is to interface SystemC with an existing external testbench environment, TestBuilder [5]. As described in [25, 24], the temporal resolution defined at boundary of the clock cycles is inadequate for SystemC verifcation. Since it does not expose fully the semantics of the SystemC simulator kernel, which gives much finer grained temporal resolution. For example, the simulation may consist of a single delta-cycle if it is driven by immediate event notifications, meaning that the simulation clock doest not advance.

There has been a lot of work on the formalization of SystemC. That means a formal model is extracted from SystemC program, so that tools like model-checkers can be applied. However, all these formalizations consider semantics of SystemC and its simulator in some form of *global* model, and they also suffer from the state explosion when dealing with industrial and large systems.

Tabakov et al. [23, 24] proposed a framework for monitoring temporal SytemC properties. This framework allows users express the verifying properties richer by fully exposing the semantics of the SystemC simulator as well as the user-code. They extend LTL by providing some extra primitives for forming the atomic propositions and let users define much finer temporal resolution. In order to realize that they proposed a technique to modify the standard SystemC kernel and apply *aspect oriented programming* (AOP) to instrument SystemC programs automatically.

SMC is recently is applied in a wide range of research areas including software engineering (e.g., verification of critical embedded systems) [12, 21, 29], system biology [14], or medical area [9]. For instance, in [8], a framework of verifying properties of mixed-signal circuits, in which analog and digital quantities interact. The bounded linear temporal logic is used to express the properties, then the design of circuit and the verifying properties is connected to a statistical model checker. This approach consists of evaluating the properties on a representative subset of behaviors, generated by simulation, and answering the question of whether the circuit satisfies the properties with a probability greater than or equal to some value.

5 SystemC Execution Trace and Extended BLTL

In order to apply SMC for verifying the desired properties of the SUV, we need to define the concept of *execution trace* and the temporal logic to reason on execution traces. In our consideration, we use the definitions of SystemC state and temporal resolution in [25, 24].

5.1 Model and Execution Trace

Considered in its entirety, the execution of a SystemC model of timed and probabilistic system is an alternation of control between the SystemC simulator kernel, the processes of the model, and the external libraries. It can be considered as a stochastic discrete-time system. The system remains in the same state between the occurrence of two points of the temporal resolution during an execution.

Definition The execution of a SystemC model is a tuple $\mathcal{M} = (T, S, S_0, \rightarrow, AP, L_{AP})$, where:

- T is a temporal resolution which indicates when a state in the execution trace should be sampled;
- $S = V \times Loc \times Arg \times Proc \times Ker \times E$ is the set of states, where $V, Loc, Arg, Proc, Ker$ and E are valuations of the variables, the locations of code labels or specific statements, argument and return values of functions in the user-code, status of all processes of the user-code, the kernel phases, and the event notifications;
- S_0 is the set of initial states;
- $\rightarrow$: $S \times S$ is the transition relation of the system. We assume there exists a probability distribution on $\rightarrow$, meaning that $\forall s \in S, \Sigma_{s' \in S} Pr(s \rightarrow s') = 1$;
- AP is a set of atomic propositions;
- $L_{AP} : S \rightarrow 2^{AP}$ is a mapping which assigns to each state the set of atomic propositions that are `true` in that state.

A state of the execution contains the state of the kernel (the kernel phases, the event notifications), the state of user-code (evaluations of variables, specific locations in the source code, the argument and return values of functions in the source code, and the processes status). Here, we consider the external libraries as black boxes, meaning that their states are not exposed. An *execution trace* $\sigma = s_0, s_1, ..., s_{n-1}$ of length n is a finite sequence of n states of $\mathcal{M}$ such that $s_i \in S$ and $s_i \rightarrow s_{i+1}$, for each $0 \leq i \leq n-1$. The *i-th suffix* of σ is the sequence $s_i, ..., s_{n-1}$ and denoted by σ^i. And the i-th state of σ is denoted by $\sigma(i)$.

