A Synchronous Approach to Threaded Program Verification

Kenneth Johnson, Loïc Besnard, Thierry Gautier, Jean-Pierre Talpin

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Synchronous Approach to Threaded Program Verification

Kenneth Johnson — Loïc Besnard — Thierry Gautier — Jean-Pierre Talpin

N° 7320

June 2010

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--7320--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

A Synchronous Approach to Threaded Program Verification

Kenneth Johnson , Loïc Besnard , Thierry Gautier , Jean-Pierre Talpin

Thème COM — Systèmes communicants
Équipes-Projets Espresso

Rapport de recherche n 7320 — June 2010 — 31 pages

Abstract: Modern systems involve a complex organization of computational processes sharing access to both processors and resources. The use of threads in programming provides a method in which lightweight processes may be given specific tasks that can be carried out either independently or in cooperation with other threads. The correct and efficient use of shared resources between threads relies on synchronisation methods, such as semaphores, mutexes, or *events*. Our work demonstrates a semi-automated method of translating threaded software to the synchronous programming language SIGNAL in order to verify the correctness of thread synchronisations in the source code.

This work is part of the FoToVP Project supported by Agence Nationale de la Recherche.

Key-words: Signal, polychrony, threads, model generation, model checking

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

A Synchronous Approach to Threaded Program Verification

Résumé : Les systémes actuels s'appuient sur une organisation complexe de processus de calcul partageant l'accés aux processeurs et aux ressources. La programmation au moyen de "threads" fournit une méthode dans laquelle des processus légers se voient attribuer des tâches spécifiques qui peuvent être menées soit indépendamment, soit en coopération avec d'autres threads. L'utilisation correcte et efficace de ressources partagées entre les threads repose sur des mécanismes de synchronisation tels que les sémaphores, les sections critiques ou les événements. Notre travail décrit une méthode semi-automatique de traduction de logiciels organisés en threads vers le langage synchrone Signal dans le but de vérifier la correction des synchronisations des programmes sources.

Ce travail s'inscrit dans le cadre du projet FoToVP (projet ANR).

Mots-clés : Signal, polychrony, threads, model generation, model checking

Contents

1 Introduction

Difficulties in providing fast and intuitive applications of formal methods to the software engineering community have resulted in the development of several tools for the automated generation of models from engineering languages such as C or Java. The aim of these tools is to allow the programmer to include formal methods as a regular task in day to day software development.

This may include formal approaches to software testing as in [18] where a preliminary automaton that accepts known system behaviour is generalised by querying the user with additional test-case scenarios. Other approaches involve a direct translation of programming code to a formal language for analysis. Java PathFinder [10] is one such tool that verifies multi-threaded java code via an automatic translation to the modelling language Promela. Using the SPIN model checker, assertions may be proved and deadlocks between threads detected.

Our work follows in this approach and introduces an "almost automatic" method of formal verification of threaded C programs. We show how programs written in C may be automatically translated into the synchronous language SIGNAL via an intermediate Static Single Assignment (SSA) representation. SSA is language independent and many compilers such as the GCC compiler perform this translation. The translation to a synchronous formalism allows us to reason about program control-flow in a formal clock calculus.

Programs containing threads execute concurrently and often interact via synchronisation commands. Extending our translation method to threads requires a hand-written specification of the scheduling policies of the operating system. As the scheduling of native threads in the operating system is non-deterministic and complex, we base our work on the deterministic and cooperative scheduling policies of the FairThreads framework. In this case, the scheduler organises execution of threads using a deterministic round-robin approach, and threads cooperate with each other using basic communication and synchronization primitives.

1.1 Outline

The content of the paper is as follows. In Section 2, we give introduce the SIGNAL programming language and review the basic scheme of translating a C program to its SSA representation.

We introduce the FairThreads framework in Section 3 and present a summary of the synchronisation commands. We illustrate their use in the cooperative scheduling of the framework and a formal specification of thread behaviour is given. We give an example which highlights the problem of thread deadlocks due to misplaced synchronisation commands. Section 4 outlines our translation method from imperative programming language to the synchronous language SIGNAL, illustrated with an example. A formal SIGNAL specification of the FairThreads scheduler is given in Section 5 and we outline its construction in detail. We show how threads communicate with the scheduler via control signals and outline the important components of the scheduler itself. Section 6 shows how our method can be used to prevent deadlocks and inefficient code by the detection of poor event synchronisations between threads. We conclude in Section 7 and give directions for further work.

2 Preliminaries

2.1 The Signal Language

SIGNAL is a multi-clocked data-flow specification language for the high-level specification of real-time systems in which computations over streams of data called *signals* are specified by systems of equations.

A signal x is a possibly infinite flow of values from simple data types such as booleans, integers or events[1] and are sampled at a discrete clock denoted $\hat{x}$. The clock of a signal is the set of *tags* representing symbolic periods in time in which a data value is present on the signal. If two signals x and y have the same clock, we call them *synchronous*. In symbols, $\hat{x} = \hat{y}$.

[1] The data type consisting of the single value `true`, representing the presence or absence of a signal.

A SIGNAL process

$$z := P(x_1, \ldots, x_n)$$

consists of the composition of simultaneous equations which equate the output signal z as a function of the input signals $x_1, \ldots, x_n$. The equations express logically consistent constraints on both the clocks and the data transmitted by the signals. Processes are constructed from *(i)* a single equation $z := f(x_1, \ldots, x_n)$, *(ii)* a synchronous composition of processes $P \mid Q$, *(iii)* or the restriction P `where` x of a signal x to the lexical scope of process P.

Signal Equations

There are five different types of equations defining primitive processes used to specify computations over signals. We list each equation used in the SIGNAL language along with its mathematical meaning and the implicit relationships between the clocks of the input and output signals. A complete account of the mathematical framework of SIGNAL is found in [14].

Equations on Data. Let f denote an n-ary function or relationship over numerical or boolean valued data. For input signals `x1`,...,`xn` the SIGNAL equation `z := f(x1,...,xn)` specifies mathematically a process equating pointwise for each tag t $z(t) = f(x_1(t), \ldots, x_n(t))$. The relationship between the clocks of the input and output signals is $\hat{z} = \hat{x_1} = \cdots = \hat{x_n}$.

Delay. For input signal `x` and constant value `a`, the equation `z := x 1 init a` specifies mathematically a process whose output is defined by $z(t_i) = a$ if t_i is the first tag t_0, and for every other tag we set $z(t_i) = x(t_{i-1})$. The relationship between the clocks of the input and output signals is $\hat{z} = \hat{x}$.

Merge. For input signals `x` and `y` the equation `z := x default y` specifies mathematically a process whose output at t is $z(t) = x(t)$ when $t \in \hat{x}$ and $z(t) = y(t)$ if $t \notin \hat{x} \wedge t \in \hat{y}$. The relationship between the clocks of the input and output signals is $\hat{z} = \hat{x} \cup \hat{y}$.

Sampling. For the input signal `x` and a signal `b` carrying boolean type values, the equation `z := x when b` specifies mathematically a process whose output $z(t)$ has the value $x(t)$ when defined and when the value $b(t)$ is defined and carries the value *true*. The relationship between the clocks of the input and output signals is $\hat{z} = \hat{x} \cap [b]$ where $[b] = \{t \in \hat{b} \mid b(t) =$ `true` $\}$. The unary form of the sampling operation given by the syntax `z := when b` specifies an event typed signal `z` which is present at instant t whenever $b(t)$ is present and true.

Equations on Clocks. In the primitive equations we have given so far, the clocks of signals are defined implicitly by the operations on the signals. The SIGNAL language allows clock relationships and contraints to be defined *explicitly* using a special clock operator. For a signal `x` we define its clock `^x` to be `^x` $\stackrel{def}{=}$ `(x=x)`, where the boolean proposition `(x=x)` is true whenever `x` is present and absent otherwise. We write `^0` for the null clock (the clock that is never present).

Combining the clock operator with the primitive equations we express clock relationships in the SIGNAL language: *(i)* The synchronisation relation $x \hat{=} y$ between the clocks of signal `x` and `y` corresponds to the process `(z := (^x = ^y)) where z`. *(ii)* clock union relationship `z := x ^+ y` corresponds to the process `z := ^x default ^y`, *(iii)* clock intersection relationship `z := x ^* y` corresponds to the process `z := ^x when ^y`.

Primitive operations on signals are composed to derived more complex operations such as the cell operation `x := y cell z init v`, that stores in `x` the most recent value carried by the signal `y` when `y` is present or when `z` is true. The cell operation is defined by the process

```
x := (|y default (x$1 init v) | x ^= y ^+ when z|).
```

2.2 C programs in SSA Representation

For control-flow analysis it is useful to represent a program by a directed graph where nodes are labeled blocks containing a sequence of statements and program control-flow between blocks is represented by an edge. Statements may be operations `x = f(y*)` or tests `if x goto L` and each block is terminated by either a `return` or `goto L` statement.

