

# INSTITUT FÜR INFORMATIK

## SyncCharts in C

Reinhard von Hanxleden

Bericht Nr. 0910

Mai 2009



CHRISTIAN-ALBRECHTS-UNIVERSITÄT  
ZU KIEL

Institut für Informatik der  
Christian-Albrechts-Universität zu Kiel  
Olshausenstr. 40  
D – 24098 Kiel

## **SyncCharts in C**

Reinhard von Hanxleden

Bericht Nr. 0910  
Mai 2009

e-mail:  
[rvh@informatik.uni-kiel.de](mailto:rvh@informatik.uni-kiel.de)

Technical Report

## Abstract

Statecharts are a well-established visual formalism for the description of reactive real-time systems. The SyncCharts dialect of Statecharts, which builds on the synchrony hypothesis, has a sound formal basis and ensures deterministic behavior. This report presents SyncCharts in C (SC), an approach on how to seamlessly and efficiently embed SyncCharts constructs into a conventional imperative programming language. SC offers deterministic concurrency and preemption via a simulation of multi-threading, inspired by reactive processing.

SC can be used as a regular programming language, requiring just a C compiler; no special tools or hardware are needed. However SC's conciseness, completeness and semantic closeness to SyncCharts make it an attractive candidate in a number of other scenarios: 1) as an intermediate target language for synthesizing graphical SyncChart models into executable code, in a more traceable manner than the traditional path through Esterel; 2) as instruction set architecture for programming precision timed (PRET) or reactive architectures; or 3) as a virtual machine instruction set. A reference implementation of SC, based on light-weight C macros, is available as open source code.

**Key words:** SyncCharts, Statecharts, Esterel, synchronous programming, code synthesis, model-based design

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Introductory Examples</b>	<b>3</b>
2.1	Reactive Control in SC—The PCO Example . . . . .	3
2.2	Signals in SC—The grcbal3 Example . . . . .	7
<b>3</b>	<b>A Tour of SC</b>	<b>12</b>
3.1	The SC Programming Model . . . . .	12
3.1.1	Synchronous threading . . . . .	12
3.1.2	Signals . . . . .	14
3.2	Multithreading Simulation . . . . .	15
3.2.1	Coarse program counters . . . . .	15
3.2.2	The dispatcher . . . . .	15
3.2.3	Thread and label structuring . . . . .	17
3.2.4	Thread scheduling . . . . .	20
3.3	SC Operators . . . . .	24
3.3.1	SC Thread Handling Operators . . . . .	24
3.3.2	SC signal operators . . . . .	26
3.3.3	SC sequential control operators . . . . .	28
3.3.4	An example of expanded macros—ABRO . . . . .	31
3.4	SC Structure . . . . .	31
3.4.1	Program files . . . . .	31
3.4.2	Functions . . . . .	32
3.4.3	Types . . . . .	33
3.4.4	Variables . . . . .	33
<b>4</b>	<b>Examples</b>	<b>35</b>
4.1	Count2Suspend . . . . .	35
4.2	Exits . . . . .	36
4.3	FilteredSR . . . . .	39
4.4	Shifter3 . . . . .	39
4.5	PreAndSuspend . . . . .	41
4.6	Reincarnation . . . . .	42
4.7	PrimeFactor . . . . .	43
<b>5</b>	<b>Related Work</b>	<b>45</b>

<b>6</b>	<b>Experimental results</b>	<b>49</b>
6.1	Conciseness of SC, Code Size . . . . .	49
6.2	SC Performance . . . . .	49
<b>7</b>	<b>Conclusions and Outlook</b>	<b>52</b>
<b>A</b>	<b>The SC files</b>	<b>57</b>
<b>B</b>	<b>Complete Examples</b>	<b>63</b>
B.1	ABRO . . . . .	63
B.2	grcbal3 . . . . .	70
B.3	PCO . . . . .	71
B.4	Count2Suspend . . . . .	72
B.5	Exits . . . . .	73
B.6	Exits-no-isatcall . . . . .	75
B.7	Exits-inlined . . . . .	77
B.8	FilteredSR . . . . .	78
B.9	PreAndSuspend . . . . .	79
B.10	PrimeFactor . . . . .	81
B.11	Reincarnation . . . . .	82
B.12	Shifter3 . . . . .	83
B.13	SurfDepth . . . . .	85

# List of Figures

2.1	The PCO (Producer-Consumer-Observer) example. . . . .	4
2.2	The <code>grcbal3</code> example. . . . .	8
3.1	The status of the whole program . . . . .	13
3.2	Execution status of a single thread . . . . .	13
3.3	The ABRO example. . . . .	18
3.4	The <code>SurfDepth</code> example. . . . .	19
3.5	ABRO tick function after macro expansion . . . . .	30
4.1	The <code>Count2Suspend</code> example. . . . .	35
4.2	The Exits example. . . . .	37
4.3	Alternative variants for the SC tick function of the Exits example (Fig. 4.2). . . . .	38
4.4	The <code>FilteredSR</code> example. . . . .	39
4.5	The <code>Shifter3</code> example. . . . .	40
4.6	The <code>PreAndSuspend</code> example. . . . .	41
4.7	The <code>Reincarnation</code> example. . . . .	42
4.8	The <code>PrimeFactor</code> example. . . . .	43
6.1	Comparison of SC with two code synthesis variants of Esterel Studio. . . . .	50

# List of Tables

2.1	SC thread operators . . . . .	5
2.2	SC signal operators and sequential control operators . . . . .	9

# Listings, Outside Figures

3.1	<code>selectCidPrio()</code> : Computation of id of thread to be dispatched, considering priorities (from <code>sc.c</code> ) . . . . .	16
3.2	<code>selectCidNoprio()</code> : Computation of id of thread to be dispatched, without considering priorities (from <code>sc.c</code> ) . . . . .	16
3.3	<code>dispatch()</code> : Variable definitions for the dispatcher (from <code>sc.h</code> ) . . . . .	16
A.1	The header file <code>sc.h</code> . . . . .	57
A.2	The main program file <code>sc.c</code> . . . . .	60
A.3	The Makefile . . . . .	61
A.4	<code>make.trace</code> : a run of <code>make</code> . . . . .	61
B.1	<code>ABRO.c</code> . . . . .	63
B.2	<code>ABRO.out</code> . . . . .	63
B.3	Assembler generated from ABRO tick function without optimizations before linking . . . . .	64
B.4	Assembler of ABRO tick function with optimizations ( <code>gcc -O3</code> ), before linking . . . . .	68
B.5	<code>grcbal3.c</code> . . . . .	70
B.6	<code>grcbal3.out</code> . . . . .	70
B.7	<code>PCO.c</code> . . . . .	71
B.8	<code>PCO.out</code> . . . . .	71
B.9	<code>Count2Suspend.c</code> . . . . .	72
B.10	<code>Count2Suspend.out</code> . . . . .	72
B.11	<code>Exits.c</code> . . . . .	73
B.12	<code>Exits.out</code> . . . . .	74
B.13	<code>Exits-no-isatcall.c</code> . . . . .	75
B.14	<code>Exits-no-isatcall.out</code> . . . . .	76
B.15	<code>Exits-inlined.c</code> . . . . .	77
B.16	<code>Exits-inlined.out</code> . . . . .	77
B.17	<code>FilteredSR.c</code> . . . . .	78
B.18	<code>FilteredSR.out</code> . . . . .	79
B.19	<code>PreAndSuspend.c</code> . . . . .	79
B.20	<code>PreAndSuspend.out</code> . . . . .	80
B.21	<code>PrimeFactor.c</code> . . . . .	81
B.22	<code>PrimeFactor.out</code> . . . . .	82
B.23	<code>Reincarnation.c</code> . . . . .	82
B.24	<code>Reincarnation.out</code> . . . . .	83
B.25	<code>Shifter3.c</code> . . . . .	83
B.26	<code>Shifter3.out</code> . . . . .	84
B.27	<code>SurfDepth.c</code> . . . . .	85



B.28 SurfDepth.out . . . . . 85

# Chapter 1

## Introduction

The control flow of reactive systems typically entails not just the sequential control flow found in traditional programming languages, such as conditionals and loops, but also exhibits concurrency and preemption. This reactive control flow is naturally expressed by the Statechart formalism introduced by David Harel [12], which extends classical finite state machines by concurrency and hierarchy/preemption. These extensions allow to keep descriptions compact and avoid the classical state explosion problem.

The graphical Statechart formalism has been originally developed to let application experts precisely describe the behavior desired for an application. Its visual nature makes this formalism accessible to non-computer scientists, without the need to be versed in a traditional programming language. However, beyond this visual *syntax*, Statecharts offer important *concepts* that can be expressed in non-visual languages as well, such as the concepts of state-based control flow, hierarchy, concurrency, and its model of time. This model of computation (MoC) of Statecharts offers a powerful abstraction mechanism compared to classical programming models. For this reason, Statechart models are typically viewed as more abstract than, say, a program written in C.

A typical design flow may start with a graphical modeling tool, which synthesizes a Statechart model into a C program, which is further compiled into some executable. However, it is also quite common to bypass the visual modeling step. Just as the code generator of a modeling tool is able to express the Statechart MoC in a C program, so it is possible for a human programmer to express Statechart behavior as a C program [27, 31]. This does not offer the visual appeal of graphical Statecharts, but has other advantages:

- no need for a modeling tool,
- high portability, and
- seamless integration with a fully featured, widely used programming language, including the type system, expression handling, control flow, access to low-level I/O, pre-processors, etc.

Even if one assumes a design flow that starts at a graphical modeling tool that supports Statecharts, it is of interest how Statechart behavior can be expressed concisely in a traditional programming language. For a number of reasons, we would like to be able to generate code that preserves the structure of the graphical model:

- it simplifies the development of the code synthesizer of the modeling tool;

- it facilitates back-annotations from the executable code into the graphical model, which allows visual animations of the running code and allows to set break points in the model; and
- it simplifies code certification for safety-critical embedded systems.

This report describes *SyncCharts in C* (SC), which is a light-weight approach to express SyncCharts [2] in C programs. SC combines the formal soundness of SyncCharts, including deterministic concurrency and preemption, with the efficiency and wide support for the C language. The main idea of SC is to emulate multi-threading, and is inspired by reactive processing [30]. As we do not have direct access to the program counter at the C language level, we keep track of individual threads via state labels, implemented as usual C program labels. These labels can also be viewed as continuations [4], or coroutine [8, 14] re-entry points. Precedence among transitions, respecting strong/weak abortions and hierarchy, and the adherence to signal dependencies are achieved by checking transition triggers in the proper order as well as assigning appropriate thread ids and priorities.

To write and execute an SC application requires neither specific tools nor special execution platforms, although both may support this concept further. All that is needed to get started is an understanding of SyncCharts (see *e. g.* the tutorial provided by André [2]), a C compiler, and the SC files. The SC files consist of one header file (`sc.h`), to be included by the application code, and one C-file (`sc.c`), to be linked in by the application. They are open source and available for free download<sup>1</sup>.

As the name suggests, SC has been developed with the SyncCharts execution model in mind. However, SC can also be viewed as a generic approach for programming light-weight, deterministic concurrent programs in C, without using SyncChart-specifics such as (valued) signals or (weak) abortions. For example, SC appears to be a suitable candidate for writing concurrent C programs that have predictable functionality and timing on PRET-like architectures [18], without having to resort to low-level synchronization mechanisms based on physical timing characteristics.

In this report, we will first work through two examples that give an overview how SC programmers can implement reactive control and signal-based communication, followed by a full tour of SC in Chapter 3 and further examples in Chapter 4. Chapter 5 discusses related work, experimental results are presented in Chapter 6. The report concludes in Chapter 7. Appendix A lists the SC files, Appendix B gives the complete code for the examples.

---

<sup>1</sup><http://www.informatik.uni-kiel.de/rtsys/sc/>

# Chapter 2

## Introductory Examples

This chapter gives a first practical introduction to SC by working through two examples. The first presents the fundamental reactive control flow mechanism supported by SC, namely concurrency and preemption. The second example makes use of signal handling and illustrates how SC supports intricate thread inter-dependencies.

### 2.1 Reactive Control in SC—The PCO Example

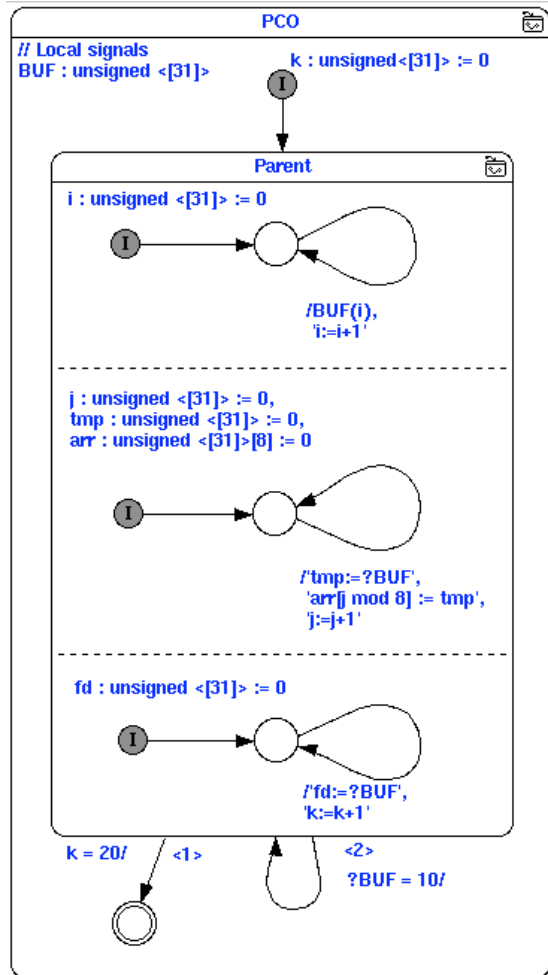
This section covers

- the general structure of SC programs,
- how SC macros are embedded in regular C code,
- the concept of deterministic, label-based simulated multi-threading, and
- deterministic preemptions.

We will illustrate these points with PCO, shown in Fig. 2.1, a simple producer-consumer example with an observer, inspired by Lickly *et al.* [18]. In addition to the original example, PCO also has a parent thread that restarts production/consumption once the buffer has the value 10, and which terminates after 20 iterations.

The SyncCharts version (Fig. 2.1a) shows a **Parent** macrostate, which is an AND (parallel) state that consists of three substates, corresponding to the producer, consumer and observer. Each substate consists of a state with a self-transition, which is triggered unconditionally and performs some action. For example, the producer state writes the current value of `i` into a buffer `BUF`, a *valued signal* in SyncCharts parlance. The consumer state reads the value of `BUF` into some variable `tmp` and then writes `tmp` into an array `arr`. The observer also reads from `BUF`. The **Parent** state re-enters itself when `BUF` has the value 10, and transitions to some final state when `k`, incremented by the observer, has reached the value 20.

Compared to an implementation that would try to achieve the same behavior with, say, Java threads, the interesting aspect of the SyncChart implementation is that the concurrency is deterministic. The three substates of **Parent** execute in lock step, and the SyncCharts semantics requires that in each execution, `BUF` must be written before it is read. Hence, the code generator of EsterelStudio, which generates C code from this (via Esterel), must schedule



(a) SyncChart, produced with EsterelStudio

```

Producer
int main() {
  DEAD (28);
  volatile unsigned int * buf =
    (unsigned int*) (0x3F800200);
  unsigned int i = 0;
  for (i = 0; ; i++) {
    DEAD (26);
    *buf = i;
  }
  return 0;
}

Consumer
int main() {
  DEAD (41);
  volatile unsigned int * buf =
    (unsigned int*) (0x3F800200);
  unsigned int i = 0;
  int arr[8];
  for (i = 0; i < 8; i++)
    arr[i] = 0;
  for (i = 0; ; i++) {
    DEAD (26);
    register int tmp = *buf;
    arr[i%8] = tmp;
  }
  return 0;
}

Observer
int main() {
  DEAD (41);
  volatile unsigned int * buf =
    (unsigned int*) (0x3F800200);
  volatile unsigned int * fd =
    (unsigned int*) (0x8000600);
  unsigned int i = 0;
  for (i = 0; ; i++) {
    DEAD (26);
    *fd = *buf;
  }
  return 0;
}

```

(b) PRET version, without preemption (from [18])

```

1 #include "sc.h"
2
3 int BUF, fd, i, j, k = 0, tmp, arr [8], idHi = 4;
4 typedef enum { TickEnd, Main, Cons, Obs, Prod }
   idtype;
5 const int ids [] = { 0, 1, 2, 3, 4 };
6 const char *id2threadname [] = { "TickEnd", "Main",
   "Cons", "Obs", "Prod" };
7
8 // ===== MAIN FUNCTION =====
9 int main()
10 {
11   int notDone, init = 1;
12
13   do {
14     notDone = tick( init ); // Call tick function
15     //sleep(1); // Slow down by 1 sec
16     init = 0;
17   } while (notDone);
18   return 0;
19 }
20
21 // ===== TICK FUNCTION =====
22 int tick( int islnit )
23 {
24   TICKSTART( islnit ); // Main thread
25
26   PCO: PAR(0, Prod, ids[Prod]);
27   PAR(0, Cons, ids[Cons]);
28   PAR(0, Obs, ids[Obs]);
29   PARE(0, Parent, id2b(Prod) | id2b(Cons) |
   id2b(Obs));
30
31   Prod: for ( i = 0; ; i++) { // Producer
32     PAUSE(L0);
33   L0: BUF = i; }
34
35   Cons: for ( j = 0; j < 8; j++) // Consumer
36     arr [j] = 0;
37   for ( j = 0; ; j++) {
38     PAUSE(L1);
39   L1: tmp = BUF;
40     arr [j % 8] = tmp; }
41
42   Obs: for ( ; ; ) { // Observer
43     PAUSE(L2);
44   L2: fd = BUF;
45     k++; }
46
47   Parent: PAUSE(L3); // Main (cont'd)
48   L3: if (k == 20) // IF iteration limit
49     TRANS(Done); // THEN terminate
50     if (BUF == 10) // IF buffer = 10
51     TRANS(PCO); // THEN restart PCO
52     goto Parent; // ELSE continue
53
54   Done: TERM;
55   TICKEND;
56 }

```

(c) Complete SC program

Figure 2.1: The PCO (Producer-Consumer-Observer) example.

Mnemonic, Operands	Notes
TICKSTART*( <i>isInitial</i> )	Start (initial) tick.
TICKEND	Finalize tick, return 1 iff there is still an enabled thread.
PAUSE*( <i>l</i> )	Deactivate current thread for this tick, continue next tick at address label <i>l</i> .
TRANS( <i>l</i> )	Abort descendant threads, jump to <i>l</i> .
SUSPEND*( <i>l</i> )	Suspend (pause) thread and its descendants, continue at <i>l</i> .
TERM*	Terminate current thread.
PAR( <i>p, l, id</i> )	PAR creates a thread with an initial priority <i>p</i> , a start address <i>l</i> , and an id <i>id</i> .
PARE*( <i>p, l, ids_desc</i> )	PARE denotes priority <i>p</i> and continuation address <i>l</i> for the spawning thread. To allow detection of normal termination of descendant threads (via JOIN), store their ids in <i>ids_desc</i> .
JOIN*( <i>l_then, l_else</i> )	If descendant threads have terminated normally, jump to <i>l_then</i> ; else pause, proceed to <i>l_else</i> .
PRIO*( <i>p, l</i> )	Set current thread priority to <i>p</i> , continue at <i>l</i> .
PPAUSE*( <i>p, l</i> )	Shorthand for PRIO( <i>p, l'</i> ); <i>l'</i> : PAUSE( <i>l</i> ) (saves one call to dispatcher).
JPPAUSE*( <i>p, l_then, l_else</i> )	Shorthand for JOIN( <i>l_then, l</i> ); <i>l</i> : PPAUSE( <i>p, l_else</i> ) (saves another call to dispatcher).

Table 2.1: SC thread operators—tick delimiters, fork/join, priority handling, and abortion and suspension. Operators marked with an asterisk may call the thread dispatcher, *i. e.*, can result in a thread context switch.

the producer before the consumer and the observer. Similarly, the transitions leaving **Parent** have deterministic behavior; in this example, they are so-called *weak abortions*, meaning that the body of the parent gets to finish its current execution before a transition is taken. An implementation with classical Java threads offers none of these assurances. To achieve the same effect would require explicit barrier synchronization. Note also that for example using Java’s `synchronized` to protect access to the shared buffer does not help, as this would only guarantee exclusive access, but no ordering.

One approach suggested recently to enforce this synchronization is to use explicit low-level time-triggered scheduling. The PRET architecture [18] offers a **DEAD** instruction which guarantees a (minimal) delay before a thread proceeds. Fig. 2.1b shows the PRET version of a reduced variant of PCO that does not have preemptions. In this PRET version, the buffer access is coordinated by giving the producer a head start before the consumer and observers (**DEAD 28** vs. **DEAD 41**), and then keeping all three running at the same rate (**DEAD 26**). To guarantee proper synchronization this way requires a timing analysis of the code and the underlying architecture, and the resulting program is fairly non-portable.

The SC version of PCO is shown in Fig. 2.1c. The **main** function contains a **while** loop that calls a **tick** function. This function computes one reaction by simulating all *enabled* threads for one tick. The return value of **tick** indicates whether the program has terminated, *i. e.*, whether all threads have become *disabled*. The **while** loop of **main** continues as long as any thread is still enabled. In this example, a call to **sleep(1)** results in a reaction rate of—approximately—once per second.

The tick function consists of regular C code and some macros. These *SC macros* are declared in `sc.h`, included in line 1. An overview of the SC Thread Handling Operators, which perform the multi-threading simulation and form the core of SC, is given in Table 2.1. The

remaining SC operators are introduced in Sec. 2.2, Table 2.2. A full discussion of all SC is presented in Sec. 3.3.

The first SC macro used in PCO, TICKSTART, performs some book keeping, depending on whether this is the initial tick or not. This is followed by a sequence of PAR/PARE macros, which fork off the children of the current thread. The current thread, started when entering tick, is the Main thread. The forked threads are Prod, Cons, and Obs.

As the forked threads are associated with the Parent state of the SyncChart, we will also refer to these as Parent's children; however, the thread that is forking them is the Main thread. In this example, the Main thread only forks these children, as the Parent macrostate is the only macrostate ever entered by Main.

Each PAR gives a thread its initial priority (here all 0), a starting label, and an id. PARE specifies a priority for the current thread (again 0), a continuation label (ParentMain), and the set of children that were just forked. Sets of threads are encoded as a bit vector, id2b maps a thread into this vector. This set is needed to properly abort Main's children when TRANS is called, see below.

Threads are declared with the `idtype` enumeration type (line 4).

The starting point of each thread is declared with an ordinary C label, named after the thread. This is just a convention; from a C perspective, these labels and the thread names have different name spaces and are different objects: one is a memory address, the other is an enumeration type index.

The code for each thread is regular C code, except that each thread contains a PAUSE macro. PAUSE indicates that a thread becomes inactive and is ready to relinquish control to the *dispatcher*. An argument to PAUSE indicates at which label the pausing thread should resume in the next tick.

The dispatcher, called by PAUSE, selects a thread for resumption. In PCO the dispatcher selects from the *active* threads, which still have work to do in the current thread, the one with the highest *thread id*. The dispatcher may also consider dynamic priorities, see Sec. 2.2, but in PCO these are all 0. Threads are mapped to their ids with the `ids` array (line 5). The TickEnd thread, which must be present in any SC program and must have the lowest id (0), returns from tick if none of the other threads are active anymore.

Taking a look at the Main thread continuation at the Parent label (line 47), we note that the transitions triggered by inspecting first `k` and then `BUF` are implemented with a TRANS macro (lines 49 and 51). This macro transfers control to the argument label, and also aborts Parent's child threads. Finally, TERM terminates the current thread (Main), and TICKEND does last book keeping before leaving tick again.

To summarize, we simulate multi-threading by keeping track of continuation points and calling a dispatcher whenever a context switch might occur. In the example, the dispatcher is called by PAUSE (thread becomes inactive for the current tick), PARE (children have been created, current thread may have changed priority), and TERM (thread has terminated). The context of a thread is very light-weight: it consists of its id (static), its continuation label (dynamic), and a priority (dynamic). Everything else is shared. The thread id encodes the order in which threads are dispatched. In PCO, the producer has to run before the consumer and the observer, hence Prod gets the highest id, which is 4. For a full discussion of PCO's precedence constraints, see Sec. 3.2.4, p. 23.

All threads are included in one C tick function, just as for example a SyncChart or Esterel program is usually synthesized into a single reaction function. This makes data sharing and

communication trivial (compare for example with the PRET communication in Fig. 2.1b), but limits modularization. This is a consequence of the label-based continuation encoding, since in C, we cannot transfer control to a label across function calls. Alternatives, such as encodings based on `setjmp/longjmp`, would provide more flexibility, but would also incur higher overhead. Note, however, that modularization is still possible insofar as “instantaneous” functionality, without any SC operator that calls the dispatcher, can still be compartmentalized into function calls. This suggests a programming model where the thread structure and their scheduling logic is summarized in a top-level tick function, and thread-local activities and data-intensive computations are modularized as function calls.

## 2.2 Signals in SC—The `grcbal3` Example

This section covers

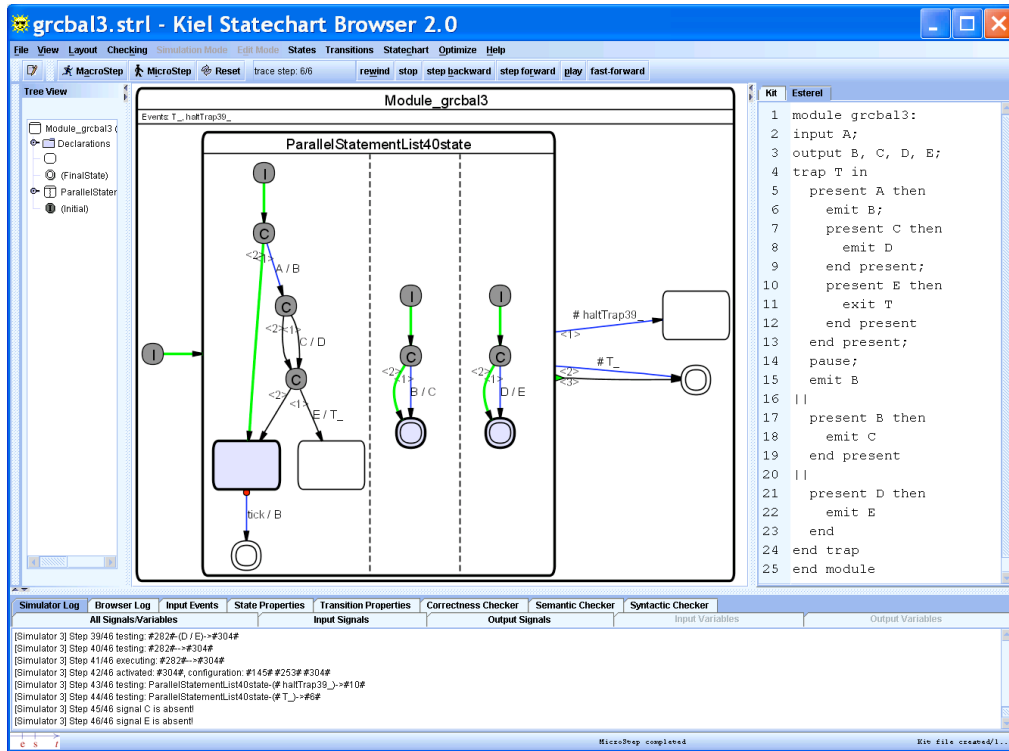
- more elaborate thread scheduling via the use of dynamic thread priorities,
- signal handling,
- a synthesis path from Esterel to SC, and
- how SC macros alone suffice to write a tick function.

Again we use an example, `grcbal3`, to illustrate these issues. Originally, this example was programmed in Esterel, and has been presented by Edwards and Zeng in their description of the Columbia Esterel Compiler [9]. Hence the name of the benchmark: GRC is the Graph Code intermediate representation of the CEC, BAL is the Bytecode Assembly Language of a virtual machine (VM) targeted by the CEC. The `grcbal3` Esterel code has been transformed into a SyncChart using KIEL [23]. Fig. 2.2a shows the Esterel version, on the right, with the generated SyncChart, in the midst of an animated simulation—the initial tick has just been executed, with no inputs present.

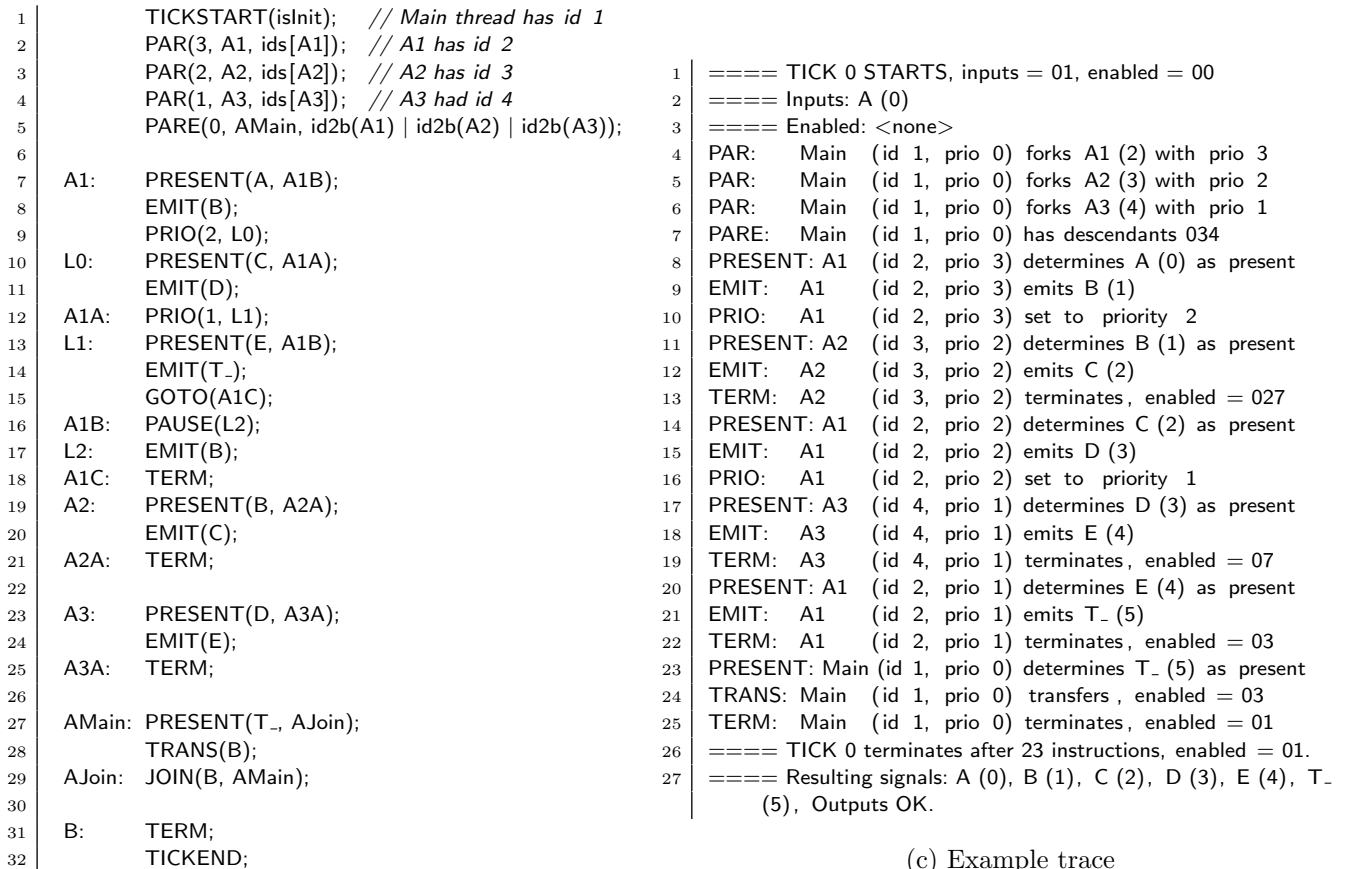
The Esterel program illustrates the use of signals to synchronize threads. It has an input signal `A` and output signals `B`...`E`. There are three concurrent threads, which are enclosed in a trap triggered by `T`. Esterel’s trap construct provides exception handling; in the example, the `exit T` statement (line 11) throws the exception. The three threads communicate back and forth via signals; for example, if `A` is present, the first thread emits a `B`, which causes the second thread to emit `C`, which in turn causes the first thread to emit a `D`.