5.2 Extended BLTL

We describe our temporal logic that is used as specification language of the verification framework. This logic is an extended version of BLTL by enriching the set of atomic propositions and the temporal resolution. We first recall the syntax and semantics for BLTL, an extension of LTL with time bounds on temporal operators. BLTL is defined by the following grammar:

$$\varphi ::= \texttt{true} | \texttt{false} | p \in AP | \varphi_1 \wedge \varphi_2 | \neg\varphi | \varphi_1 \texttt{U}_\texttt{t} \varphi_2.$$

The temporal modalities `F` (the "eventually", sometimes in the future) and `G` (the "always", from now on forever) can be derived from the "until" `U` as $\texttt{F}_\texttt{t}\varphi = \texttt{true}\texttt{U}_\texttt{t}\varphi$ and $\texttt{G}_\texttt{t}\varphi = \neg\texttt{F}_\texttt{t}\neg\varphi$, respectively. The semantics of BLTL is defined w.r.t finite sequences of states of $\mathcal{M}$. We denote the fact that ω, a finite sequence of states, satisfies the BLTL formula φ by $\omega \models \varphi$.

- $\omega^k \models \texttt{true}$ and $\omega^k \not\models \texttt{false}$
- $\omega^k \models p, p \in AP$ if and only if $p \in L(\omega(k))$
- $\omega^k \models \varphi_1 \wedge \varphi_2$ if and only if $\omega^k \models \varphi_1$ and $\omega^k \models \varphi_2$
- $\omega^k \models \neg\varphi$ if and only if $\omega^k \not\models \varphi$
- $\omega^k \models \varphi_1 \mathtt{U_t} \varphi_2$ if and only if there exists natural i such that $\omega^{k+i} \models \varphi_2$, $\Sigma_{j<i}(t_j - t_{j-1}) \leq t$, and for each $0 \leq j < i, \omega^{k+j} \models \varphi_1$

We now present the new set of atomic propositions that is formed by proposed primitives in [25]. These primitives are declared in the configuration file of our framwork and let users define properties about the states of user-code, and SystemC kernel. We have the following:

SystemC_expr	::=	*model_expr*|*kernel_expr*
model_expr	::=	*loc_expr* | *arg_expr* | *proc_expr*
loc_expr	::=	[`before`|`after`]{*code_label* | *syntax_expr*} | *func_name* : {`entry`|`exit`|`call`|`return`}
arg_expr	::=	*func_name* : *nonnegative_integer*
proc_expr	::=	*proc_name*.`proc_state`
kernel_expr	::=	*phase_expr* | *event_expr*
phase_expr	::=	`kernel_phase`
event_expr	::=	*event_name*.`notified`

A *model_ expr* is a Boolean expression about the state of the user-code that can be the location counter (*loc_ expr*), the argument and return values of a function, or the status of a process (*proc_ expr*). Using the location counter, users can refer to a state of user-code before or after a specific statement or a code label is immediately executed. For example, assume that we want to specify the property *"always the value of the variable a is different from 0 whenever it is used as a divisor within t seconds"*. We first define the primitive `plocation loc1` "`/*a`" : `before` (the matching statements are regular expressions) that declares the Boolean variable `loc1` that holds the value `true` immediately before the execution of all statements that contain the devision operator "/" followed by zero of more spaces, and the variable "a". Then the property is expressed as follows:

$$G_t(loc1 \rightarrow (a! = 0))$$

Primitives `entry`, `exit`, `call`, and `return` refer to the location immediately before the first executable statement, the location immediately after the last execuatable statement in a function, the location that contains the function call, and the location immediately after the function call, respectively. The primitive `location send_start` "%`producer` :: `send`()" : `call` and `location send_start` "%`producer` :: `send`()" : `return` declare Boolean variables `send_start` and `send_done` that hold the value `true` immediately before and after a function call of the function `send`() of the module `producer`, respectively. Similarly, the primitive `location rcv` "%`consumer` :: `receive`()" : `return` declares a Boolean variable `rcv` that holds the value `true` immediately after a function call of the function `receive`() of the module `consumer`. After that, users can specify the property *"*`send`() *remains blocked until* `receive`() *has returned within t seconds"* as follows:

$$G_t(send\_start \rightarrow (!send\_done\ \mathtt{U_t}\ rcv))$$

Specification of post-condition of a function can be done by exposing the argument and return values of this function. The primitive `value float ret` "`float bar` :: `foo`(...)" : `0`, for example,

declares the float variable `ret` is assigned the return value of `foo`(). The primitive `value float vari` "`float bar` :: `foo`(...)" : `i` declares float variable `vari` such that it is equal to the i-th argument of `foo` whenever the function starts executing.

For each process name, the primitive *proc_ expr* indicates the status of this process in the simulator kernel that can be *waiting*, *runnable*, or *running*. The *kernel_ expr* consists of the primitives to expose the current state of the kernel (*phase_ expr*) (e.g., end of delta-cycle notification) and when a specific event is notified (*event_ expr*). For instance, the primitive declaration `eventclock wevent write_event.notified` declares a Boolean variable `wevent` that holds immediately when `write_event` is notified. Then this variable can be used to define a temporal resolution as `temporal_resolution wevent` in the configuration file of our framework.