A program is said to be in *static single assignment form* whenever each variable in the program appears only once on the left hand side of an assignment. Following [7], a program is converted to SSA form by replacing assignments of a program variable x with assignments to new versions $x_1, x_2, \ldots$ of x, uniquely indexing each assignment. Each use of the original variable x in a program block is replaced by the indexed variable x_i when the block is reachable by the i^{th} assignment. For variables in blocks reachable by more than one program block, a special ϕ operator is used to choose the new variable value depending on the program control-flow. For example, $x_3 = \phi(x_1, x_2)$ means "x_3 takes the value x_1 when the flow comes from the block where x_1 is defined, and x_2 otherwise". This is needed to represent C programs where a variable can be assigned in both branches of a conditional statement or in the body of a loop. We display the grammar rules for C programs in SSA form in Figure 1

```
<program>  ::=  L:<block>;<program> | L:<block>
  <block>  ::=  <stm>;<block> | <term>
    <stm>  ::=  x = f(y*) | x = phi(y*)| if x goto L
   <term>  ::=  goto L | return
```

Figure 1: Grammar Rules for C programs in SSA form

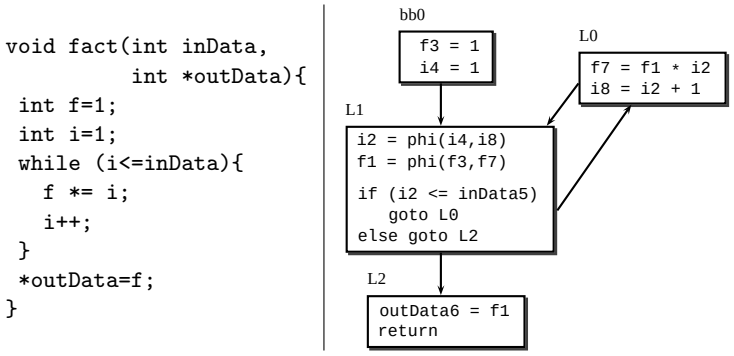

Figure 2: From C to SSA form

2.3 Example: the `fact` program

The left side of Figure 2 depicts a C program `fact` which takes an integer value input `inDate` and outputs its factorial. On the right is its SSA form represented as a control-flow diagram consisting of four blocks labelled `bb0`, `L0`, `L1` and `L2`.

The block `bb0` is the entry point of the program which initialises the variables `f3` and `i4` then passes control to block `L1`. The `phi` operator sets the value of the variables `i2` and `f1` depending on the source of control flow, either from block `bb0` or `L0`. If the terminal condition `i2 <= inData5` is satisfied control goes to `L0` where `f7` is updated with the new factorial value and the index counter is incremented. Once the index counter is equal to the `inData5` the loop terminates and control goes to block `L2` where the output is set to the factorial value in `f1` and returned.

3 The FairThreads Framework

FairThreads [3, 5] is a framework for concurrent and parallel programming of software systems mixing both cooperative and preemptive threads. In a purely cooperative context, *schedulers* are defined to which threads may dynamically link and unlink. Threads attached to a scheduler *cooperate*

with each other by willingly yielding their control of the processor to another thread. They can synchronise and communicate data with other threads using events. The scheduling of cooperative threads follows a simple round-robin approach, and with all threads linked to a single scheduler, the system runs in a deterministic fashion with a simple well defined semantics [4].

The execution sequence of a FairThreads scheduler is decomposed into a series of execution *instants*[2] during which each thread linked to the scheduler has an equal opportunity to

1. run until its next cooperation point, and
2. respond to events generated by threads linked to the scheduler.

Cooperation points come in two flavours: explicit, when a thread calls a synchronisation primitive or implicit, when a thread is waiting for an event to be received. In FairThreads, events are used for thread synchronisation and to communicate data between threads and can be associated with one or more data values. Events generated by a thread are broadcast to all other threads linked to the scheduler. In doing this, each thread witnesses the presence and absence of events in exactly the same way and all threads waiting for an event have the possibility to react to it during the same instant. At the end of each instant, events generated during the instant are *reset* or cleared and threads who missed an event may react to its absence in the next instant. A summary of the FairThreads synchronisation primitives is given in Figure 3.

`ft_thread_await(e)`	Wait for event `e`
`ft_thread_await_n(e,k)`	Wait until event `e` is received or issue a timeout after k instants
`ft_thread_generate_value(e,v)`	Generate event `e` with value `v`
`ft_thread_get_value(e,k,v)`	Get the k^{th} value associated with event `e` and store in `v`
`ft_thread_cooperate_n(k)`	Yield control from the calling thread to the scheduler for k instants
`ft_thread_join_n(t,k)`	Suspend execution of the calling thread until thread `t` has terminated or issue a timeout after k instants

Figure 3: Fairthreads Commands

3.1 Cooperative Thread Scheduling in FairThreads

We present two examples to illustrate the scheduling of cooperative threads in FairThreads.

3.1.1 Example: Good Synchronisation

Figure 4 depicts threads A and B each represented by a sequence of blocks containing program code and dashed arrows represents control-flow between threads. The control-flow dictated by the scheduler is deterministic, and arrows labelled start and finish show the beginning and end points of the execution sequence.

Starting with thread A, the code in A1 is executed sequentially until blocked by the synchronisation command `await(e)`. This command blocks the thread until event e is received, and so the scheduler selects the next thread to be run. Thread B starts by executing the code in B1 and generates event e by the command `generate(e)`, and continues to execute code in B2. Using the `cooperate` command, control is returned to thread A which receives event e. Unblocked, thread A executes A2 and terminates, leaving thread B to execute the code in block B3 and then terminate. Both threads have finished running and terminate normally.

[2]Note that the word instant is used in both the SIGNAL language and the FairThreads framework but with entirely different meanings

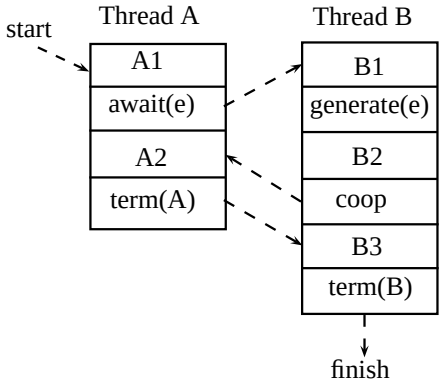

Figure 4: Scheduling Threads A and B

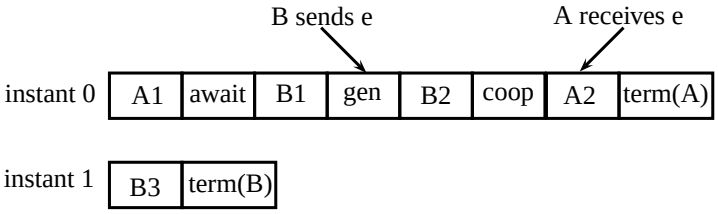

Figure 5: Instant decomposition of Threads A and B

Figure 5 displays the series of instants produced by the scheduler executing threads A and B. In instant 0, thread B sends event e and eventually cooperates, leaving thread A to receive the event, finish running and terminate in the same instant. Since all threads have had an opportunity to run and respond to all sent events, the instant ends. In instant 1 thread B runs until it terminates.

3.1.2 Example: Bad Synchronisation

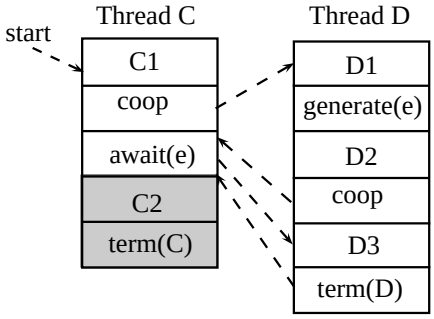

Figure 6: Scheduling Threads C and D

Another scheduling example between threads C and D is presented in Figure 6. Execution begins with thread C running the code in C1 and then cooperating, allowing thread D to start and execute D1. Continuing the sequential execution of D, an event e is generated by the command `generate(e)` and then D2 is executed. Executing the `cooperate` command, control is returned to thread C where it awaits event e, blocked by command `await(e)`. Control immediately returns to thread D which executes D3 and then terminates. The scheduler returns control to thread C

but execution is blocked, waiting for an event that will never appear. The shaded blocks C2 and term(C) are never executed.

To see why these threads failed to synchronise properly we examine the instant decomposition in Figure 7. In instant 0 thread C begins running and then cooperates. Thread D begins running and generates an event e and eventually cooperates. As both threads C and D have already run to their cooperation point, the instant is finished and event e is cleared to prepare for a new instant. In instant 1 thread C awaits an event e and is thus blocked, leaving thread D to run and terminate. With thread C blocked and no other threads left for the scheduler to run, the program is deadlocked.

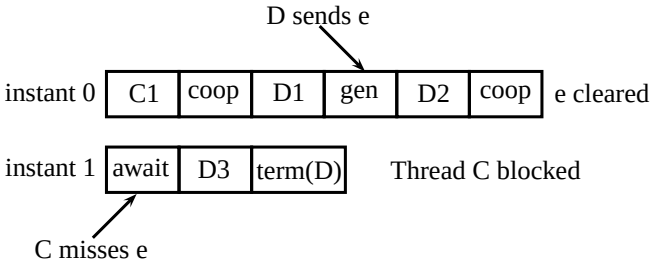

Figure 7: Instant decomposition of Threads C and D

3.2 Modelling FairThreads Scheduling Behaviour

The behaviour of each thread linked to the scheduler is formally described by the state machine depicted in Figure 8. Each state of the machine corresponds to a possible state the thread may be in during the course of execution and each transition models the effect of the FairThreads operations on the thread. It is the responsibility of the scheduler to observe the state of each of the threads and determine which thread is to be executed next, according to the scheduling policy. For our discussion we ignore the state codes displayed in the figure, returning to them later in Section 5.

