Electronic Communications of the EASST
Volume X (2010)

Proceedings of the
10th International Workshop on
Automated Verification of Critical Systems
(AVoCS 2010)

A Synchronous Approach to Threaded Program Verification

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

16 pages

Guest Editors: Jens Bendisposto, Michael Leuschel, Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A Synchronous Approach to Threaded Program Verification

Kenneth Johnson[1]**, Loïc Besnard, Thierry Gautier, and Jean-Pierre Talpin**

INRIA Centre Rennes-Bretagne Atlantique/CNRS IRISA.
Campus de Beaulieu. 35042 Rennes Cedex, FRANCE
[1]kenneth.johnson@irisa.fr

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 specialised commands or events communicated between threads. Our work demonstrates a semi-automated method of translating cooperatively threaded software to the synchronous programming language SIGNAL in order to verify the correctness of thread synchronisations in the source code.

Keywords: Signal, polychrony, threads, model generation, model checking

1 Introduction

Modern software systems are complex and involve multi-threaded programs featuring thread communication and shared resources. Difficulties in detecting design errors such as deadlocks in concurrent systems using traditional software validation or simulation have resulted in the development of a wide variety of model-checking tools [HP00, HD01, AB01] which generate a formal model of the system in order to verify system properties.

We introduce a new approach to specifying and verifying concurrent systems based on the synchronous model of computation [BB91]. The synchronous formalism has been successfully applied to the mathematical modelling of critical real-time systems based on, for example, data-flow equations. It features a family of languages with well-developed toolsets for specification, model checking and code generation for simulation.

Our method automatically translates C programs 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.

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 use the deterministic and cooperative scheduling policies of the FairThreads [Boua, Bou06] framework. In the cooperative case, the scheduler organises execution of threads using a deterministic round-robin approach, and threads cooperate with each other using basic communication and synchronisation primitives. The simplification of the concurrency model used for scheduling threads aids both the writing of a scheduler specification, and the task of verifying its properties [PK09].

The content of the paper is as follows. In Section 2, we 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 using software tools to detect 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 or integers and is 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, $\hat{x} = \hat{y}$.

A SIGNAL process $z := P(x_1,\dots,x_n)$ consists of the composition of simultaneous equations which equate the output signal z as a function of the input signals $x_1,\dots,x_n$. 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 $z := F(x_1,\dots,x_n)$ for primitive process F, *(ii)* a synchronous composition $P \mid Q$ of processes P and 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 used by the SIGNAL language to define primitive processes specifying computations over signals. We list each equation 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 [LTL03].

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 whose output $z(t)$ for each tag $t \in \hat{z}$ is defined pointwise by $z(t) = f(x_1(t),\dots,x_n(t))$. The relationship between the clocks of the input and output signals is $\hat{z} = \hat{x_1} = \dots = \hat{x_n}$.

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 the signal x is present *and* b is present with $b(t)$ carrying the value `true`. The relationship between the clocks of the input and output signals is $\hat{z} = \hat{x} \cap [b]$ where $[b] = \{t \in \hat{b} \mid b(t) = \texttt{true}\}$.

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* by these operations. Clocks are represented by pure signals of type `event`, carrying a single boolean value `true`, denoting the presence of the signal. For a signal `x`, the clock operator `ˆx` returns a new signal of type `event` defined by the boolean expression `x=x`. We write ˆ0 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 using `event` signals: *(i)* The synchronisation relation `x ˆ= y` between the clocks of signal `x` and `y` corresponds to `ˆx = ˆy`, *(ii)* clock union relationship `x ˆ+ y` corresponds to `ˆx default ˆy`, *(iii)* clock intersection relationship `x ˆ* y` corresponds to `ˆx when ˆy`. Furthermore, the unary form of the sampling operation `when b` returns an event typed signal representing the clock of $[b]$.

Primitive operations on signals are composed to specify complex processes. Consider the `cell` process

```
process cell = (?integer x,boolean b!integer z)
(|z := x default (z$1 init 0)
 |z ˆ= x ˆ+ when b |)
```

which takes input signals `x` and `b` and produces an output signal `z`. The first equation specifies the data carried by `z` to be the value of `x`. If `x` is not present, the process outputs the previous value, initialised to 0. The second equation gives the clock of the process. The output signal `z` is present whenever `x` is present *or* whenever the boolean signal `b` is present and carries the value `true`.