The SyncChart synthesized by KIEL is equivalent to the Esterel version. However, as SyncCharts do not provide traps, they have to be emulated with weak abortions. This translation is always possible, and in `grcbal3` this can be done in a straightforward fashion, via a weak abort triggered by a fresh signal `T_`. The transition that implements this is shown in the lower right of the SyncChart, which leads to a final state (double circle). The `#`-mark means that the transition is *immediate*, meaning it can be triggered from the initial instant on. Note that the synthesis process produces a superfluous state, reachable via `haltTrap39`—which is nowhere emitted, hence it can be safely eliminated. This is a result of the general rule for transforming traps, which has to handle nested traps and trap actions [23], and a lack of a subsequent optimization in KIEL that would remove such clearly unreachable states.





(a) Screen shot of KIEL [24], as it synthesizes a SyncChart from the original Esterel code [9]



(b) SC tick function

(c) Example trace

Figure 2.2: The grcbal3 example.

Mnemonic, Operands	Notes
SIGNAL( $S$ )	Initialize a local signal $S$ .
EMIT( $S$ )	Emit signal $S$ .
PRESENT( $S, l_{else}$ )	If $S$ is present, proceed normally; else, jump to $l_{else}$ .
EMITINT( $S, val$ )	Emit valued signal $S$ , of type integer, with value $val$ .
EMITINTMUL( $S, val$ )	Emit valued signal $S$ , of type integer, combined with multiplication, with value $val$ .
VAL( $S, reg$ )	Retrieve value of signal $S$ , into register/variable $reg$ .
PRESENTPRE( $S, l_{else}$ )	If $S$ was present in previous tick, proceed normally; else, jump to $l_{else}$ . If $S$ is a signal local to thread $t$ , consider last preceding tick in which $t$ was active, <i>i. e.</i> , not suspended.
VALPRE( $S, reg$ )	Retrieve value of signal $S$ at previous tick, into register/variable $reg$ .
GOTO( $l$ )	Jump to label $l$ .
CALL( $l, l_{ret}$ )	Call function $l$ (eg, an on exit function), return to $l_{ret}$ .
RET	Return from function call.
ISAT( $id, l_{state}, l$ )	If thread $id$ is at state $l_{state}$ , then proceed to next instruction (which might be an on exit function of associated with $id$ at state $l_{state}$ ). Else, jump to label $l$ .
ISATCALL( $id, l_{state}, l_{action}, l$ )	Shorthand for ISAT( $id, l_{state}, l$ ); CALL( $l_{action}, l$ )

Table 2.2: SC signal operators (pure signals, valued signals, and accesses to the previous tick) and SC sequential control operators (jumps and exit actions).

Fig. 2.2b shows the tick function of the SC version of `grcbal3`. In addition to the SC concurrency operators already introduced in Sec. 2.1 and Table 2.1, `grcbal3` makes use of SC *signal operators*. An overview of these and some other, sequential control operators is given in Table 2.2.

To better understand this example’s operation, consider also the execution trace shown in Fig. 2.2c. All SC macros (apart from `TICKSTART` and `TICKEND`) log their operation to `stdout` if instructed to do so via a preprocessor directive. The trace illustrates the operation of `grcbal3` in case input signal `A` is present. The first line shows the input signals (`A`) and the enabled threads (initially none) as bit vector, in octal notation with leading 0. `TICKSTART`, `PAR`, and `PARE` are as explained for the `PCO` example (Sec. 2.1). One difference, however, is that threads `A1`, `A2` and `A3`, which correspond to the three concurrent substates embedded in the macrostate in the `SyncChart` version, are started with priorities 3, 2, and 1, respectively. This priority is used by the dispatcher, which always resumes the active thread with the highest priority; if there are multiple such threads with the same, highest priority, then the highest thread id decides. In `PCO`, all threads had priority 0, hence there only the thread id matters to the dispatcher.

After `Main` has forked its children, `PARE` calls the dispatcher, see line 5 in the program, line 7 in the trace. This starts `A1` (thread id 2), as it has the highest priority. `A1` determines `A` as present and emits signal `B`. The `PRIO` directive lowers `A1`’s priority to 2, specifies `L0` as continuation, and calls the dispatcher. Now `A2` (id 3) is started, as it has the same priority as `A1`, but a higher thread id. `A2` determines `B` as present and hence emits `C`. Then the `TERM` operator *terminates* `C`, meaning that it is deactivated (does not resume in the current tick) and disabled (will not be resumed in the next tick). Therefore `TERM` calls the dispatcher, without specifying a continuation label. The set of remaining enabled threads is encoded in a

bit vector, see line 13 of the trace. The vector octal 027, binary 10111, has bits 0 (rightmost bit, indicating thread `TickEnd`), 1 (`Main`), 2 (`A1`) and 4 (`A3`) set.

In this fashion, control is passed back and forth between `Parent`'s children until they have all have completed their tick, and the `Main` thread, running at priority 0, resumes; see line 23 of the trace. It determines that `T_` is present, which corresponds in the original Esterel program to a thrown exception (exit `T`), hence the program has to terminate. This is done by first aborting `Parent`'s children with `TRANS` (in this case unnecessary, as they have all terminated already), transferring control to label `B`, and then terminating `Main`.

As the trace indicates (line 26), a total of 23 SC instructions have been executed, and solely the always-enabled `TickEnd` thread is still enabled. The trace also shows the signals emitted by the reaction. In this example, the `main` function calling the `tick` function not only sets the inputs (currently read in from an array), but also compares the generated output to a reference output (“`Outputs OK`”). See the complete code in Listing B.5 for how this is done.

One last operator in `grcbal3` not explained yet is the `JOIN` in line 29. Here the `Main` thread checks whether all of its children have terminated. If so, then `Main` also terminates, according to the semantics of `SyncChart` macrostates, and similarly Esterel's concurrency operator `||`.

To summarize, `grcbal3` illustrates how thread ids and priorities can be used to schedule threads in an arbitrary fashion. In this case, we have used this to schedule threads such that signal dependencies, imposed by the Esterel/`SyncCharts` semantics, are adhered to. This semantics requires that within a tick all potential signal emitters run before a signal is tested. This is similar to the situation in the producer-consumer example, just that in `grcbal3` there is not just one buffer to synchronize on, but four output signals.

This example is, admittedly, fairly intricate, as it has also been designed to illustrate the scheduling challenges that Esterel poses to a compiler. For an inexperienced SC programmer it may therefore be non-obvious how to assign priorities and thread ids properly such that signal dependency rules are adhered to; see Sec. 3.2.4 for a full discussion. There are several possible alternatives to unchecked manual priority/id assignment:

- one might relegate thread id and priority assignment to a separate analysis pass, similar to an Esterel compiler (feasible, but it would require a separate tool);
- one might use `SyncCharts`—or Esterel—as entry language for SC, and do the signal dependence analysis there (also possible, but this would lose the direct embedding in C); or,
- one might add run-time checks to the SC operators that ensure that no signals that have been tested already in a tick are emitted in a tick (a reasonable consistency check, easy to implement—but it does not offer a guarantee as a static analysis would do).

However, one should also note that such intricate dependencies appear to be rather rare. We can distinguish three types of programs:

**Dynamically scheduled** programs that require dynamic scheduling of threads, which entails run-time alterations of thread priorities (via `PRIORIO`);

**Statically scheduled** programs that require just static scheduling, which can be handled with thread id assignment; and

**Unscheduled** programs that do not impose scheduling constraints at all.

From the 10 benchmarks currently included in the SC distribution, most provided by André [2], only `grcbal3` and `Exits` belong to the first category.

# Chapter 3

## A Tour of SC

SC consists of a programming model, which is implemented with simulated multi-threading, a set of SC operators, and a convention on how to structure an SC program. These concepts are explained in the next sections.

### 3.1 The SC Programming Model

SC programs follow the *synchronous programming model* established in SyncCharts and other synchronous programming languages. This programming model is characterized by two main concepts explained in the following, the synchronous threading model and signals. The first concept is essential to SC; the second one is also provided by SC, but can be regarded as optional for the SC programmer.

#### 3.1.1 Synchronous threading

The main concept that must be understood to program in SC is that of a *logical tick*, or *logical instant*. An SC program conceptually consists of a number of concurrent threads, whose concurrent execution is grouped (synchronized) by a progression of logical ticks. A tick boundary of a thread is usually denoted by the PAUSE operator, which thus denotes implicit synchronization points among thread; no thread can start the next tick, before all concurrent threads have completed the current tick.

In Statecharts parlance, the tick boundaries (PAUSE operators) collectively reached by the currently active threads form the *configuration* of a system. The system progresses from one stable configuration to the next. The *synchrony hypothesis* states that the computation of a reaction does not consume any time. In other words, the progression from one configuration to the next configuration is considered to not consume physical time; physical time only advances while the system rests in a stable configuration. This synchrony hypothesis is of course an abstraction from reality, where computations of course do consume time, but this abstraction allows a compositional, formally grounded semantics.

#### Program execution states

Figure 3.1 illustrates the execution states of the whole program, using the SyncChart formalism. Upon program start, the main thread is enabled (forked), and the program is considered

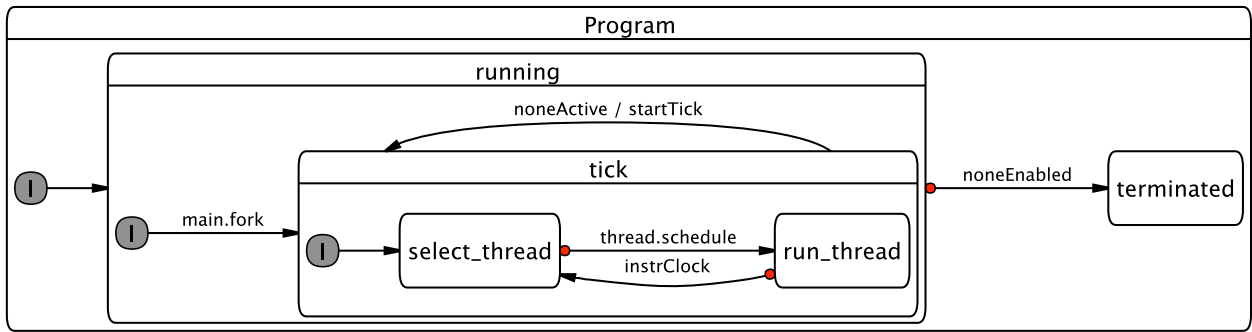


Figure 3.1: The status of the whole program (from [17]).

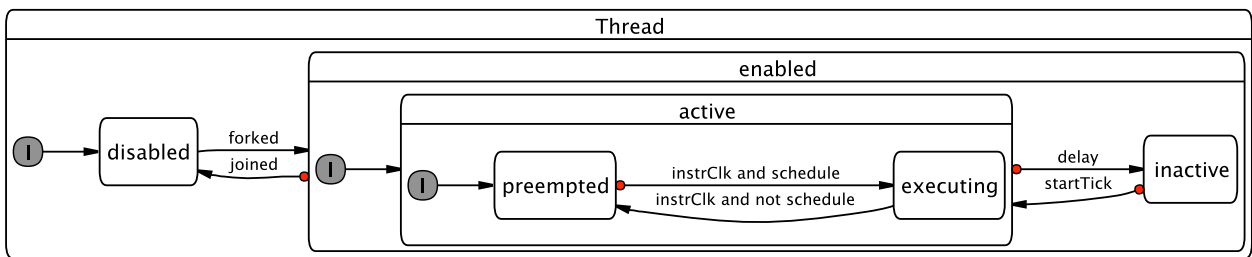


Figure 3.2: Execution status of a single thread (from [17]).

*running*. Entering the tick state corresponds to a call to the tick function, described in Section 3.4. The state is left (the tick function terminates) when no threads are active anymore. If all threads have become disabled (all have terminated), the whole program becomes terminated.

Note that the thread logic and the overall program logic of SC are very close to the Kiel Esterel Processor [17]. A slight difference arises within the tick state: in the KEP, each instruction cycle is started by a thread selection step, whereas in SC, threads run freely until they encounter an SC operator that implies a call to the dispatcher and a possible thread context switch. The operators in question that can thus possibly cause a thread context switch are marked in Table 2.1.

### Thread execution states

The execution status of a thread is illustrated in Figure 3.2. Two flags are needed to describe the status of a thread. One flag indicates whether the thread is *disabled* or *enabled*. Initially, only the main thread is enabled. Other threads become enabled whenever they are forked (with PAR, see Section 3.3.1), and become disabled again when they terminate themselves (TERM) or get aborted by a transition that leaves the parent state (using TRANS). The other flag indicates whether the thread should still be scheduled within the current logical tick (the thread is *active*) or not (*inactive*).

A thread is *active* if it still has work to do in the current tick, otherwise it is *inactive*. The order of execution among active threads is statically determined by a *thread id* and a *thread*

*priority*. The *thread dispatcher* starts/resumes the active thread with the highest priority, this thread then becomes *executing*. Threads that are active but not executing are considered *preempted*.

Priorities can be shared among threads; if there are multiple threads with the same, highest priority, the thread with the highest id wins. To ensure that there is a unique thread to be chosen, the thread ids must not be shared among threads that can be active concurrently. SC requires each thread to be given a thread id and initial priority upon its creation (with `PAR`). SC also provides an operator to change the priority of a running thread (`PRIO`). With these mechanisms, the programmer can enforce arbitrary, deterministic thread schedules. For simplicity, these schedules are usually determined statically; however, in terms of expressiveness of the SC operators, it would also be possible to create dynamic schedules.

### 3.1.2 Signals

Another concept that is characteristic of synchronous languages is that of *signals*, which can be used for broadcast communication among threads. SC programmers do not have to use signals, they might achieve the same effect with the appropriate use of standard C variables. However, the explicit use of signals for thread control, for example to trigger preemptions, might help to clarify the interaction and synchronization patterns across thread boundaries.

Signals are *absent* per default. They become *present* for the current tick if a thread *emits* the signal in this tick. Any thread (including the emitting thread) can test for the presence of a signal and can change control flow accordingly, including not only conditional branches but also various forms of preemption.

SC provides a full range of signal handling operators, including local, valued, and combined signals, and tests for signal presence across tick boundaries (the `PRE` operator). SC assumes that signals become visible within the tick they are emitted, and also, unlike some other approaches [7], allows to test for signal absence in the current tick, not just in the next tick.

A word of caution: it is a common assumption of (strictly) synchronous programs that signals have a unique, well-defined presence/absence status for the duration of a tick. This effectively means that we must not test for the presence of a signal if it may still be emitted within that tick; see also Sec. 3.2.4. In other words, all writes must be performed before any reads are done. Compilers dedicated to synchronous languages perform a static *signal dependency analysis* of the program and try to compute a—usually static—schedule that orders threads (or thread segments) accordingly. A compiler rejects the program if it cannot find such a schedule. As mentioned in Sec. 2.2, one could envisage an analysis tool that performs a signal dependency analysis on an SC program and checks that the encoded schedule respects all signal dependencies. In the presence of arbitrary C control flow, this analysis would have to be conservative. If we were to use SC as intermediate language for synthesizing code from a visual SyncCharts model, it would be the responsibility of the code synthesis tool to perform a dependency analysis on the model and to schedule threads accordingly. Using plain SC, as presented here, we do not perform such a compilation or analysis; we just use a regular C compiler and C does not have a concept of signal dependencies and consistency. It is thus the responsibility of the programmer to schedule threads accordingly, using thread ids and priorities. On the other hand, this also provide the options to weaken the requirement of strict synchrony, and to program with a signal model that corresponds *e. g.* to the original Statecharts semantics. Note also that the program will in any case be deterministic, there are no race conditions that can produce different outputs for the same inputs.

## 3.2 Multithreading Simulation

To simulate multi-threading, we must be able to keep track of the locus of control of each thread, and we need a dispatcher that performs the context switches.

### 3.2.1 Coarse program counters

In a VM or hardware implementation of the SC operators, one could have direct access to a program counter that denotes the locus of control. As we are working here at the C level, we do not have that option. Instead, we annotate the C program with regular C labels, at all possible thread continuation points. In a way, these denote thread-level “basic blocks,” but unlike traditional basic blocks, they do not denote sequences of straight line code, but instead they delineate sequences of code in which no thread context switch can happen.

Using gcc’s computed goto extension, we can store these program labels in an ordinary C array. In SC, this array of *coarse program counters* is `pc[idMax]`, declared in `sc.h` (see Section 3.4.1).

Whenever a thread calls an SC operator that might result in a context switch to another thread, we must save a *continuation point* for the thread in its program counter. In our implementation, this operation is folded into the SC operators, so that it suffices to pass the continuation point of the thread along as argument to the SC operator, which then performs the book keeping.

Again, if we would implement the SC operators in a VM or as reactive processing ISA, we could do away with that label parameter; indeed, the KEP ISA does not have such a label parameter. However, passing this continuation label explicitly also gives some additional freedom, *i. e.*, optimization opportunities, as the passed label does not necessarily have to point to the instruction immediately following the operator. This can often be used to save a jump instruction. For example, the Esterel `halt` instruction can be implemented as simply `l: PAUSE(l)`, with some unique label `l`.

### 3.2.2 The dispatcher

As explained in Section 3.1.1, the dispatcher starts/resumes the active thread with the highest priority; if there are multiple active threads with the same, highest priority, the thread with the highest id wins.

As the dispatcher may be called rather frequently—namely, whenever we perform an SC operator that can result in a context switch, see also Table 2.1—we should strive for an efficient implementation of the dispatcher. The dispatcher should be as general as necessary and as fast as possible. As the demands of SC programs on the dispatcher may vary—in particular, they may or may not use priorities—SC provides different dispatchers, which can be selected by the application, via a `#define USEPRIO` C preprocessor directive.

The dispatcher consists of two parts:

1. Computation of the current (to be dispatched) thread id, `cid`.
2. A jump to the corresponding program counter, stored in `pc[cid]`.



Listing 3.1: `selectCidPrio()`: Computation of id of thread to be dispatched, considering priorities (from `sc.c`)

```

1 // For enabled threads with highest prio, highest id "wins"
2 void selectCidPrio () {
3     int id;
4     int cprio = -1;
5
6     for (id = idHi; id >= 0; id--) {
7         if (isActive(id) && (pr[id] > cprio)) {
8             cid = id;
9             cprio = pr[id];
10        }
11    }

```

The most general version for the computation of `cid` is implemented in `selectCidPrio`, see Listing 3.1. A loop iterates through thread ids, starting from the highest id (given by `idHi`, see Section 3.4.4) downwards to id 0. In the loop body, we check whether the currently examined thread id is active, and if so, whether it has a higher priority than the highest priority encountered so far. The run time of this implementation is linear in the number of thread ids in use.

Listing 3.2: `selectCidNoprio()`: Computation of id of thread to be dispatched, without considering priorities (from `sc.c`)

```

1 // Which is actually faster depends on application
2 void selectCidNoprio () {
3     int act;
4
5     act = active;
6     for (cid = 0; act != 0; act >>= 1)
7         cid++;

```

As it turns out, many SC programs do not require the usage of priorities for proper thread scheduling, that is, all priorities can be 0; if there are any scheduling constraints, they are simple enough to be resolved via thread ids alone. In this case, the computation of `cid` can be simplified to the implementation in `selectCidNoprio`, see Listing 3.2. This function only considers whether a thread is active or not. In the current SC implementation, this information is stored in a bit vector `active`. Hence it suffices to set `cid` to the position of the highest set bit in `active`. The implementation of `selectCidNoprio` uses the obvious algorithm, with a run time linear in the position of the highest bit. Note that there are also alternatives that run logarithmic to bit vector size<sup>1</sup>. Which algorithm is actually faster depends on the application.

Listing 3.3: `dispatch()`: Variable definitions for the dispatcher (from `sc.h`)

```

1 #ifndef USEPRIO
2 // Version 1: for arbitrary priorities
3 #define dispatch() selectCidPrio (); \
4     goto *pc[cid]
5
6 #elif ((defined __i386__ || defined __amd64__ || defined __x86_64__) && defined __GNUC__)
7 // Version 2a: all priorities = 0, x86 + gcc available
8 // Use fast Bit Scan Reverse assembler instruction
9 #define dispatch() \
10     __asm volatile (" bsrl %1,%0\n" \
11                    : "=r" (cid) \
12                    : "c" (active) \
13                    ); \

```

<sup>1</sup>See eg <http://graphics.stanford.edu/~seander/bithacks.html#IntegerLog>

```

14 |     goto *pc[cid]
15 |
16 | #else
17 | // Version 2b: all priorities = 0, x86 + gcc not available
18 | #define dispatch() selectCidNoprio(); \
19 |     goto *pc[cid]
20 | #endif

```

Fortunately, many processor instruction sets provide an assembler instruction that does exactly this, to detect the index of the highest set bit. The x86 does this with the Bit Scan Reverse (BSR) instruction. Using the gcc assembler escape, we can embed this instruction into C and thus obtain an even faster dispatcher. This consists of just a couple of instructions and has constant run time. This dispatcher variant is not implemented as a separate function, but instead as a macro, to be expanded/inlined by the preprocessor. This does not unduly increase the code size, and saves the function call overhead at run time. It also alleviates the need to link against `sc.c` that defines the alternative dispatcher functions (see Section 3.4.1). Listing 3.3 shows how the dispatcher is defined.

Note that the current implementation of SC, based on bit vectors implemented as simple integers, assumes that the number of concurrent threads does not exceed the word size. The same limitation applies to signals, whose presence/absence status is also implemented as integer-based bit vectors. Neither limitation has posed any problems in the applications considered so far. However, it should be rather straightforward to lift either limitation, and to use bit vectors of arbitrary size or some other unrestricted data structure.

### 3.2.3 Thread and label structuring

To illustrate how the thread structure is derived from a SyncChart, consider the ABRO example in Fig. 3.3 [2, Fig. 5-12]. ABRO is arguably the “hello-world” program of synchronous programming and has the following behavior: Two concurrent threads wait for signals A and B; once these have occurred, in any order, output O is emitted. If R is present, the behavior is reset. As the transition triggered by R is a *strong abort*, the transition takes priority over the internal behavior of ABO: if R is present in a tick, ABO does not get to execute in that tick.

A note on the SC implementation: as there is normal termination leaving ABO, there is no need for a JOIN on thread AB.

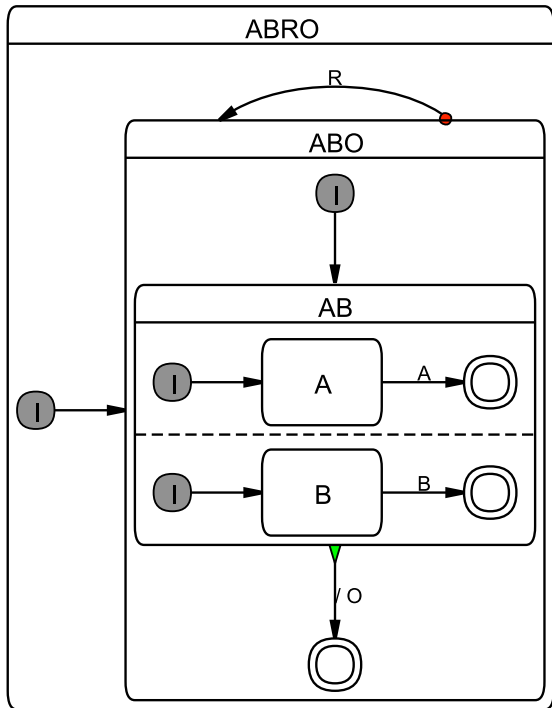
#### Thread structure

Each macrostate of degree of concurrency  $n$  has  $n$  *embedded* threads. In ABRO:

- ABO has one embedded thread (AB), and
- AB has two embedded threads (WaitA and WaitB).

#### Naming threads and their initial label

- TickEnd is the thread returning from the tick function, at macro TICKEND. It must have id 0, as it has to run after all other threads.
- Main is the main thread, activated in the initial tick upon entering the tick function.



(a) SyncChart

```

1 // Thread ids: AB=1, WaitB=2, WaitA=3, Main=4
2   TICKSTART(islnit);
3
4 ABO:  PAR(0, AB, ids[AB]);
5       PARE(0, ABOmain, id2b(AB) | id2b(WaitA) | id2b(WaitB));
6
7 AB:   PAR(0, WaitA, ids[WaitA]);
8       PAR(0, WaitB, ids[WaitB]);
9       PARE(0, ABmain, id2b(WaitA) | id2b(WaitB));
10
11 WaitA: PAUSE(L0);
12 L0:    PRESENT(A, WaitA);
13        TERM;
14
15 WaitB: PAUSE(L1);
16 L1:    PRESENT(B, WaitB);
17        TERM;
18
19 ABmain: JOIN(Done, ABmain);
20 Done:  EMIT(O);
21        TERM;
22
23 ABOmain: PAUSE(L2);
24 L2:    PRESENT(R, ABOmain);
25        TRANS(ABO);
26
27        TICKEND;

```

(b) SC tick function

Figure 3.3: The ABRO example.

- Other threads are named after their first state.
- The initial label of a thread is named after the thread.

Note that the last two rules are in most cases redundant. However, it can be the case that a thread does not commence directly at its first state, in particular if the initial transition has to perform some action; see for example the initial transition of thread *S1* in *PrimeFactor* (Fig. 4.8). In such cases, the recommended convention is:

- The thread and its entry point should still be named after its first state *S*, according to the SyncChart diagram.
- However, the entry point of the state should be renamed to *Ssurf*.

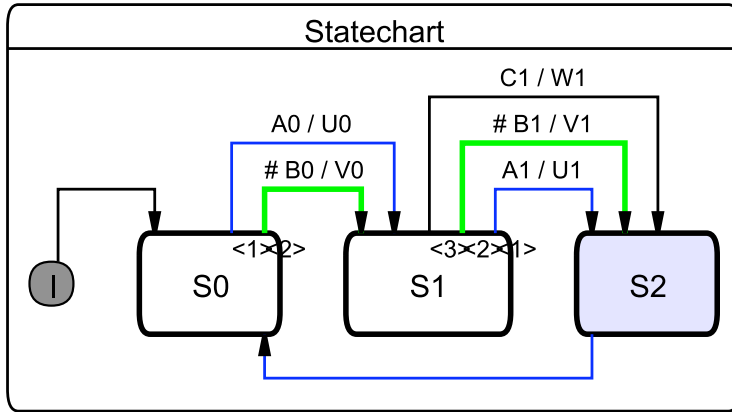
The latter part is derived from the surface/depth distinction, elaborated on in the following.

### Surface vs. depth

In SyncCharts, as well as in Esterel, one distinguishes

**Immediate transitions** which can potentially be taken in the same tick as their source state is entered, and

**Delayed transitions** which will only become enabled from the next tick onwards.



(a) SyncChart, during simulation—also illustrating KIEL’s horizontal/vertical layout.

```

1 // Thread ids: Main=1
2   TICKSTART(islnit);
3
4   GOTO(S0surf);
5
6 S0depth:PRESENT(A0, S0surf);
7   EMIT(U0);
8   GOTO(S1surf);
9 S0surf: PRESENT(B0, L0);
10  EMIT(V0);
11  GOTO(S1surf);
12 L0:  PAUSE(S0depth);
13
14 S1surf: PRESENT(B1, L4);
15  GOTO(L2);
16 S1depth:PRESENT(A1, L1);
17  EMIT(U1);
18  GOTO(S2);
19 L1:  PRESENT(B1, L3);
20 L2:  EMIT(V1);
21  GOTO(S2);
22 L3:  PRESENT(C1, L4);
23  EMIT(W1);
24  GOTO(S2);
25 L4:  PAUSE(S1depth);
26
27 S2:  PAUSE(S0surf);
28
29  TICKEND;

```

(b) SC tick function

Figure 3.4: The SurfDepth example.

Transitions are by default delayed; immediate transition triggers are indicated by a # mark (see also Fig. 2.2a).

One also distinguishes the

**Surface** of a statement, which is what is executed in the initial tick and which includes only the immediate transitions, and the

**Depth** of a statement, which is where execution commences in subsequent ticks and which includes immediate as well as delayed transitions.

Consider for example in PrimeFactor (Fig. 4.8b) state S1, which has an immediate transition triggered by B, and a delayed transition triggered by A. The former is tested in the surface of S1, at label S1surf, as well as later at the depth (which is commenced at label S1depth), whereas the latter transition is only tested at the depth.

It appears that in most cases code can be structured such that there is no need for code duplication between surface and depth. However, this cannot always be avoided. Consider the SurfDepth example in Fig. 3.4. The transitions from S0 to S1 can be ordered such that the transition-number priority can be honored (test A0 before B0) as well as the immediate (B0)/non-immediate (A0) distinction. However, this is not possible with the transitions from S1 to S2. The only immediate transition, triggered by B1, has a transition number between

the two other, delayed transitions. Hence we must duplicate the test for **B1**, once at the surface label **S1surf**, once in the depth code starting at **S1depth**.