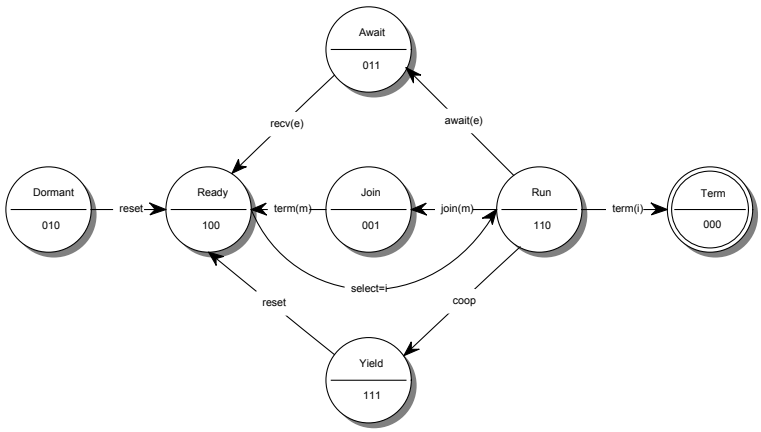

Figure 8: State Machine of Thread i

Before the scheduler starts, we assume that all threads are attached and are in the *Dormant* state. When the scheduler starts, a *reset* event is issued signaling the beginning of a new execution instant and each of the threads are placed in the *Ready* state. Following the round-robin scheduling policy, the i^{th} thread is selected and moves into the *Run* state. This thread now has control of the processor and may run until it terminates *term(i)*. If an *await(e)* command is executed, the thread is blocked and moves into the *Await* state until the event e is present. Similarly when the command

join(m) is executed the thread is placed in the *Join* state until thread m terminates. When the thread executes the command *coop* it moves into the *Yield* state until the scheduler determines that the end of the execution instant has occurred and a *reset* event is issued, returning all yielded threads to the *Ready* state.

3.3 A FairThreads Example

We present an example of a FairThreads program listed in Figure 9 where a scheduler is defined and a new event is created in the scheduler. Connected to the scheduler are two threads `1` and `2` whose

```
void main(){
   ft_scheduler_t sched = ft_scheduler_create();
   ft_event_t e = ft_event_create(sched);
   ft_thread_create(sched,thread1,NULL,NULL);
   ft_thread_create(sched,thread2,NULL,NULL);
   ft_scheduler_start(sched);
}
```

Figure 9: FairThreads Example

function bodies are listed in Figure 10. Thread `1` initializes variables and waits for events to occur on the scheduler by executing the `ft_thread_await(e)` command. When an event is present the thread is unblocked and the value associated with the event is retrieved and the factorial function `fact` is computed, after which the thread terminates. If no event should occur, the thread is blocked indefinitely.

Thread `2` initializes and receives an integer value from some input device via the `getValue` function. A new event with the value associated with it is generated and the thread cooperates and afterwards terminates.

```
  void thread1(){
   int v,f;
 // ft_thread_cooperate_n(1);
   ft_thread_await(e);
   ft_thread_get_value(e,0,&v);
   fact(v,&f);
   finalizethread1();
  }
```

```
void thread2(){
 int num;
 initthread2();
 num = getValue();
 ft_thread_generate_value(e,(void*)&num);
 ft_thread_cooperate_n(1);
 finalizethread2();
}
```

Figure 10: Threads connected to FairThreads Scheduler

The sequence of executions performed by the FairThreads scheduler for threads `1` and `2` corresponds to the example illustrated in Section 3.1.1 with the instant decomposition identical to that in Figure 5. The addition of an `ft_thread_coop_n(1)` command to thread `1` results in the event generated by thread `2` to be missed, causing a deadlock. This execution sequence is identical to the situation of the second example in Section 3.1.2.

A study of this example gives the fundamental reason for the occurrence of synchronisation problems in cooperative threads: events are generated, but misplaced synchronisation commands make it impossible for any thread to receive and react to them. By defining a SIGNAL specification modelling the behaviour of the scheduler and threads, we show how it is possible to give a formal method to automatically detect poorly placed synchronisation commands in threaded programs.

4 From Imperative Programs to a Synchronous Formalism

Translating an imperative program to a synchronous paradigm involves decomposing potentially *unbounded* computations produced by while-loop constructions into a sequence of *bounded and*

finite computations to be performed in a series of logical instants. The main point to be made about this method is that it allows us to reason about program control-flow and computation using a well-founded formal calculus on clocks based on a synchronous model of computation.

We outline our approach to translating imperative programming code into the synchronous formalism SIGNAL using the `fact` program as an example. For a full account of the translation process and implementation details of this method, the interested reader may consult [11] and [2].

The SIGNAL process `proFact` listed in Figure 11 models the sequence of computations performed by the `fact` program.

The input of the process is the integer valued signal `inData` and the output signal `outData` carries the factorial value once the computation has finished. In lines 22, 25-27 the control-flow in the SSA diagram is modelled by defining block labels as boolean typed signals which are present and carry a true value whenever control is in the corresponding block. Block `bb0` is the entry point of the program and is defined to be true for the beginning of the computation and then false afterwards. When control-flow is possible from two different blocks in the diagram the `phi` statement is used, and this is naturally modelled by the merging of two signals using the SIGNAL command `default`.

The computations performed in each block are specified in lines 9-10, 12-13, 18 and 21. Each block statement is sampled over the block's corresponding boolean signal using the SIGNAL command `when` and is thus performed only when control-flow is at that block.

The while-loop computation is performed in a series of logical instants, each consisting of a single iteration. For each instant the statements in block `L0` are computed and the loop condition is checked. The result of the condition is carried by the signal `next_L0`. This value determines whether or not another iteration is required in the *next* instant. Accordingly, the boolean signal `L0` modelling the body of the loop is defined as the previous value of `next_L0`.

5 Modelling FairThreads in Signal

In this section we describe our approach of using SIGNAL processes to model and simulate the operation of a software system executing threaded programs.

The model we develop simulates the FairThreads cooperative scheduling policy and is composed of a collection of processes representing key components of the system: the operating system organizing the scheduling of the thread execution, mechanisms for communication between components, and of course the threads themselves. The components of the software system and their interconnections with two threads are depicted in Figure 12. For simplicity we illustrate our method with two threads.

Our approach is based on the generation of a SIGNAL model from a threaded program that maintains information on the state of each thread and event. For this reason we require that they be statically created in the program and our translation scheme assigns each a numerical value.

In Figure 12, the boxes labelled Thread1 and Thread2 depict the SIGNAL processes generated from threads statically created and attached to the scheduler. For example, we may use threads `1` and `2` in the FairThreads example of Section 3.3. These processes are automatically generated by extending the method described in Section 4: an SSA representation of the original threaded code is generated, and then it is translated into the synchronous formalism of a SIGNAL process. The key idea of this translation is that the SSA blocks are modelled by boolean signals, which are present and carry a true value whenever control is in the corresponding block.

5.1 Thread Control Signals

This idea is extended to processes modelling threads in a simple way. In the SSA representation of the threads, synchronisation primitives (commands from the FairThreads library) are considered as external function calls. This means that they are isolated from the regular programming constructs of the language and placed into separate labelled blocks in the SSA representation.

Now, synchronisation primitives are special commands that are used for cooperation and communication with other threads. They often change the state of the thread executing the command

```
1 process proFact = (?integer inData; !integer outData;)
2  (| (| pK__1 := inData_5 ^+ i_2
3      | pK__2 := Z_i_2 ^+ Z_f_1
4      |)
5   | (| Z_f_1 := f_1$1
6      | Z_i_2 := i_2$1
7      |)
8   | inData_5 ^= L0 ^= f_1 ^= i_2 ^= bb_0
9   | (| f_3 := 1 when bb_0
10     | i_4 := 1 when bb_0
11     |)
12  | (| f_7 := ((Z_f_1 cell pK__2)*(Z_i_2 cell pK__2)) when L0
13     | i_8 := (Z_i_2+1) when L0
14     |)
15  | (| i_2 := i_8 default (i_4 default Z_i_2)
16     | f_1 := f_7 default (f_3 default Z_f_1)
17     |)
18  | outData_6 := f_1 when L2
19  | when bb_0 ^= inData
20  | (| inData_5 := inData cell (^bb_0) |)
21  | (| outData := (outData_6 cell L2) when L2 |)
22  | (| bb_0 := (not (^bb_0))$1 init true
23     | next_L0 := (((i_2 cell pK__1)<=(inData_5 cell pK__1)) when L1)
24                     default false
25     | L0 := next_L0$1 init false
26     | L1 := (true when L0) default (true when bb_0)
27     | L2 := (not ((i_2 cell pK__1)<=(inData_5 cell pK__1)))
28    when L1
29     |)
30  |)
31 where ... end
```

Figure 11: Listing of SIGNAL process `proFact`

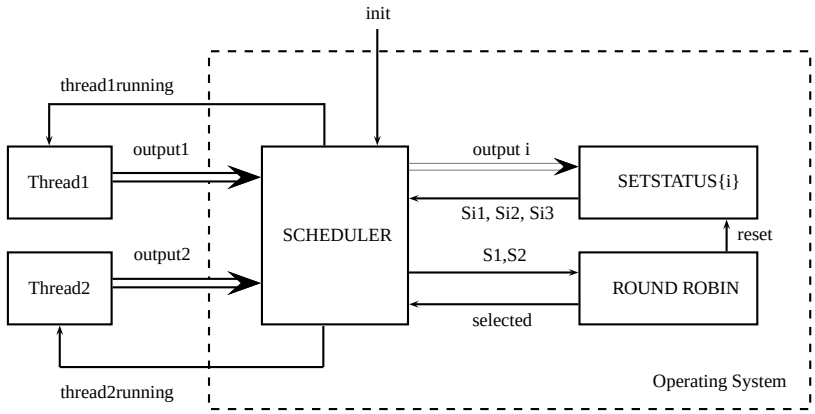

Figure 12: Components of a Threaded Software System

(cf. Figure 8). In order to notify the operating system of a state change, threads must communicate with the operating system.