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 labelled blocks containing a sequence of statements and program control-flow between blocks is represented by an edge. Statements may be operations `x = f(`**y**`)` for a list of variables **y**, or tests `if x goto L` and each block is terminated by either a `return` or `goto L` statement.

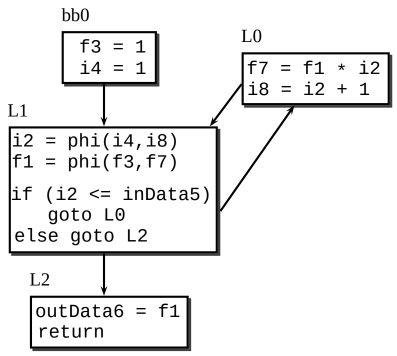
A program is said to be in *static single assignment form* whenever each variable in the program appears only once on the left hand side of an assignment. Following [CFR+91], a program is converted to SSA form by replacing assignments of a program variable x with assignments to new versions $\ldots, x_i, \ldots$ of x, uniquely indexing each assignment in the program. Each use of the

```
<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

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. 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.

```
void fact(int inData,
          int *outData){
 int f=1;
 int i=1;
 while (i<=inData){
   f *= i;
   i++;
 }
 *outData=f;
}
```



```
bb0:
f3 = 1
i4 = 1

L0:
f7 = f1 * i2
i8 = i2 + 1

L1:
i2 = phi(i4,i8)
f1 = phi(f3,f7)
if (i2 <= inData5)
   goto L0
else goto L2

L2:
outData6 = f1
return
```

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 [Boua, Bou06] is a framework for concurrent and parallel programming of software systems mixing both cooperative and preemptive threads. Our work deals with threads in a purely cooperative context, where *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 created in the scheduler. 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 [Boub].

The execution sequence of threads linked to a FairThreads scheduler is decomposed into a series of execution *instants*[1] during which each thread has an equal opportunity to run until its next cooperation point, and 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. 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. Once each thread has had an opportunity to run cooperatively or has otherwise been blocked, the instant is complete. At the end of each instant, events that were generated are *reset* or cleared and threads who missed an event may react to its absence in the next instant. A subset of the FairThreads synchronisation primitives used in our work is given in Figure 3.

`await(e)`	Wait for event `e`
`generate_value(e,v)`	Generate event `e` with value `v`. Multiple values are associated to `e` using multiple calls.
`get_value(e,k,v)`	Get k^{th} value associated with event `e` and store in `v`
`cooperate`	Yield control back to scheduler
`join(t)`	Suspend calling thread until thread `t` has terminated

Figure 3: Fairthreads Commands

3.1 Cooperative Thread Scheduling in FairThreads

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

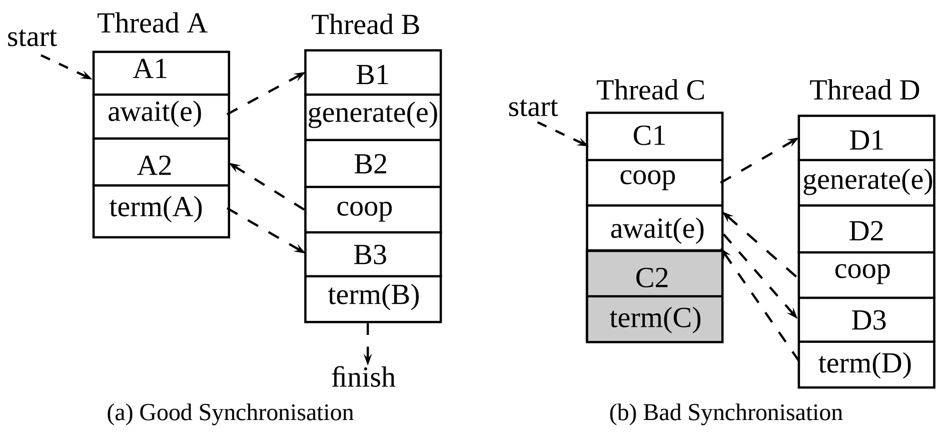