Finally, users can define their own temporal resolutions used by the temoral operator in BLTL formulas. These temporal resolutions indicate when a state in the execution trace should be sampled and let users define the time bounds as a duration of simulation time or a number of transitions steps in a execution trace. For example, a Booleans variable *passby* represents the fact that the lifts pass by in a multi-lift system, then the BLTL formula $\mathtt{F}_{1000}(passby)$ can express property *"the lifts pass by within 1000 seconds"* if the time resolution is defined at the boundary of 1s clock simulation, or property *"the lifts pass by within 1000 times the event e is notified"* if the time resolution is defined at the boundary of the notification of the event e. In case the time bounds are defined by the number of state changes, the semantics of formula with until operator is defined as $\omega^k \models \varphi_1 \mathtt{U}_t \varphi_2$ if and only if there exists natural i such that $\omega^{k+i} \models \varphi_2$, $i \leq t$, and for each $0 \leq j < i, \omega^{k+j} \models \varphi_1$.

6 Implementation

Our implementation contains two components: a monitor and aspect-advice generator and a statistical model checker. The monitor uses the similar techniques described in [23] for observing a set of execution traces of the SUV corresponding to the verifying properties in our extended BLTL. The monitor has a `step`() function that is called at every sampling point defined by the temporal resolution during the simulation of the SystemC model. The `step`() function will record the values of all primitives in the formula as a state of the system. For each extended primitive, the monitor implements a corresponding callback function, which assignes correct values for this primitive. The monitor also implements corresponding callback function for the defined temporal resoution, which determines state change in the system and calls the `step`() function. The tool also generates an aspect-advice file that is used by AspectC^{++} [10] to instrument the SUV.

In order to apply statistical model checking, the instrumented SUV and the monitors are compiled into an executable specification. The checker is implemented in Java as a plugin of Plasma Lab [2] which establishes an interface between Plasma Lab and the executable specification. In the current version, the communication between the executable specification and the checker is done via standard input and output. Based on the time bounds and the structure of the verifying formula, the checker requests the monitor generating an execution trace with a specific length. The checker uses various statistical hypothesis testing algorithms including sequential hypothesis testing, Monte Carlo simulation, or 2-sided Chernoff bound.

Running our verification framework consists of two phases: i) User writes a configuration file containing all properties to be verified with the declarations of all using primitives, as well as other necessary information. The tool will generates the monitors and an aspect-advice file that is then used by AspectC^{++} to generate the instrumented SUV. Finally, the instrumented SUV code and the monitors are compiled together and linked with the SystemC kernel into an executable specification. This executable model is simulated to produce execution traces to be verified by checker in the second phase; ii) In the second phase, a plugin for Plasma Lab is used to verify the properties of the SystemC model. Given the executable specification in the output of the first phase, it is then run to execute the simulation with inputs provided by user. The inputs can be generated using any standard stimuli-generation technique. For every property, the checker of Plasma Lab verifies the execution traces producing by the corresponding monitor to check the validity of this property.

7 Experimental Evaluation

We now apply our implementation to the multi-lift system above. The SystemC model and the configuration are given as follows:

- *Temporal Resolution*: the clock cycle of the SystemC simulation is set to be 1 second and the temporal resolution is set at boundary of clock cycles. That means a state in the execution trace is sampled at rate 1 $\frac{sample}{s}$.
- *Model*: The model consists two modules: *liftsystem* which contains N lifts and M floors, and *user* which generates external and internal requests. The *liftsystem* object has multiple data structures, e.g., a vector of integer for user requests from external button panels, a two dimension vector of integer for requests from internal button panels, a vector of *lift_status_t* data structure for the status of the lifts. For an external request from a certain floor, the controller will assign this request to the nearest lift (the distance is computed by the number of floors). There are $2M-2+NM$ different requests with N lifts and M floors ($2M-2$ are external requests and NM are internal requests). We model the

Number of lifts,floors,users	Probability	Number of traces
5,30,2	0.2114	23026
5,30,4	0.2371	23026
10,30,4	0.2802	23026
10,30,5	0.3216	23026
10,30,6	0.3821	23026

Table 1: Results for F_{1000} *passby*

behavior of one user as follows: every second, a user generates a request. For simplicity, the probability that it is an external request is $\frac{2}{N} \times \frac{1}{2M-2}$, and the probability that it is an internal request is $\frac{N-2}{N} \times \frac{1}{NM}$. A lift is modeled as follows. If there is an external or internal request at the current floor, it serves and clears this request. Otherwise, it checks whether it should continue traveling in the same direction, or change direction, or simply idle. For each 2 seconds, a lift can travel 1 floor.