Note that this duplication concerns not only the test of the transition trigger, but also possible transition actions. In **SurfDepth** this applies to the emission of **V1**. We here chose to minimize code size and to follow the write-things-once principle as much as possible, by sharing one **EMIT(V1)<sub>20</sub>** statement between the transition tests at lines 14 and 19, with the **GOTO(L2)<sub>15</sub>** statement. An alternative would be to have another **EMIT(V1)** directly after the test at line 14, followed by a **GOTO(S2)**. This increases the program size by one statement (the **EMIT(V1)**), but saves one statement at run time (the **GOTO(S2)**).

### Label naming

- The (surface) entry point of a macrostate  $S$  gets label  $S$ , or exceptionally (see above) label  $Ssurf$ .

*Example:* label **ABO**.

*Example:* in **PrimeFactor** (Fig. 4.8), label **S1surf**.

- The depth entry point of a state  $S$  gets label  $Sdepth$ .

*Example:* in **PrimeFactor** (Fig. 4.8), label **S1depth**.

- For macrostate  $S$ , the entry point of the code that checks transitions attached to  $S$  gets label  $Smain$ .

*Examples:* **ABmain**, **ABOmain**.

### 3.2.4 Thread scheduling

As SC is embedded into plain C, SC programs are deterministic, and there is no need for classical synchronization among threads using semaphores or similar concepts. However, as mentioned in Sec. 3.1.2, certain *scheduling rules* must be followed when encoding a specific SyncChart in SC, to adhere to the original, synchronous semantics.

This is related to the scheduling problem that a compiler for synchronous languages faces, and one might use similar concepts to address this. For example, one might transform a SyncChart into something like a CKAG (Concurrent KEP Assembler Graph [17]) that expresses scheduling constraints, and then transcribe this into an SC program. As we here—so far—assume a human programmer that writes an SC program, we do not describe the scheduling task in algorithmic terms, but instead give precedence constraints that must be fulfilled. As stated in Sec. 2.2, this is a non-trivial problem in the general case; however, it appears that most SC programs exhibit relatively few scheduling constraints.

### Precedence of operations

Let  $Op_1$  and  $Op_2$  be two operations that must be performed in an SC program. For example, in **ABRO**, let  $Op_1$  be the test for the presence of signal **R** of the **Main** thread in line 24, abbreviated as  $Op_1 = \mathbf{Main}_{24, \text{PRESENT}(\mathbf{R})}$ , and let  $Op_2 = \mathbf{WaitA}_{12, \text{PRESENT}(\mathbf{A})}$ . In this case,  $Op_1$ , which corresponds to the strong abort transition on **ABO**, must be executed before  $Op_2$ , which is a transition nested within **ABO**. We say that  $Op_1$  *has precedence* over  $Op_2$ , and write this as  $Op_1 \succ Op_2$ , in this example  $\mathbf{Main}_{24, \text{PRESENT}(\mathbf{R})} \succ \mathbf{WaitA}_{12, \text{PRESENT}(\mathbf{A})}$ .

In the following, we will use the terms statements and operations interchangeably. A note on notation: we may use the line-number-in-subscript notation throughout the report to refer to specific statements in a program. We may also abbreviate statements, for example by omitting label arguments.

We are now ready to define the precedence constraints imposed by a SyncChart. It is  $Op_1 \succ Op_2$  if

1.  $Op_1$  and  $Op_2$  can be executed in the same tick, and
2. one of the following conditions holds:

**Outer-inner precedence**  $Op_1$  tests the trigger of a strong abort or suspension associated with a state  $S$ , and  $Op_2$  belongs to a descendant (inner state) of  $S$ .

*Example:* in ABRO,  $\text{Main}_{24,\text{PRESENT}(\text{R})} \succ \text{WaitA}_{12,\text{PRESENT}(\text{A})}$ .

**Inner-outer precedence**  $Op_1$  belongs to a descendant (inner state) of  $S$ , and  $Op_2$  tests for normal termination or tests the trigger of a weak abort associated with a state  $S$

*Example:* in ABRO,  $\text{WaitA}_{12,\text{PRESENT}(\text{A})} \succ \text{AB}_{19,\text{JOIN}}$ .

**Transition-number precedence**  $Op_1$  and  $Op_2$  are associated with transitions that are associated with the same state, and the transition  $Op_1$  is associated with has a higher priority than  $Op_2$ . Here, we refer to transition priorities indicated in the SyncChart with numbers (*increasing* number for *decreasing* priority).

Note that SyncCharts already impose some ordering on the transition numbers within the transitions associated with the same state. Highest priority (lowest number) have strong aborts, followed by suspension, followed by weak aborts, followed by normal termination.

*Example:* in Exits (Fig. 4.2), considering the transitions associated with state M10, the strong abort has precedence over normal termination, *i. e.*,  $\text{M10}_{27,\text{PRESENT}(\text{A})} \succ \text{M10}_{20,\text{JPPAUSE}}$ ; note that the JOIN of the normal termination is folded in with PRIO and PAUSE.

**Write-read precedence**  $Op_1$  writes (emits) a signal, which is read (tested for presence) by  $Op_2$ .

Here, with “signal” we refer to general shared variables for which writer-reader precedence should be respected within a tick. This applies to signals in the sense of SyncCharts or Esterel, operated on via the SC signal operators (Table 2.2), but also to shared C variables, such as the buffer BUF in the PCO example (Fig. 2.1).

*Example:* in grcbal3 (Fig. 2.2b),  $\text{A1}_{8,\text{EMIT}(\text{B})} \succ \text{A2}_{19,\text{PRESENT}(\text{B})}$ .

*Example:* in PCO (Fig. 2.1c),  $\text{Prod}_{33,\text{BUF}=1} \succ \text{Cons}_{39,\text{tmp}=\text{BUF}}$ .

We summarily refer to the first three types of precedence constraints as *structural constraints*, whereas the last one is a *signal constraint*.

## Fulfillment of precedence constraints

We also classify a precedence constraint  $Op_1 \succ Op_2$  as follows:

**Intra-thread precedence**  $Op_1$  and  $Op_2$  belong to the same thread.

In this case, the constraint must be fulfilled via the sequential ordering of the operations within a thread.

**Inter-thread precedence**  $Op_1$  and  $Op_2$  belong to concurrent threads.

In this case, the constraint must be fulfilled via an appropriate assignment of static thread ids and, if necessary, dynamic priorities.

## Thread precedence

We can lift the notion of precedence from individual operations to the threads that they belong to. For an operation  $Op$ , let  $thrd(Op)$  be the thread associated with  $Op$ . For example, in ABRO, it is  $thrd(\text{WaitA}_{12, \text{PRESENT}(A)}) = \text{WaitA}$ . Then, for  $Op_1, Op_2$  with  $Op_1 \succ Op_2$  and  $t_1 = thrd(Op_1), t_2 = thrd(Op_2)$ , this implies  $t_1 \succ t_2$ . In ABRO,  $\text{Main}_{24, \text{PRESENT}(R)} \succ \text{WaitA}_{12, \text{PRESENT}(A)}$  implies  $\text{Main} \succ \text{WaitA}$ . In other words,  $\text{Main}$  should be scheduled before  $\text{WaitA}$ .

In some cases it is convenient to use a mixed notation that orders an individual operation with another thread. For example,  $\text{Main}_{24, \text{PRESENT}(R)} \succ \text{WaitA}$  expresses that the presence test on R must run before thread  $\text{WaitA}$ .

## Static vs. dynamic scheduling

For an SC program  $P$  derived from a SyncChart and a pair of operations in  $P$ , the SyncChart either specifies a fixed order in which the operations must be performed, or it does not specify an order at all. All precedence constraints on individual operations are static. In other words,  $\succ$  is a partial order with respect to individual operations in  $P$ .

However, at the thread level, it may be the case that for a pair of threads  $T_1$  and  $T_2$  in  $P$  there exist operations in  $T_1$  and  $T_2$  that induce  $T_1 \succ T_2$ , and simultaneously other operations that induce  $T_2 \succ T_1$ . In other words,  $\succ$  is not necessarily a partial order with respect to threads in  $P$ .

If  $\succ$  is a partial order in  $P$  at the thread level, then it is possible to schedule all threads statically by just assigning them thread ids that respect  $\succ$ . There is no need for dynamic priorities, all thread priorities can remain at 0. As noted in Sec. 2.2, it appears that most programs belong to this category of statically schedulable programs.

If  $\succ$  is not a partial order in  $P$  at the thread level, then one should still assign thread ids in a way that static precedences between threads are met; however, one must use positive thread priorities as well to resolve the remaining dynamic precedences.

Furthermore, immediate transitions must be properly distinguished from delayed transitions—see also Sec. 3.2.3.

In the following, we will illustrate how precedence constraints are met in SC programs with the examples introduced so far, ABRO, PCO, and `grcbal3`. Chapter 4 provides further examples.

## Precedence constraints in ABRO

In ABRO, there are the following structural inter-thread constraints at the thread level.

1. Strong abortion on ABO (*outer-inner*):  
Main  $\succ$  AB, Main  $\succ$  WaitA, Main  $\succ$  WaitB
2. Normal termination on AB (*inner-outer*):  
WaitA  $\succ$  AB, WaitB  $\succ$  AB

This thread precedence relation induces a partial order, which can be fulfilled with the following thread id assignment: AB = 1, WaitB = 2, WaitA = 3, Main = 4. We generally omit stating explicitly the id of TickEnd, since it always must be 0; however, the program must still declare the TickEnd thread and assign id 0 to it.

Note that the order between WaitB and WaitA could as well have been reversed. Here, WaitA has been given the higher priority such that the order of execution between WaitA and WaitB is consistent with the order in which they appear in the program, as an aid in helping to understand the execution trace. Apart from this small consideration for the human observer, it makes no difference in which order threads are executed that do not have a precedence constraint between them. The order in which the threads appear in the program has no semantic relevance.

## Precedence constraints in PCO

In PCO (Fig. 2.1c, p. 4), there are the following inter-thread constraints at the thread level.

1. Weak abortions on Parent (*inner-outer*):  
Prod  $\succ$  Main, Cons  $\succ$  Main, Obs  $\succ$  Main
2. Writer on BUF before reader on BUF (*write-read*):  
Prod  $\succ$  Cons, Prod  $\succ$  Obs

Note that the first constraint is again a structural constraint, but the second is a signal constraint. Again, this precedence relation induces a partial order at the thread level, which is observed by the following thread id assignment: Main = 1, Cons = 2, Obs = 3, Prod = 4.

## Precedence constraints in grcbal3

As pointed out in Sec. 2.2, grcbal3 (Fig. 2.2) is a relatively complex example that requires dynamic scheduling. In other words,  $\succ$  is not a partial order at the thread level. We will therefore use the mixed operation/thread notation to capture the constraints as concisely as possible while still permitting an ordering, without contradictions.

1. Weak abortion and normal termination on macrostate (*inner-outer*):  
A1  $\succ$  Main, A2  $\succ$  Main, A3  $\succ$  Main

This inter-thread structural constraint is met by assigning Main the thread id 1 (the lowest possible, apart from the TickEnd thread) and priority 0.



2. Precedence of weak abortion over normal termination (*transition-number*):

$\text{Main}_{27, \text{PRESENT}(T\_)} \succ \text{Main}_{29, \text{JOIN}}$

This intra-thread structural constraint is met by ordering the operations in the program accordingly.

3. Communication via signal B (*write-read*):

$\text{A1}_{8, \text{EMIT}(B)} \succ \text{A2}_{19, \text{PRESENT}(B)}$

This constraint is met by executing the first operation at priority 3, as induced by the  $\text{Main}_{2, \text{PAR}(3, \text{A1}, \text{id}_s[\text{A1}]}$  statement, and the second at priority 2.

4. Communication via signal C (*write-read*):

$\text{A2}_{20, \text{EMIT}(C)} \succ \text{A1}_{10, \text{PRESENT}(C)}$

This constraint is met by executing both operations at priority 2, as induced by  $\text{A1}_{9, \text{PRIO}(2)}$  and  $\text{Main}_{3, \text{PAR}(2, \text{A2}, \text{id}_s[\text{A2}]}$ , and assigning A2 a higher thread id than A1.

5. Communication via signal D (*write-read*):

$\text{A1}_{11, \text{EMIT}(D)} \succ \text{A3}_{23, \text{PRESENT}(D)}$

This constraint is met by executing the first operation at priority 2 and the second at priority 1.

6. Communication via signal E (*write-read*):

$\text{A3}_{24, \text{EMIT}(E)} \succ \text{A1}_{13, \text{PRESENT}(E)}$

This constraint is met by executing both operations at priority 1 and assigning A3 a higher thread id than A1.

### 3.3 SC Operators

There are three classes of SC operators: *SC Thread Handling Operators*, *SC Signal Operators*, and *SC Sequential Control Operators*.

#### 3.3.1 SC Thread Handling Operators

An overview of the SC Thread Handling Operators, which perform the multi-threading simulation and form the core of SC, is given in Table 2.1, p. 5.

##### Tick start and end

TICKSTART and TICKEND do some book keeping. For example, in the initial tick, TICKSTART initializes the TickEnd thread (see Section 3.4.3) and activates the Main thread; in subsequent ticks, TICKSTART activates the enabled threads.

TICKEND determines whether there are still any enabled threads, apart from the never disabled TickEnd thread.

## Pausing, suspending, aborting and terminating a thread

PAUSE pauses the currently active thread. This entails setting the program counter of the current thread to the label provided as argument, to deactivate the current thread, and to call the dispatcher.

SUSPEND suspends (“freezes”, “steals the clock from”) the current thread and its descendants for the current tick and calls the dispatcher. For an example, see `Count2Suspend`, Fig. 4.1, p. 35.

Differences between PAUSE and SUSPEND:

- PAUSE deactivates just the current thread, whereas SUSPEND also deactivates its descendants. The latter exploits that the PCs of the descendants must reside at tick boundaries, *i. e.*, there is nothing more to do for the descendants in the current tick.
- Unlike PAUSE, SUSPEND must do some signal handling in case local signals and `pre` are used, as explained in Section 3.3.2.

TERM terminates the current thread by disabling it.

TRANS performs an abortion of the current thread and its descendants, by simply disabling them, and transfers control to the specified label  $l$ . In SyncCharts, this corresponds to a (weak or strong abort) transition from the current state to some other state.

Whether TRANS corresponds to a weak or strong abort is merely a question of whether TRANS is executed *before* the descendant threads have computed the tick (*strong abort*) or *after* the descendants have run (*weak abort*). See also the outer-inner vs. inner-outer precedences discussed in Sec. 3.2.4. Again, this is an implication of the SyncChart semantics and can be viewed as a (reasonable) convention. Nothing would prevent a programmer to break with this convention and schedule a TRANS arbitrarily, in the middle of the execution of the descendant. This would not break determinism (we still have a sequential C program), but it would probably make the flow of the program more difficult to comprehend.

To summarize, a thread voluntarily relinquishes control for the remainder of the tick via PAUSE, SUSPEND, or TERM. A thread may also be aborted when a (transitive) parent performs a TRANS.

## Fork and join

A sequence of PAR statements, followed by a PARE statement, together form a *fork*. Each PAR creates a child thread by initializing its program counter and its priority and enabling it. PARE then registers the descendant threads with the current (parent) thread and calls the dispatcher. The parent thread must know about its descendant threads to detect their termination, and also possibly to terminate them in case the parent is aborted. The set of descendants includes the newly created child threads, and, in case these will possibly fork threads as well, their descendants (transitively) as well. Note that the latter is necessary for abortions, but not for normal termination, as normal termination should respect the hierarchical ordering (grandchildren should terminate before children terminate normally).

JOIN performs the corresponding *join* operation, which checks whether all descendant threads have terminated normally. If they have terminated, control transfers immediately to the  $l_{then}$  label. This corresponds to a *normal termination transition* in SyncCharts. By (reasonable) convention, normal termination transitions have the lowest priority, and there

can be only one such normal termination transition. See also the notes on transition-number precedence, Sec. 3.2.4. This means that after performing an unsuccessful JOIN, there is nothing else the current thread has to do for the current tick, and it pauses. We exploit this by folding the PAUSE into the else-branch of the JOIN. That is, if the descendant threads have not terminated, we execute a `PAUSE(lelse)`.

Notes:

- One might also decide to break with the SyncChart convention of pausing after an unsuccessful join, and to supply an SC JOIN variant that does not automatically pause. This would be trivial to implement, but so far there has no need arisen to do so.
- A JOIN is only required if the corresponding SyncChart does have a normal termination transition. If the parent thread never terminates, or if it is only terminated through abortion (via TRANS), no JOIN is required.
- Normally, a fork spawns off *child threads* of the current thread, and the current thread keeps executing, at the label specified by PARE. However, we may also construct a fork without PARE, which just consists of a sequence of PAR statements that effectively create *sibling threads* of the current thread. Here the current thread simply keeps executing after the PAR statements. Since there is no PARE, the dispatcher will not be called, so the current thread should be the one with the highest priority/thread id of the sibling threads. See `Shifter3` (Fig. 4.5) for an example.

### Thread priority handling

The initial priority of a thread is assigned upon creation of the thread, as argument to PAR. It may be necessary to change the priority of a thread later at run time. This is done with the PRIO operator. Note that within a tick, it is only meaningful to lower the priority of a thread, not to raise it, since if a thread is already executing, there is no effect when raising its priority [17]. We can use priority lowering to yield to other threads. However, as thread priorities are preserved across tick boundaries, we may want to raise a priority at the end of a tick, to start the next tick with a higher priority.

The PRIO operator entails a call to the dispatcher, as now another active thread might be the one with the highest priority, or at the same priority but with a higher thread id. However, there are common situations where the next operator to be executed by the thread that has just called PRIO is another operator that necessitates the dispatcher. For example, in the aforementioned scenario where we raise a priority to start the next tick with a higher priority, PRIO is followed immediately by PAUSE. In this case, the first call to the dispatcher is superfluous. Therefore, there are two *combined operators*, PPAUSE and JPPAUSE, that combine PRIO with other operators. These are not just syntactic sugar, but optimize performance, and code size. For example, JPPAUSE combines PRIO with a JOIN and a PAUSE, thus reducing three potential calls to the dispatcher to just one call.

### 3.3.2 SC signal operators

If an SC program wants to use signals (see Section 3.1.2), it can use the operators shown in Table 2.2. Signals must be declared in the `signaltype` (see Section 3.4.3).

## Global vs. local signals, reincarnation

We can classify signals as follows:

**Global signals** Signals get initialized once at the beginning of each tick. This is the default in SC.

**Local signals** Signals can be declared for the scope of a SyncChart macrostate. This implies that signals are initialized whenever the macrostate is entered. This is achieved with the `SIGNAL` operator, see below.

An interesting aspect of local signals is the possibility of *reincarnation*, or *schizophrenia*. A loop around the macrostate declaring a local signal may provoke the simultaneous existence of two different “incarnations” of the local signal [2]. This is illustrated in the **Reincarnation** example (see Figure 4.7).

## Pure signals

As explained in Section 3.1.2, signals can be *present* or *absent*. *Pure signals* just have this presence status, unlike valued signals, which also carry a value (see next section).

The `SIGNAL` operator initializes a local signal, as explained above, by setting its status to absent.

The `EMIT` operator sets a signal present.

The `PRESENT` operator checks for the presence of a signal. If it is, control proceeds normally to the next statement (*then branch*), otherwise it jumps to the specified label  $l_{else}$  (*else branch*).

## Valued signals

Valued signals carry a value of a certain type. So far, SC implements just integer valued signals, extensions to other types (or a more generic typing mechanism) would be straightforward. The `EMITINT` operator emits a signal  $S$  (makes it present) and assigns it a value  $val$ .

If an application uses valued integer signals, the signal declaration in `signaltype` (see Section 3.4.3) has to order the valued signals before the pure signals. The number of valued signals, say  $n$ , must be declared with a “`#define valSigIntCnt n`” directive.

The `VAL` operator retrieves the value of  $S$  and stores it in a register (an ordinary C variable). In SyncCharts/Esterel, this is done with the `?S` notation. It would also have been straightforward to implement `VAL` as a function that returns the value directly, which might seem a bit more natural from the C perspective. However, to stay in the spirit of operators that could also be used for an ISA, `VAL` requests an explicit “destination register.”

**Combine functions** Adhering to the synchronous, deterministic SyncCharts semantics, signals have a unique presence/absence status throughout a tick. This is no problem for pure signals, in so far as the execution of multiple `EMIT` statements within one tick has no further effect, we just set an already present statement to present again. For valued signals, the situation is slightly more complicated, as valued signals are considered to carry a unique value throughout a tick as well. This at first sight conflicts with the possibility of executing multiple valued emissions within one tick, as these valued emissions might occur with different values.

But SyncCharts (as Esterel) offers an elegant way out of this dilemma, by way of *combine functions*. These functions must be binary, commutative, associative functions that can be used to combine multiple values into one uniquely determined value. For example, we may use addition or *max* as combine functions. Subtraction would not be allowed, as it is not associative, and we cannot, in general, make any assumptions on the order in which values are supplied to the combine function.

So far, SC implements multiplication as combine function. `EMITINTMUL` emits an integer signal, combined with multiplication. Again, it would be straightforward to extend this to other combine functions, or to implement a generic mechanism.

As mentioned above in the context of signal reincarnation, it is possible that statements are executed multiple times within a macro tick. This can lead to interesting—but still explainable and deterministic—behavior when using combined valued signals, as illustrated in the `PrimeFactor` example (see Figure 4.8).

### Crossing tick boundaries (PRE)

In general, we are interested in the presence status (and perhaps value) of a signal for the current tick. However, to implement delays, or sometimes to break “dependency cycles,” we may want to access the status/value of a signal in the previous tick. This functionality is provided in SyncCharts/Esterel with the `pre` operator, and SC provides this functionality as well.

If this functionality is used, SC has to do some further book keeping, and this has to be indicated in the application with a `#define usePRE` directive.

`PRESENTPRE` is like `present`, but refers to the presence status of *S* not in the current tick, but in the previous tick.

`VALPRE` is like `VAL`, but again refers to the previous tick.

**Pre, suspend, and local signals** There is an interesting interaction between `pre`, suspension, and local signal declaration. Recall that suspension “steals the clock” from a thread (Section 3.3.1). If a thread has declared local signals and wants to access their status in the previous tick (via `PRESENTPRE` or `VALPRE`), “stealing the clock” from a the thread means that in the next tick when the thread is not suspended any more, the “previous tick” refers to the previous tick in which the thread was not suspended yet. See the `PreAndSuspend` example (Figure 4.6) for illustration.

To handle this case properly, the SC program has to do some bookkeeping. Specifically, it must keep track of local signals of states that might be suspended. To let SC do this, the application must provide a mapping from thread ids to lists of signals that are declared local to the thread, or its descendants. This mapping must be given by the `sigsDescs[]` array, see the complete listing of `PreAndSuspend`, Listing B.19, line 36. Whenever a thread *i* is suspended, the signals given in `sigsDescs[i]` are added to a list of signals (`sigsFreeze`) whose status is preserved into the next tick.

### 3.3.3 SC sequential control operators

The lower part of Table 2.2 lists further SC operators dedicated to sequential control. The `GOTO` is just what it says, implemented directly as a C `goto`. It is listed as an SC operator merely for completeness.

SyncCharts allow entry and exit actions to be associated with a state, these can also be used in SC, as explained in the following.

### Entry actions

An *entry action* associated with a state  $S$  is performed whenever  $S$  is entered. This can be implemented in SC basically as a code sequence that immediately precedes the entry point of  $S$ , and redirecting transitions to  $S$  to the beginning of the entry action. Hence, no special SC operators are needed for entry actions.

### Exit actions

An *exit action* is performed whenever the state is left. This also includes abortions, of the state itself or one of its (transitive) parents. This makes exit actions more powerful than entry actions, and their implementation does require specific SC operators.

An aborted thread does not regain control, so the aborting thread must ensure that any exit actions associated with an aborted thread are still performed. Again, there is a clear rule on what should happen when multiple exit actions might be performed: when macrostates with exit actions are nested, the exit actions are executed in the innermost to outermost order.

SC provides two operators for writing exit actions.  $\text{CALL}(l, l_{ret})$  is an unconditional function call to label  $l$ . As we do not have direct access to a program counter, we must also explicitly specify the return address  $l_{ret}$ .  $\text{CALL}$  can be used whenever it is clear that the exit action must be called. For an exit action associated with state  $S$ , this could be for example at a regular exit point of  $S$ , or before  $S$  aborts and transfers to another state via  $\text{TRANS}$ . It could also be at an abortion of a parent state  $T$ , if  $S$  must be active whenever  $T$  is, *i. e.*, there are no sibling states of  $S$ .

The  $\text{RET}$  instruction returns from a function call, by transferring control to the  $l_{ret}$  label supplied to the last call instruction. Note that since exit actions are not nested, there is no need for a return address stack, it suffices to just remember one return address (implemented as global variable `returnAddress`). However, should one want to use the SC call mechanism also for nested calls, it would be straightforward to implement a stack instead of a simple return address variable..

The interesting case, as already mentioned, are abortions. Consider the situation where  $S$  has some sibling states, and the parent  $T$  gets aborted. When  $T$  gets aborted the exit action of  $S$  must be performed if  $S$  is active; otherwise, when a sibling of  $S$  is active, the exit action of  $S$  must not be performed. To implement this behavior, the  $\text{ISAT}(id, l_{state}, l)$  operator can be used. It checks whether thread  $id$  is at state  $l_{state}$ ; if this is the case, control proceeds to the next instruction, which then commences the `on exit` function of associated with  $id$  at state  $l_{state}$ . Else, control proceeds to label  $l$ .

SC also provides  $\text{ISATCALL}(id, l_{state}, l_{action}, l)$  as a shorthand for  $\text{ISAT}(id, l_{state}, l)$ ;  $\text{CALL}(l_{action}, l)$ . For example, in the Exits code (Fig. 4.2b), the  $\text{ISATCALL}$  at label L3, which is reached upon normal termination of state M10, conditionally calls the exit action of M2. An equivalent SC program that does not make use of this shorthand is shown in Fig. 4.3a.

```

1  int tick(int islnit)
2  {
3      // TICKSTART(islnit);
4      if ( islnit ) {
5          tickCnt = 0;
6          pc[TickEnd] = &&TickEndLabel;
7          pr[TickEnd] = 0;
8          enabled = (1 << ids[TickEnd]);
9          active = enabled;
10         cid = ids[Main];
11         enabled |= (1 << cid);
12         active |= (1 << cid);
13     } else { active = enabled;
14         __asm volatile (" bsr _%1,%0\n":"=r"(cid):"c"(active)
15         );
16         goto *pc[cid];
17     }
18     // PAR(0, AB, ids[AB]);
19 ABO: pc[ids[AB]] = &&AB;
20     pr[ids[AB]] = 0;
21     enabled |= (1 << ids[AB]);
22     active |= (1 << ids[AB]);
23
24     // PARE(0, ABomain, id2b(AB) | id2b(WaitA) | id2b(
25         WaitB));
26     pc[cid] = &&ABomain;
27     pr[cid] = 0;
28     descsc[cid] = (1 << ids[AB]) | (1 << ids[WaitA]) | (1
29         << ids[WaitB]);
30     __asm volatile (" bsr _%1,%0\n":"=r"(cid):"c"(active));
31     goto *pc[cid];
32
33     // PAR(0, WaitA, ids[WaitA]);
34 AB: pc[ids[WaitA]] = &&WaitA;
35     pr[ids[WaitA]] = 0;
36     enabled |= (1 << ids[WaitA]);
37     active |= (1 << ids[WaitA]);
38
39     // PAR(0, WaitB, ids[WaitB]);
40     pc[ids[WaitB]] = &&WaitB;
41     pr[ids[WaitB]] = 0;
42     enabled |= (1 << ids[WaitB]);
43     active |= (1 << ids[WaitB]);
44
45     // PARE(0, ABMain, id2b(WaitA) | id2b(WaitB));
46     pc[cid] = &&ABmain;
47     pr[cid] = 0;
48     descsc[cid] = (1 << ids[WaitA]) | (1 << ids[WaitB]);
49     __asm volatile (" bsr _%1,%0\n":"=r"(cid):"c"(active));
50     goto *pc[cid];
51
52     // PAUSE(L0);
53 WaitA:pc[cid] = &&L0;
54     active &= ~(1 << cid);
55     __asm volatile (" bsr _%1,%0\n":"=r"(cid):"c"(active));
56     goto *pc[cid];
57
58     // PRESENT(A, WaitA);
59 L0: if (!( signals & (1 << A))) goto WaitA;
60
61     // TERM;
62     enabled &= ~(1 << cid);
63     active &= ~(1 << cid);
64     __asm volatile (" bsr _%1,%0\n":"=r"(cid):"c"(active));
65     goto *pc[cid];
66
67     // PAUSE(L1);
68 WaitB:pc[cid] = &&L1;
69     active &= ~(1 << cid);
70     __asm volatile (" bsr _%1,%0\n":"=r"(cid):"c"(active));
71     goto *pc[cid];
72
73     // PRESENT(B, WaitB);
74 L1: if (!( signals & (1 << B))) goto WaitB;
75
76     // TERM;
77     enabled &= ~(1 << cid);
78     active &= ~(1 << cid);
79     __asm volatile (" bsr _%1,%0\n":"=r"(cid):"c"(active));
80     goto *pc[cid];
81
82     // JOIN(Done, ABmain);
83 ABmain: if (((enabled & descsc[cid]) == 0)) goto Done;
84     pc[cid] = &&ABmain;
85     active &= ~(1 << cid);
86     __asm volatile (" bsr _%1,%0\n":"=r"(cid):"c"(active));
87     goto *pc[cid];
88
89     // EMIT(O);
90 Done: signals |= (1 << O);
91
92     // TERM;
93     enabled &= ~(1 << cid);
94     active &= ~(1 << cid);
95     __asm volatile (" bsr _%1,%0\n":"=r"(cid):"c"(active));
96     goto *pc[cid];
97
98     // PAUSE(L2);
99 ABomain:pc[cid] = &&L2;
100     active &= ~(1 << cid);
101     __asm volatile (" bsr _%1,%0\n":"=r"(cid):"c"(active));
102     goto *pc[cid];
103
104     // PRESENT(R, ABomain);
105 L2: if (!( signals & (1 << R))) goto ABomain;
106
107     // TRANS(ABO);
108     enabled &= ~descsc[cid];
109     active &= ~descsc[cid];
110     goto ABO;
111
112     // TICKEND
113 TickEndLabel: return (enabled != (1 << ids[TickEnd]));
114 }

```

Figure 3.5: ABRO tick function after macro expansion (produced by gcc -E).

### 3.3.4 An example of expanded macros—ABRO

Fig. 3.3.4 shows the ABRO tick function from Fig. 3.3b after macro expansion. For better readability, comments are added, extraneous braces and semicolons are removed, and line breaks and indentation were reformatted.