The isolation of synchronisation primitives provides an easy method of modelling these communication requirements: the signals modelling the blocks containing FairThreads commands are defined as *output signals* of the thread process. An output signal is present whenever the command it is modelling is currently being executed in the thread, and carries the parameter value supplied to

the command. Additional output signals are required for commands with more than one parameter. If a command does not appear in a thread then the clock of its output signal is defined to be the null clock (the thread never executes that command).

The output signals for thread `1` and `2` are depicted in Figure 12 as double arrows labelled output1 and output2. We list all ten output signals in Figure 13 and their corresponding FairThreads commands and parameters.

For example, the command `ft_thread_await_n(e,k)` requires two control signals: `integer await` which carries the number of the event e, and `integer awaitTimeout` carries the length of time k to wait for the event before continuing execution. The output signal `endProcessing` is not associated with a command but rather the last block in the SSA diagram. This signal is present whenever the thread executes the last instruction and thus informs the operating system that the thread has terminated.

FairThreads Command	**Output Signal**
`ft_thread_await_n(e,k)`	`integer await, awaitTimeout`
`ft_thread_generate_value(e,v)`	`integer genEvent, genEventValue`
`ft_thread_get_value(e,k,v)`	`integer getValueFromEvent, getithValueOfEvent`
`ft_thread_cooperate_n(k)`	`integer cooperate`
`ft_thread_join_n(t,k)`	`integer join, joinTimeout`
	`event endProcessing`

Figure 13: Output Signals for Thread Communication

In addition to output signals used to announce the thread status, an input signal is required connecting the operating system to the thread to notify it that it has been selected to start or resume its execution. The input signals of threads `1` and `2` are depicted in the diagram by arrows labelled thread1running and thread2running.

These special input and output signals are called *control signals* and they provide an interface between the operating system and the thread.

5.2 The Operating System

The role of the operating system model is to observe control signals, maintain and update the state of each thread and events generated by threads, and deterministically select and notify threads to execute, as required by the specifications of the FairThreads scheduling policy. The operating system model is composed of many components, and we highlight the most essential ones and their interconnections, depicted inside the dashed box in Figure 12.

The box labelled `SCHEDULER` represents a SIGNAL process which takes as input the control signals for each of the threads and selects and notifies the next thread to execute via the output signals `thread1running` and `thread2running`. This notification is dependent on the state of each thread and which thread, if any, is ready to run.

5.2.1 The State of Threads and Events

Now, the state of a thread is often determined by synchronisation commands. For example, a thread that executes the command `await(e)` is in the *Await* state and is not able to run: another thread must be selected by the scheduler. Our model is required to maintain and update each thread state according to the signals present on the control signals. The states in the finite state machine depicted in Figure 8 are represented by a unique three-valued boolean code, and thus a thread i is described by a tuple

```
boolean Si1,Si2,Si3.
```

For each separate thread i, it is the task of the process `SETSTATUS{i}` to observe the thread's control signals depicted by the double arrow labelled `outputi` on the figure, determine the values carried by the state signals in the *next* logical instant, and output the resulting state, depicted by the arrow labelled `Si1, Si2, Si3` in the figure. This process essentially models the behaviour of the state machine in Figure 8 by enumerating every possible state transition and setting the next state variables accordingly. For example, suppose thread i is currently in the *Run* state when a signal is present on the `await` control signal. According to the state diagram the next state is *Await*, which is specified in SIGNAL by the equations listed in Figure 14. Note also that the equations specify that each thread is initialized to the *Dormant* state.

```
|      Si1   := NEXT_Si1  $ 1 init false
|      Si2   := NEXT_Si2  $ 1 init true
|      Si3   := NEXT_Si3  $ 1 init false
| NEXT_Si1   := ... default false when await ...
| NEXT_Si2   := ... default  true when await ...
| NEXT_Si3   := ... default  true when await ...
```

Figure 14: Listing from SIGNAL process `SETSTATUS`

Control signals also supply the scheduler with parameter values which must be maintained and updated accordingly by the scheduler. The execution of the command `Await(e)` results in the control signal `await` carrying the number associated with event `e`. As this value is necessary to describe the execution state of the thread the scheduler maintains the value in the signal

```
integer Sia.
```

The other synchronous command parameters are handled similarly, with each having a special state signal associated with it.

By using *state signals* to carry the values of the current thread states, and the parameters of the synchronous primitives, a complete description of the execution state for all threads in the program is given. We denote all state signals for a thread i by `Si`.

The scheduler must maintain the state of events generated by any of the executing threads. Thus, each event created via the command `generate(e,v)` is specified in the scheduler by the event signals

```
boolean pe; integer ve
```

whereby the signal `pe` is present and carries the value `true` when event represented by the number `e` is present, and `ve` carries the integer value `v` associated with the event.

The operating system components we describe require synchronous state signals such that we may perform basic operations and comparisons on the values they carry. The state signals for threads and events are synchronous with the master clock of the software system given by the input signal `init`. Intuitively speaking, each state signal must be present to specify the thread and event state throughout the entire execution sequence.

5.2.2 Deterministic Thread Selection

The box labelled `ROUNDROBIN` in Figure 12 depicts the process responsible for modelling the FairThreads cooperative round-robin selection outlined in Section 3. This process observes the current state of each thread, the number of the thread currently executing, and transmits the number of the selected thread to the scheduler whereby the thread is notified via its input control signal. All state signals `S1,S2` maintained by the scheduler are provided as input signals and the number of the selected thread is output to the scheduler via the signal labelled `selected`.

To model correct scheduling behaviour, the `ROUNDROBIN` process specifies the duration of each FairThreads instant. Recall that once each thread has had an opportunity to run and yields control back to the scheduler or has otherwise been blocked, the instant is complete and a new instant begins. This means all events generated during the instant are reset and threads in the *Yield* state are returned to *Ready*.

This behaviour is modelled simply by the SIGNAL equation in Figure 15, where the `reset` signal is present whenever all threads are in a blocked or terminated state. Since a signal presence on `reset` causes a thread state change (cf. Figure 8) it is an input signal to the process `SETSTATUS`.

```
| reset := when initialize default
           when ((yield1 or term1 or await1 or join1) and
                 (yield2 or term2 or await2 or join2))
```

Figure 15: Specification of the reset signal for two threads

We provide a complete listing of the FairThreads scheduler specification in SIGNAL for the good and bad synchronisation examples in Section 3.3 in Appendix A.

6 Threaded Program Verification in Signal

In this paper we have outlined a semi-automatic translation of imperative threaded programs into a synchronous formalism that allows us to reason about control-flow sequences of threads in terms of a formal clock calculus. Using the SIGNAL compiler included in the Polychrony toolset, we are able to check *static* properties such as contrary clock constraints, cycles, null clocks, exclusive clocks.

To check *dynamic* properties, we use the model checker SIGALI [17] which is an interactive tool specialized on algebraic reasoning in $\mathbf{Z}/3\mathbf{Z} = \{0, 1, -1\}$ logic. The compiler can automatically generate sets of dynamic polynomial equations from SIGNAL specifications that defines an automaton describing the dynamical behaviour of the specification. The software tool can analyze this automaton and prove properties such as liveness, reachability, and deadlock. It provides an analysis of the logical and synchronisation properties of boolean signals, where the values carried by the signals are encoded by the three values: 1 for present and true, -1 for present and false, and 0 for absent. This is practical in the sense that true numerical verification quickly results in state spaces that are no longer manageable, however it requires, depending on the nature of the underlying model, major or minor modifications prior to formal verification. For many properties, numerical values are not needed at all and can be abstracted away thus speeding up verification. When verification of numerical manipulations is sought, an abstraction to boolean values can be performed, that is sufficient in most cases.