- *Initial State*: at the begining, every lifts are idle and at floor 0.
- *Transition Relation*: Given the state of the model and the SystemC kernel s_k at clock cycle t_k, when a user request is given at clock cycle t_{k+1}, the kernel computes the state of the model and the kernel at s_{k+1}. The probability distribution from $s_k \rightarrow s_{k+1}$ is included by the probability distribution of the input request at t_{k+1} and the semantics of the kernel.

We define a Boolean variable `passby` which is true iff the lifts pass by. We considered the verifying BLTL formula $\mathtt{F}_{1000}$ *passby*, i.e., what is the probability that the lifts pass by within 1000 seconds of simulation. We applied the 2-sided Chernoff bound with the accuracy $\epsilon = 0.01$ and the level of confidence $1 - \alpha = 0.98$. Table 1 shows the probability of the verifying property with different parameters of the model that are denoted by the numbers of lifts, floors, and users. We can observe that the probability increases when the number of users increases. This means that with more requests per second, it is likely that the lifts pass by. Similarly, the probability increases when the number of lifts increases. In the size of model, these results can be compared with those reported in [22], in which the state space explosion occurs when there are more than 3 lifts and more than 4 floors.

In our experiments with different time bounds of model with 10 lifts, 30 floors, and 5 users, which reported in Table 2. Intutively, when the model runs for longer time, meanings that more requests the model processed, the probability that the lifts pass by is increases as well.

Lastly, we did a few experiments on the performance of the nearest assignment and the random asignment algorithms that are reported in Table 3. The time bounds are 100, 200, and 500 in the rows number 1, 2, and 3, respectively. The nearest assignment always performs better than the random assigment in all cases as we expect.

Time bounds	Probability
100	0.0310
500	0.1987
1000	0.3216
2000	0.3724
4000	0.5710

Table 2: Results for F_t *passby* with different time bounds and algorithms

Number of lifts,floors,users	Probability (Nearest)	Probability (Random)
10,10,2	0.0321	0.8129
10,30,4	0.1206	0.8770
10,30,5	0.1987	0.9983

Table 3: Results for F_t *passby* with different algorithms

8 Conclusions

This paper presents the first attempt to verify non-trivial properties of SystemC model with statistical model checking techniques. In comparison the probabilistic technique [22], our technique allows us to handle large insdustrial systems modeling in SystemC as well as to expose a rich set of user-code primitives by automatically instrumenting the user-code with AspectC^{++}.

The framework contains two main components: a *monitor* that observes executions of the SUV based on the verifying properties, and a statistical model checker implementing a set of hypothesis testing algorithms. We use the similar techniques proposed by Tabakov et al. [24] to automatically generate the monitor corresponding to the user-defined verifying properties and the SUV. The statistical model checker is implemented as a plugin of the checker Plasma Lab [2]. Users express desired properties in (BLTL), our framework allows users to adapt BLTL for SystemC by adding the extended primitives to form atomic propositions, and user-defined temporal resolutions. The mechanism presented here does not require the users have background in aspect oriented programing or intrument the user-code manually.

In this work, we consider a external library is a black box, meaning that we do not consider the state of the external libraries. Thus, arguments passing to a function in the external library cannot be monitored. As future work, we would like to allow users to monitor the state of the external libraries with the future version of AspectC^{++}. We also plan to apply statistical model checking to verify temporal properties of SystemC-AMS (Analog/Mixed-Signal).

References

[1] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In *FSTTCS'06*, volume 4337, pages 260–272. LNCS, Springer, 2006.

[2] B. Boyer, K. Corre, A. Legay, and S. Sedwards. Plasma lab: A flexible, distributable statistical model checking library. In *QEST'13*, pages 160–164, 2013.

[3] A. Braun, J. Gerlach, and W. Rosentiel. Checking temporal properties in systemc specifications. In *HLDVT'02*, pages 23–27. IEEE International, 2002.

[4] D. Bustan, S. Rubin, and M. Vardi. Verifying omega-regular properties of markov chains. In *CAV'04*, volume 3114, pages 189–201. LNCS, Springer, 2004.