## 3.4 SC Structure

### 3.4.1 Program files

There are two variants possible, the *minimal variant* that does not link in `sc.c`, and the (*extended variant*) that does link it in.

#### The minimal files variant

An SC program consists of at least the following files:

**sc.h** A header file that defines a number of types, global variables and the SC macros.

**APP.c** A C file that defines an application *APP* (for example, `ABRO.c`). This must include `sc.h`.

The above is sufficient, if no separate, alternative dispatcher routine is required, which in turn requires that

1. the application does not depend on thread priorities, and
2. the dispatcher can be implemented with a Bit Scan Reverse (BSR) assembler instruction embedded in the code. This instruction is accessible on x86 architectures when using `gcc`.

In this minimal files version, the *APP.c* file must define a `main` function.

To produce an executable, it suffices to compile just *APP.c*. For example, “`gcc PCO.c -o PCO`” produces an executable `PCO`.

#### The extended files variant

This variant should be used if

- an alternative dispatcher is required, because
  - the application needs thread priorities, or
  - BSR is not available,
- or if one wants the convenience of using a pre-defined `main` function that for example compares the output of the tick function with a given sequence of reference outputs.

This extended files variant uses the following additional file:



**sc.c** A C file that contains the `main` function, alternative dispatcher functions (`selectCidPrio` and `selectCidNoprio`), and an auxiliary function for tracing (`vec2names`) that converts a bit vector to a string of thread or signal names.

Note that the `main` function assumes that signals are used, and hence calls signal-related functions that must be provided by *APP.c* (see Section 3.4.2). This means that when the extended files variant is used, for example, because the application uses thread priorities and hence an alternative dispatcher function is needed, *APP.c* must define these signal-related functions (which can be empty). This is slightly awkward and could be avoided for example by spreading the functions in `sc.c` across several files. Another alternative would be to pre-define alternative `main` functions (or rather functions called by `main`, which in turn can be selected via a macro mechanism in *APP.c*, similar to the selection of the appropriate dispatcher). However, to keep things simple, the functions are at this point all in `sc.c`.

To produce an executable, `sc.c` and *APP.c* must be compiled and linked. For example, “`gcc ABRO.c sc.c -o ABRO`” produces an executable `ABRO`.

The files `sc.c` and `sc.h` are part of the SC software package, *APP.c* must be written by the SC programmer.

## 3.4.2 Functions

### Minimal files variant

In the minimal files variant, *APP.c* must not provide any specific function—except, as usual in C, a `main` function. However, it is good practice to modularize the program by providing the following function:

**tick()** This function is the top-level function that describes the behavior of the application. One call to `tick` completes when all active threads have reached the end of a logical tick (indicated by the `PAUSE` operator) or have terminated (indicated by `TERM`). The `main` function, defined in `sc.c`, calls `tick` repeatedly, until all threads defined in `tick` have terminated.

### Extended files variant

If the `main` function provided by `sc.c` is used, the `tick` function, described in Section 3.4.2, is not optional, but mandatory, as it is called by `main` defined in `sc.c`. In addition, `main` calls the following functions, which therefore must be provided in *APP.c*:

**getInputs()** This function is called before `tick` is called and defines input signals. If no signals are used, this function is empty.

**checkOutputs()** This function is called after the `tick` function and can be used to define reference outputs. These are then compared with the outputs actually computed. If no signals are used, this function is also empty.

**printval(int id)** A function to print valued signal, with index `id`. If no valued signals are used, this function is empty.

### 3.4.3 Types

#### Minimal files variant and no signal usage

An SC program has to define the following type:

**idtype** An enumeration type that declares the *thread names*. This must contain the name TickEnd.

TickEnd is a special, degenerated thread that does nothing but finish a tick. This is implemented by assigning its program counter the label defined by the TICKEND operator (see Section 3.3.1). The TickEnd thread should only execute when no other thread is active anymore. It therefore must be assigned the lowest thread id (statically, in **idtype**, see Section 3.4.3), and the lowest priority (at run time, by the TICKSTART operator, see Section 3.3.1).

We here exploit that C enumeration types correspond to a sequence of integers, starting at 0 and increasing by 1. Thus **idtype** serves as a mapping from *thread names*, used in the SC program, to *thread identifiers*. These identifiers in turn serve as indices to thread-related information, in particular their *thread id* (via the **ids** array, see below). Thread identifiers are unique to each thread occurring in the program, implicitly defined via the **idtype**. In contrast, thread ids may be shared between threads, as long as these threads cannot be concurrent. In other words, thread identifiers have to be unique at compile time (statically), whereas thread ids may be shared, but have to be unique at run time (dynamically).

In the examples used here, there is no sharing of thread ids, as all threads used may be concurrent. Furthermore, it is often (but not always) the case that the thread id is identical to the thread identifier.

#### Extended files variant, or usage of signals

In the extended files variant, or if signals are used, *APP.c* also must define the following type:

**signaltype** An enumeration type that declares the signal names.

### 3.4.4 Variables

#### Minimal files variant and no signal usage

An SC program has to define the following variables:

**idHi** Defines the highest thread id in use.

**ids** Integer array that maps thread indices to ids. This must map thread TickEnd (to be included in the **idtype**, see Section 3.4.3) to id 0.

**id2threadname[]** An array of strings that maps thread ids to thread names. This should correspond to the **idtype** enumeration type (see Section 3.4.3).

#### Extended files variant, or usage of signals

In the extended files variant, or if signals are used, *APP.c* also must define the following variable:

**s2signame** An array of strings that maps signal ids to signal names. This should correspond to the defined **signaltype** enumeration.

Furthermore, in the extended files variant, *APP.c* must define the following variables, which are used by **main**:

**runMax** The number of runs to be executed.

**tickMax** The maximal number of ticks to execute per run.

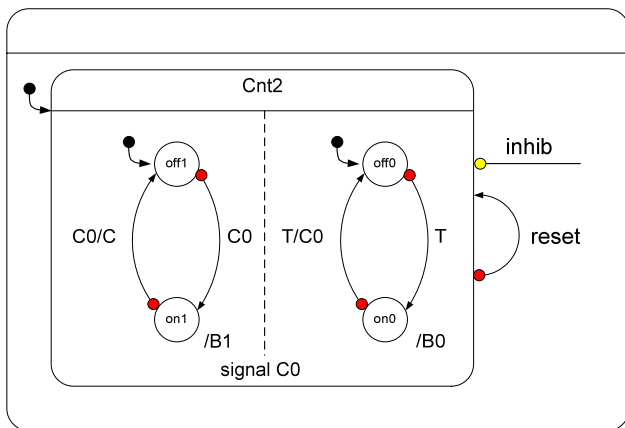
# Chapter 4

## Examples

This chapter contains a selection of further examples provided by André [2].

### 4.1 Count2Suspend

Count2Suspend [2, Fig. 8-5], shown in Fig. 4.1, illustrates the use of suspension. The 2-bit counter in macrostate Cnt2 counts up whenever input signal T is present—except when *inhib*



(a) SyncChart

```

1 // Thread ids: Off1=1, Off0=2, Main=3
2   TICKSTART(islnit);
3
4 Cnt2:  PAR(0, Off0, ids [Off0]);
5        PAR(0, Off1, ids [Off1]);
6        PARE(0, Cnt2Main, id2b(Off0) | id2b(Off1));
7
8 Off0:  PAUSE(L0);
9 L0:    PRESENT(T, Off0);
10 On0:  EMIT(B0);
11       PAUSE(L1);
12 L1:    PRESENT(T, On0);
13       EMIT(C0);
14       GOTO(Off0);
15
16 Off1:  PAUSE(L2);
17 L2:    PRESENT(C0, Off1);
18 On1:  EMIT(B1);
19       PAUSE(L3);
20 L3:    PRESENT(C0, On1);
21       EMIT(C);
22       GOTO(Off1);
23
24 Cnt2Main: PAUSE(L4);
25 L4:    PRESENT(reset, L5);
26       TRANS(Cnt2);
27 L5:    PRESENT(inhib, Cnt2Main);
28       SUSPEND(L5);
29
30   TICKEND;

```

(b) SC tick function

Figure 4.1: The Count2Suspend example.

suspends (“freezes”) operation of `Cnt2`. The `Main87,SUSPEND` statement performs the according control of the execution of `Cnt2`, see also p. 25.

### Precedence constraints

1. Strong abort and suspension on `Cnt2` (*outer-inner*):  
 $\text{Main} \succ \text{Off0}, \text{Main} \succ \text{Off1}$
2. Communication via a signal `C0` (*writer-reader*):  
 $\text{Off0} \succ \text{Off1}$

These constraints induce a partial order, met by the thread id assignment in the SC code.

## 4.2 Exits

Exits [2, Fig. 8-8], shown in Fig. 4.2, illustrates the handling of exit actions. These are implemented with the `CALL` operator, which calls exit actions unconditionally, and `ISATCALL`, which calls exit actions if the corresponding state is active (and now gets aborted). See also the descriptions of these operators on p. 29.

Alternative tick functions for Exits are shown in Fig. 4.3. The code shown in Fig. 4.3a differs from Fig. 4.2b only in that the shorthand `ISATCALL` is expanded into separate `ISAT` and `CALL` operations. The code in Fig. 4.3b inlines the exit actions. This violates the Write-Things-Once principle, but in this case makes the code shorter, as the exit actions consist of simple `EMIT` operations. However, rather surprisingly, this inlining actually degrades performance, by about 10%.

### Precedence constraints

1. Strong abort on `M0` (*outer-inner*):  
 $\text{Main} \succ \text{M10}, \text{Main} \succ \text{M2}, \text{Main} \succ \text{M11}$

We meet this by assigning `Main` the highest thread id (4). Furthermore, as thread `M10`, a child of `Main`, will eventually raise its priority to 1, `Main` is also assigned this priority, in `Main6,PARE`, after forking `M10`.

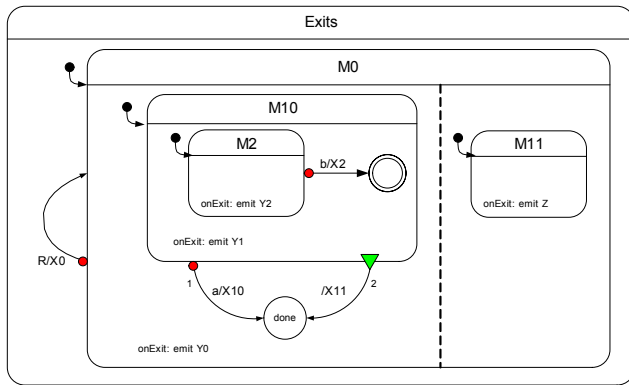
2. Strong abort on `M10` (*outer-inner*):

$$\text{M10}_{27,\text{PRESENT(A)}} \succ \text{M2}$$

This is met by assigning `M10` priority 1, with `M1020,JPPAUSE(1,...)`, while `M2` has priority 0. Note that this strong abort is not immediate, but delayed. Hence it is part of the depth of `M10` (see Sec. 3.2.3), and it is sufficient if `M10` enters its depth with priority 1, but not its surface (label `M10main`).

3. Normal termination on `M10` (*inner-outer*):

$$\text{M2} \succ \text{M10}_{20,\text{JPPAUSE}}$$



(a) SyncChart

```

1 // Thread ids: M11=1, M10=2, M2=3, Main=4
2 TICKSTART(isInIt);
3 M0: PAR(0, M10, ids[M10]);
4   PAR(0, M11, ids[M11]);
5     PARE(1, M0main, id2b(M10) | id2b(M11) | id2b(M2));
6
7 M10: PAR(0, M2, ids[M2]);
8     PARE(0, M10main, id2b(M2));
9
10 M2: PAUSE(M2depth);
11 M2depth:PRESENT(B, M2);
12     CALL(M2exit, L1);
13 M2exit: EMIT(Y2);
14     RET;
15 L1: EMIT(X2);
16     TERM;
17
18 L2: PRIO(0, M10main);
19 M10main:JPPAUSE(1, L3, M10depth);
20 L3: ISATCALL(ids[M2], M2depth, M2exit, L4);
21 L4: CALL(M10exit, L5);
22 M10exit:EMIT(Y1);
23     RET;
24 L5: EMIT(X11);
25     TRANS(Done);
26 M10depth:PRESENT(A, L2);
27     ISATCALL(ids[M2], M2depth, M2exit, L7);
28 L7: CALL(M10exit, L8);
29 L8: EMIT(X10);
30     TRANS(Done);
31 Done: PAUSE(Done);
32
33 M11: PAUSE(M11);
34 M11exit:EMIT(Z);
35     RET;
36
37 M0main: PAUSE(L9);
38 L9: PRESENT(R, M0main);
39     ISATCALL(ids[M2], M2depth, M2exit, L10);
40 L10: ISATCALL(ids[M10], M10depth, M10exit, L11);
41 L11: CALL(M11exit, L12);
42 L12: EMIT(Y0); // Only place to call exit action of M0
43     EMIT(X0);
44     TRANS(M0);
45
46 TICKEND;

```

(b) SC tick function

Figure 4.2: The Exits example.

```

1 // Thread ids: M11=1, M10=2, M2=3, Main=4
2     TICKSTART(islnit);
3 M0:   PAR(0, M10, ids[M10]);
4       PAR(0, M11, ids[M11]);
5       PARE(1, M0main, id2b(M10) | id2b(M11) | id2b(
6         M2));
7 M10:  PAR(0, M2, ids[M2]);
8       PARE(0, M10main, id2b(M2));
9
10 M2:   PAUSE(M2depth);
11 M2depth:PRESENT(B, M2);
12       CALL(M2exit, L1);
13 M2exit: EMIT(Y2);
14       RET;
15 L1:   EMIT(X2);
16       TERM;
17
18 L2:   PRIO(0, M10main);
19 M10main:JPPAUSE(1, L3, M10depth);
20 L3:   ISAT(ids[M2], M2depth, L4);
21       CALL(M2exit, L4);
22 L4:   CALL(M10exit, L5);
23 M10exit:EMIT(Y1);
24       RET;
25 L5:   EMIT(X11);
26       TRANS(Done);
27 M10depth:PRESENT(A, L2);
28       ISAT(ids[M2], M2depth, L7);
29       CALL(M2exit, L7);
30 L7:   CALL(M10exit, L8);
31 L8:   EMIT(X10);
32       TRANS(Done);
33 Done: PAUSE(Done);
34
35 M11:  PAUSE(M11);
36 M11exit:EMIT(Z);
37       RET;
38
39 M0main: PAUSE(L9);
40 L9:   PRESENT(R, M0main);
41       ISAT(ids[M2], M2depth, L10);
42       CALL(M2exit, L10);
43 L10:  ISAT(ids[M10], M10depth, L11);
44       CALL(M10exit, L11);
45 L11:  CALL(M11exit, L12);
46 L12:  EMIT(Y0);
47       EMIT(X0);
48       TRANS(M0);
49
50     TICKEND;

```

(a) Tick function without ISATCALL

```

1 // Thread ids: M11=1, M10=2, M2=3, Main=4
2     TICKSTART(islnit);
3 M0:   PAR(0, M10, ids[M10]);
4       PAR(0, M11, ids[M11]);
5       PARE(1, M0main, id2b(M10) | id2b(M11) | id2b(
6         M2));
7 M10:  PAR(0, M2, ids[M2]);
8       PARE(0, M10main, id2b(M2));
9
10 M2:   PAUSE(M2depth);
11 M2depth:PRESENT(B, M2);
12       EMIT(Y2);
13       EMIT(X2);
14       TERM;
15
16 L2:   PRIO(0, M10main);
17 M10main:JPPAUSE(1, L3, M10depth);
18 L3:   ISAT(ids[M2], M2depth, L4);
19       EMIT(Y2);
20 L4:   EMIT(Y1);
21       EMIT(X11);
22       TRANS(Done);
23 M10depth:PRESENT(A, L2);
24       ISAT(ids[M2], M2depth, L7);
25       EMIT(Y2);
26 L7:   EMIT(Y1);
27       EMIT(X10);
28       TRANS(Done);
29 Done: PAUSE(Done);
30
31 M11:  PAUSE(M11);
32
33 M0main: PAUSE(L9);
34 L9:   PRESENT(R, M0main);
35       ISAT(ids[M2], M2depth, L10);
36       EMIT(Y2);
37 L10:  ISAT(ids[M10], M10depth, L11);
38       EMIT(Y1);
39 L11:  EMIT(Z);
40       EMIT(Y0);
41       EMIT(X0);
42       TRANS(M0);
43
44     TICKEND;

```

(b) Tick function with inlined exit actions

Figure 4.3: Alternative variants for the SC tick function of the Exits example (Fig. 4.2).

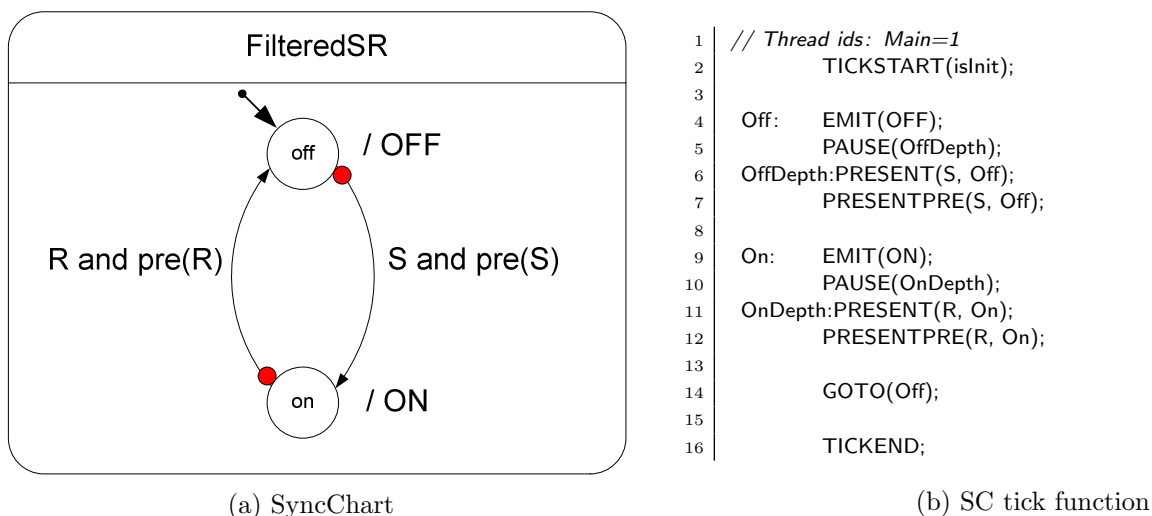


Figure 4.4: The FilteredSR example.

This is met by assigning  $M10$  a lower id than  $M2$ , and executing both  $M2$  and the test for normal termination with priority 0. To ensure the latter, we set the priority of  $M10$  at  $M10_{9,PARE}$ , for the test in the tick when  $M10$  is entered, and at  $M10_{19,PRIO(0)}$ , for the test in subsequent ticks.

4. Precedence of strong abortion over normal termination on  $M10$  (*transition-number*):

$$M10_{27,PRESENT(A)} \succ M10_{20,JPPAUSE}$$

This intra-thread structural constraint is met by ordering the operations in the program accordingly.

### 4.3 FilteredSR

FilteredSR [2, Fig. 8-18], shown in Fig. 4.4), illustrates the use of PRE on pure signals.

We here use *signal expressions*, in this case signal conjunction. In full regular C, such expressions can be built up the usual way with C's logical operators ( $!$ ,  $\&\&$ ,  $\|\|$ ). If we want to restrict ourselves to plain SC operators, we can encode these expressions with control flow, as is done here.

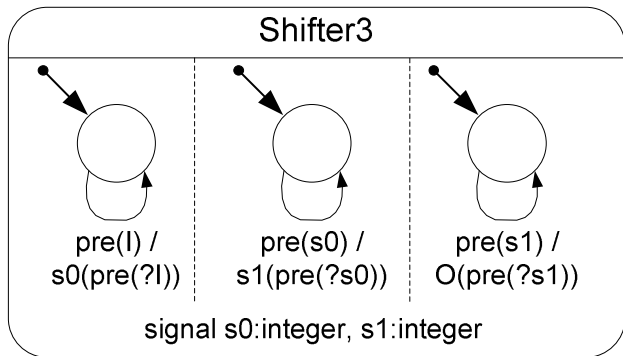
**Precedence constraints** As there is just the **Main** thread and each state has just one outgoing transition, there are no precedence constraints.

### 4.4 Shifter3

Shifter3 [2, Fig. 8-19], shown in Fig. 4.5, illustrates the use of PRE on valued signals.

There are three top-level concurrent threads. There are two alternatives to implement this:





(a) SyncChart

```

1 // Thread ids: Main=1, Shift1=2, ShiftO=3
2   int reg0;
3
4   TICKSTART(islnit);
5
6   PAR(0, Shift1, ids[Shift1]);
7   PAR(0, ShiftO, ids[ShiftO]);
8   GOTO(Shift0);
9
10  Shift0depth:PRESENTPRE(l, Shift0);
11    VALPRE(l, reg0);
12    EMITINT(S0, reg0);
13  Shift0: PAUSE(Shift0depth);
14
15  Shift1depth:PRESENTPRE(S0, Shift1);
16    VALPRE(S0, reg0);
17    EMITINT(S1, reg0);
18  Shift1: PAUSE(Shift1depth);
19
20  ShiftOdepth:PRESENTPRE(S1, ShiftO);
21    VALPRE(S1, reg0);
22    EMITINT(O, reg0);
23  ShiftO: PAUSE(ShiftOdepth);
24
25  TICKEND;

```

(b) SC tick function

Figure 4.5: The Shifter3 example.

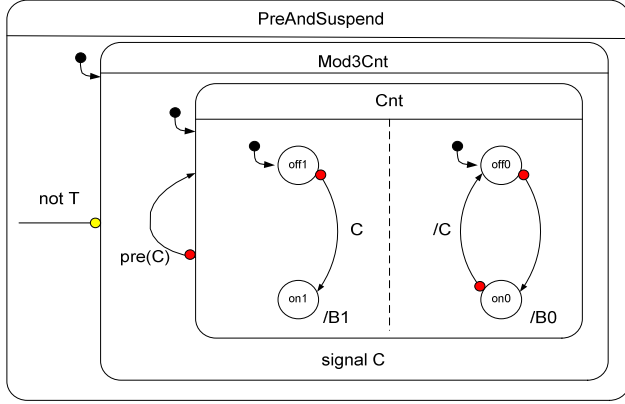
1. The **Main** thread is interpreted as the top-level macrostate **Shifter3**. It spawns off three children (**Shift0**, **Shift1**, **ShiftO**), and then terminates (with **TERM**), as it has nothing more to do.
2. The **Main** thread is interpreted as one of the concurrent subthreads of macrostate **Shifter3**, say **Shift0**. It spawns off two concurrent threads (**Shift1**, **ShiftO**); see also the last note in Sec. 3.3.1.

We here implement the second alternative, as it reduces the number of required threads by

1. Further notes on this methods of spawning concurrent threads:

- This is a fork without a **PARE**, hence there is no call to the dispatcher before **Main** delves into the code region implementing the **Shift0** substate. In this case this is unproblematic, as there are no precedence constraints to be obeyed (see below). In general, if we use this technique of spawning concurrent threads (instead of child threads) and there is a particular thread that must be executed next, we must make sure that the current thread takes on the role this particular thread.
- If there are no precedence constraints, it is suggested to let the spawning thread continue with the code region that follows after the fork, in this case **Shift0**. If the spawning thread can start at the beginning of that code region, this saves a **GOTO**. In this example we do not have this saving, as we still have to jump to the **Shift0** label.

A further optimization implemented here: Starting the code fragment belonging to state **Shift0** with its depth (**Shift0depth**) allows to save a final **GOTO** by folding it into **PAUSE(Shift0depth)**<sub>13</sub>. Similarly for the other concurrent states.



(a) SyncChart

Instant	1	2	3	4	5	6	7	8	9	10	11	12	13
<b>T</b>	-	+	-	+	-	+	-	-	+	-	+	+	-
<b>B0</b>	-	+	-	-	-	-	-	-	+	-	-	-	-
<b>B1</b>	-	-	-	+	-	-	-	-	-	-	+	-	-
<b>C</b>	-	-	+	+	-	+	+	-	+	+	-	+	-

(b) Sample execution trace [2]

```

1 // Thread ids: Off1=1, Off0=2, Cnt=3, Main=4
2 TICKSTART(isInIt);
3
4 PAR(0, Cnt, ids [Cnt]);
5 PARE(0, Mod3CntMain, id2b(Cnt) | id2b(Off1) | id2b(Off0
6 ));
7 Cnt: PAR(0, Off1, ids [Off1]);
8 PAR(0, Off0, ids [Off0]);
9 PARE(0, CntMain, id2b(Off1) | id2b(Off0));
10
11 Off1: PAUSE(L0);
12 L0: PRESENT(C, Off1);
13
14 On1: EMIT(B1);
15 PAUSE(On1);
16
17 L1: EMIT(C);
18 Off0: PAUSE(On0);
19
20 On0: EMIT(B0);
21 PAUSE(L1);
22
23 CntDepth:PRESENTPRE(C, CntMain);
24 TRANS(Cnt);
25 CntMain:PAUSE(CntDepth);
26
27 Mod3CntMain:PAUSE(Mod3CntDepth);
28 Mod3CntDepth:PRESENT(T, L2);
29 GOTO(Mod3CntMain);
30 L2: SUSPEND(Mod3CntDepth);
31
32 TICKEND;

```

(c) SC tick function

Figure 4.6: The PreAndSuspend example.

**Precedence constraints** There are no precedence constraints, even though there is signal-based communication via  $S0$  and  $S1$ ; all triggers are delayed (via  $PRE$ ), hence any written signals will not be read before the next tick.

## 4.5 PreAndSuspend

PreAndSuspend [2, Fig. 8-20], shown in Fig. 4.6, illustrates the proper handling of  $PRE$  in conjunction with suspension and local signals. See also the execution trace in Fig. 4.6b.

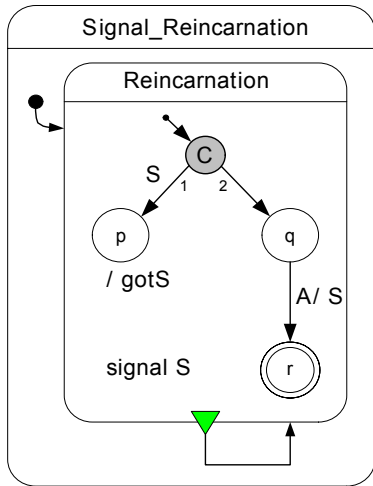
For the signal  $C$ , declared locally at state  $Mod3Cnt$ ,  $PRE(C)$  refers to the presence value of  $C$  in the previous tick in which  $Mod3Cnt$  was active (not suspended).

### Precedence constraints

1. Suspension of  $Mod3Cnt$  (*outer-inner*):

Main  $\succ$  Cnt

2. Strong abort on Cnt (*outer-inner*):



(a) SyncChart

```

1 // Thread ids: Main=1
2   TICKSTART(isInIt);
3
4 Reinc: SIGNAL(S);
5   PRESENT(S, Q);
6 P:   EMIT(gotS);
7   PAUSE(P);
8 Q:   PAUSE(L0);
9 L0:  PRESENT(A, Q);
10   EMIT(S);
11   GOTO(Reinc);
12
13   TICKEND;

```

(b) SC tick function

Figure 4.7: The Reincarnation example.

$\text{Cnt} \succ \text{Off0}, \text{Cnt} \succ \text{Off1}$

3. Communication via  $C$  (*writer-reader*)

$\text{Off0} \succ \text{Off1}$

These constraints induce a partial order, met by the thread id assignment in the SC code.

## 4.6 Reincarnation

Reincarnation [2, Fig. 8-22], shown in Fig. 4.7, illustrates the **SIGNAL** instruction to handle signal reincarnation.

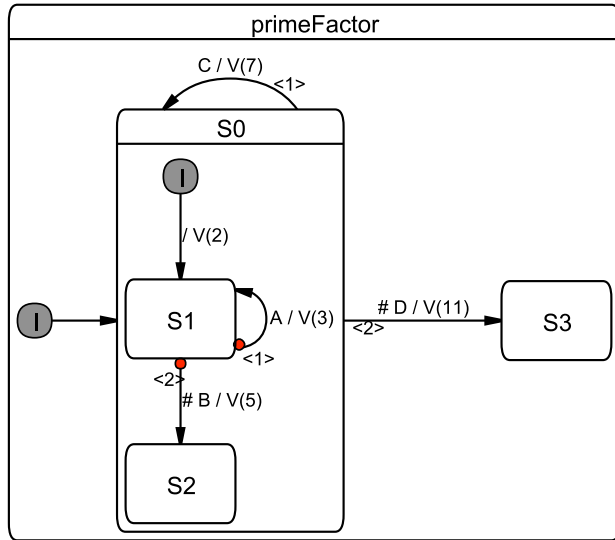
The canonical encoding in SC would have the **Main** thread spawn off an inner thread that computes the behavior of the **Reincarnation** state. However, as the macrostate **Reincarnation** only has a normal termination transition attached to it, all the **Main** thread does is to reenter itself once **Reincarnation** has terminated. This can be streamlined by transferring control from all terminating states within **Reincarnation** to the entry of **Reincarnation**. In this case, there is just one such terminating state, namely **r**. Hence, there is no need anymore for a separate parent thread that checks for termination. In other words, the **Main** thread can directly run the **Reincarnation** state.

### Precedence constraints

1. Normal termination of **Reincarnation** (*inner -outer*):

This gets folded into the sequential code of **Main**.

2. There is also the conditional pseudo state with two outgoing transitions, which in principle constitutes a transition-number precedence. In this case, this corresponds to a simple if-then-else branch, encoded in  $\text{PRESENT}(S, Q)_5$ .



(a) SyncChart

```

1 // Thread ids: Main=1, S1=2
2   TICKSTART(islnit);
3
4 S0:  PAR(0, S1, ids[S1]);
5     PARE(0, S0main, id2b(S1));
6
7 S1:  EMITINTMUL(V, 2);
8
9 S1surf: PRESENT(B, S1depth);
10     EMITINTMUL(V, 5);
11     GOTO(S2);
12 S1depth: PAUSE(L0);
13 L0:  PRESENT(A, S1surf);
14     EMITINTMUL(V, 3);
15     GOTO(S1surf);
16
17 S2:  PAUSE(S2);
18
19 S0main: PRESENT(D, S0depth);
20     EMITINTMUL(V, 11);
21     TRANS(S3);
22 S0depth: PAUSE(L1);
23 L1:  PRESENT(C, S0main);
24     EMITINTMUL(V, 7);
25     TRANS(S0);
26
27 S3:  PAUSE(S3);
28     TICKEND;

```

(b) SC tick function

Figure 4.8: The PrimeFactor example.