For example, using SIGALI with only slight modifications to the FairThreads scheduling model, we can prove important dynamical properties of thread execution. The boolean signal

```
thread1running := S11 and S12 and (not S13)
```

is defined over the state signals of thread 1 and carries a true value whenever thread 1 is currently in the *Run* state. The signal `thread2running` is defined likewise. We add to the model a testing signal `exclusiveRunning` shown in Figure 16 and using SIGALI prove that there is no state in the automaton such that the property is true. Thus we conclude that our model does not schedule more than one thread to run at a time.

```
| exclusiveRunning := thread1Running and thread2Running
| Sigali(Never(B_True(exclusiveThreadRun)))
```

Figure 16: Property One. Exclusive Thread Execution

Program	Size Never(RECV)	State vars	States	Reachable	Transitions	Time
Ex. 3.1.1	false	34	2^{34}	84	548.931	1.18s
Ex. 3.1.2	true	30	2^{30}	19	286.755	2.92s

Table 1: Verification Time and Complexity

We use SIGALI to detect missed events in threaded programs which result in missynchronisations of threads or at the very least ineffective code. Encoding the state signals `S1a` and `S2a` as boolean signals we define the property

```
RECV := (S1a or S2a) and pe1
```

which is true when either thread 1 or thread 2 are waiting for event 1 and the event is present. By defining a special signal `boolean observer` which records the history of values of this property, we use SIGALI in Figure 17 to prove `RECV` is sometimes true for Example 3.1.1 and never true for Example 3.1.2. Table 1 summarises these results.

```
|observer := RECV default observer$1 init false
|observer ^= initialize
|Sigali(Never(B_True(observer)))
```

Figure 17: Property Two. Detecting Missed Events

7 Conclusion and Future Work

In this paper we have presented a deterministic threaded framework called FairThreads in which threads are attached to a scheduler and cooperative with each other for access to resources and the processor. To illustrate the scheduling policy of FairThreads two examples are presented. The first example exhibits correct synchronisation between threads using an event to send and receive data. The second example is a variation of this, but with a synchronisation error: a FairThreads primitive `ft_thread_cooperate` is misplaced causing the event never to be received. These types of synchronisation problems cause deadlocks and are difficult to detect in complex programs with several threads.

Our solution to detecting these types of synchronisation problems semi-automatically is based on the synchronous data-flow language SIGNAL. Imperative programs written in software engineering languages such as the C programming language are automatically converted into an SSA representation by the GCC compiler. We outline a method of translating the SSA code to SIGNAL where each SSA block is assigned a boolean signal, enabling us to reason about control-flow in a formal calculus on signals and clocks. This translation is automatic and we describe how the notion of control signals extends our method to programs which use synchronisation commands in the FairThreads library.

The FairThreads scheduler is modelled in SIGNAL and we highlight the most important components. The model was written by hand and is specific to the FairThreads scheduling policy. The SIGNAL compiler is used for code generation which simulates the scheduling of the two examples we have presented. Our model is general and is easily extended to an arbitrary number of threads and events. As we need to maintain the states of all threads and events, our method is limited to the static creation of threads and generation of events.

The SIGNAL compiler automatically generates a polynomial dynamical system which defines an automaton simulating the behaviour of the FairThreads specification. Using the SIGALI software tool we prove dynamical properties of our specification. Formal properties defined over boolean signals modelling thread states are easily verified by the tool, and we give the exclusive thread execution property as an example. More complex properties are formulated over the thread state

signals and boolean signals modelling generated events allowing us to verify the threads are free from deadlocks arising from missed events.

Though the work presented here was focused specifically on the scheduling policies of FairThreads, we envisage a generalisation of the scheduler model to simulate other scheduling policies. For example, we may consider modelling and verification tasks in the context of other cooperative threaded frameworks such as SHIM [8].