[5] Cadence. Design systems, inc.: Testbuilder user guide. In *Product Version 1.0*, 2001.

[6] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd. Surviving the soc revolution: A guide to platform-based design. In *Kluwer Academic Publishers, Norwell, USA*, 1999.

[7] F. Ciesinski and M. Grober. On probabilistic computation tree logic. In *Validation of Stochastic Systems*, volume 2925, pages 147–188. LNCS, Springer, 2004.

[8] E. Clarke, A. Donze, and A. Legay. Statistical model checking of mixed-analog circuits with an application to a third order delta-signam modulator. In *HVC'08*, 2008.

[9] Cmacs. http://cmacs.cs.cmu.edu/.

[10] A. Gal, W. Schroder-Preikschat, and O. Spinczyk. Aspectc++: Language proposal and prototype implementation. In *OOPSLA'01*, 2001.

[11] S. V. W. Group. SystemC verification standard specification [version 1.0e], 2003.

[12] H. Hermanns, B. Watcher, and L. Zhang. Probabilistic cegar. In *CAV'08*, volume 5123, pages 162–175. LNCS, Springer, 2008.

[13] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. In *TACAS'06*, volume 3920, pages 441–444. LNCS, Springer, 2006.

[14] S. Jha, E. Clarke, C. Langmead, A. Legay, A. Platzer, and P. Zuliani. A bayesian approach to model checking biological systems. In *CMSB'09*, volume 5688, pages 218–234. LNCS, Springer, 2009.

[15] J. Katoen, E. Hahn, H. Hermanns, D. Jansen, and I. Zapreev. The ins and outs of the probabilistic model checker mrmc. In *QEST'09*. IEEE CS Press, 2009.

[16] K. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[17] A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: An overview. In *RV'10*, volume 6418, pages 122–135. LNCS, Springer, 2010.

[18] R. Lipsett, C. Schaefer, and C. Ussery. *VHDL: Hardware description and design*. Kluwer Academic Publishers, 1993.

[19] J. Rutten, M. Kwiatkowska, G. Norman, and D. Parker. Mathematical techniques for analyzing concurrent and probabilistic systems. In *CRM Monograph Series*, volume 23. American Mathematical Society, Providence, 2004.

[20] K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In *CAV'05*, volume 3576, pages 266–280. LNCS, Springer, 2004.

[21] K. Sen, M. Viswanathan, and G. Agha. Vesta: A statistical model-checker and analyzer for probabilistic systems. In *QUEST'05*, pages 251–252. IEEE Computer Society, 2005.

[22] J. Sun, Y. Liu, S. Song, J. Dong, and X. Li. Prts: An approach for model checking probabilistic real-time hierarchical systems. In *ICFEM'11*, volume 6991, pages 147–162. LNCS, Springer, 2011.

[23] D. Tabakov and M. Vardi. Monitoring temporal systemc properties. In *Formal Methods and Models for Codesign*, pages 123–132. IEEE, 2010.

[24] D. Tabakov and M. Vardi. Automatic aspectization of systemc. In *MISS'12*. ACM, 2012.

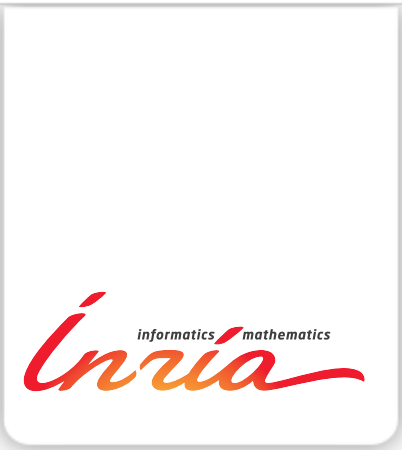
[25] D. Tabakov, M. Vardi, G. Kamhi, and E. Singerman. A temporal language for systemc. In *FMCAD'08*, pages 1–9. IEEE Press, 2008.

[26] M. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *FOCS'85*, pages 327–338, 1985.

[27] H. Younes. Ymer: A statistical model checker. In *CAV'05*, volume 3576, pages 429–433, 2005.

[28] H. Younes, M. Kwiatkowska, G. Norman, and D. Parker. Numerical vs statistical probabilistic model checking. In *STTT'06*, volume 8(3), pages 216–228, 2006.

[29] H. Younes and R. Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. In *COMPUT'06*, volume 204(9), pages 1368–1409, 2006.



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399