## 4.7 PrimeFactor

PrimeFactor [2, Fig. 8-25], shown in Fig. 4.8, illustrates the use of valued signals and the proper handling of reincarnation/schizophrenia.

- S0 needs no JOIN, as it never terminates normally.
- S2: PAUSE(S2) encodes a final, but non-terminating state; this corresponds to Esterel's halt.

### Precedence constraints

1. Normal termination transitions of S0 (*inner -outer*):

S1  $\succ$  Main

This is met by thread id assignment.

2. Ordering of transitions of S0 (*transition-number*):

Main<sub>23,PRESENT(C)</sub>  $\succ$  Main<sub>19,PRESENT(D)</sub>

This is met by statement ordering.

3. Ordering of transitions of **S1** (*transition-number*):

$$\mathbf{S1}_{13, \text{PRESENT(A)}} \succ \mathbf{S1}_{9, \text{PRESENT(B)}}$$

This is also met by statement ordering.

To summarize, all scheduling constraints are handled by proper ordering of the transition predicate tests, and by the fact that the id of the inner state (**S0**, id 1) is higher than the priority of the surrounding root thread.

# Chapter 5

## Related Work

**Statechart variants** Since the original Statecharts proposal [13], numerous dialects of Statecharts have been developed and Statecharts have also been incorporated into the Unified Modeling Language (UML). Statecharts are supported by a multitude of modeling tools; the first commercial tool was Statemate [13], other established tools today are SCADE/Esterel Studio (Esterel Technologies), Matlab/Simulink/Stateflow (The Mathworks), ASCET (ETAS), or Rational Rose (IBM). These tools all implement the fundamental Statechart concepts of concurrency, hierarchy, and signal broadcast. However, their underlying MoCs also have some subtle, but important differences, in particular regarding their handling of concurrency. In fact, while the visual *syntax* of Statecharts appears fairly simple and straightforward, it is not at all obvious what their *semantics* should be [5].

Most Statechart dialects in use today, including UML Statecharts, have the limitation that they do not offer deterministic concurrency. Concurrent states are often implemented as concurrent threads, thus inheriting the non-determinism associated with thread scheduling [16]. This can be alleviated by adopting a *strictly synchronous* semantics, which precisely states how computations should proceed [6]. The synchronous MoC implements the *synchrony hypothesis*, which abstracts from concrete run-time behavior by assuming that the computation of a reaction does not take any time. The *strict* interpretation of synchrony also adopts a fixed point semantics, which means that the status of events (sometimes also referred to as signals) must be consistent throughout a reaction. Strictly synchronous Statechart dialects are Argos [19] and SyncCharts [2], also known as Safe State Machines (SSMs). There is also the *loose* interpretation of synchrony, which does assume that physical time does not progress during a reaction, but does not require that the system progresses along fixed points. Instead, it allows the presence/absence status of events to change during a reaction. This is less restrictive than strict synchrony, but degrades compositionality and may lead to infinite computations. The original Statechart proposal implemented loose synchrony.

**Expressing Statecharts in C/C++** As mentioned in the introduction, it is already common practice to express Statecharts in a classical programming language. Samek describes how to express UML Statecharts in C/C++ [27]. As in UML Statecharts, this approach does not provide deterministic concurrency. Wagner *et al.* describe how to implement FSMs in C [31], but these are flat automata without any concurrency.

**Synchronous language extensions** There have been several proposals to extend traditional programming languages by synchronous constructs. Reactive C [10] is an extension of C inspired by Esterel. It employs the concepts of computational instants (ticks) and preemptions, but does not provide true concurrency; Reactive C’s `merge` operator emulates concurrency by running threads sequentially, in their textual order.

FairThreads [7] extend this by true concurrency, implemented via native threads. They also offer macros to express automata. SC does not use native threads, but does its own, light-weight thread book keeping. Another difference is that the signal mechanism provided by FairThreads does not allow reaction to signal absence, whereas SC does allow this (see `grcbal3`).

The Esterel-C Language (ECL) [15] is another proposal to extend C by Esterel-like constructs. A C program is annotated with Esterel-like constructs for signal handling and reactive control flow, and from this program the ECL compiler derives an Esterel part and a purely sequential C part. SC is in the same spirit of annotating C with synchronous operators, but differs from ECL in that it does not resort to a separate language (Esterel).

Another recent proposal for a synchronous extension of C is Precision Timed C (PRET-C) [25]. PRET-C focuses on temporal predictability and assumes a target architecture with specific support for thread scheduling and abort handling. PRET-C provides a minimal set of C extensions, namely a concurrency operator, which runs threads with static priorities, a delayed abortion operator, and an EOT operator that delineates ticks. An associated compiler produces a corresponding intermediate format, the Timed Concurrent Control Flow Graph, where each thread at each EOT tests whether it is aborted or not with `Checkabort` nodes.

Lusteral, presented by Mendler and Pouzet [20], also tries to capture the essence of synchronous programming in a small number of operators. It combines elements of the synchronous languages Lustre, Esterel and Signal and embeds them in Haskell. As this is a functional language, it allows to express the semantics of the Lusteral operators nicely as higher-order functions.

**Compiling synchronous programs** As SC expresses synchronous, control-oriented concurrency by means of a—ultimately sequential—C program, executing an SC program raises similar issues as they arise when synthesizing a synchronous language into sequential code. There have been numerous proposals for this, in particular for the Esterel language [22, 9]. It is a common procedure to translate an Esterel program into a C program, but the resulting C program usually bears little resemblance to the original Esterel program. For example, the C code might be a flat automaton, or it might simulate a hardware circuit.

Probably the closest in spirit to SC is the BAL virtual machine [9], which proposes a high-level ISA that captures the Esterel semantics as closely as possible; see also the comparison done in Chapter 6.

Another interesting approach is the dynamic list code generation [9], which produces C code that executes concurrently running threads by dispatching small groups of instructions that can run without a context switch. These blocks are dispatched by a scheduler that uses linked lists of pointers to code blocks that will be executed in the current cycle. While the fundamentals of that code generation are very different from the SC approach, their use of pointers and `gcc`’s computed `gotos` has inspired the label-based “coarse grain program counter” approach presented here.

**The PRET and SHIM programming models** As discussed in Sec. 2.1, SC is also related to the programming model proposed for the Precision Timed Architecture (PRET) proposed by Edwards and Lee [18], but does not rely on low-level timing for synchronization.

Another related programming model is SHIM [29], proposed for software/hardware integration, which provides Kahn process networks with CSP-like rendezvous communication and exception handling. It uses a separate compiler to convert a SHIM program into sequential C code. SHIM, like SC, has been inspired by synchronous languages, but it does not use a synchronous programming model, instead relying on communication channels for synchronization.

**Code generation from Statecharts/SyncCharts** As SC can be used as a target format when synthesizing Statecharts into a sequential program, this work also relates to code generation from Statecharts. Three different methods of compiling Statecharts are common: compilation into an object oriented language using the state pattern [1], dynamic simulation [32], and flattening into finite state machines. Since flattening can suffer from state explosion, often a combination of flattening and dynamic simulation is used. All of these methods incur relatively high overhead and typically make use of a run time system to achieve concurrency, and usually the result is not deterministic.

For SyncCharts, it is also possible to translate the Statechart model into an equivalent textual Esterel program [11]. Such a translation was proposed by André [3] together with the initial definition of SyncCharts and their semantics. This transformation, with additional unpublished optimizations, is implemented in Esterel Studio. The resulting Esterel program can then be translated into software or hardware [22]. As discussed in Chapter 6, this path via Esterel to C is here used for experimental comparison. A drawback of this approach is that the original structure of SyncCharts cannot always be preserved in the Esterel code, as Esterel does not allow the arbitrary control flow that can be expressed by SyncChart transitions; this also can induce the need for additional signals, to encode the next active state. This structure is even less preserved in a C program compiled from the Esterel program.

**Compilation for reactive processors** One approach to synthesize SyncCharts into a textual program that does preserve the original structure is to generate code directly for a reactive processor [30], as done by the state machine to KEP compiler (**smakc!**) [28]. Unlike the instruction set architecture (ISA) of traditional processors, which provide only sequential control flow operators such as branches and jumps, the ISA of reactive processors directly expresses concurrency and preemption. The **smakc!** compiler targets the Kiel Esterel Processor [17], which implements synchronous concurrency via multi-threading. This multi-threading approach, which is also realized for example in the StarPro processor [33], has the advantage of allowing high degrees of concurrency without excessive resource requirements.

The SC operators have been inspired by the KEP ISA, and adopt the KEP's mechanism of priority-based multi-threading. However, the SC operators have been developed with SyncCharts in mind, rather than Esterel, and they make minimal assumptions on the execution platform. The main resulting differences between SC and the KEP ISA are:

- SC provides a **TRANS** operator that implements an arbitrary state transition;
- SC does not provide Esterel's exception handling via traps;
- SC does not rely on special watcher units to implement aborts.



A motivation for the KEP's watcher units was to avoid `Checkabort` instructions [26, 25], as these introduce an overhead—both in terms of code size as execution speed—at each tick, in all threads, proportional to the abort nesting depth. Interestingly, SC needs neither watchers nor `Checkaborts`, by giving parent threads the power to abort their descendants with the `TRANS` operator.

# Chapter 6

## Experimental results

### 6.1 Conciseness of SC, Code Size

The main goal in developing SC was to develop a concise embedding of SyncChart behavior into C. It is difficult to measure “conciseness” precisely, as this compares a visual language against a textual one. A better point of reference might be Esterel code. For example, `grcbal3` in Esterel takes 25 lines (see Fig. 2.2a); in SC, it takes 28 lines (Fig. 2.2b). This indicates a comparable level of conciseness, which is remarkable in that the SC operators are embedded in the imperative, sequential programming model of C.

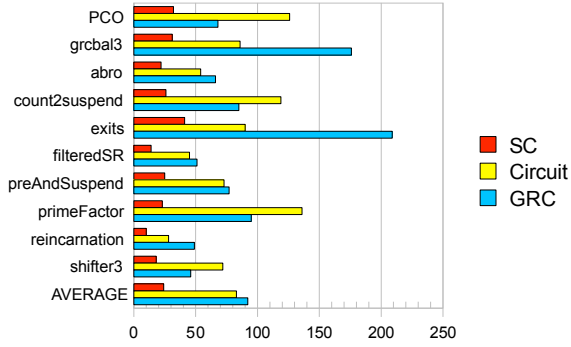
Another interesting point of comparison is the BAL VM instruction set, as it has been designed specifically to encode Esterel programs in as little memory as possible [9]. To encode `grcbal3`, BAL uses 74 instructions, of complexity comparable to the SC operators. The SC version makes do with 28 instructions, and these are also arguably easier to relate to an Esterel program or a SyncChart than the BAL assembler. This makes SC an attractive alternative candidate for a VM instruction set.

Fig. 6.1a compares the size of the SC tick functions for a number of benchmarks, taken from André [2], with the size of the C code generated by EsterelStudio. Two synthesis variants are considered, one based on circuit simulation, the other based on GRC. As can be seen, SC is often less than half the size of the synthesized C code.

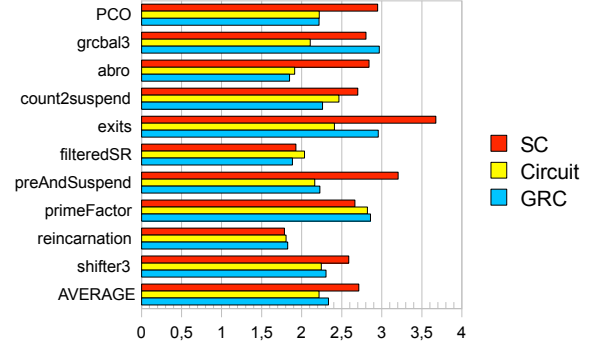
Fig. 6.1b compares sizes of object code for the tick function. Here, the SC code is larger on average, just in two cases it is slightly better than both E-Studio results. However, considering the sizes of the executable on an x86 architecture, shown in Fig. 6.1c, SC is ahead again. All results were obtained with `gcc -O3`.

### 6.2 SC Performance

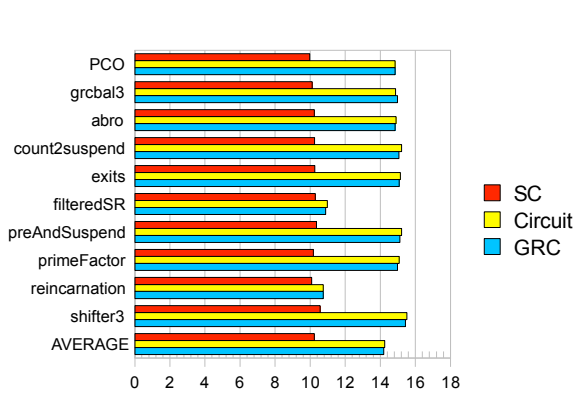
The development of SC has not been motivated primarily by performance concerns, but still it is interesting to see how it compares. On the negative side, SC basically just interprets a SyncChart, it cannot perform any global optimizations or partial evaluations at compile time, as do for example the EsterelStudio synthesis tools. On the positive side, SC code has no scalability problems, neither in terms of code size (like the flat automaton synthesis approach) nor in terms of run time. It only does work that needs to be done, in the sense that no unnecessary code regions are executed. This is different than for example the widely



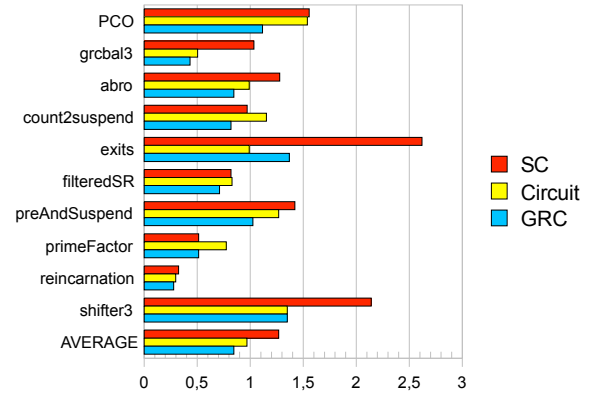
(a) Size of tick function in C source code, line count without empty lines and comments



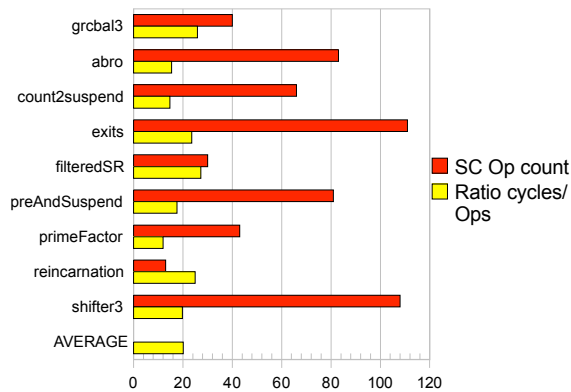
(b) Size of tick function object code, in Kbytes



(c) Size of executable, in Kbytes



(d) Accumulated run times of tick function, in thousands of clock cycles



(e) SC operations count, ratio to clock cycles

Figure 6.1: Comparison of SC with two code synthesis variants of Esterel Studio.

used circuit simulation approach, where always the whole circuit is simulated, irrespective of which regions are active. Furthermore, the SC context switches are very light weight, as 1) each thread requires very little information (see Sec. 2.1), and 2) the dispatcher is fast in typical scenarios (see Sec. 3.2.2). Therefore, SC certainly requires less overhead than a traditional thread-based implementation, where a context switch itself already takes thousands of instructions.

A more challenging point of reference are the monolithic C functions synthesized from SyncCharts. Figure 6.1d compares the run times of the tick functions, on an Intel Core 2 Duo architecture. For the measurements, a representative input trace was executed, outputs were compared against a reference trace, and the execution times of the individual calls to the tick functions were accumulated. Timings were done in numbers of processor cycles, using the x86 `rdtsc` (Read Time Stamp Counter) instruction. The machine runs at 2.4 GHz, so most of the runs took less than 1  $\mu$ s. As to be expected, SC does not beat any of the advanced synthesis techniques. In the `exits` example, which makes heavy use of exit actions, modularized into separate procedure calls rather than inlining (and possibly duplicating) them, the performance is even 2–2.5x worse. Overall, however, SC is roughly comparable, and in four of the ten benchmarks it is faster than the Circuit approach. As applications get larger, one should expect that SC stays comparable (at least), as again it does not have scalability problems. Also, one should expect that in practice, the run time of SC programs is dominated by regular C operations, not the SC operators.

A last statistic is shown in Figure 6.1e, which counts the SC operations executed, as listed in the output traces in the appendix. It also shows the ratio to the clock cycles from Fig. 6.1d. The ratio varies somewhat, but on average twenty clock cycles are needed to perform one SC operation.

# Chapter 7

## Conclusions and Outlook

SyncCharts in C are a light-weight approach to embed deterministic reactive control flow constructs into a widely used programming language. With a relatively small number of primitives it is possible to cover the complete SyncCharts language. The multi-threaded, priority-based approach has been inspired by synchronous reactive processing; hence, originally, this approach required a special compiler and a special architecture to implement. For example, the KEP has watchers that check for preemption in parallel to normal operation, a reactive processing unit that resolves control priorities on the fly, and a dispatcher that selects the next thread for execution at the beginning of each instruction cycle. Therefore, it was not obvious from the onset that it would be possible to achieve the same behavior by isolated SC operators, embedded in regular imperative sequential code, on a standard architecture, at a competitive performance. As it turns out, standard architectures already provide features that can be used to advantage, even if they are not directly available on the C level, such as the x86 `bsr` instruction that can be used for fast dispatching. A number of issues that pose challenges in implementing synchronous programs, such as schizophrenia or reaction to signal absence, are also unproblematic.

Considering the formal semantics of SC, as it is expressed in terms of C, one might take the stance that the semantics of the SC operators is expressed by the C statements they consist of, none of which touch on any of the many semantic uncertainties of C. In terms of mental complexity, this should not be as daunting as one might think; as of SC version 1.2, the file `sc.h` that defines all SC operators (except the general versions of the dispatcher, which are defined as functions in `sc.c`), is 567 lines long (see Listing A.1), of which 173 lines are comments, 49 lines are related to tracing, and 127 lines are empty. This leaves 218 lines of C code that explain what the 12 SC thread operators, 11 signal operators, and 5 sequential control operators do. Still, it should be worthwhile to formalize the semantics at a more abstract level, to allow formal reasoning about them.

SC is freely available, and can be used as is for writing reactive applications in C. However, there are a number of interesting further projects that should be pursued. As already mentioned, SC seems a viable candidate for synthesizing visual SyncCharts into code, especially if traceability is required, or as input language for PRET architectures. It would also be an interesting exercise to add something like a `DEAD` timing primitive [18] to SC. Unlike PRET architectures, traditional architectures probably cannot do this cycle-accurate; however, using something like `nanosleep` or the x86 `rtsc` instruction, it should be possible to get fairly close. One might use this to pad calls of the `tick` function to reduce the reaction jitter, replacing for example the crude call to `sleep` in PCO (Fig. 2.1c, line 15). A related issue is the WCRT

analysis for SC, which could build on earlier work [21,25]. Another question not addressed at all so far is how the SC approach could be used to extract true parallelism from a program, *e. g.* for programming multi-core processors. This should be feasible, *e. g.* by an alternative thread id/priority assignment scheme that expresses when things can be run in parallel; but it is an interesting question how to make this fast and how to minimize global synchronization overheads.

## Acknowledgments

Numerous discussions have helped shape SC and this report. I would like to thank in particular Alain Girault, Michael Mendler, Partha Roop, Robert de Simone and Claus Traulsen.

Christian Schneider has conducted the experiments reported on in Chapter 6.

# Bibliography

- [1] J. Ali and J. Tanaka. Converting Statecharts into Java code. In *Proceedings of the Fourth World Conference on Integrated Design and Process Technology (IDPT '99)*, Dallas, Texas, June 2000. Society for Design and Process Science (SDPS).
- [2] C. André. Semantics of SyncCharts. Technical Report ISRN I3S/RR-2003-24-FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.
- [3] C. André. Computing SyncCharts reactions. In *SLAP 2003: Synchronous Languages, Applications and Programming, A Satellite Workshop of ECRST 2003*, volume 88, pages 3 – 19, 2004.
- [4] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 2007.
- [5] M. v. d. Beeck. A comparison of Statecharts variants. In H. Langmaack, W. P. de Roever, and J. Vytupil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. Springer-Verlag, 1994.
- [6] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The Synchronous Languages Twelve Years Later. In *Proceedings of the IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, Jan. 2003.
- [7] F. Boussinot. Fairthreads: mixing cooperative and preemptive threads in C. *Concurrency and Computation: Practice and Experience*, 18(5):445–469, Apr. 2006.
- [8] M. E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.
- [9] S. A. Edwards and J. Zeng. Code generation in the Columbia Esterel Compiler. *EURASIP Journal on Embedded Systems*, Article ID 52651, 31 pages, 2007.
- [10] Frederic Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, 1991.
- [11] S. M. G. Berry and J.-P. Rigault. Esterel: Towards a synchronous and semantically sound high-level language for real-time applications. In *IEEE Real-Time Systems Symposium*, pages 30–40, 1983. IEEE Catalog 83CH1941-4.
- [12] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

- [13] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, Apr. 1990.
- [14] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pages 993–998, 1977.
- [15] L. Lavagno and E. Sentovich. ECL: a specification environment for system-level design. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 511–516, New York, NY, USA, 1999. ACM Press.
- [16] E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.
- [17] X. Li, M. Boldt, and R. von Hanxleden. Mapping Esterel onto a multi-threaded embedded processor. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, San Jose, CA, October 21–25 2006.
- [18] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of Compilers, Architectures, and Synthesis of Embedded Systems (CASES'08)*, Atlanta, USA, Oct. 2008.
- [19] F. Maraninchi and Y. Rémond. Argos: An automaton-based synchronous language. *Computer Languages*, 27(27):61–92, 2001.
- [20] M. Mendler and M. Pouzet. Uniform and modular composition of data-flow & control-flow in the lazy  $\lambda$ -calculus. Presentation at the International Open Workshop on Synchronous Programming (SYNCHRON'08), Aussois, France, Dec. 2008.
- [21] M. Mendler, R. von Hanxleden, and C. Traulsen. Wcrt algebra and interfaces for esterel-style synchronous processing. In *Proceedings of the Design, Automation and Test in Europe (DATE'09)*, Nice, France, Apr. 2009.
- [22] D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*. Springer, May 2007.
- [23] S. Prochnow, C. Traulsen, and R. von Hanxleden. Synthesizing Safe State Machines from Esterel. In *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, Ottawa, Canada, June 2006.
- [24] S. Prochnow and R. von Hanxleden. Statechart development beyond WYSIWYG. In *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, Nashville, TN, USA, Oct. 2007.
- [25] P. S. Roop, S. Andalam, R. von Hanxleden, S. Yuan, and C. Traulsen. Tight WCRT analysis of synchronous C programs. Technical Report 0912, Christian-Albrechts-Universität Kiel, Department of Computer Science, May 2009.
- [26] P. S. Roop, Z. Salcic, and M. W. S. Dayaratne. Towards Direct Execution of Esterel Programs on Reactive Processors. In *4th ACM International Conference on Embedded Software (EMSOFT 04)*, Pisa, Italy, Sept. 2004.



- [27] M. Samek. *Practical UML Statecharts in C/C++ Event-Driven Programming for Embedded Systems*. Newnes, 2008.
- [28] F. Starke, C. Traulsen, and R. von Hanxleden. Executing Safe State Machines on a reactive processor. Technical Report 0907, Christian-Albrechts-Universität Kiel, Department of Computer Science, Kiel, Germany, Mar. 2009.
- [29] O. Tardieu and S. A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *Proceedings of the Proceedings of the International Conference on Embedded Software (EMSOFT'06)*, Seoul, Korea, Oct. 2006.
- [30] R. von Hanxleden, X. Li, P. Roop, Z. Salcic, and L. H. Yoong. Reactive processing for reactive systems. *ERCIM News*, 66:28–29, Oct. 2006.
- [31] F. Wagner, R. Schmuki, P. Wolstenholme, and T. W. Thomas. *Modeling Software with Finite State Machines: A Practical Approach*. Auerbach Publications, 2006.
- [32] A. Wasowski. On efficient program synthesis from Statecharts. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compilers, and Tools for Embedded Systems (LCTES'03)*, volume 38, issue 7, June 2003. ACM SIGPLAN Notices.
- [33] S. Yuan, S. Andalam, L. H. Yoong, P. S. Roop, and Z. Salcic. STARPro—a new multi-threaded direct execution platform for Esterel. In *Proceedings of Model Driven High-Level Programming of Embedded Systems (SLA++P'08)*, Budapest, Hungary, Apr. 2008.

# Appendix A

## The SC files

The complete SC package consists of two files:

**sc.h** The header file, to be included by each application `<application>.c`

**sc.c** The file including the main program, to be linked with `<application>.o`

This section also includes the Makefile and a run of calling make.

Listing A.1: The header file `sc.h`

```
1 // Definition of SyncChart C macros
2 // Header file included by ksmc.c and <application>.c
3
4 // See http://www.informatik.uni-kiel.de/rtsys/sc/ for further
5 // information, including licensing
6
7 // Release 1.2
8 // Reinhard v. Hanxleden
9 // rvh@informatik.uni-kiel.de
10 // Initial version: 5 March 2009
11 // Current version: 20 May 2009
12
13 #include <stdio.h>
14
15 // =====
16 // Instruction counting/Tracing
17
18 // Check whether externflags has been defined (eg from gcc command line)
19 // If so, suppress tracing and instruction counting
20 // This then results in compact macro-expanded source code and executable
21 #ifndef externflags
22 // Comment the following line out to suppress detailed tracing.
23 #define mytrace
24 #define instrCnt
25 #endif
26
27 // Increment/decrement SC instruction counter
28 // Decrement is needed in some places to avoid duplicate counting
29 #ifdef instrCnt
30 #define instrCntIncr tickInstrCnt++;
31 #define instrCntDecr tickInstrCnt--;
32 #else
33 #define instrCntIncr
34 #define instrCntDecr
35 #endif
36
37
38 // If tracing is turned on, print trace string
39 #ifndef mytrace
40 #define trace0(f) printf(f);
41 #define trace1(f, a) printf(f, a);
42 #define trace2(f, a, b) printf(f, a, b);
43 #define trace3(f, a, b, c) printf(f, a, b, c);
44 #define trace4(f, a, b, c, d) printf(f, a, b, c, d);
45 #else
46
47 #define trace0(f)
48 #define trace1(f, a)
49 #define trace2(f, a, b)
50 #define trace3(f, a, b, c)
51 #define trace4(f, a, b, c, d)
52 #endif
53
54
55 // Count instruction (optionally), print trace string prefix (optionally)
56 // Trace string prefix takes a string s (typically denoting the instruction
57 // ) and identifies the executing thread, both by name and thread id
58 #define traceThread(s) \
59 instrCntIncr \
60 trace4("%-9s%-6s_(id_%d,_prio_%d)_" , s, id2threadname[cid], cid, pr[cid])
61
62
63 // Print trace prefix + suffix
64 // s is string denoting instruction (eg, "PAUSE:")
65 // f is format string for trace suffix
66 // a, b, ... are arguments for format string
67 #define trace0t(s, f) traceThread(s) trace0(f);
68 #define trace1t(s, f, a) traceThread(s) trace1(f, a);
69 #define trace2t(s, f, a, b) traceThread(s) trace2(f, a, b);
70 #define trace3t(s, f, a, b, c) traceThread(s) trace3(f, a, b, c);
71 #define trace4t(s, f, a, b, c, d) traceThread(s) trace4(f, a, b, c, d);
72
73
74
75 // =====
76 // Type definitions
77
78 typedef void *labeltype; // Computed goto -- a la gcc
79 typedef unsigned int bitvector; // 32 bits on IA32
80 typedef bitvector signalvector; // 32 signals on IA32
81 typedef bitvector threadvector; // 32 threads on IA32
82
83
84 // =====
85 // Global variables
86
87 signalvector signals; // Bit mask for signals
88 threadvector enabled; // Bit mask for enabled threads
89 threadvector active; // Bit mask for active threads
90
91 #define idMax 8*sizeof(threadvector) // Number of threads
92
93 int runCnt; // Counts program runs
94 int cid; // Id of current thread
95 int tickCnt; // Counts program ticks
96 int tickInstrCnt; // Instructions in one tick
97 labeltype pc[idMax]; // Pseudo program counters
98 int pr[idMax]; // Priorities
99 threadvector descs[idMax]; // Descendants of thread
100 labeltype TickEndLabel; // Label to stop program
101 labeltype returnAddress; // For function calls (eg Exit Actions)
102
103
104 // =====
105 // Declarations of constants and variables
106
107 // Constants defined in <application>.c
108 int idHi; // Highest thread id in use
109 int runMax; // # of runs to execute
110 int tickMax; // # of ticks to execute
111
112 // Note: the mapping from thread ids to names (labels) is not
```

```

113 // necessarily unique, as we may want to reuse thread ids for threads
114 // that cannot be active concurrently.
115 // We here adopt the convention to use the label of the thread that
116 // has the given id and appears first in the given program.
117 // Note: because of this possible thread id sharing, ids (declared in
118 // <application>.c) is an array, instead of just an enumerated type.
119 extern const char *id2threadname[]; // Names of threads
120 extern const char *s2signame[]; // Names of signals
121
122
123 // =====
124 // Declarations of functions defined in <application>.c:
125
126 // Initialize signals to inputs for one tick
127 void getInputs();
128
129 // Set reference outputs and check valued signals, if there are any
130 int checkOutputs(signalvector *tickOutputs);
131
132 // Print value of a signal, if it has one
133 void printVal(int id);
134
135 // Compute one tick.
136 // Returns 1 if some thread is still active in current tick
137 int tick(int islnit);
138
139 // Functions defined in ksm.p.c
140 void selectCidPrio();
141 void selectCidNoprio();
142
143
144 // =====
145 // Dispatcher
146
147 #ifndef USEPRIO
148 // Version 1: for arbitrary priorities
149 #define dispatch() selectCidPrio(); \
150 goto *pc[cid]
151
152 #elif ((defined __i386__ || defined __amd64__ || defined __x86_64__) &&
153 defined __GNUC__)
154 // Version 2a: all priorities = 0, x86 + gcc available
155 // Use fast Bit Scan Reverse assembler instruction
156 #define dispatch() \
157 __asm volatile (" bsr %1,%0\n" \
158 : "=r" (cid) \
159 : "c" (active) \
160 ); \
161 goto *pc[cid]
162
163 #else
164 // Version 2b: all priorities = 0, x86 + gcc not available
165 #define dispatch() selectCidNoprio(); \
166 goto *pc[cid]
167 #endif
168
169 // =====
170 // Low-level routines
171
172 // Encoding of signal/thread <u> (some non-negative int) in bitvector.
173 // This implementation is fast and simple, BUT limits the max thread
174 // ID and max signal ID to the word width of the machine (eg 32).
175 #define u2b(u) (1 << u)
176
177 // Mapping thread id <id> (of enumeration type idtype) to bitvector.
178 #define id2b(id) u2b(ids[id])
179
180
181 // =====
182 // Keeping track of the thread status
183
184 // Thread enabling/disabling
185 #define enable(id) \
186 enabled |= u2b(id); \
187 active |= u2b(id)
188
189 #define enableNit(id) \
190 enabled = u2b(id); \
191 active = enabled
192
193 #define disable(id) \
194 enabled &= ~u2b(id); \
195 active &= ~u2b(id)
196
197 #define disableSet(idset) \
198 enabled &= ~idset; \
199 active &= ~idset
200