Figure 4: Thread Synchronisation Examples

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

3.1.1 Example: Good Synchronisation

Figure 4a depicts threads A and B each represented by a sequence of blocks containing program code and dashed arrows that represent 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. The command `Term` signifies the finalisation of thread execution and control is returned to the scheduler.

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.

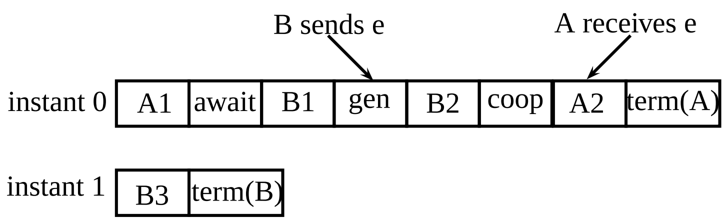


Figure 5: Instant decomposition of Threads A and B

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

Another scheduling example between threads C and D is presented in Figure 4b. 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 6. 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.

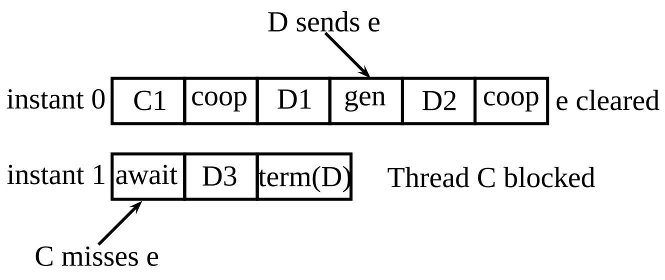


Figure 6: Instant decomposition of Threads C and D

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 7. 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.

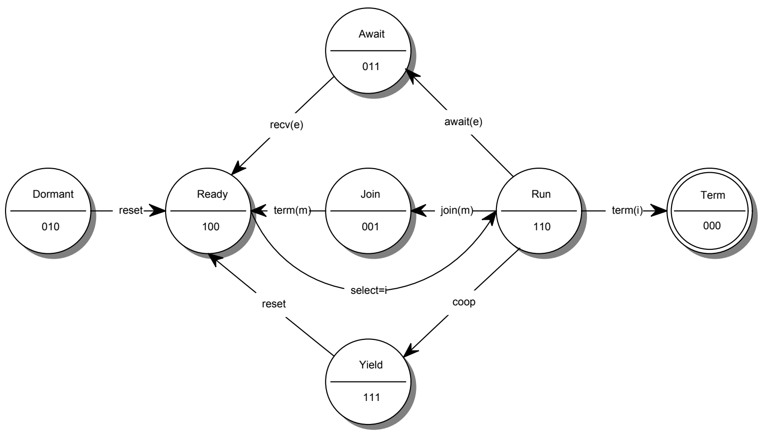


Figure 7: State Machine of Thread *i*

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 8 where a scheduler is defined and a new event is created in the scheduler. Connected to the scheduler are two threads `1` and `2`

```
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 8: FairThreads Example

whose function bodies are listed in Figure 9. Thread `1` initializes variables and waits for events to occur on the scheduler by executing `await(e)`. 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(){                      | void thread2(){
 int v,f;                            |  int num;
// ft_thread_cooperate();            |  initthread2();
 ft_thread_await(e);                 |  num = getValue();
 ft_thread_get_value(e,0,&v);        |  ft_thread_generate_value(e,(void*)&num);
 fact(v,&f);                         |  ft_thread_cooperate();
 finalizethread1();                  |  finalizethread2();
}                                    | }
```

Figure 9: 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 a `cooperate` 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 missed events due to 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*

```
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 10: Listing of SIGNAL process proFact

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 [KTBB06] and [BGM$^+$09].

The SIGNAL process `proFact` listed in Figure 10, is generated from the control-flow diagram. The equations model the sequence of computations defined by the control-flow of 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

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. For simplicity we illustrate our method with two threads and depict the components of the software system and their interconnections in Figure 11.