A Signal Specification Listing

We present the complete SIGNAL specification of the FairThreads scheduler including the threads from in Examples 3.1.1 and 3.1.2. This specification is modified for the purpose of SIGALI verification.

```
process FairThreadsRapportBinaryExamples1and2 = ( ? ! integer TASKID;)
(| initialize := INITIALIZING()
| TASKID := APPLICATION(initialize)
|)
where
boolean initialize;
type ft_event_t:=integer;
type timeout_type:=integer;
%-------------------------------------------------------------------------%
process exiting = (? event e;);
%-------------------------------------------------------------------------%
%  FairThreads primitives %
process ft_thread_generate_value = (? ft_event_t e; integer n; !)
spec (|e ^= n|);
process ft_thread_cooperate_n    = (? integer x; !);
action  ft_thread_cooperate      = (?  !);
process ft_thread_await_n        = (? ft_event_t e; integer n;!integer r)
spec (|e ^= n ^= r|);
process ft_thread_await          = (? ft_event_t e;!) ;
process ft_thread_get_value = (? ft_event_t e; integer n; ! integer value)
  spec (|e ^= n ^= value|);
process ft_thread_join           = (? integer n);
%  Printings %
process printStates = (? boolean S11, S12, S13, S21, S22, S23,S31,S32,S33; !) ;
process printBool                = (? strings; boolean b;)
spec (| strings ^= b |);
process printInteger             = (? strings; integer b;)
spec (| strings ^= b |);
process printEvent                = (? strings; boolean b1,b2;)
spec (| strings ^= b1 ^= b2 |);

%  --------------- State managing ------------%
process IN_RUNNING = ( ? boolean S1, S2, S3; ! boolean B)
(| B := S1 and S2 and (not S3) |);
process IN_TERM    = ( ? boolean S1, S2, S3; ! boolean B)
(| B:= (not S1 and not S2 and not S3) |);
process IN_READY   = ( ? boolean S1, S2, S3; ! boolean B)
(| B:= (S1 and (not S2) and not S3) |);
process IN_YIELD   = ( ? boolean S1, S2, S3; ! boolean B)
(| B:= (S1 and S2 and S3) |);
process IN_DORMANT = ( ? boolean S1, S2, S3; ! boolean B)
(| B:= (not S1) and S2 and (not S3) |);
process IN_JOIN    = ( ? boolean S1, S2, S3; ! boolean B)
(| B:= (not S1) and (not S2) and S3 |);
```

```
process IN_AWAIT    = ( ? boolean S1, S2, S3; ! boolean B)
(| B:= (not S1) and S2 and S3 |);
%  --------------- State managing ------------%
%
We modify the roundrobin process as follows.
selected is encoded by two booleans B1 and B2 and the integer
values are encoded as follows.
B1 B2
0   0  0
1   0  1
2   1  0
3   1  1
Next, we define some processes to make the specification easier to read.%
%-------------------------------------------------------------------------%
process IS_EQUAL = (?boolean B1,B2,C1,C2!boolean answer;)
(|answer := (B1=C1) and (B2=C2)
|);
%-------------------------------------------------------------------------%
process SET_ONE = (?!boolean B1,B2;)
(|B1 := false
|B2 := true
|);
%-------------------------------------------------------------------------%
process ONE = (?boolean B1,B2 !boolean answer)
(|answer := (not B1) and B2
|);
%-------------------------------------------------------------------------%
process SET_TWO = (?!boolean B1,B2;)
(|B1 := true
|B2 := false
|);
%-------------------------------------------------------------------------%
process TWO = (?boolean B1,B2 !boolean answer)
(|answer := B1 and (not B2)
|);
%-------------------------------------------------------------------------%
process SET_THREE = (?!boolean B1,B2;)
(|B1 := true
|B2 := true
|);
%-------------------------------------------------------------------------%
process THREE = (?boolean B1,B2 !boolean answer)
(|answer := B1 and B2
|);
%-------------------------------------------------------------------------%
process JOIN = {integer numTask1, numTask2;}
(? integer join;  %  thread number%
 integer joinn; %  timeout %
 event e1, e2;  %  terminate %
! event joinresume;
 integer sw;%thread number that we are waiting for. not waiting sw=-1%
 integer jc;%once this value has reached 0, issue joinresumei%
 )
 %  if join timed out or   the thread terminated, issue a resume event.%
(| (| sw := n_sw $ 1 init (-1)
  | n_sw := join  default (-1 when joinresume) default sw
  |)
|  (| jc := n_jc $1 init (8888888)
```

```
   | n_jc := joinn default ((jc-1) when (jc>=0)) default jc
   |)
| joinresume := when ((sw=numTask1) when e1) default
 (when (sw=numTask2) when e2) default when (jc=0)
|)
where
integer n_jc ,n_sw;
end;
%-------------------------------------------------------------------------%
process Qe = {boolean index_evt1,index_evt2; string s1,s2;}
  (? boolean reset;
 boolean GE11,GE12;integer GV1;
 boolean GE21,GE22;integer GV2;
 boolean GE31,GE32;integer GV3;
   ! boolean pe1;integer ve1;)
(| eventarrived := when (IS_EQUAL(GE11,GE12,index_evt1,index_evt2)) default
 when (IS_EQUAL(GE21,GE22,index_evt1,index_evt2)) default
 when (IS_EQUAL(GE31,GE32,index_evt1,index_evt2))
| (| zpe1 := pe1$1 init false
  | pe1 := true when eventarrived default
  false when reset        default
   zpe1
  |)
| (| zve1 := ve1$1 init 9999999
  | ve1 := ((GV1 default GV2 default GV3) when eventarrived)
default (9999999 when not pe1) default zve1
  |)
| printBool(s1,pe1)
| printInteger(s2,ve1)
|)
where
event eventarrived;
boolean zpe1;
integer zve1;
end;
%-------------------------------------------------------------------------%
process AWAIT1EVENT = (? boolean await1,await2;%number of event we are
                        waiting for %
 integer awaitn;%number of ticks we should wait before resuming%
 event awaitresume;
 ! boolean Sa1,Sa2; %event we are waiting for. otherwise Sa1=F,Sa2=F%
 integer ac; )  % when value has reached 0, issue awaitresumei%
(| (| zSa1 := Sa1 $ 1 init false
  | zSa2 := Sa2 $ 1 init false
  | Sa1 := await1  default false when awaitresume default zSa1
  | Sa2 := await2  default false when awaitresume default zSa2
  |)
| (| ac := zac $1 init (-1)
  | zac :=  awaitn default (ac-1) when ( (ac > -1)) default (-1)
  |)


| printBool("awaitresume = ",awaitresume)
| printEvent("event: ",await1,await2)
|)
where
integer zac;
boolean zSa1,zSa2;
```

```
end;

%-------------------------------------------------------------------------%
process SET_STATUS={ boolean INIT_STATUS1, INIT_STATUS2, INIT_STATUS3;}
(? boolean initialize;
 event reset, is_selected;
boolean await1;
 integer awaitn;
 event coop;
 integer join,joinn;
 event terminate,joinresume,awaitresume;boolean Continue;
! boolean Si1, Si2,Si3;
 event start;)
%           Si1  Si2  Si3
 TERM       0    0    0
 DORMANT    0    1    0
 READY      1    0    0
 RUN        1    1    0
 JOINE      0    0    1
 AWAITE     0    1    1
 YIELD      1    1    1
%
(| Si1   := NEXT_Si1  $ 1 init INIT_STATUS1
| Si2   := NEXT_Si2  $ 1 init INIT_STATUS2
| Si3   := NEXT_Si3  $ 1 init INIT_STATUS3


| DORMANT := IN_DORMANT(Si1, Si2, Si3)
| RUNNING := IN_RUNNING (Si1, Si2, Si3)
| TERM    := IN_TERM (Si1, Si2, Si3)
| JOIN    := IN_JOIN(Si1,Si2,Si3)
| AWAIT   := IN_AWAIT(Si1,Si2,Si3)

| NEXT_Si1 := ((true when DORMANT)
 default (true when RUNNING when Continue)
 default (true when reset  when (not (AWAIT OR JOIN)))
 default (false when ^await1)
 default (false when ^awaitn)
 default (true when is_selected)
 default (false when terminate)
 default (true when coop)
 default (true when joinresume)
 default (false when ^join)
 default (false when ^joinn)
 default (true when awaitresume)) when not TERM
 default Si1

| NEXT_Si2 := ((false when DORMANT)
 default (true when RUNNING when Continue)
 default (false when reset when (not (AWAIT OR JOIN)))
 default (true when ^await1)
 default (true when ^awaitn)
 default (true when is_selected)
 default (false when terminate)
 default (true when coop)
 default (false when joinresume)
 default (false when ^join)
 default (false when ^joinn)
```

```
 default (false when awaitresume)) when not TERM
 default Si2

| NEXT_Si3 :=( (false when DORMANT)
default (false when RUNNING when Continue)
default (false when reset when (not (AWAIT OR JOIN)))
default (true when ^await1)
default (true when ^awaitn)
default (false when is_selected)
default (false when terminate)
default (true when coop)
default (false when joinresume)
default (true when ^join)
default (true when ^joinn)
default (false when awaitresume))  when not TERM
default Si3
|)
where
boolean NEXT_Si1,NEXT_Si2,NEXT_Si3,DORMANT,RUNNING,JOIN,AWAIT,TERM;
end;
%-------------------------------------------------------------------------%
process ROUNDROBIN = (? boolean initialize;
boolean S11,S12,S13,S21,S22,S23,S31,S32,S33;
boolean Continue1, Continue2, Continue3;
  ! boolean B1,B2;
event reset;)

(| ready1 := IN_READY(S11, S12, S13)
| ready2 := IN_READY(S21, S22, S23)
| ready3 := IN_READY(S31, S32, S33)

| (| term1 := IN_TERM (S11, S12, S13)
  | term2 := IN_TERM (S21, S22, S23)
  | term3 := IN_TERM (S31, S32, S33)
  | term12 := term1 and term2
  | term13 := term1 and term3
  | term23 := term2 and term3
  |)
| (| yield1 := IN_YIELD (S11, S12, S13)
  | yield2 := IN_YIELD (S21, S22, S23)
  | yield3 := IN_YIELD (S31, S32, S33)
  |)
| (| no_running1 := not IN_RUNNING (S11, S12, S13)
  | no_running2 := not IN_RUNNING (S21, S22, S23)
  | no_running3 := not IN_RUNNING (S31, S32, S33)
  |)
|
(| await1 := IN_AWAIT (S11, S12, S13)
 | await2 := IN_AWAIT (S21, S22, S23)
 | await3 := IN_AWAIT (S31, S32, S33)
 |)
|
(| join1 := IN_JOIN (S11, S12, S13)
 | join2 := IN_JOIN  (S21, S22, S23)
 | join3 := IN_JOIN  (S31, S32, S33)
 |)

| (| resetEvents(when reset)
```

```
|reset := when initialize default when
          ((yield1 or term1 or await1 or join1) and
 (yield2 or term2 or await2 or join2) and
 (yield3 or term3 or await3 or join3))
  |)
| no_running := when ( no_running1 and no_running2 and no_running3 )
|B1 ^= B2
|B1 ^= when no_running

|zB1 := B1 $1 init true %initialised to THREE%
|zB2 := B2 $1 init true
| B2 := ( (true when reset)
   default  zB2 when ( Continue1 or Continue2 or Continue3)
 default (true when ready1 when ((zB2 =true) or ((zB2=false) and
(term3 when ^B2)) or (term23  when ^B2) ))
 default (false when ready2 when ((zB2 =true) or ((zB2=true) and
(term1 when ^B2)) or (term13  when ^B2) ))
 default (true when ready3 when ((zB2 =false) or ((zB2=true) and
(term2 when ^B2)) or (term12  when ^B2) ))
 default false ) when no_running

| B1 := ( (true when reset)
  default  zB1 when ( Continue1 or Continue2 or Continue3)
 default (false when ready1 when ((zB1 =true) or ((zB1=true) and
 (term3 when ^B1)) or (term23  when ^B1) ))
 default (true when ready2 when ((zB1 =false) or ((zB1=true) and
 (term1 when ^B1)) or (term13  when ^B1) ))
 default (true when ready3 when ((zB1 =true) or ((zB1=false) and
 (term2 when ^B1)) or (term12  when ^B1) ))
 default false ) when no_running
|printBool("Continue1 = ",Continue1)
|printBool("Continue2 = ",Continue2)
|printBool("Continue3 = ",Continue3)
|)
where
boolean zB1,zB2;
event resetStatus1,resetStatus2,resetStatus3;
event no_running;
boolean ready1, ready2, ready3;
boolean yield1,yield2,yield3;
boolean join1, join2, join3;
boolean await1, await2, await3;
boolean no_running1, no_running2, no_running3;
boolean term1,term2,term3,term12, term13, term23;
process resetEvents = (? event b!);
process eventPresent = (? integer event_index ! boolean value)
 spec (|event_index ^= value|);
end;
%-------------------------------------------------------------------------%
process UNDEFINED_THREAD = (? ! boolean await1; timeout_type awaitn;
  integer coop;
  integer join; timeout_type joinn;
  boolean GE1,GE2; integer genevalue;
  ft_event_t getValueFromEvent; integer getIthvalueOfEvent;
  event terminate;)
(| await1 := false when (^0)
| awaitn :=  0 when (^0)
| coop := 0 when (^0)
```

```
| join := 0 when (^0)
| joinn := 0 when (^0)
| GE1 := false when (^0)
| GE2 := false when (^0)
| genevalue := 0 when  (^0)
| getValueFromEvent:= 0 when  (^0)
| getIthvalueOfEvent:= 0 when  (^0)
| terminate := ^0
|);
%-------------------------------------------------------------------------%
process SCHEDULER_1_EVENT={boolean T1, T2, T3}
%  Ti at true when the i-th thread exists %
( ? boolean initialize;boolean await11;integer awaitn1;event coop1;
integer join1,joinn1;
boolean GE11,GE12;integer genevalue1;event terminate1;
boolean await21;integer awaitn2;event coop2;integer join2,joinn2;
boolean GE21,GE22;integer genevalue2;event terminate2;
boolean await31;integer awaitn3;event coop3;integer join3,joinn3;
boolean GE31,GE32;integer genevalue3;event terminate3;
!  boolean THREAD1RUNNING,THREAD2RUNNING,THREAD3RUNNING;)
(| (pe1,ve1) := Qe {false,true, "pe1 = ", "ve1 = "}
   (reset,GE11,GE12,genevalue1,GE21,GE22,genevalue2,GE31,
   GE32,genevalue3)
| initialize ^= pe1 ^= ve1
%============================== JOIN%
| (|(joinresume1,S1w,jc1) :=
JOIN{2,3}(join1,joinn1,terminate2,terminate3)
|(joinresume2,S2w,jc2) :=
JOIN{1,3}(join2, joinn2, terminate1, terminate3)
|(joinresume3,S3w,jc3) :=
JOIN{1,2}(join3, joinn3, terminate1, terminate2)
|)
%============================== SCHEDULING%
| (| (S11, S12, S13, start1) :=
SET_STATUS{false, T1, false}(initialize,reset,when
(ONE(B1,B2)),await11,awaitn1,coop1,join1,joinn1,
terminate1,joinresume1,awaitresume11,Continue1)
| (S21, S22, S23, start2) := SET_STATUS{false, T2, false}