```

```

201 #define isEnabled(id) (enabled & u2b(id))
202
203 #define isEnabledNotOnly(id) (enabled != id2b(id))
204
205 #define isEnabledNoneOf(idset) ((enabled & idset) == 0)
206
207 // Thread (de-)activation
208 #define activate(id) active |= u2b(id)
209
210 #define deactivate(id) active &= ~u2b(id)
211
212 #define isActive(id) (active & u2b(id))
213
214
215 // =====
216 // Tick start and end
217
218 // Start a tick (an instant).
219 // IF this is the initial tick (<islni> is set),
220 // THEN initialize things and continue with following instruction,
221 // ELSE call dispatcher to resume where we left off
222 #define TICKSTART(islni) \
223 freezePreClear \
224 if ( islni ) { \
225 tickCnt = 0; \
226 pc[TickEnd] = &&TickEndLabel; \
227 pr[TickEnd] = 0; \
228 enableNit (ids[TickEnd]); \
229 cid = ids[Main]; \
230 enable( cid ); \
231 setPreNit \
232 setValNit \
233 } else { \
234 active = enabled; \
235 dispatch(); \
236 }
237
238 // Complete a tick.
239 // Return 0 iff computation has terminated
240 #define TICKEND \
241 TickEndLabel: setPre \
242 return isEnabledNotOnly(TickEnd);
243
244
245 // =====
246 // Pausing, suspending, aborting and terminating a thread}
247
248
249 // Pause a thread.
250 // <label> typically points to instruction to be executed after pause
251 // stmt. Therefore, this argument should be superfluous in low-level ISA.
252 // HOWEVER, supplying the label explicitly may save a subsequent GOTO.
253 // For example, <label>: pause(<label>) can be used as shorthand for
254 // HALT, which corresponds to a final, but non-terminating state. (A
255 // terminating state would be encoded by TERM).
256 #define PAUSE(label) { \
257 traceIt("PAUSE:", " pauses, active==_0%\n", active) \
258 pc[cid] = &&label; \
259 deactivate( cid ); \
260 dispatch(); }
261
262
263 // Suspend current thread, continue at label
264 // Note: suspension is implemented by deactivating the current thread
265 // as well as its descendants. This exploits that the PCs of the
266 // descendants
267 // must reside at tick boundaries.
268 #define SUSPEND(label) { \
269 traceIt("SUSPEND:", " suspends itself and descendants_0%\n", descsc[cid]) \
270 active &= ~descsc[cid]; \
271 freezePre \
272 instrCntDecr \
273 PAUSE(label) }
274
275 // Transition to <label>, kill descendant threads (implements abortion)
276 #define TRANS(label) { \
277 disableSet( descsc[ cid ] ); \
278 traceIt("TRANS:", " transfers, enabled==_0%\n", enabled) \
279 goto label; }
280
281
282 // Terminate a thread.
283 // Note: a HALT would be implemented as <label>: PAUSE(<label>)
284 #define TERM { \
285 disable( cid ); \
286 traceIt("TERM:", " terminates, enabled==_0%\n", enabled) \
287 dispatch(); }

```

```

288 // =====
289 // Handling concurrency
290 // =====
291 // Spawn a thread of priority <p>, starting at <label>, with id <id>
292 #define PAR(p, label, id) {
293     trace3t("PAR:", "forks_%s_(%d)_with_prio_%d\n", id2threadname[id], id, p)
294     pc[id] = &&label;
295     pr[id] = p;
296     enable(id); }
297 // Denote parent thread, starting at <label>
298 // Current thread gets priority <p>, continues at <label>, has descendants
299 // <d>
300 // Descendants are used
301 // - to check for termination (with JOIN)
302 // - to be disabled upon abortion (with TRANS)
303 #define PARE(p, label, d) {
304     trace1t("PARE:", "has_descendants...0%o\n", d)
305     pc[cid] = &&label;
306     pr[cid] = p;
307     descsc[cid] = d;
308     dispatch(); }
309 // Join completed child threads.
310 // IF all descendants have terminated,
311 // THEN jump to <thenlabel>,
312 // ELSE jump to <elselabel>
313 #define JOIN(thenlabel, elselabel) {
314     trace1t("JOIN:", "%s\n",
315         isEnabledNoneOf(descsc[cid]) ? "joins" : "does_not_join")
316     if (isEnabledNoneOf(descsc[cid]))
317         goto thenlabel;
318     instrCntDecr
319     PAUSE(elselabel); }
320 // Set priority of a thread.
321 // <label> points to instruction after prio stmt
322 // This argument should be superfluous in low-level ISA
323 #define PRIO(p, label) {
324     trace1t("PRIO:", "set_to_ priority _%d\n", p)
325     pr[cid] = p;
326     pc[cid] = &&label;
327     dispatch(); }
328 // =====
329 // Efficient shorthands for thread handling
330 // =====
331 // Set a priority, then pause (this sets "prionext").
332 // The is a more efficient alternative to
333 // PRIO(p, label1); label1: PAUSE(label)
334 // as no context switch is needed immediately before the PAUSE
335 #define PPAUSE(p, label) {
336     trace1t("PPAUSE:", "sets_prio_to_%d\n", p)
337     pr[cid] = p;
338     instrCntDecr
339     PAUSE(label) }
340 // IF all descendants have terminated,
341 // THEN jump to <thenlabel>,
342 // ELSE set priority, pause, and continue at <elselabel>
343 // The is a more efficient alternative to
344 // JOIN(thenlabel, label1); label1: PPAUSE(p, elselabel)
345 // as no context switch is needed immediately before the PAUSE
346 #define JPPAUSE(p, thenlabel, elselabel) {
347     trace1t("JPPAUSE:", "%s\n",
348         isEnabledNoneOf(descsc[cid]) ? "joins" : "does_not_join")
349     if (isEnabledNoneOf(descsc[cid]))
350         goto thenlabel;
351     instrCntDecr
352     PPAUSE(p, elselabel) }
353 // =====
354 // Signal initialization, emission and testing
355 // =====
356 // Initialize a local signal (handles reincarnation)
357 #define SIGNAL(s) {
358     trace2t("SIGNAL:", "initializes_%s_(%d)\n", s2signame[s], s)
359     signals &= ~u2b(s); }
360 // Emission of a pure signal <s>
361 #define EMIT(s) {
362     trace2t("EMIT:", "emits_%s_(%d)\n", s2signame[s], s)
363     signals |= u2b(s); }
364 // Test for presence of signal.
365 // IF <s> is present,
366 // THEN proceed to next instruction,
367 // ELSE jump to <label>
368 #define PRESENT(s, label) {
369     trace3t("PRESENT:", "determines_%s_(%d)_as_%s\n",
370         s2signame[s], s, (signals & u2b(s)) ? "present" : "absent")
371     if (!(signals & u2b(s)))
372         goto label; }
373 // =====
374 // Handling valued signals
375 // The following is compiled conditionally depending on valSigIntCnt
376 // <application>.c must define valSigIntCnt if valued signals are used
377 #ifdef valSigIntCnt
378     int i; // Counter for looping through valued sigs
379 // At beginning of initial tick:
380 // Initialize valued signals (-1 is for "undefined")
381 #define setValInit
382     for (i = 0; i < valSigIntCnt; i++)
383         valSigInt[i] = -1;
384 #else // #ifdef valSigIntCnt
385 #define setValInit
386 #endif
387 // Emission of a valued signal <s>, type integer
388 #define EMITINT(s, val) {
389     valSigInt[s] = val;
390     trace3t("EMITInt:", "emits_%s_(%d)_value_%d\n",
391         s2signame[s], s, val)
392     signals |= u2b(s); }
393 // Emission of a valued signal <s>, type integer, combined with *
394 #define EMITINTMUL(s, val) {
395     valSigIntMult[s] *= val;
396     trace4t("EMITInt*:", "emits_%s_(%d)_value_%d_result_%d\n",
397         s2signame[s], s, val, valSigIntMult[s])
398     signals |= u2b(s); }
399 // Retrieve value of signal <s> into <reg>
400 #define VAL(s, reg) {
401     trace3t("VAL:", "determines_value_of_%s_(%d)_as_%d\n",
402         s2signame[s], s, valSigInt[s])
403     reg = valSigInt[s]; }
404 // =====
405 // Handling PRE
406 // The following is compiled conditionally depending on usePRE
407 // <application>.c must define usePRE if PRE is used
408 #ifdef usePRE
409     signalvector sigsPre; // Signals from previous tick
410     signalvector sigsFreeze; // Signals that are frozen, due to suspension
411 // At beginning of initial tick:
412 // Initialize previous signals
413 #define setPreInit
414     sigsPre = 0;
415     setPreValInit;
416 // At end of tick:
417 // Copy current signals (unless frozen) to previous signals
418 #define setPre
419     sigsPre = (sigsPre & sigsFreeze) | (signals & ~sigsFreeze);
420     setPreVal
421 // When suspending current thread:
422 // Add signals local to current thread or its descendants
423 // to list of signals to freeze
424 #define freezePre
425     sigsFreeze |= sigsDescs[cid];
426 // At beginning of tick:
427 // Clear list of signals to freeze
428 #define freezePreClear
429     sigsFreeze = 0;

```

```

464 #else // #ifndef usePRE
465 #define setPreInit
466 #define setPre
467 #define freezePre
468 #define freezePreClear
469 #endif // #ifndef usePRE
470
471 // Test for presence of signal in previous tick.
472 // IF <s> was present in previous tick,
473 // THEN proceed to next instruction,
474 // ELSE jump to <label>
475 // #define PRESENTPRE(s, label) {
476 //     trace3t("PRESENTPRE:", "determines_previous_value_of_%s_(%d)_as_%s\n",
477 //           s2signame[s], s, (sigsPre & u2b(s)) ? "present" : "absent")
478 //     if (!(sigsPre & u2b(s)))
479 //         goto label; }
480
481
482 // =====
483 // Handling valued signals in conjunction with PRE
484
485 #ifndef usePRE
486 #define valSigIntCnt
487 // At beginning of initial tick:
488 // Initialize previous signal values
489 #define setPreValInit
490     for (i = 0; i < valSigIntCnt; i++)
491         valSigIntPre[i] = -1;
492
493 // At end of tick:
494 // Copy values of current signals (unless frozen) to previous signals
495 #define setPreVal
496     for (i = 0; i < valSigIntCnt; i++)
497         if (!(sigsFreeze & u2b(i)))
498             valSigIntPre[i] = valSigInt[i];
499
500 #else // #ifndef valSigIntCnt
501 #define setPreValInit
502 #define setPreVal
503 #endif // #ifndef valSigIntCnt
504 #endif // #ifndef usePRE
505
506 // Retrieve previous value of signal <s> into <reg>
507 #define VALPRE(s, reg) {
508     trace3t("VALPRE:", "determines_value_of_%s_(%d)_as_%d\n",
509           s2signame[s], s, valSigIntPre[s])
510     reg = valSigIntPre[s]; }
511
512 // =====
513 // Control flow: jumps
514
515 // Just a goto that also gets counted as instruction
516 #define GOTO(label) {
517     instrCntIncr
518     goto label; }
519
520 // =====
521 // Support for Exit Actions
522
523 // IF thread <id> is active and at state <statelabel>,
524 // THEN proceed to next instruction,
525 // ELSE jump to <label>
526 // Can use this if an Exit Action _may_ have to be performed
527 #define ISAT(id, statelabel, label) {
528     if (isEnabled(id) && (pc[id] == &&statelabel)) {
529         trace0t("ISAT:", "is_at_probed_label\n")
530     } else {
531         trace0t("ISAT:", "is_not_at_probed_label\n")
532         goto label;
533     }
534 }
535
536 // Call a function at <label>, return to <retlabel>
537 // Use this if an Exit Action _must_ be performed
538 #define CALL(label, retlabel) {
539     trace0t("CALL:", "calls_function\n")
540     returnAddress = &&retlabel;
541     goto label; }
542
543 // Return from a function call
544 #define RET {
545     trace0t("RET:", "returns_from_function\n")
546     goto *returnAddress; }
547
548 // Conditionally call a function.
549 // IF thread <id> is active and at state <statelabel>,
550 // THEN call function at <label>;
551 // Return to <retlabel>
552 // Use this if an Exit Action _may_ have to be performed
553 // Shorthand for ISAT(id, statelabel, retlabel); CALL(label, retlabel);
554 #define ISATCALL(id, statelabel, label, retlabel) {
555     if (isEnabled(id) && (pc[id] == &&statelabel)) {
556         trace0t("ISATCALL:", "does_call_function\n")
557         returnAddress = &&retlabel;
558         goto label;
559     }
560     trace0t("ISATCALL:", "does_not_call_function\n")
561     goto retlabel; }
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

Listing A.2: The main program file sc.c

## Listing A.3: The Makefile

```

69     id++;
70 }
71
72 if ( first ) {
73     printf (" <none>");
74 }
75
76 printf ("%s", suffix);
77 #endif
78 }
79
80 // =====
81 // The main program
82 // Returns 0 iff outputs generated by program match reference trace
83 int main()
84 {
85     int runInstrCnt; // Instructions in one run
86     int runInstrCnt = 0; // Instructions accumulated over all runs
87     int outputsOK = 1; // Outputs of simulation correct?
88     int notDone; // Current run not done yet?
89     int init; // Is initial tick?
90     signalvector tickInputs; // Input values for a tick
91     signalvector tickOutputs; // Reference output values for a tick
92     signalvector tickSignals; // Reference signal values for a tick
93
94 // Execute all runs
95 for (runCnt = 0; (runCnt < runMax) && outputsOK; runCnt++) {
96     printf ("#####_RUN_%d_STARTS_#####\n",
97         runCnt);
98
99     runInstrCnt = 0;
100    tickCnt = 0;
101    init = 1;
102    enabled = 0;
103
104    do { // Execute all ticks of one run
105        tickInstrCnt = 0;
106        getInputs();
107        tickInputs = signals;
108
109        trace3("====_TICK_%d_STARTS,_inputs_=%0o,_enabled_=%0o\n",
110            tickCnt, tickInputs, enabled);
111        vec2names("====_Inputs:_", tickInputs, s2signame, "\n");
112        vec2names("====_Enabled:_", enabled, id2threadname, "\n");
113
114        notDone = tick(init); // Call automaton function
115        init = 0;
116
117        runInstrCnt += tickInstrCnt;
118        trace3("====_TICK_%d_terminates_after_%d_instructions,_enabled_=_",
119            0%o.\n",
120            tickCnt, tickInstrCnt, enabled);
121        vec2names("====_Resulting_signals:_", signals, s2signame, "");
122        outputsOK = checkOutputs(&tickOutputs);
123        if (outputsOK) {
124            tickSignals = tickInputs | tickOutputs;
125            if (signals == tickSignals) {
126                trace0(" ,_Outputs_OK.\n\n");
127            } else {
128                vec2names(" ,_Outputs_NOT_OK_--_expected_signals_",
129                    tickSignals, s2signame, "!\n\n");
130            }
131            outputsOK = 0;
132        } else {
133            notDone = 0;
134        }
135
136        tickCnt++;
137        if (tickCnt >= tickMax) {
138            printf ("====_Executed_tickMax_=%d_ticks!\n", tickMax);
139            notDone = 0;
140        }
141    } while (notDone && outputsOK);
142
143    printf ("#####_RUN_%d_terminates_after_%d_instructions\n\n",
144        runCnt, runInstrCnt);
145    runInstrCnt += runInstrCnt;
146 };
147
148 printf ("#####_All_runs_terminate_after_%d_instructions\n\n",
149     runInstrCnt);
150 return !outputsOK;
151 }

```

```

1 progs := ABRO Count2Suspend Exits Exits-no-isatcall Exits-inlined
2     FilteredSR grcbal3 \
3     PreAndSuspend PrimeFactor Reincarnation Shifter3 SurfDepth PCO
4
5 downloads := Makefile make.trace sc.c sc.h $(progs:=.c) $(progs:=.out)
6
7 CCFLAGS := -Wall
8
9 all : $(progs:=.out)
10
11 allprogs : $(progs)
12
13 PCO: PCO.c sc.h Makefile
14     gcc $(CCFLAGS) PCO.c -o PCO
15
16 # Want to compile ABRO-C without tracing, to not interfere with kbd input
17 ABRO-C: ABRO-C.c sc.c sc.h Makefile
18     gcc $(CCFLAGS) -D externflags -D instrCnt ABRO-C.c sc.c -o $@
19
20 %.out2: %.c sc.c sc.h Makefile
21     gcc $(CCFLAGS) $*.c sc.c -o $*
22     $* > $@
23
24 %.out: %.c
25     $* > $@
26
27 %-expanded.c: %.c
28     gcc $(CCFLAGS) -D externflags -E $*.c > $@
29
30 %-expanded-flags.c: %.c
31     gcc $(CCFLAGS) -E $*.c > $@
32
33 %.s: %.c sc.h Makefile
34     gcc $(CCFLAGS) -D externflags -O3 -S $*.c
35
36 %-unopt.s: %.c sc.h Makefile
37     gcc $(CCFLAGS) -o $@ -D externflags -S $*.c
38
39 %.o: %.c sc.h Makefile
40     gcc $(CCFLAGS) -D externflags -O3 -c -o $*.o $*.c
41
42 %.asm: %.o
43     #objdump -d $*.o > $@
44     otool -tv $*.o > $@
45
46 %-linked.asm: %.c sc.c sc.h Makefile
47     gcc $(CCFLAGS) -D externflags -O3 $*.c sc.c -o $*
48     #objdump -d $*.exe > $@
49     otool -tv $* > $@
50
51 %: %.c sc.c sc.h Makefile
52     gcc $(CCFLAGS) $*.c sc.c -o $*
53
54 make.trace:
55     (time make) >& $@
56
57 # Example to match either or expression :
58 # grep -E 'trace\|' sc.h
59 %.stats:
60     echo "Line_count_of_$*:"
61     wc $*
62     echo "Comment_line_count_of_$*:"
63     grep -c "^_*/" $*
64     echo "Empty_line_count_of_$*:"
65     grep -c "$$" $*
66     echo "Trace_related_line_count_of_$*_discouting_multi-line_trace_
67     commands_(9),_counting_again_comments_(4):"
68     grep -c "trace" $*
69
70 sc.tar.gz: $(downloads)
71     tar -cf sc.tar $(downloads)
72     gzip -f sc.tar
73     ls -l $@
74
75 BIBLIO_REMOTEPATH=biblio@rtsys.informatik.uni-kiel.de:/home/biblio/
76     public.html/downloads
77
78 %p: %
79     scp $* $(BIBLIO_REMOTEPATH)/
80
81 clean:
82     -rm *~ *-expanded.c *.stackdump *.o
83
84 realclean: clean
85     -rm *.exe *.out

```

#### Listing A.4: make.trace: a run of make

```
1 | make[1]: Entering directory '/cygdrive/c/Dokumente und Einstellungen/rvh/  
  | Eigene Dateien/shared/papers/scc'  
2 | gcc abro.c scc.c -o abro  
3 | abro > abro.out  
4 | gcc count2suspend.c scc.c -o count2suspend  
5 | count2suspend > count2suspend.out  
6 | gcc exits.c scc.c -o exits  
7 | exits > exits.out  
8 | gcc filteredSR.c scc.c -o filteredSR  
9 | filteredSR > filteredSR.out  
10 | gcc grcbal3.c scc.c -o grcbal3  
11 | grcbal3 > grcbal3.out  
12 | gcc preAndSuspend.c scc.c -o preAndSuspend  
13 | preAndSuspend > preAndSuspend.out  
14 | gcc primeFactor.c scc.c -o primeFactor  
15 | primeFactor > primeFactor.out  
16 | gcc reincarnation.c scc.c -o reincarnation  
17 | reincarnation > reincarnation.out  
18 | gcc shifter3.c scc.c -o shifter3  
19 | shifter3 > shifter3.out  
20 | make[1]: Leaving directory '/cygdrive/c/Dokumente und Einstellungen/rvh/  
  | Eigene Dateien/shared/papers/scc'  
21 |  
22 | real    0m5.718s  
23 | user    0m3.210s  
24 | sys     0m1.690s
```

# Appendix B

## Complete Examples

### B.1 ABRO

Listing B.1: ABRO.c

```

1 // ABRO — the "hello world" for SSMs
2 // Example from Charles Andr, Semantics of SyncCharts,
3 // ISRN I3S/RR—2003—24—FR, April 2003, Figure 5—12
4 //
5 // rvh, 17 mar 2009
6 #include "sc.h"
7
8 #define RUNMAX 2 // # of runs to execute
9 #define TICKMAX 5 // # of ticks to execute
10
11 int runMax = RUNMAX; // # of runs to execute
12 int tickMax = TICKMAX; // # of ticks to execute
13
14 // =====
15 // Program—specific definitions
16
17 // Signals
18 typedef enum {A, B, R, O} signaltyp;
19 const char *s2signame[] = {"A", "B", "R", "O"};
20
21 // Thread ids
22 // Note: WaitA gets a higher id than WaitB (rather than the other way
23 // around) simply to let WaitA execute first, to make the trace match
24 // the syntactical flow of the program
25 int idHi = 4; // Highest thread id in use
26 typedef enum { TickEnd, AB, WaitB, WaitA, Main } idtype;
27 const int ids[] = {0, 1, 2, 3, 4};
28 const char *id2threadname[] = {"TickEnd", "AB", "WaitB", "WaitA", "Main"};
29
30 // Inputs for RUNMAX runs of TICKMAX ticks
31 signalvector inputs[RUNMAX][TICKMAX] =
32 { {0, u2b(A), u2b(B), u2b(R), 0},
33 {u2b(A) | u2b(B), u2b(A) | u2b(B), 0, u2b(R), u2b(A) | u2b(B) | u2b(R)} };
34
35 // Expected outputs
36 signalvector outputs[RUNMAX][TICKMAX] =
37 { {0, 0, u2b(O), 0, 0},
38 {0, u2b(O), 0, 0, 0} };
39
40 void getInputs()
41 {
42 signals = inputs[runCnt][tickCnt];
43 }
44
45 // Set reference outputs and check valued signals, if there are any.
46 // Return 1 unless valued signal outputs are wrong.
47 // No valued signals here, therefore always return 1.
48 int checkOutputs(signalvector *tickOutputs)
49 {
50 *tickOutputs = outputs[runCnt][tickCnt];
51 return 1;
52 }
53
54 // No valued signals to print
55 void printVal(int id)
56 {
57 }
58
59 // Returns 1 if some thread is still active in current tick

```

```

61 // Note: No JOIN on thread embedded in Main
62 int tick(int islnit)
63 {
64 // Thread ids: AB=1, WaitB=2, WaitA=3, Main=4
65 TICKSTART(islnit);
66
67 ABO: PAR(0, AB, ids[AB]);
68 PARE(0, ABomain, id2b(AB) | id2b(WaitA) | id2b(WaitB));
69
70 AB: PAR(0, WaitA, ids[WaitA]);
71 PAR(0, WaitB, ids[WaitB]);
72 PARE(0, ABmain, id2b(WaitA) | id2b(WaitB));
73
74 WaitA: PAUSE(L0);
75 L0: PRESENT(A, WaitA);
76 TERM;
77
78 WaitB: PAUSE(L1);
79 L1: PRESENT(B, WaitB);
80 TERM;
81
82 ABmain: JOIN(Done, ABmain);
83 Done: EMIT(O);
84 TERM;
85
86 ABomain: PAUSE(L2);
87 L2: PRESENT(R, ABomain);
88 TRANS(ABO);
89
90 TICKEND;
91 }
92
93 // Local Variables :
94 // compile—command: "make ABRO; ABRO"
95 // End:

```

Listing B.2: ABRO.out

```

1 ##### RUN 0 STARTS #####
2 ===== TICK 0 STARTS, inputs = 00, enabled = 00
3 ===== Inputs: <none>
4 ===== Enabled: <none>
5 PAR: Main (id 4, prio 0) forks AB (1) with prio 0
6 PARE: Main (id 4, prio 0) has descendants 016
7 PAUSE: Main (id 4, prio 0) pauses, active = 023
8 PAR: AB (id 1, prio 0) forks WaitA (3) with prio 0
9 PAR: AB (id 1, prio 0) forks WaitB (2) with prio 0
10 PARE: AB (id 1, prio 0) has descendants 014
11 PAUSE: WaitA (id 3, prio 0) pauses, active = 017
12 PAUSE: WaitB (id 2, prio 0) pauses, active = 07
13 JOIN: AB (id 1, prio 0) does not join
14 PAUSE: AB (id 1, prio 0) pauses, active = 03
15 ===== TICK 0 terminates after 9 instructions, enabled = 037.
16 ===== Resulting signals: <none>, Outputs OK.
17
18 ===== TICK 1 STARTS, inputs = 01, enabled = 037
19 ===== Inputs: A (0)
20 ===== Enabled: TickEnd (0), AB (1), WaitB (2), WaitA (3), Main (4)
21 PRESENT: Main (id 4, prio 0) determines R (2) as absent
22 PAUSE: Main (id 4, prio 0) pauses, active = 037
23 PRESENT: WaitA (id 3, prio 0) determines A (0) as present
24 TERM: WaitA (id 3, prio 0) terminates, enabled = 027
25 PRESENT: WaitB (id 2, prio 0) determines B (1) as absent
26 PAUSE: WaitB (id 2, prio 0) pauses, active = 07
27 JOIN: AB (id 1, prio 0) does not join
28 PAUSE: AB (id 1, prio 0) pauses, active = 03

```



```

29 ===== TICK 1 terminates after 7 instructions, enabled = 027.
30 ===== Resulting signals: A (0), Outputs OK.
31
32 ===== TICK 2 STARTS, inputs = 02, enabled = 027
33 ===== Inputs: B (1)
34 ===== Enabled: TickEnd (0), AB (1), WaitB (2), Main (4)
35 PRESENT: Main (id 4, prio 0) determines R (2) as absent
36 PAUSE: Main (id 4, prio 0) pauses, active = 027
37 PRESENT: WaitB (id 2, prio 0) determines B (1) as present
38 TERM: WaitB (id 2, prio 0) terminates, enabled = 023
39 JOIN: AB (id 1, prio 0) joins
40 EMIT: AB (id 1, prio 0) emits O (3)
41 TERM: AB (id 1, prio 0) terminates, enabled = 021
42 ===== TICK 2 terminates after 7 instructions, enabled = 021.
43 ===== Resulting signals: B (1), O (3), Outputs OK.
44
45 ===== TICK 3 STARTS, inputs = 04, enabled = 021
46 ===== Inputs: R (2)
47 ===== Enabled: TickEnd (0), Main (4)
48 PRESENT: Main (id 4, prio 0) determines R (2) as present
49 TRANS: Main (id 4, prio 0) transfers, enabled = 021
50 PAR: Main (id 4, prio 0) forks AB (1) with prio 0
51 PARE: Main (id 4, prio 0) has descendants 016
52 PAUSE: Main (id 4, prio 0) pauses, active = 023
53 PAR: AB (id 1, prio 0) forks WaitA (3) with prio 0
54 PAR: AB (id 1, prio 0) forks WaitB (2) with prio 0
55 PARE: AB (id 1, prio 0) has descendants 014
56 PAUSE: WaitA (id 3, prio 0) pauses, active = 017
57 PAUSE: WaitB (id 2, prio 0) pauses, active = 07
58 JOIN: AB (id 1, prio 0) does not join
59 PAUSE: AB (id 1, prio 0) pauses, active = 03
60 ===== TICK 3 terminates after 11 instructions, enabled = 037.
61 ===== Resulting signals: R (2), Outputs OK.
62
63 ===== TICK 4 STARTS, inputs = 00, enabled = 037
64 ===== Inputs: <none>
65 ===== Enabled: TickEnd (0), AB (1), WaitB (2), WaitA (3), Main (4)
66 PRESENT: Main (id 4, prio 0) determines R (2) as absent
67 PAUSE: Main (id 4, prio 0) pauses, active = 037
68 PRESENT: WaitA (id 3, prio 0) determines A (0) as absent
69 PAUSE: WaitA (id 3, prio 0) pauses, active = 017
70 PRESENT: WaitB (id 2, prio 0) determines B (1) as absent
71 PAUSE: WaitB (id 2, prio 0) pauses, active = 07
72 JOIN: AB (id 1, prio 0) does not join
73 PAUSE: AB (id 1, prio 0) pauses, active = 03
74 ===== TICK 4 terminates after 7 instructions, enabled = 037.
75 ===== Resulting signals: <none>, Outputs OK.
76
77 ===== Executed tickMax = 5 ticks!
78 ##### RUN 0 terminates after 41 instructions
79
80 ##### RUN 1 STARTS #####
81 ===== TICK 0 STARTS, inputs = 03, enabled = 00
82 ===== Inputs: A (0), B (1)
83 ===== Enabled: <none>
84 PAR: Main (id 4, prio 0) forks AB (1) with prio 0
85 PARE: Main (id 4, prio 0) has descendants 016
86 PAUSE: Main (id 4, prio 0) pauses, active = 023
87 PAR: AB (id 1, prio 0) forks WaitA (3) with prio 0
88 PAR: AB (id 1, prio 0) forks WaitB (2) with prio 0
89 PARE: AB (id 1, prio 0) has descendants 014
90 PAUSE: WaitA (id 3, prio 0) pauses, active = 017
91 PAUSE: WaitB (id 2, prio 0) pauses, active = 07
92 JOIN: AB (id 1, prio 0) does not join
93 PAUSE: AB (id 1, prio 0) pauses, active = 03
94 ===== TICK 0 terminates after 9 instructions, enabled = 037.
95 ===== Resulting signals: A (0), B (1), Outputs OK.
96
97 ===== TICK 1 STARTS, inputs = 03, enabled = 037
98 ===== Inputs: A (0), B (1)
99 ===== Enabled: TickEnd (0), AB (1), WaitB (2), WaitA (3), Main (4)
100 PRESENT: Main (id 4, prio 0) determines R (2) as absent
101 PAUSE: Main (id 4, prio 0) pauses, active = 037
102 PRESENT: WaitA (id 3, prio 0) determines A (0) as present
103 TERM: WaitA (id 3, prio 0) terminates, enabled = 027
104 PRESENT: WaitB (id 2, prio 0) determines B (1) as present
105 TERM: WaitB (id 2, prio 0) terminates, enabled = 023
106 JOIN: AB (id 1, prio 0) joins
107 EMIT: AB (id 1, prio 0) emits O (3)
108 TERM: AB (id 1, prio 0) terminates, enabled = 021
109 ===== TICK 1 terminates after 9 instructions, enabled = 021.
110 ===== Resulting signals: A (0), B (1), O (3), Outputs OK.
111
112 ===== TICK 2 STARTS, inputs = 00, enabled = 021
113 ===== Inputs: <none>
114 ===== Enabled: TickEnd (0), Main (4)
115 PRESENT: Main (id 4, prio 0) determines R (2) as absent
116 PAUSE: Main (id 4, prio 0) pauses, active = 021
117 ===== TICK 2 terminates after 2 instructions, enabled = 021.

```