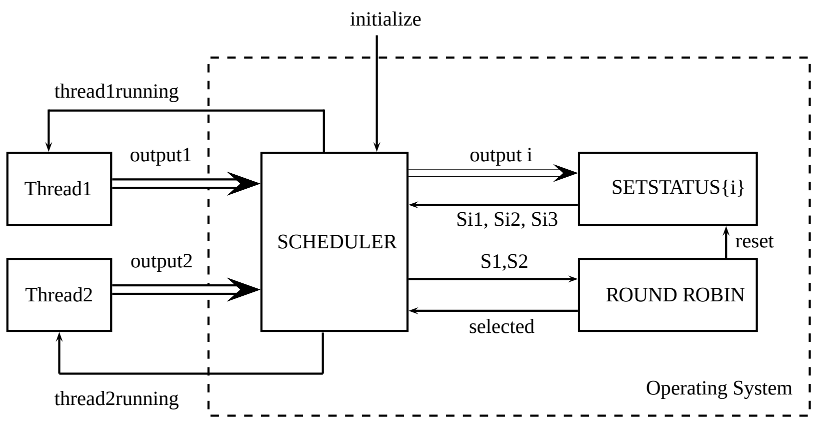


Figure 11: Components of a Threaded Software System

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 11, 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 (cf. Figure 7). 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 11 as double arrows labelled output1 and output2. We list the output signals in Figure 12 and their corresponding FairThreads commands and parameters. In particular, 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 commands may be extended to include a parameter denoting a timeout value. For example, the command `await(e)` is extended to `await_n(e,k)`, where *k* is a timeout value. The modelling of these commands 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. For brevity, these details have been omitted, and the interested reader should refer to [JBGT10].

FairThreads Command	**Output Signal**
`await(e)`	`integer await`
`generate_value(e,v)`	`integer genEvent,` `genEventValue`
`get_value(e,k,v)`	`integer getValueFromEvent,` `getithValueOfEvent`
`cooperate`	`integer cooperate`
`join(t)`	`integer join`
	`event endProcessing`

Figure 12: 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 11.

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.

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 7 are represented by a unique three-valued boolean code, and thus a thread *i* is described by a tuple of signals `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 7 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

```
|      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 ...
```

Note also that the equations specify that each thread is initialized to the *Dormant* state.

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

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 `initialize`. Intuitively speaking, each state signal must be present to specify the thread and event state throughout the entire execution sequence.

5.2.1 Deterministic Thread Selection

The box labelled `ROUNDROBIN` in Figure 11 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 by the SIGNAL equation:

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

where `reset` 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 7) it is an input signal to the process `SETSTATUS`. We provide a complete listing of the FairThreads scheduler SIGNAL specification in [JBGT10] for the examples we have presented.

6 Threaded Program Verification in SIGNAL

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. The compiler can automatically generate sets of dynamic polynomial equations from SIGNAL specifications that defines an automaton describing the dynamical behaviour of the specification. Dynamic properties are verified using the model checker SIGALI [MR02] which is an interactive tool specialized on algebraic reasoning in $\mathbf{Z}/3\mathbf{Z} = \{0,1,-1\}$ logic. It can analyze the 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

Program	Complexity				Time (Result)	
	State vars	States	Reachable	Transitions	PROP1	PROP2
Ex. 3.1.1	34	2^{34}	84	548 931	0.09s (true)	1.18s (false)
Ex. 3.1.2	30	2^{30}	19	286 755	0.09s (true)	2.92s (true)

Table 1: Verification Complexity, Time and Results

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.

Using SIGALI with only slight modifications to our 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 the value `true` whenever thread `1` is currently in the *Run* state. The signal `thread2running` is defined likewise. We add to the model a testing signal

```
| TWORUN := thread1Running and thread2Running
| Sigali(Never(B_True(TWORUN)))  %Property PROP1%
```

and use SIGALI to prove property `PROP1`: it is never the case that the signal `TWORUN` carries the value `true`. Thus, we conclude that our model does not schedule more than one thread to run at a time.