(initialize,reset,when (TWO(B1,B2)),await21,awaitn2,coop2,
join2,joinn2,terminate2,joinresume2,awaitresume21,Continue2)
| (S31, S32, S33, start3) := SET_STATUS{false, T3, false}
(initialize,reset,when (THREE(B1,B2)),await31,awaitn3,coop3,
join3,joinn3,terminate3,joinresume3,awaitresume31,Continue3)
|)
| printStates(S11,S12,S13,S21,S22,S23,S31,S32,S33)
|Continue1 :=  true when (await11 when pe1) default false
|Continue2 :=  true when (await21 when pe1) default false
|Continue3 :=  true when (await31 when pe1) default false
|Continue1 ^=Continue2 ^= Continue3 ^= initialize
|(B1,B2, reset) := ROUNDROBIN(initialize,
S11,S12,S13,S21,S22,S23,S31,S32,S33,
 Continue1,  Continue2,  Continue3)
| THREAD1RUNNING := IN_RUNNING(S11, S12, S13)
| THREAD2RUNNING := IN_RUNNING(S21, S22, S23)
| THREAD3RUNNING := IN_RUNNING(S31, S32, S33)
|exclusiveThreadRun := (THREAD1RUNNING and THREAD2RUNNING) or
(THREAD1RUNNING and THREAD3RUNNING) or
(THREAD2RUNNING and THREAD3RUNNING)
```

```
|Sigali(Never(B_True(exclusiveThreadRun)))

|  jc1 ^= jc2 ^= jc3 ^= S1w ^= S2w ^= S3w ^= ^0
|  S11a ^= S12a ^= S21a ^= S22a ^= S31a ^= S32a ^= ac1 ^= ac2 ^= ac3
 ^= % jc1 ^= jc2 ^= jc3 ^= S1w ^= S2w ^= S3w ^=%
 S11 ^= S12 ^= S13 ^= S21 ^= S22 ^= S23 ^= S31 ^=
 S32 ^= S33 ^= initialize ^= pe1 ^= ve1
| exiting (when IN_TERM(S11, S12, S13) and IN_TERM(S21, S22, S23)
  and IN_TERM(S31, S32, S33))
|printBool ("S11a = ",S11a)
|printBool ("S21a = ",S21a)
|awaitresume11 :=   when (S11a) and pe1
|awaitresume21 :=   when (S21a) and pe1
|awaitresume31 :=   when (S31a) and pe1
|S11a := NEXT_S11a $ 1 init false
|NEXT_S11a := await11 default false when awaitresume11 default S11a
|S21a := NEXT_S21a $ 1 init false
|NEXT_S21a := await21 default false when awaitresume21 default S21a
|S31a := NEXT_S31a $ 1 init false
|NEXT_S31a := await31 default false when awaitresume31 default S31a
|obj1 := ((S11a or S21a or S31a) and pe1)  default obj1$1 init false
|obj1 ^= initialize
|Sigali(Never(B_True(obj1)))
|) where
boolean NEXT_S11a,NEXT_S12a,NEXT_S21a,NEXT_S22a,NEXT_S31a,NEXT_S32a;
boolean Continue1,Continue2,Continue3;
use SIGALI;
boolean exclusiveThreadRun,obj1 ;
event reset;
event awaitresume11,awaitresume21, awaitresume31;
event start1, start2,start3;
event joinresume1,joinresume2,joinresume3;
boolean S11, S12, S13, S21, S22, S23, S31, S32, S33;
boolean S11a,S12a,S21a,S22a,S31a,S32a;  % which event are we waiting for%
  integer ac1,ac2,ac3;  % countdown values for the awaitn command%
  integer S1w,S2w,S3w;     %thread we waiting for to terminate?%
  integer jc1,jc2,jc3;     %countdown values for the joinn command%

  boolean  pe1; %true if event1 present%
  integer ve1;      %value of event 1%
boolean B1,B2;
end;
%-------------------------------------------------------------------------%
process FT_SCHEDULER_2TASKS_1_EVENT =(? boolean initialize;
boolean await11; timeout_type awaitn1; integer coop1;
integer join1; timeout_type joinn1;
boolean GE11,GE12; integer genevalue1; ft_event_t getValueFromEvent1;
integer getIthvalueOfEvent1;event terminate1;
boolean await21; timeout_type awaitn2; integer coop2;
integer join2; timeout_type joinn2;
boolean GE21,GE22; integer genevalue2; ft_event_t getValueFromEvent2;
integer getIthvalueOfEvent2;event terminate2;
! boolean THREAD1RUNNING,THREAD2RUNNING;)
(| (THREAD1RUNNING,THREAD2RUNNING,THREAD3RUNNING) :=
SCHEDULER_1_EVENT {true, true, false} (initialize,
  await11,awaitn1, ^coop1,join1,joinn1,GE11,GE12,genevalue1,terminate1,
  await21,awaitn2, ^coop2,join2,joinn2,GE21,GE22,genevalue2,terminate2,
  await31,awaitn3, ^coop3,join3,joinn3,GE31,GE32,genevalue3,terminate3 )
```

```
| (await31,awaitn3,coop3,join3,joinn3,GE31,GE32,genevalue3,
getValueFromEvent3,getIthvalueOfEvent3, terminate3) := UNDEFINED_THREAD()
|) where
boolean THREAD3RUNNING;
boolean await31;
timeout_type awaitn3;
integer coop3;
integer join3;
timeout_type joinn3;
boolean GE31,GE32;
integer genevalue3;
ft_event_t getValueFromEvent3;
integer getIthvalueOfEvent3;
event terminate3;
end;
%-------------------------------------------------------------------------%
process INITIALIZING=
 (? ! boolean initialize;)
 (| initialize := (not ^initialize) $1 init true
  |);
%-------------------------------------------------------------------------%
process APPLICATION =( ? boolean initialize;! integer TASKID;)
(|(await11,awaitn1,coop1,join1,joinn1,GE11,GE12,genevalue1,
getValueFromEvent1, getIthvalueOfEvent1, endProcessing1) :=
EXAMPLE2_thread1(when THREAD1RUNNING)
|(await21,awaitn2,coop2,join2,joinn2,GE21,GE22,genevalue2,
getValueFromEvent2, getIthvalueOfEvent2, endProcessing2) :=
  EXAMPLE2_thread2(when THREAD2RUNNING)
| (THREAD1RUNNING,THREAD2RUNNING) := FT_SCHEDULER_2TASKS_1_EVENT{}
  (initialize,await11,awaitn1,coop1,join1,joinn1,GE11,GE12,genevalue1,
  getValueFromEvent1,getIthvalueOfEvent1, endProcessing1,
await21,awaitn2,coop2,join2,joinn2,GE21,GE22,genevalue2,
getValueFromEvent2,getIthvalueOfEvent2, endProcessing2)
| TASKID := 1 when THREAD1RUNNING default 2 when THREAD2RUNNING
|)
where
boolean THREAD1RUNNING,THREAD2RUNNING;
boolean await11,await21,await31;
integer awaitn1,awaitn2,awaitn3;
integer getValueFromEvent1, getIthvalueOfEvent1, getValueFromEvent2,
   getIthvalueOfEvent2, getValueFromEvent3, getIthvalueOfEvent3;
integer coop1,coop2,coop3;
integer join1,join2,join3;
integer joinn1,joinn2,joinn3;
boolean GE11,GE12,GE21,GE22,GE31,GE32;
integer genevalue1,genevalue2,genevalue3;
event endProcessing1, endProcessing2, endProcessing3;
%-------------------------------------------------------------------------%
process EXAMPLE1_thread2 % D% =
( ? event running;
  ! boolean _await1;
integer _await_timeout;
integer _cooperate;
integer _join_thread;
integer _join_timeout;
boolean GE1,GE2;
integer _gen_event_value;
ft_event_t _get_value_fromevent;
```

```
integer _get_ithvalue_ofevent;
event endProcessing;
)
  (| pK__2 ^= pK__1 ^= bb_0
   | (| (| pK__4 :: initthread2()
 | pK__4 ^= when bb_0
 |)
  | D_3218_1 := getValue() % when bb_0 ----%  | D_3218_1 ^= when bb_0
  | num_2 := D_3218_1 when bb_0
  |)
   | ft_thread_generate_value(1 when pK__3,num_2 when pK__3)
   | ft_thread_cooperate_n(1 when pK__2)
   | (| pK__5 :: finalizethread2()
  | pK__5 ^= when pK__1
  |)
   | (| bb_0 := ((not (^bb_0))$1 init true) when running
  | pK__3 := true when bb_0 when running
  | next_pK__2 := ((true when pK__3) default false) when running
  | pK__2 := (next_pK__2$1 init false) when running
  | next_pK__1 := ((true when pK__2) default false) when running
  | pK__1 := (next_pK__1$1 init false) when running
  |)
   | (| _await1 := ^0)
  | _await_timeout := ^0
  | _cooperate := 1 when pK__2
  | _join_thread := ^0
  | _join_timeout := ^0
%          | _gen_event := 1 when pK__3%
| GE1 := false when pK__3
| GE2 := true when pK__3
  | _gen_event_value := num_2  when pK__3
  | _get_value_fromevent := ^0
  | _get_ithvalue_ofevent := ^0
  | endProcessing := when pK__1
  |)
   |)
  where
  label pK__4,pK__5;
  boolean bb_0,pK__3,next_pK__2,pK__2,next_pK__1,pK__1;
  integer D_3218_1;
  integer num_2;
  end
%ThreadD%;
%-----------------------------------------------------------------------------%
process EXAMPLE1_thread1 % C% =
( ? event running;
  ! boolean _await1;
integer _await_timeout;
integer _cooperate;
integer _join_thread;
integer _join_timeout;
boolean GE1,GE2;
integer _gen_event_value;
ft_event_t _get_value_fromevent;
integer _get_ithvalue_ofevent;
event endProcessing;
)
  pragmas
```

```
 Local_Virtual_SSA_Code {value}
  end pragmas
  (| (| pK__9 := pK__6 ^+ Z_value |)
   | (| Z_value := value$1  | value ::= defaultvalue Z_value|)
   | pK__7 ^= pK__6 ^= value ^= bb_0
   | ft_thread_await(1 when bb_0)
   | value ::= ft_thread_get_value(1 when pK__7, 1)
| (| value_0_2 := (Z_value cell pK__9) when (pK__6 cell pK__9) when pK__6
  | D_3214_3 := compute_factorial(value_0_2 when pK__6)
  |printInteger("Value of Factorial is: ",D_3214_3)
  | f_4 := D_3214_3 when pK__6
  |)
   | (| bb_0 := ((not (^bb_0))$1 init true) when running
  | next_pK__7 := ((true when bb_0) default false) when running
  | pK__7 := (next_pK__7$1 init false) when running
  | next_pK__6 := ((true when pK__7) default false) when running
  | pK__6 := (next_pK__6$1 init false) when running
  |)
   | (|     _await1 := true when bb_0
 | _await_timeout := -1 when bb_0
  | _cooperate := ^0
  | _join_thread := ^0
  | _join_timeout := ^0

  | GE1 := false when (^0) | GE2 := false when (^0)
  | _gen_event_value := ^0
  | _get_value_fromevent := 1 when pK__7
  | _get_ithvalue_ofevent := 0 when pK__7
  | endProcessing := when pK__6
  |)
   |)
  where
  event pK__9;
  boolean bb_0,next_pK__7,pK__7,next_pK__6,pK__6;
  Z_value;
  integer value_0_2;
  integer D_3214_3;
  integer f_4;
  shared integer value;

 process compute_factorial =
  ( ? integer inData;
! integer outData;)
  spec (| inData ^= outData |) external;

  end
%ThreadC%;
%-------------------------------------------------------------------------%
process EXAMPLE2_thread2 =
( ? event running;
  ! boolean _await1;
integer _await_timeout;
integer _cooperate;
integer _join_thread;
integer _join_timeout;
boolean GE1,GE2;
integer _gen_event_value;
ft_event_t _get_value_fromevent;
```

```
integer _get_ithvalue_ofevent;
event endProcessing;
)
  (| pK__2 ^= pK__1 ^= bb_0
   | (| (| pK__4 :: initthread2()
 | pK__4 ^= when bb_0
 |)
  | D_3218_1 := getValue() % --when bb_0 % | D_3218_1 ^= when bb_0
  | num_2 := D_3218_1 when bb_0
  |)
   | ft_thread_generate_value(1 when pK__3,num_2 when pK__3)
   | ft_thread_cooperate_n(1 when pK__2)
   | (| pK__5 :: finalizethread2()
  | pK__5 ^= when pK__1
  |)
   | (| bb_0 := ((not (^bb_0))$1 init true) when running
  | pK__3 := true when bb_0 when running
  | next_pK__2 := ((true when pK__3) default false) when running
  | pK__2 := (next_pK__2$1 init false) when running
  | next_pK__1 := ((true when pK__2) default false) when running
  | pK__1 := (next_pK__1$1 init false) when running
  |)
   | (| _await1 := false when (^0)
  | _await_timeout := ^0
  | _cooperate := 1 when pK__2
  | _join_thread := ^0
  | _join_timeout := ^0
%          | _gen_event := 1 when pK__3%
|GE1 := false when pK__3
|GE2 := true when pK__3
  | _gen_event_value := num_2 when pK__3
  | _get_value_fromevent := ^0
  | _get_ithvalue_ofevent := ^0
  | endProcessing := when pK__1
  |)
   |)
  where
  label pK__4,pK__5;
  boolean bb_0,pK__3,next_pK__2,pK__2,next_pK__1,pK__1;
  integer D_3218_1;
  integer num_2;
  end
%ThreadD%;
%-----------------------------------------------------------------------------%
process EXAMPLE2_thread1 =
( ? event running;
  ! boolean _await1;
integer _await_timeout;
integer _cooperate;
integer _join_thread;
integer _join_timeout;
boolean GE1,GE2;
integer _gen_event_value;
ft_event_t _get_value_fromevent;
integer _get_ithvalue_ofevent;
event endProcessing;
)
pragmas
```

```
 Local_Virtual_SSA_Code {value}
end pragmas
(| (| pK__11 := pK__6 ^+ Z_value |)
| (| Z_value := value$1 | value ::= defaultvalue  Z_value |)
| pK__8 ^= pK__7 ^= pK__6 ^= value ^= bb_0
| (| pK__9 :: ft_thread_cooperate()
  | pK__9 ^= when bb_0
  |)
| ft_thread_await(1 when pK__8)
| value ::= ft_thread_get_value(1 when pK__7, 1)
| (| value_0_2 := (Z_value cell pK__11) when
  (pK__6 cell pK__11) when pK__6
  | D_3214_3 := compute_factorial(value_0_2) when pK__6
  | f_4 := D_3214_3 when pK__6
  |)
   | (| bb_0 := ((not (^bb_0))$1 init true) when running
  | next_pK__8 := ((true when bb_0) default false) when running
  | pK__8 := (next_pK__8$1 init false) when running
  | next_pK__7 := ((true when pK__8) default false) when running
  | pK__7 := (next_pK__7$1 init false) when running
  | next_pK__6 := ((true when pK__7) default false) when running
  | pK__6 := (next_pK__6$1 init false) when running
  |)
   | ( %_await := 1 when pK__8%
  | _await1 := true when pK__8
  | _await_timeout := -1 when pK__8
  | _cooperate := 1 when bb_0
  | _join_thread := ^0
  | _join_timeout := ^0
%          | _gen_event := ^0%
   | GE1 := false when (^0) | GE2 := false when (^0)
  | _gen_event_value := ^0
  | _get_value_fromevent := 1 when pK__7
  | _get_ithvalue_ofevent := 0 when pK__7
  | endProcessing := when pK__6
  |)
   |)
  where
  event pK__11;
  label pK__9;
  boolean bb_0,next_pK__8,pK__8,next_pK__7,pK__7,next_pK__6,pK__6;
  Z_value;
  integer value_0_2;
  integer D_3214_3;
  integer f_4;
  shared integer value; % -- %
  process compute_factorial =
  ( ? integer inData;
! integer outData;)
  spec (| inData ^= outData |) external;

  end
%ThreadC%;
end %RapportExample2%;
%-------------------------------------------------------------------------%
action initthread2              = (?!);
action finalizethread2          = (?!);
process getValue                 = (?!integer value);
```

```
process nextValueExist          = (?!integer value);

process extern_compute          =
  (? integer inData; !integer outData;)
spec (|inData ^= outData|);
process getEOI                  = (?!integer value);
process yield                   = (? integer numTask, val;)
spec (| numTask ^= val |);
process join                    = (? integer numTask, val;)
spec (| numTask ^= val |);
process SendEndprocessing       = (?integer x;);
end;
```