```

118 ===== Resulting signals: <none>, Outputs OK.
119
120 ===== TICK 3 STARTS, inputs = 04, enabled = 021
121 ===== Inputs: R (2)
122 ===== Enabled: TickEnd (0), Main (4)
123 PRESENT: Main (id 4, prio 0) determines R (2) as present
124 TRANS: Main (id 4, prio 0) transfers, enabled = 021
125 PAR: Main (id 4, prio 0) forks AB (1) with prio 0
126 PARE: Main (id 4, prio 0) has descendants 016
127 PAUSE: Main (id 4, prio 0) pauses, active = 023
128 PAR: AB (id 1, prio 0) forks WaitA (3) with prio 0
129 PAR: AB (id 1, prio 0) forks WaitB (2) with prio 0
130 PARE: AB (id 1, prio 0) has descendants 014
131 PAUSE: WaitA (id 3, prio 0) pauses, active = 017
132 PAUSE: WaitB (id 2, prio 0) pauses, active = 07
133 JOIN: AB (id 1, prio 0) does not join
134 PAUSE: AB (id 1, prio 0) pauses, active = 03
135 ===== TICK 3 terminates after 11 instructions, enabled = 037.
136 ===== Resulting signals: R (2), Outputs OK.
137
138 ===== TICK 4 STARTS, inputs = 07, enabled = 037
139 ===== Inputs: A (0), B (1), R (2)
140 ===== Enabled: TickEnd (0), AB (1), WaitB (2), WaitA (3), Main (4)
141 PRESENT: Main (id 4, prio 0) determines R (2) as present
142 TRANS: Main (id 4, prio 0) transfers, enabled = 021
143 PAR: Main (id 4, prio 0) forks AB (1) with prio 0
144 PARE: Main (id 4, prio 0) has descendants 016
145 PAUSE: Main (id 4, prio 0) pauses, active = 023
146 PAR: AB (id 1, prio 0) forks WaitA (3) with prio 0
147 PAR: AB (id 1, prio 0) forks WaitB (2) with prio 0
148 PARE: AB (id 1, prio 0) has descendants 014
149 PAUSE: WaitA (id 3, prio 0) pauses, active = 017
150 PAUSE: WaitB (id 2, prio 0) pauses, active = 07
151 JOIN: AB (id 1, prio 0) does not join
152 PAUSE: AB (id 1, prio 0) pauses, active = 03
153 ===== TICK 4 terminates after 11 instructions, enabled = 037.
154 ===== Resulting signals: A (0), B (1), R (2), Outputs OK.
155
156 ===== Executed tickMax = 5 ticks!
157 ##### RUN 1 terminates after 42 instructions
158
159 ##### All runs terminate, after 83 instructions

```

Listing B.3: Assembler generated from ABRO tick function (see Fig. 3.3b and Fig. 3.3.4) without optimizations (plain gcc), before linking

```

1  .tick :
2      pushl   %ebp
3      movl   %esp, %ebp
4      pushl   %esi
5      pushl   %ebx
6      subl   $16, %esp
7      call   L31
8      ".L0000000003$pb":
9  L31:
10     popl   %ebx
11     cmpl   $0, 8(%ebp)
12     je     L10
13     leal  L_tickCnt$non_Jazy_ptr - ".L0000000003$pb"(%ebx), %eax
14     movl  (%eax), %eax
15     movl  $0, (%eax)
16     leal  L_pc$non_Jazy_ptr - ".L0000000003$pb"(%ebx), %eax
17     movl  (%eax), %eax
18     leal  L12 - ".L0000000003$pb"(%ebx), %edx
19     movl  %edx, (%eax)
20     leal  L_pr$non_Jazy_ptr - ".L0000000003$pb"(%ebx), %eax
21     movl  (%eax), %eax
22     movl  $0, (%eax)
23     leal  _ids - ".L0000000003$pb"(%ebx), %eax
24     movl  (%eax), %ecx
25     movl  $1, %eax
26     sall  %cl, %eax
27     movl  %eax, %edx
28     leal  L_enabled$non_Jazy_ptr - ".L0000000003$pb"(%ebx), %eax
29     movl  (%eax), %eax
30     movl  %edx, (%eax)
31     leal  L_enabled$non_Jazy_ptr - ".L0000000003$pb"(%ebx), %eax
32     movl  (%eax), %eax
33     movl  (%eax), %edx
34     leal  L_active$non_Jazy_ptr - ".L0000000003$pb"(%ebx), %eax
35     movl  (%eax), %eax
36     movl  %edx, (%eax)
37     leal  _ids - ".L0000000003$pb"(%ebx), %eax

```

38	movl	16(%eax), %edx	127	movl	\$1, %eax
39	leal	L_cid\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax	128	sall	%cl, %eax
40	movl	(%eax), %eax	129	orl	%eax, %edx
41	movl	%edx, (%eax)	130	leal	_ids - "L0000000003\$pb"(%ebx), %eax
42	leal	L_cid\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax	131	movl	8(%eax), %ecx
43	movl	(%eax), %eax	132	movl	\$1, %eax
44	movl	(%eax), %ecx	133	sall	%cl, %eax
45	movl	\$1, %eax	134	orl	%edx, %eax
46	sall	%cl, %eax	135	movl	%eax, %edx
47	movl	%eax, %edx	136	leal	L_descs\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax
48	leal	L_enabled\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax	137	movl	(%eax), %eax
49	movl	(%eax), %eax	138	movl	%edx, (%eax,%esi,4)
50	movl	(%eax), %eax	139	leal	L_active\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax
51	orl	%eax, %edx	140	movl	(%eax), %eax
52	leal	L_enabled\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax	141	movl	(%eax), %ecx
53	movl	(%eax), %eax	142	bsrl	%ecx, %edx
54	movl	%edx, (%eax)	143		
55	leal	L_cid\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax	144	leal	L_cid\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax
56	movl	(%eax), %eax	145	movl	(%eax), %eax
57	movl	(%eax), %ecx	146	movl	%edx, (%eax)
58	movl	\$1, %eax	147	leal	L_cid\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax
59	sall	%cl, %eax	148	movl	(%eax), %eax
60	movl	%eax, %edx	149	movl	(%eax), %edx
61	leal	L_active\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax	150	leal	L_pc\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax
62	movl	(%eax), %eax	151	movl	(%eax), %eax
63	movl	(%eax), %eax	152	movl	(%eax,%edx,4), %eax
64	orl	%eax, %edx	153	movl	%eax, -12(%ebp)
65	leal	L_active\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax	154	jmp	L29
66	movl	(%eax), %eax	155	L10:	
67	movl	%edx, (%eax)	156	leal	L_enabled\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax
68			157	movl	(%eax), %eax
69	L13:	leal	158	movl	(%eax), %edx
70		_ids - "L0000000003\$pb"(%ebx), %eax	159	leal	L_active\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax
71		4(%eax), %ecx	160	movl	(%eax), %eax
72		L_pc\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax	161	movl	%edx, (%eax)
73		(%eax), %eax	162	leal	L_active\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax
74		L15 - "L0000000003\$pb"(%ebx), %edx	163	movl	(%eax), %eax
75		%edx, (%eax,%ecx,4)	164	movl	(%eax), %ecx
76		leal	165	bsrl	%ecx, %edx
77		_ids - "L0000000003\$pb"(%ebx), %eax	166		
78		4(%eax), %edx	167	leal	L_cid\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax
79		leal	168	movl	(%eax), %eax
80		L_pr\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax	169	movl	%edx, (%eax)
81		(%eax), %eax	170	leal	L_cid\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax
82		\$0, (%eax,%edx,4)	171	movl	(%eax), %eax
83		leal	172	movl	(%eax), %edx
84		_ids - "L0000000003\$pb"(%ebx), %eax	173	leal	L_pc\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax
85		4(%eax), %ecx	174	movl	(%eax), %eax
86		\$1, %eax	175	movl	(%eax,%edx,4), %eax
87		sall	176	movl	%eax, -12(%ebp)
88		%cl, %eax	177	jmp	L29
89		movl	178	L14:	
90		%eax, %edx	179	jmp	L30
91		leal	180	L29:	
92		L_enabled\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax	181	L30:	
93		(%eax), %eax	182	jmp	* - 12(%ebp)
94		leal	183	L15:	
95		_ids - "L0000000003\$pb"(%ebx), %eax	184	leal	_ids - "L0000000003\$pb"(%ebx), %eax
96		4(%eax), %ecx	185	movl	12(%eax), %ecx
97		\$1, %eax	186	leal	L_pc\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax
98		sall	187	movl	(%eax), %eax
99		%cl, %eax	188	leal	L17 - "L0000000003\$pb"(%ebx), %edx
100		movl	189	movl	%edx, (%eax,%ecx,4)
101		%eax, %edx	190	leal	_ids - "L0000000003\$pb"(%ebx), %eax
102		leal	191	movl	12(%eax), %edx
103		L_active\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax	192	leal	L_pr\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax
104		(%eax), %eax	193	movl	(%eax), %eax
105		leal	194	movl	\$0, (%eax,%edx,4)
106		L_cid\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax	195	leal	_ids - "L0000000003\$pb"(%ebx), %eax
107		(%eax), %eax	196	movl	12(%eax), %ecx
108		leal	197	movl	\$1, %eax
109		L16 - "L0000000003\$pb"(%ebx), %edx	198	sall	%cl, %eax
110		(%eax), %eax	199	movl	%eax, %edx
111		%edx, (%eax,%ecx,4)	200	leal	L_enabled\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax
112		leal	201	movl	(%eax), %eax
113		L_cid\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax	202	movl	(%eax), %eax
114		(%eax), %edx	203	orl	%eax, %edx
115		leal	204	leal	L_enabled\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax
116		L_pr\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax	205	movl	(%eax), %eax
117		(%eax), %eax	206	movl	%edx, (%eax)
118		\$0, (%eax,%edx,4)	207	leal	_ids - "L0000000003\$pb"(%ebx), %eax
119		leal	208	movl	12(%eax), %ecx
120		L_cid\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax	209	movl	\$1, %eax
121		(%eax), %eax	210	sall	%cl, %eax
122		leal	211	movl	%eax, %edx
123		L16 - "L0000000003\$pb"(%ebx), %edx	212	leal	L_active\$non_lazy_ptr - "L0000000003\$pb"(%ebx), %eax
124		(%eax), %eax	213	movl	(%eax), %eax
125		%edx, (%eax,%ecx,4)	214	movl	(%eax), %eax
126		leal	215	orl	%eax, %edx
		_ids - "L0000000003\$pb"(%ebx), %eax			
		12(%eax), %ecx			

216	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	305	movl	(%eax), %eax
217	movl	(%eax), %eax	306	leal	L20 — "L0000000003\$pb"(%ebx), %edx
218	movl	(%eax), %eax	307	movl	%edx, (%eax,%ecx,4)
219	leal	_ids — "L0000000003\$pb"(%ebx), %eax	308	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
220	movl	8(%eax), %ecx	309	movl	(%eax), %eax
221	leal	L_pc\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	310	movl	(%eax), %ecx
222	movl	(%eax), %eax	311	movl	\$1, %eax
223	leal	L18 — "L0000000003\$pb"(%ebx), %edx	312	sall	%cl, %eax
224	movl	%edx, (%eax,%ecx,4)	313	notl	%eax
225	leal	_ids — "L0000000003\$pb"(%ebx), %eax	314	movl	%eax, %edx
226	movl	8(%eax), %edx	315	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
227	leal	L_pr\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	316	movl	(%eax), %eax
228	movl	(%eax), %eax	317	movl	(%eax), %eax
229	movl	\$0, (%eax,%edx,4)	318	andl	%eax, %edx
230	leal	_ids — "L0000000003\$pb"(%ebx), %eax	319	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
231	movl	8(%eax), %ecx	320	movl	(%eax), %eax
232	movl	\$1, %eax	321	movl	%edx, (%eax)
233	sall	%cl, %eax	322	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
234	movl	%eax, %edx	323	movl	(%eax), %eax
235	leal	L_enabled\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	324	movl	(%eax), %ecx
236	movl	(%eax), %eax	325	bsrl	%ecx,%edx
237	movl	(%eax), %eax	326		
238	orl	%eax, %edx	327	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
239	leal	L_enabled\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	328	movl	(%eax), %eax
240	movl	(%eax), %eax	329	movl	%edx, (%eax)
241	movl	%edx, (%eax)	330	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
242	leal	_ids — "L0000000003\$pb"(%ebx), %eax	331	movl	(%eax), %eax
243	movl	8(%eax), %ecx	332	movl	(%eax), %edx
244	movl	\$1, %eax	333	leal	L_pc\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
245	sall	%cl, %eax	334	movl	(%eax), %eax
246	movl	%eax, %edx	335	movl	(%eax,%edx,4), %eax
247	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	336	movl	%eax, -12(%ebp)
248	movl	(%eax), %eax	337	jmp	L14
249	movl	(%eax), %eax	338	L20:	
250	orl	%eax, %edx	339	leal	L_signals\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
251	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	340	movl	(%eax), %eax
252	movl	(%eax), %eax	341	movl	(%eax), %eax
253	movl	%edx, (%eax)	342	andl	\$1, %eax
254	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	343	testl	%eax, %eax
255	movl	(%eax), %eax	344	je	L17
256	movl	(%eax), %ecx	345	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
257	leal	L_pc\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	346	movl	(%eax), %eax
258	movl	(%eax), %eax	347	movl	(%eax), %ecx
259	leal	L19 — "L0000000003\$pb"(%ebx), %edx	348	movl	\$1, %eax
260	movl	%edx, (%eax,%ecx,4)	349	sall	%cl, %eax
261	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	350	notl	%eax
262	movl	(%eax), %eax	351	movl	%eax, %edx
263	movl	(%eax), %edx	352	leal	L_enabled\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
264	leal	L_pr\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	353	movl	(%eax), %eax
265	movl	(%eax), %eax	354	movl	(%eax), %eax
266	movl	\$0, (%eax,%edx,4)	355	andl	%eax, %edx
267	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	356	leal	L_enabled\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
268	movl	(%eax), %eax	357	movl	(%eax), %eax
269	movl	(%eax), %esi	358	movl	%edx, (%eax)
270	leal	_ids — "L0000000003\$pb"(%ebx), %eax	359	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
271	movl	12(%eax), %ecx	360	movl	(%eax), %eax
272	movl	\$1, %eax	361	movl	(%eax), %ecx
273	movl	%eax, %edx	362	movl	\$1, %eax
274	sall	%cl, %edx	363	sall	%cl, %eax
275	leal	_ids — "L0000000003\$pb"(%ebx), %eax	364	notl	%eax
276	movl	8(%eax), %ecx	365	movl	%eax, %edx
277	movl	\$1, %eax	366	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
278	sall	%cl, %eax	367	movl	(%eax), %eax
279	orl	%edx, %eax	368	movl	(%eax), %eax
280	movl	%eax, %edx	369	andl	%eax, %edx
281	leal	L_descs\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	370	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
282	movl	(%eax), %eax	371	movl	(%eax), %eax
283	movl	%edx, (%eax,%esi,4)	372	movl	%edx, (%eax)
284	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	373	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
285	movl	(%eax), %eax	374	movl	(%eax), %eax
286	movl	(%eax), %ecx	375	movl	(%eax), %ecx
287	bsrl	%ecx,%edx	376	bsrl	%ecx,%edx
288			377		
289	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	378	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
290	movl	(%eax), %eax	379	movl	(%eax), %eax
291	movl	%edx, (%eax)	380	movl	%edx, (%eax)
292	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	381	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
293	movl	(%eax), %eax	382	movl	(%eax), %eax
294	movl	(%eax), %edx	383	movl	(%eax), %edx
295	leal	L_pc\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	384	leal	L_pc\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
296	movl	(%eax), %eax	385	movl	(%eax), %eax
297	movl	(%eax,%edx,4), %eax	386	movl	(%eax,%edx,4), %eax
298	movl	%eax, -12(%ebp)	387	movl	%eax, -12(%ebp)
299	jmp	L14	388	jmp	L14
300	L17:		389	L18:	
301	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	390	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
302	movl	(%eax), %eax	391	movl	(%eax), %eax
303	movl	(%eax), %ecx	392	movl	(%eax), %ecx
304	leal	L_pc\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	393	leal	L_pc\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax

394	movl	(%eax), %eax	483	leal	L_descs\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
395	leal	L22—"L0000000003\$pb"(%ebx), %edx	484	movl	(%eax), %eax
396	movl	%edx, (%eax,%ecx,4)	485	movl	(%eax,%edx,4), %edx
397	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	486	leal	L_enabled\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
398	movl	(%eax), %eax	487	movl	(%eax), %eax
399	movl	(%eax), %ecx	488	movl	(%eax), %eax
400	movl	\$1, %eax	489	andl	%edx, %eax
401	sall	%cl, %eax	490	testl	%eax, %eax
402	notl	%eax	491	je	L24
403	movl	%eax, %edx	492	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
404	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	493	movl	(%eax), %eax
405	movl	(%eax), %eax	494	movl	(%eax), %ecx
406	movl	(%eax), %eax	495	leal	L_pc\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
407	andl	%eax, %edx	496	movl	(%eax), %eax
408	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	497	leal	L19—"L0000000003\$pb"(%ebx), %edx
409	movl	(%eax), %eax	498	movl	%edx, (%eax,%ecx,4)
410	movl	%edx, (%eax)	499	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
411	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	500	movl	(%eax), %eax
412	movl	(%eax), %eax	501	movl	(%eax), %ecx
413	movl	(%eax), %ecx	502	movl	\$1, %eax
414	bsrl	%ecx,%edx	503	sall	%cl, %eax
415			504	notl	%eax
416	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	505	movl	%eax, %edx
417	movl	(%eax), %eax	506	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
418	movl	%edx, (%eax)	507	movl	(%eax), %eax
419	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	508	movl	(%eax), %eax
420	movl	(%eax), %eax	509	andl	%eax, %edx
421	movl	(%eax), %edx	510	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
422	leal	L_pc\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	511	movl	(%eax), %eax
423	movl	(%eax), %eax	512	movl	%edx, (%eax)
424	movl	(%eax,%edx,4), %eax	513	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
425	movl	%eax, —12(%ebp)	514	movl	(%eax), %eax
426	jmp	L14	515	movl	(%eax), %ecx
427	L22:		516	bsrl	%ecx,%edx
428	leal	L_signals\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	517		
429	movl	(%eax), %eax	518	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
430	movl	(%eax), %eax	519	movl	(%eax), %eax
431	shrl	%eax	520	movl	%edx, (%eax)
432	andl	\$1, %eax	521	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
433	testl	%eax, %eax	522	movl	(%eax), %eax
434	je	L18	523	movl	(%eax), %edx
435	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	524	leal	L_pc\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
436	movl	(%eax), %eax	525	movl	(%eax), %eax
437	movl	(%eax), %ecx	526	movl	(%eax,%edx,4), %eax
438	movl	\$1, %eax	527	movl	%eax, —12(%ebp)
439	sall	%cl, %eax	528	jmp	L14
440	notl	%eax	529	L24:	
441	movl	%eax, %edx	530	leal	L_signals\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
442	leal	L_enabled\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	531	movl	(%eax), %eax
443	movl	(%eax), %eax	532	movl	(%eax), %eax
444	movl	(%eax), %eax	533	movl	%eax, %edx
445	andl	%eax, %edx	534	orl	\$8, %edx
446	leal	L_enabled\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	535	leal	L_signals\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
447	movl	(%eax), %eax	536	movl	(%eax), %eax
448	movl	%edx, (%eax)	537	movl	%edx, (%eax)
449	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	538	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
450	movl	(%eax), %eax	539	movl	(%eax), %eax
451	movl	(%eax), %ecx	540	movl	(%eax), %ecx
452	movl	\$1, %eax	541	movl	\$1, %eax
453	sall	%cl, %eax	542	sall	%cl, %eax
454	notl	%eax	543	notl	%eax
455	movl	%eax, %edx	544	movl	%eax, %edx
456	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	545	leal	L_enabled\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
457	movl	(%eax), %eax	546	movl	(%eax), %eax
458	movl	(%eax), %eax	547	movl	(%eax), %eax
459	andl	%eax, %edx	548	andl	%eax, %edx
460	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	549	leal	L_enabled\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
461	movl	(%eax), %eax	550	movl	(%eax), %eax
462	movl	%edx, (%eax)	551	movl	%edx, (%eax)
463	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	552	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
464	movl	(%eax), %eax	553	movl	(%eax), %eax
465	movl	(%eax), %ecx	554	movl	(%eax), %ecx
466	bsrl	%ecx,%edx	555	movl	\$1, %eax
467			556	sall	%cl, %eax
468	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	557	notl	%eax
469	movl	(%eax), %eax	558	movl	%eax, %edx
470	movl	%edx, (%eax)	559	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
471	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	560	movl	(%eax), %eax
472	movl	(%eax), %eax	561	movl	(%eax), %eax
473	movl	(%eax), %edx	562	andl	%eax, %edx
474	leal	L_pc\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	563	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
475	movl	(%eax), %eax	564	movl	(%eax), %eax
476	movl	(%eax,%edx,4), %eax	565	movl	%edx, (%eax)
477	movl	%eax, —12(%ebp)	566	leal	L_active\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax
478	jmp	L14	567	movl	(%eax), %eax
479	L19:		568	movl	(%eax), %ecx
480	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax	569	bsrl	%ecx,%edx
481	movl	(%eax), %eax	570		
482	movl	(%eax), %edx	571	leal	L_cid\$non_lazy_ptr — "L0000000003\$pb"(%ebx), %eax

```

572     movl    (%eax), %eax
573     movl    %edx, (%eax)
574     leal   L_cid$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
575     movl    (%eax), %eax
576     movl    (%eax), %edx
577     leal   L_pc$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
578     movl    (%eax), %eax
579     movl    (%eax,%edx,4), %eax
580     movl    %eax, -12(%ebp)
581     jmp    L14
582
L16:
583     leal   L_cid$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
584     movl    (%eax), %eax
585     movl    (%eax), %ecx
586     leal   L_pc$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
587     movl    (%eax), %eax
588     leal   L26 - "L00000000003$pb"(%ebx), %edx
589     movl    %edx, (%eax,%ecx,4)
590     leal   L_cid$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
591     movl    (%eax), %eax
592     movl    (%eax), %ecx
593     movl    $1, %eax
594     sall   %cl, %eax
595     notl   %eax
596     movl    %eax, %edx
597     leal   L_active$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
598     movl    (%eax), %eax
599     movl    (%eax), %eax
600     andl   %eax, %edx
601     leal   L_active$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
602     movl    (%eax), %eax
603     movl    %edx, (%eax)
604     leal   L_active$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
605     movl    (%eax), %eax
606     movl    (%eax), %ecx
607     bsrl   %ecx,%edx
608
609     leal   L_cid$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
610     movl    (%eax), %eax
611     movl    %edx, (%eax)
612     leal   L_cid$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
613     movl    (%eax), %eax
614     movl    (%eax), %edx
615     leal   L_pc$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
616     movl    (%eax), %eax
617     movl    (%eax,%edx,4), %eax
618     movl    %eax, -12(%ebp)
619     jmp    L14
620
L26:
621     leal   L_signals$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
622     movl    (%eax), %eax
623     movl    (%eax), %eax
624     shr    $2, %eax
625     andl   $1, %eax
626     testl  %eax, %eax
627     je     L16
628     leal   L_cid$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
629     movl    (%eax), %eax
630     movl    (%eax), %edx
631     leal   L_descs$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
632     movl    (%eax), %eax
633     movl    (%eax,%edx,4), %eax
634     movl    %eax, %edx
635     notl   %edx
636     leal   L_enabled$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
637     movl    (%eax), %eax
638     movl    (%eax), %eax
639     andl   %eax, %edx
640     leal   L_enabled$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
641     movl    (%eax), %eax
642     movl    %edx, (%eax)
643     leal   L_cid$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
644     movl    (%eax), %eax
645     movl    (%eax), %edx
646     leal   L_descs$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
647     movl    (%eax), %eax
648     movl    (%eax,%edx,4), %eax
649     movl    %eax, %edx
650     notl   %edx
651     leal   L_active$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
652     movl    (%eax), %eax
653     movl    (%eax), %eax
654     andl   %eax, %edx
655     leal   L_active$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
656     movl    (%eax), %eax
657     movl    %edx, (%eax)
658     jmp    L13
659
L12:
660     leal   L_ids - "L00000000003$pb"(%ebx), %eax

```

```

661     movl    (%eax), %ecx
662     movl    $1, %eax
663     sall   %cl, %eax
664     movl    %eax, %edx
665     leal   L_enabled$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
666     movl    (%eax), %eax
667     movl    (%eax), %eax
668     cmpl   %eax, %edx
669     setne  %al
670     movzbl %al, %eax
671     addl   $16, %esp
672     popl   %ebx
673     popl   %esi
674     leave
675     ret

```

## Listing B.4: Assembler of ABRO tick function with optimizations (gcc -O3), before linking

```

1     .tick :
2     pushl  %ebp
3     movl   %esp, %ebp
4     pushl  %edi
5     pushl  %esi
6     pushl  %ebx
7     subl   $12, %esp
8     movl   8(%ebp), %eax
9     call   L39
10    "L00000000003$pb":
L39:
11
12     popl   %ebx
13     testl  %eax, %eax
14     je     L10
15     movl   L_tickCnt$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
16     movl   L_pc$non_lazy_ptr - "L00000000003$pb"(%ebx), %edi
17     movl   L_active$non_lazy_ptr - "L00000000003$pb"(%ebx), %edx
18     movl   L_pr$non_lazy_ptr - "L00000000003$pb"(%ebx), %esi
19     movl   $0, (%eax)
20     leal   L12 - "L00000000003$pb"(%ebx), %eax
21     movl   L_cid$non_lazy_ptr - "L00000000003$pb"(%ebx), %ecx
22     movl   %eax, (%edi)
23     movl   L_enabled$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
24     movl   $1, (%edx)
25     movl   $0, (%esi)
26     movl   $4, (%ecx)
27     movl   $1, (%eax)
28     movl   $17, (%edx)
29     movl   $17, (%eax)
30
L13:
31     movl   (%edx), %ecx
32     leal   L17 - "L00000000003$pb"(%ebx), %eax
33     movl   %eax, 4(%edi)
34     movl   L_enabled$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
35     movl   $0, 4(%esi)
36     orl   $2, %ecx
37     movl   %ecx, (%edx)
38     movl   L_cid$non_lazy_ptr - "L00000000003$pb"(%ebx), %edx
39     orl   $2, (%eax)
40     movl   (%edx), %eax
41     leal   L18 - "L00000000003$pb"(%ebx), %edx
42     movl   %edx, (%edi,%eax,4)
43     movl   L_descs$non_lazy_ptr - "L00000000003$pb"(%ebx), %edx
44     movl   $0, (%esi,%eax,4)
45     movl   $14, (%edx,%eax,4)
46     bsrl  %ecx,%ecx
47
48     movl   L_cid$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
49     movl   %ecx, (%eax)
50     movl   (%edi,%ecx,4), %eax
51     .align 4,0x90
52
L36:
53     jmp    *%eax
54     .align 4,0x90
55
L28:
56     movl   L_signals$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
57     testb  $4, (%eax)
58     jne   L38
59
L18:
60     movl   L_cid$non_lazy_ptr - "L00000000003$pb"(%ebx), %ecx
61     leal   L28 - "L00000000003$pb"(%ebx), %eax
62     movl   $-2, %esi
63     movl   (%ecx), %edx
64     movl   %eax, (%edi,%edx,4)
65     movl   L_active$non_lazy_ptr - "L00000000003$pb"(%ebx), %eax
66     movl   %edx, %ecx
67     roll  %cl, %esi
68     andl  (%eax), %esi

```

69	movl %esi, (%eax)	158	bsrl %ecx,%ecx
70	movl %esi, %ecx	159	
71	bsrl %ecx,%edx	160	movl L_cid\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %edx
72		161	movl (%edi,%ecx,4), %eax
73	movl L_cid\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax	162	movl %ecx, (%edx)
74	movl %edx, (%eax)	163	jmp *%eax
75	movl (%edi,%edx,4), %eax	164	.align 4,0x90
76	jmp *%eax	165	L22:
77	.align 4,0x90	166	movl L_signals\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax
78	L19:	167	testb \$1, (%eax)
79	movl L_cid\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %ecx	168	je L19
80	leal L22—"L0000000003\$pb"(%ebx), %eax	169	movl L_cid\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %edx
81	movl \$-2, %esi	170	movl (%edx), %eax
82	movl (%ecx), %edx	171	movl \$-2, %edx
83	movl %eax, (%edi,%edx,4)	172	movl %eax, %ecx
84	movl L_active\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax	173	movl L_enabled\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax
85	movl %edx, %ecx	174	roll %cl, %edx
86	roll %cl, %esi	175	movl %edx, %ecx
87	andl (%eax), %esi	176	andl %edx, (%eax)
88	movl %esi, (%eax)	177	movl L_active\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax
89	movl %esi, %ecx	178	andl (%eax), %ecx
90	bsrl %ecx,%edx	179	movl %ecx, (%eax)
91		180	bsrl %ecx,%ecx
92	movl L_cid\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax	181	
93	movl %edx, (%eax)	182	movl L_cid\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %edx
94	movl (%edi,%edx,4), %eax	183	movl (%edi,%ecx,4), %eax
95	jmp *%eax	184	movl %ecx, (%edx)
96	.align 4,0x90	185	jmp *%eax
97	L20:	186	.align 4,0x90
98	movl L_cid\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %ecx	187	L24:
99	leal L24—"L0000000003\$pb"(%ebx), %eax	188	movl L_signals\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax
100	movl \$-2, %esi	189	testb \$2, (%eax)
101	movl (%ecx), %edx	190	je L20
102	movl %eax, (%edi,%edx,4)	191	movl L_cid\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %edx
103	movl L_active\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax	192	movl (%edx), %eax
104	movl %edx, %ecx	193	movl \$-2, %edx
105	roll %cl, %esi	194	movl %eax, %ecx
106	andl (%eax), %esi	195	movl L_enabled\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax
107	movl %esi, (%eax)	196	roll %cl, %edx
108	movl %esi, %ecx	197	movl %edx, %ecx
109	bsrl %ecx,%edx	198	andl %edx, (%eax)
110		199	movl L_active\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax
111	movl L_cid\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax	200	andl (%eax), %ecx
112	movl %edx, (%eax)	201	movl %ecx, (%eax)
113	movl (%edi,%edx,4), %eax	202	bsrl %ecx,%ecx
114	jmp *%eax	203	
115	.align 4,0x90	204	
116	L10:	205	movl L_cid\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %edx
117	movl L_enabled\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %edx	206	movl (%edi,%ecx,4), %eax
118	movl L_active\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax	207	movl %ecx, (%edx)
119	movl (%edx), %ecx	208	jmp *%eax
120	movl %ecx, (%eax)	209	.align 4,0x90
121	bsrl %ecx,%ecx	210	L21:
122		211	movl L_enabled\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax
123	movl L_pc\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %edi	212	movl L_cid\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %ecx
124	movl L_cid\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %edx	213	movl (%eax), %edx
125	movl (%edi,%ecx,4), %eax	214	movl (%ecx), %esi
126	movl %ecx, (%edx)	215	movl L_descs\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax
127	jmp *%eax	216	testl %edx, (%eax,%esi,4)
128	.align 4,0x90	217	je L26
129	L17:	218	leal L21—"L0000000003\$pb"(%ebx), %eax
130	leal L19—"L0000000003\$pb"(%ebx), %eax	219	movl %esi, %ecx
131	movl \$3, %ecx	220	movl %eax, (%edi,%esi,4)
132	movl L_pr\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %esi	221	movl L_active\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax
133	movl %eax, (%edi,%ecx,4)	222	movl \$-2, %edx
134	movl L_enabled\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax	223	roll %cl, %edx
135	movl L_active\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %edx	224	andl (%eax), %edx
136	movl \$0, (%esi,%ecx,4)	225	movl %edx, (%eax)
137	movl (%eax), %eax	226	movl %edx, %ecx
138	movl %eax, -24(%ebp)	227	bsrl %ecx,%edx
139	leal L20—"L0000000003\$pb"(%ebx), %eax	228	
140	movl (%edx), %ecx	229	movl L_cid\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax
141	movl \$2, %edx	230	movl %edx, (%eax)
142	movl %eax, (%edi,%edx,4)	231	movl (%edi,%edx,4), %eax
143	movl L_enabled\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax	232	jmp *%eax
144	movl \$0, (%esi,%edx,4)	233	.align 4,0x90
145	orl \$12, -24(%ebp)	234	L12:
146	orl \$12, %ecx	235	movl L_enabled\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax
147	-24(%ebp), %edx	236	cmpl \$1, (%eax)
148	movl %edx, (%eax)	237	setne %al
149	movl L_active\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax	238	addl \$12, %esp
150	movl %ecx, (%eax)	239	popl %ebx
151	movl L_cid\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax	240	movzbl %al, %eax
152	movl (%eax), %edx	241	popl %esi
153	leal L21—"L0000000003\$pb"(%ebx), %eax	242	popl %edi
154	movl %eax, (%edi,%edx,4)	243	leave
155	movl L_descs\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax	244	ret
156	movl \$0, (%esi,%edx,4)	245	L26:
157	movl \$12, (%eax,%edx,4)	246	movl L_signals\$non_lazy_ptr—"L0000000003\$pb"(%ebx), %eax
			movl %esi, %ecx