We may also detect missed events in threaded programs which result in missynchronisations of threads or at the very least ineffective code. Recall that the state signals `S1a` and `S2a` carry the number of the event that thread `1` and thread `2` are awaiting, respectively. Our verification method requires us to encode these integer signals as boolean signals and we note that this task is greatly simplified when we consider specifications with a single event. We define the property

```
| RECV := (S1a or S2a) and p1
```

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 and is synchronous with the master clock `^initialize` of the software system, we add to the model the following:

```
|observer := RECV default observer$1 init false
|observer ^= initialize
|Sigali(Never(B_True(observer))) %Property PROP2%
```

and use SIGALI to prove property `PROP2`. For Example 3.1.1, the signal `RECV` sometimes carries the value `true`, hence `PROP2` is `false`, showing that event 1 is received. For Example 3.1.2 we prove that the signal `RECV` never carries the value `true`, showing that event 1 is never received, and `PROP2` is `true`. Table 1 summarises our verification speed and results.

7 Conclusion and Future Work

In this paper we considered FairThreads threads attached to a scheduler that cooperate for access to resources and the processor. Our solution to semi-automatically detecting deadlocks due to missed events is based on the synchronous data-flow language SIGNAL. We outlined a translation method enabling us to reason about program control-flow in a formal clock calculus. The scheduler model was hand-written specific to the FairThreads scheduling policy. Software tools were used for generating simulations of the examples we have presented and the SIGALI model checker was used for proving dynamical properties of our model.

The examples in our experiments were chosen to illustrate a fundamental synchronisation error which occurs independent of complexities in program control. Further work is required to consider more elaborate examples, and to test the scalability of our approach. Indeed, as with other approaches to model checking, we would require sophisticated methods of program abstraction which is often difficult for large programs. Therefore we present our work as a "proof-of-concept" for the use of synchronous languages for formal modelling in software engineering.

In addition to these points, we envisage further generalisations of the scheduler model to simulate other scheduling policies and to include other synchronisation mechanisms such as mutexes, where tokens are used to guarantee the mutual exclusive access to shared resources in the software system. We may consider modelling and verification tasks in the context of other cooperative threaded frameworks such as SHIM [ET06].

Bibliography

[AB01] C. Artho, A. Biere. Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs. *Software Engineering Conference, Australian* 0:0068, 2001.

[BB91] A. Benveniste, G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE* 79(9):1270–1282, Sep. 1991.

[BGM+09] L. Besnard, T. Gautier, M. Moy, J. Talpin, K. Johnson, 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.

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

[Boub] F. Boussinot. Operational Semantics of Cooperative Fair Threads. www-sop.inria.fr/meije/rp/FairThreads/FTC/documentation/semantics.pdf.

[Bou06] F. Boussinot. FairThreads: mixing cooperative and preemptive threads in C: Research Articles. *Concurrency and Computation: Practice and Experience* 18(5):445–469, 2006.

[CFR+91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, F. 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.

[ET06] S. Edwards, 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.

[HD01] J. Hatcliff, M. Dwyer. Using the Bandera Tool Set to Model-check Properties of Concurrent Java Software. In *LNCS*. Pp. 39–58. Springer-Verlag, 2001.

[HP00] K. Havelund, T. Pressburger. Model Checking JAVA Programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer* 2(4):366–381, 2000.

[JBGT10] K. Johnson, L. Besnard, T. Gautier, J.-P. Talpin. A Synchronous Approach to Threaded Program Verification. Technical report, INRIA, 2010.
http://hal.archives-ouvertes.fr/inria-00492694/PDF/RR-7320.pdf

[KTBB06] H. Kalla, J.-P. Talpin, D. Berner, 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*. Pp. 426–436. March 2006.

[LTL03] P. Le Guernic, J.-P. Talpin, J.-C. Le Lann. Polychrony for System Design. *Journal for Circuits, Systems and Computers* 12(3):261–304, April 2003.

[MR02] H. Marchand, E. Rutten. Signal and Sigali User's Manual. http://www.irisa.fr/espresso/Polychrony, 2002.

[PK09] P. Parizek, T. Kalibera. Platform-Specific Restrictions on Concurrency in Model Checking of Java Programs. In *FMICS '09: Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems*. Pp. 117–132. Springer-Verlag, Berlin, Heidelberg, 2009.