References

[1] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, Sep. 1991.

[2] L. Besnard, T. Gautier, M. Moy, J.P. Talpin, K. Johnson, and F. Maraninchi. Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form. In *Ninth International Workshop on Automated Verification of Critical Systems (AVOCS'09)*, Electronic Communications of the EASST, September 2009.

[3] F. Boussinot. FairThreads in C. `www-sop.inria.fr/mimosa/rp/FairThreads/FTC/index.html`.

[4] F. Boussinot. Operational semantics of cooperative fair threads. `www-sop.inria.fr/meije/rp/FairThreads/FTC/documentation/semantics.pdf`.

[5] F. Boussinot. FairThreads: mixing cooperative and preemptive threads in C: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(5):445–469, 2006.

[6] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[7] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[8] S.A. Edwards and O. Tardieu. SHIM: a deterministic model for heterogeneous embedded systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(8):854 –867, aug. 2006.

[9] INRIA Espresso Group. Polychrony/SME Public Web Site. http://www.irisa.fr/espresso/Polychrony.

[10] K. Havelund and T. Pressburger. Model checking java programs using JAVA pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[11] H. Kalla, J.-P. Talpin, D. Berner, and L. Besnard. Automated translation of C/C++ models into a synchronous formalism. In *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, pages 426–436, March 2006.

[12] M. Le Borgne, H. Marchand, É. Rutten, and M. Samaan. Formal verification of signal programs: Application to a power transformer station controller. *Algebraic Methodology and Software Technology*, pages 271 – 285, 1996.

[13] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, Sep. 1991.

[14] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers*, 12(3):261–304, April 2003.

[15] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic Systems*, 10:325 – 346, 2000.

[16] H. Marchand and M. Le Borgne. The supervisory control problem of discrete event systems using polynomial methods. Technical Report 1271, INRIA, 1999. http://hal.inria.fr/inria-00072869/en/.

[17] H. Marchand and E. Rutten. Signal and Sigali user's manual. http://www.irisa.fr/espresso/Polychrony, 2002.

[18] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Improving dynamic software analysis by applying grammar inference principles. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):269–290, 2008.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)

ISSN 0249-6399