```

247     movl   $-2, -16(%ebp)
248     roll  %cl, -16(%ebp)
249     andl  -16(%ebp), %edx
250     orl   $8, (%eax)
251     movl  L_enabled$non_lazy_ptr - "L0000000003$pb"(%ebx), %eax
252     movl  -16(%ebp), %ecx
253     movl  %edx, (%eax)
254     movl  L_active$non_lazy_ptr - "L0000000003$pb"(%ebx), %edx
255     andl  (%edx), %ecx
256     movl  %ecx, (%edx)
257     bsrl  %ecx, %ecx
258
259     movl  L_cid$non_lazy_ptr - "L0000000003$pb"(%ebx), %eax
260     movl  %ecx, (%eax)
261     movl  (%edi, %ecx, 4), %eax
262     jmp   *%eax
L38:
264     movl  L_cid$non_lazy_ptr - "L0000000003$pb"(%ebx), %edx
265     movl  L_descs$non_lazy_ptr - "L0000000003$pb"(%ebx), %ecx
266     movl  L_pr$non_lazy_ptr - "L0000000003$pb"(%ebx), %esi
267     movl  (%edx), %eax
268     movl  L_enabled$non_lazy_ptr - "L0000000003$pb"(%ebx), %edx
269     movl  (%ecx, %eax, 4), %eax
270     movl  L_active$non_lazy_ptr - "L0000000003$pb"(%ebx), %ecx
271     notl  %eax
272     andl  %eax, (%edx)
273     movl  %ecx, %edx
274     andl  %eax, (%ecx)
275     jmp  L13

```

## B.2 grcbal3

Listing B.5: grcbal3.c

```

1 // Example from Figure 1a in Stephen A. Edwards and Jia Zeng,
2 // Code Generation in the Columbia Esterel Compiler,
3 // EURASIP Journal on Embedded Systems, Volume 2007,
4 // Article ID 52651, 31 pages, doi:10.1155/2007/52651
5 //
6 // rvh, 5 mar 2009
7
8
9 // This program illustrates the usage of priorities /thread ids to
10 // handle signal dependencies among concurrent threads.
11 // It is encoded in 28 instructions ; BAL uses 74 instructions .
12 // (cf. Figure 8b in Edwards/Zeng'07).
13
14 // Binary on IA32:
15 // After linking : 1045 Bytes unoptimized, 586 optimized
16
17 #define USEPRIO // Select appropriate dispatcher
18 #include "sc.h"
19
20 // =====
21 // Program-specific definitions
22
23 #define RUNMAX 2 // # of runs to execute
24 #define TICKMAX 2 // # of ticks to execute
25
26 int runMax = RUNMAX; // # of runs to execute
27 int tickMax = TICKMAX; // # of ticks to execute
28
29 // Signals
30 typedef enum {A, B, C, D, E, T_} signaltype;
31 const char *s2signame[] = {"A", "B", "C", "D", "E", "T_"};
32
33 // Thread ids
34 int idHi = 4; // Highest thread id in use
35 typedef enum { TickEnd, Main, A1, A2, A3 } idtype;
36 const int ids[] = {0, 1, 2, 3, 4};
37 const char *id2threadname[] = {"TickEnd", "Main", "A1", "A2", "A3"};
38
39 // Inputs for RUNMAX runs of TICKMAX ticks
40 signalvector inputs[RUNMAX][TICKMAX] =
41   {{u2b(A), 0},
42    {0, 0}};
43
44 // Expected outputs
45 signalvector outputs[RUNMAX][TICKMAX] =
46   {{u2b(B) | u2b(C) | u2b(D) | u2b(E) | u2b(T_), 0},
47    {0, u2b(B)}};
48
49 void getInputs()
50 {
51   signals = inputs[runCnt][tickCnt];

```

```

52 }
53
54 // Set reference outputs and check valued signals, if there are any.
55 // Return 1 unless valued signal outputs are wrong.
56 // No valued signals here, therefore always return 1.
57 int checkOutputs(signalvector *tickOutputs)
58 {
59   *tickOutputs = outputs[runCnt][tickCnt];
60   return 1;
61 }
62
63 // No valued signals to print
64 void printVal(int id)
65 {
66 }
67
68 // Returns 1 if some thread is still active in current tick
69 int tick(int isInit)
70 {
71   TICKSTART(isInit); // Main thread has id 1
72   PAR(3, A1, ids[A1]); // A1 has id 2
73   PAR(2, A2, ids[A2]); // A2 has id 3
74   PAR(1, A3, ids[A3]); // A3 has id 4
75   PARE(0, AMain, id2b(A1) | id2b(A2) | id2b(A3));
76
77   A1: PRESENT(A, A1B);
78       EMIT(B);
79       PRIO(2, L0);
80   L0: PRESENT(C, A1A);
81       EMIT(D);
82   A1A: PRIO(1, L1);
83   L1: PRESENT(E, A1B);
84       EMIT(T_);
85       GOTO(A1C);
86   A1B: PAUSE(L2);
87   L2: EMIT(B);
88   A1C: TERM;
89   A2: PRESENT(B, A2A);
90       EMIT(C);
91   A2A: TERM;
92
93   A3: PRESENT(D, A3A);
94       EMIT(E);
95   A3A: TERM;
96
97   AMain: PRESENT(T_, AJoin);
98          TRANS(B);
99   AJoin: JOIN(B, AMain);
100
101   B: TERM;
102   TICKEND;
103 }
104
105 // Local Variables :
106 // compile-command: "make grcbal3; grcbal3"
107 // End:

```

Listing B.6: grcbal3.out

```

1 ##### RUN 0 STARTS #####
2 ==== TICK 0 STARTS, inputs = 01, enabled = 00
3 ==== Inputs: A (0)
4 ==== Enabled: <none>
5 PAR: Main (id 1, prio 0) forks A1 (2) with prio 3
6 PAR: Main (id 1, prio 0) forks A2 (3) with prio 2
7 PAR: Main (id 1, prio 0) forks A3 (4) with prio 1
8 PARE: Main (id 1, prio 0) has descendants 034
9 PRESENT: A1 (id 2, prio 3) determines A (0) as present
10 EMIT: A1 (id 2, prio 3) emits B (1)
11 PRIO: A1 (id 2, prio 3) set to priority 2
12 PRESENT: A2 (id 3, prio 2) determines B (1) as present
13 EMIT: A2 (id 3, prio 2) emits C (2)
14 TERM: A2 (id 3, prio 2) terminates, enabled = 027
15 PRESENT: A1 (id 2, prio 2) determines C (2) as present
16 EMIT: A1 (id 2, prio 2) emits D (3)
17 PRIO: A1 (id 2, prio 2) set to priority 1
18 PRESENT: A3 (id 4, prio 1) determines D (3) as present
19 EMIT: A3 (id 4, prio 1) emits E (4)
20 TERM: A3 (id 4, prio 1) terminates, enabled = 07
21 PRESENT: A1 (id 2, prio 1) determines E (4) as present
22 EMIT: A1 (id 2, prio 1) emits T_ (5)
23 TERM: A1 (id 2, prio 1) terminates, enabled = 03
24 PRESENT: Main (id 1, prio 0) determines T_ (5) as present
25 TRANS: Main (id 1, prio 0) transfers, enabled = 03
26 TERM: Main (id 1, prio 0) terminates, enabled = 01
27 ==== TICK 0 terminates after 23 instructions, enabled = 01.
28 ==== Resulting signals: A (0), B (1), C (2), D (3), E (4), T_ (5), Outputs
    OK.

```

```

29
30 ##### RUN 0 terminates after 23 instructions
31
32 ##### RUN 1 STARTS #####
33 ===== TICK 0 STARTS, inputs = 00, enabled = 00
34 ===== Inputs: <none>
35 ===== Enabled: <none>
36 PAR: Main (id 1, prio 0) forks A1 (2) with prio 3
37 PAR: Main (id 1, prio 0) forks A2 (3) with prio 2
38 PAR: Main (id 1, prio 0) forks A3 (4) with prio 1
39 PARE: Main (id 1, prio 0) has descendants 034
40 PRESENT: A1 (id 2, prio 3) determines A (0) as absent
41 PAUSE: A1 (id 2, prio 3) pauses, active = 037
42 PRESENT: A2 (id 3, prio 2) determines B (1) as absent
43 TERM: A2 (id 3, prio 2) terminates, enabled = 027
44 PRESENT: A3 (id 4, prio 1) determines D (3) as absent
45 TERM: A3 (id 4, prio 1) terminates, enabled = 07
46 PRESENT: Main (id 1, prio 0) determines T_ (5) as absent
47 JOIN: Main (id 1, prio 0) does not join
48 PAUSE: Main (id 1, prio 0) pauses, active = 03
49 ===== TICK 0 terminates after 12 instructions, enabled = 07.
50 ===== Resulting signals: <none>, Outputs OK.
51
52 ===== TICK 1 STARTS, inputs = 00, enabled = 07
53 ===== Inputs: <none>
54 ===== Enabled: TickEnd (0), Main (1), A1 (2)
55 EMIT: A1 (id 2, prio 3) emits B (1)
56 TERM: A1 (id 2, prio 3) terminates, enabled = 03
57 PRESENT: Main (id 1, prio 0) determines T_ (5) as absent
58 JOIN: Main (id 1, prio 0) joins
59 TERM: Main (id 1, prio 0) terminates, enabled = 01
60 ===== TICK 1 terminates after 5 instructions, enabled = 01.
61 ===== Resulting signals: B (1), Outputs OK.
62
63 ===== Executed tickMax = 2 ticks!
64 ##### RUN 1 terminates after 17 instructions
65
66 ##### All runs terminate, after 40 instructions

```

## B.3 PCO

Listing B.7: PCO.c

```

1 #include "sc.h"
2
3 int BUF, fd, i, j, k = 0, tmp, arr [8], idHi = 4;
4 typedef enum { TickEnd, Main, Cons, Obs, Prod } idtype;
5 const int ids [] = { 0, 1, 2, 3, 4 };
6 const char *id2threadname [] = { "TickEnd", "Main", "Cons", "Obs", "Prod"
7 };
8
9 // ===== MAIN FUNCTION =====
10 int main()
11 {
12     int notDone, init = 1;
13
14     do {
15         notDone = tick(init); // Call tick function
16         //sleep(1); // Slow down by 1 sec
17         init = 0;
18     } while (notDone);
19     return 0;
20 }
21
22 // ===== TICK FUNCTION =====
23 int tick(int isInit)
24 {
25     TICKSTART(isInit); // Main thread
26
27     PCO: PAR(0, Prod, ids[Prod]);
28         PAR(0, Cons, ids[Cons]);
29         PAR(0, Obs, ids[Obs]);
30         PARE(0, Parent, id2b(Prod) | id2b(Cons) | id2b(Obs));
31
32     Prod: for (i = 0; i < 4; i++) { // Producer
33         PAUSE(L0);
34     }
35     L0: BUF = i; }
36
37     Cons: for (j = 0; j < 8; j++) // Consumer
38         arr[j] = 0;
39         for (j = 0; j < 8; j++) {
40             PAUSE(L1);
41             L1: tmp = BUF;
42                 arr[j % 8] = tmp; }

```

```

42 Obs: for (; ; ) { // Observer
43     PAUSE(L2);
44     L2: fd = BUF;
45         k++; }
46
47 Parent: PAUSE(L3); // Main (cont'd)
48 L3: if (k == 20) // IF iteration limit
49     TRANS(Done); // THEN terminate
50     if (BUF == 10) // IF buffer = 10
51     TRANS(PCO); // THEN restart PCO
52     goto Parent; // ELSE continue
53
54 Done: TERM;
55     TICKEND;
56 }
57
58 // Local Variables :
59 // compile-command: "make prod-cons; prod-cons"
60 // End:

```

Listing B.8: PCO.out

```

1 PAR: Main (id 1, prio 0) forks Prod (4) with prio 0
2 PAR: Main (id 1, prio 0) forks Cons (2) with prio 0
3 PAR: Main (id 1, prio 0) forks Obs (3) with prio 0
4 PARE: Main (id 1, prio 0) has descendants 034
5 PAUSE: Prod (id 4, prio 0) pauses, active = 037
6 PAUSE: Obs (id 3, prio 0) pauses, active = 017
7 PAUSE: Cons (id 2, prio 0) pauses, active = 07
8 PAUSE: Main (id 1, prio 0) pauses, active = 03
9 PAUSE: Prod (id 4, prio 0) pauses, active = 037
10 PAUSE: Obs (id 3, prio 0) pauses, active = 017
11 PAUSE: Cons (id 2, prio 0) pauses, active = 07
12 PAUSE: Main (id 1, prio 0) pauses, active = 03
13 PAUSE: Prod (id 4, prio 0) pauses, active = 037
14 PAUSE: Obs (id 3, prio 0) pauses, active = 017
15 PAUSE: Cons (id 2, prio 0) pauses, active = 07
16 PAUSE: Main (id 1, prio 0) pauses, active = 03
17 PAUSE: Prod (id 4, prio 0) pauses, active = 037
18 PAUSE: Obs (id 3, prio 0) pauses, active = 017
19 PAUSE: Cons (id 2, prio 0) pauses, active = 07
20 PAUSE: Main (id 1, prio 0) pauses, active = 03
21 PAUSE: Prod (id 4, prio 0) pauses, active = 037
22 PAUSE: Obs (id 3, prio 0) pauses, active = 017
23 PAUSE: Cons (id 2, prio 0) pauses, active = 07
24 PAUSE: Main (id 1, prio 0) pauses, active = 03
25 PAUSE: Prod (id 4, prio 0) pauses, active = 037
26 PAUSE: Obs (id 3, prio 0) pauses, active = 017
27 PAUSE: Cons (id 2, prio 0) pauses, active = 07
28 PAUSE: Main (id 1, prio 0) pauses, active = 03
29 PAUSE: Prod (id 4, prio 0) pauses, active = 037
30 PAUSE: Obs (id 3, prio 0) pauses, active = 017
31 PAUSE: Cons (id 2, prio 0) pauses, active = 07
32 PAUSE: Main (id 1, prio 0) pauses, active = 03
33 PAUSE: Prod (id 4, prio 0) pauses, active = 037
34 PAUSE: Obs (id 3, prio 0) pauses, active = 017
35 PAUSE: Cons (id 2, prio 0) pauses, active = 07
36 PAUSE: Main (id 1, prio 0) pauses, active = 03
37 PAUSE: Prod (id 4, prio 0) pauses, active = 037
38 PAUSE: Obs (id 3, prio 0) pauses, active = 017
39 PAUSE: Cons (id 2, prio 0) pauses, active = 07
40 PAUSE: Main (id 1, prio 0) pauses, active = 03
41 PAUSE: Prod (id 4, prio 0) pauses, active = 037
42 PAUSE: Obs (id 3, prio 0) pauses, active = 017
43 PAUSE: Cons (id 2, prio 0) pauses, active = 07
44 PAUSE: Main (id 1, prio 0) pauses, active = 03
45 PAUSE: Prod (id 4, prio 0) pauses, active = 037
46 PAUSE: Obs (id 3, prio 0) pauses, active = 017
47 PAUSE: Cons (id 2, prio 0) pauses, active = 07
48 PAUSE: Main (id 1, prio 0) pauses, active = 03
49 PAUSE: Prod (id 4, prio 0) pauses, active = 037
50 PAUSE: Obs (id 3, prio 0) pauses, active = 017
51 PAUSE: Cons (id 2, prio 0) pauses, active = 07
52 TRANS: Main (id 1, prio 0) transfers, enabled = 03
53 PAR: Main (id 1, prio 0) forks Prod (4) with prio 0
54 PAR: Main (id 1, prio 0) forks Cons (2) with prio 0
55 PAR: Main (id 1, prio 0) forks Obs (3) with prio 0
56 PARE: Main (id 1, prio 0) has descendants 034
57 PAUSE: Prod (id 4, prio 0) pauses, active = 037
58 PAUSE: Obs (id 3, prio 0) pauses, active = 017
59 PAUSE: Cons (id 2, prio 0) pauses, active = 07
60 PAUSE: Main (id 1, prio 0) pauses, active = 03
61 PAUSE: Prod (id 4, prio 0) pauses, active = 037
62 PAUSE: Obs (id 3, prio 0) pauses, active = 017
63 PAUSE: Cons (id 2, prio 0) pauses, active = 07
64 PAUSE: Main (id 1, prio 0) pauses, active = 03
65 PAUSE: Prod (id 4, prio 0) pauses, active = 037
66 PAUSE: Obs (id 3, prio 0) pauses, active = 017

```



```

67 PAUSE: Cons (id 2, prio 0) pauses, active = 07
68 PAUSE: Main (id 1, prio 0) pauses, active = 03
69 PAUSE: Prod (id 4, prio 0) pauses, active = 037
70 PAUSE: Obs (id 3, prio 0) pauses, active = 017
71 PAUSE: Cons (id 2, prio 0) pauses, active = 07
72 PAUSE: Main (id 1, prio 0) pauses, active = 03
73 PAUSE: Prod (id 4, prio 0) pauses, active = 037
74 PAUSE: Obs (id 3, prio 0) pauses, active = 017
75 PAUSE: Cons (id 2, prio 0) pauses, active = 07
76 PAUSE: Main (id 1, prio 0) pauses, active = 03
77 PAUSE: Prod (id 4, prio 0) pauses, active = 037
78 PAUSE: Obs (id 3, prio 0) pauses, active = 017
79 PAUSE: Cons (id 2, prio 0) pauses, active = 07
80 PAUSE: Main (id 1, prio 0) pauses, active = 03
81 PAUSE: Prod (id 4, prio 0) pauses, active = 037
82 PAUSE: Obs (id 3, prio 0) pauses, active = 017
83 PAUSE: Cons (id 2, prio 0) pauses, active = 07
84 PAUSE: Main (id 1, prio 0) pauses, active = 03
85 PAUSE: Prod (id 4, prio 0) pauses, active = 037
86 PAUSE: Obs (id 3, prio 0) pauses, active = 017
87 PAUSE: Cons (id 2, prio 0) pauses, active = 07
88 PAUSE: Main (id 1, prio 0) pauses, active = 03
89 PAUSE: Prod (id 4, prio 0) pauses, active = 037
90 PAUSE: Obs (id 3, prio 0) pauses, active = 017
91 PAUSE: Cons (id 2, prio 0) pauses, active = 07
92 PAUSE: Main (id 1, prio 0) pauses, active = 03
93 PAUSE: Prod (id 4, prio 0) pauses, active = 037
94 PAUSE: Obs (id 3, prio 0) pauses, active = 017
95 PAUSE: Cons (id 2, prio 0) pauses, active = 07
96 TRANS: Main (id 1, prio 0) transfers, enabled = 03
97 TERM: Main (id 1, prio 0) terminates, enabled = 01

```

## B.4 Count2Suspend

Listing B.9: Count2Suspend.c

```

1 // Example from Charles Andr, Semantics of SyncCharts,
2 // ISBN 135/RR-2003-24-FR, April 2003, Figure 8-5
3 //
4 // rvh, 19 mar 2009
5 #include "sc.h"
6
7 #define RUNMAX 1 // # of runs to execute
8 #define TICKMAX 10 // # of ticks to execute
9
10 int runMax = RUNMAX; // # of runs to execute
11 int tickMax = TICKMAX; // # of ticks to execute
12
13
14 // =====
15 // Program-specific definitions
16
17 // Signals
18 typedef enum {T, inhib, reset, B0, B1, C, C0} signaltype;
19 const char *s2signame[] = {"T", "inhib", "reset", "B0", "B1", "C", "C0"};
20
21 // Thread ids
22 // Note: Off0 gets a higher id than Off1 (rather than the other way
23 // around) simply to let Off0 execute first, to make the trace match
24 // the syntactical flow of the program
25 int idHi = 3; // Highest thread id in use
26 typedef enum { TickEnd, Off1, Off0, Main } idtype;
27 const int ids[] = { 0, 1, 2, 3 };
28 const char *id2threadname[] = { "TickEnd", "Off1", "Off0", "Main" };
29
30 // Inputs for RUNMAX runs of TICKMAX ticks
31 signalvector inputs[RUNMAX][TICKMAX] =
32 { {u2b(T), 0, u2b(T), u2b(T) | u2b(inhib), 0,
33 u2b(T) | u2b(reset), 0, u2b(T), u2b(T) | u2b(inhib), u2b(T)}
34 };
35
36 // Expected outputs
37 signalvector outputs[RUNMAX][TICKMAX] =
38 { {0, 0, u2b(B0), 0, u2b(B0),
39 0, 0, u2b(B0), 0, u2b(B1) | u2b(C0)} };
40
41 void getInputs()
42 {
43 signals = inputs[runCnt][tickCnt];
44 }
45
46 // Set reference outputs and check valued signals, if there are any.
47 // Return 1 unless valued signal outputs are wrong.
48 // No valued signals here, therefore always return 1.
49 int checkOutputs(signalvector *tickOutputs)

```

```

50 {
51 *tickOutputs = outputs[runCnt][tickCnt];
52 return 1;
53 }
54
55 // No valued signals to print
56 void printVal(int id)
57 {
58 }
59
60 // Returns 1 if some thread is still active in current tick
61 int tick(int islnit)
62 {
63 // Thread ids: Off1=1, Off0=2, Main=3
64 TICKSTART(islnit);
65
66 Cnt2: PAR(0, Off0, ids[Off0]);
67 PAR(0, Off1, ids[Off1]);
68 PARE(0, Cnt2Main, id2b(Off0) | id2b(Off1));
69
70 Off0: PAUSE(L0);
71 L0: PRESENT(T, Off0);
72 On0: EMIT(B0);
73 PAUSE(L1);
74 L1: PRESENT(T, On0);
75 EMIT(C0);
76 GOTO(Off0);
77
78 Off1: PAUSE(L2);
79 L2: PRESENT(C0, Off1);
80 On1: EMIT(B1);
81 PAUSE(L3);
82 L3: PRESENT(C0, On1);
83 EMIT(C);
84 GOTO(Off1);
85
86 Cnt2Main:PAUSE(L4);
87 L4: PRESENT(reset, L5);
88 TRANS(Cnt2);
89 L5: PRESENT(inhib, Cnt2Main);
90 SUSPEND(L5);
91
92 TICKEND;
93 }
94
95 // Local Variables :
96 // compile-command: "make count2suspend; count2suspend"
97 // End:

```

Listing B.10: Count2Suspend.out

```

1 ##### RUN 0 STARTS #####
2 ===== TICK 0 STARTS, inputs = 01, enabled = 00
3 ===== Inputs: T (0)
4 ===== Enabled: <none>
5 PAR: Main (id 3, prio 0) forks Off0 (2) with prio 0
6 PAR: Main (id 3, prio 0) forks Off1 (1) with prio 0
7 PARE: Main (id 3, prio 0) has descendants 06
8 PAUSE: Main (id 3, prio 0) pauses, active = 017
9 PAUSE: Off0 (id 2, prio 0) pauses, active = 07
10 PAUSE: Off1 (id 1, prio 0) pauses, active = 03
11 ===== TICK 0 terminates after 6 instructions, enabled = 017.
12 ===== Resulting signals: T (0), Outputs OK.
13
14 ===== TICK 1 STARTS, inputs = 00, enabled = 017
15 ===== Inputs: <none>
16 ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Main (3)
17 PRESENT: Main (id 3, prio 0) determines reset (2) as absent
18 PRESENT: Main (id 3, prio 0) determines inhib (1) as absent
19 PAUSE: Main (id 3, prio 0) pauses, active = 017
20 PRESENT: Off0 (id 2, prio 0) determines T (0) as absent
21 PAUSE: Off0 (id 2, prio 0) pauses, active = 07
22 PRESENT: Off1 (id 1, prio 0) determines C0 (6) as absent
23 PAUSE: Off1 (id 1, prio 0) pauses, active = 03
24 ===== TICK 1 terminates after 7 instructions, enabled = 017.
25 ===== Resulting signals: <none>, Outputs OK.
26
27 ===== TICK 2 STARTS, inputs = 01, enabled = 017
28 ===== Inputs: T (0)
29 ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Main (3)
30 PRESENT: Main (id 3, prio 0) determines reset (2) as absent
31 PRESENT: Main (id 3, prio 0) determines inhib (1) as absent
32 PAUSE: Main (id 3, prio 0) pauses, active = 017
33 PRESENT: Off0 (id 2, prio 0) determines T (0) as present
34 EMIT: Off0 (id 2, prio 0) emits B0 (3)
35 PAUSE: Off0 (id 2, prio 0) pauses, active = 07
36 PRESENT: Off1 (id 1, prio 0) determines C0 (6) as absent
37 PAUSE: Off1 (id 1, prio 0) pauses, active = 03

```

```

38  ===== TICK 2 terminates after 8 instructions, enabled = 017.
39  ===== Resulting signals: T (0), B0 (3), Outputs OK.
40
41  ===== TICK 3 STARTS, inputs = 03, enabled = 017
42  ===== Inputs: T (0), inhibit (1)
43  ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Main (3)
44  PRESENT: Main (id 3, prio 0) determines reset (2) as absent
45  PRESENT: Main (id 3, prio 0) determines inhibit (1) as present
46  SUSPEND: Main (id 3, prio 0) suspends itself and descendants 06
47  PAUSE: Main (id 3, prio 0) pauses, active = 011
48  ===== TICK 3 terminates after 3 instructions, enabled = 017.
49  ===== Resulting signals: T (0), inhibit (1), Outputs OK.
50
51  ===== TICK 4 STARTS, inputs = 00, enabled = 017
52  ===== Inputs: <none>
53  ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Main (3)
54  PRESENT: Main (id 3, prio 0) determines inhibit (1) as absent
55  PAUSE: Main (id 3, prio 0) pauses, active = 017
56  PRESENT: Off0 (id 2, prio 0) determines T (0) as absent
57  EMIT: Off0 (id 2, prio 0) emits B0 (3)
58  PAUSE: Off0 (id 2, prio 0) pauses, active = 07
59  PRESENT: Off1 (id 1, prio 0) determines C0 (6) as absent
60  PAUSE: Off1 (id 1, prio 0) pauses, active = 03
61  ===== TICK 4 terminates after 7 instructions, enabled = 017.
62  ===== Resulting signals: B0 (3), Outputs OK.
63
64  ===== TICK 5 STARTS, inputs = 05, enabled = 017
65  ===== Inputs: T (0), reset (2)
66  ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Main (3)
67  PRESENT: Main (id 3, prio 0) determines reset (2) as present
68  TRANS: Main (id 3, prio 0) transfers, enabled = 011
69  PAR: Main (id 3, prio 0) forks Off0 (2) with prio 0
70  PAR: Main (id 3, prio 0) forks Off1 (1) with prio 0
71  PARE: Main (id 3, prio 0) has descendants 06
72  PAUSE: Main (id 3, prio 0) pauses, active = 017
73  PAUSE: Off0 (id 2, prio 0) pauses, active = 07
74  PAUSE: Off1 (id 1, prio 0) pauses, active = 03
75  ===== TICK 5 terminates after 8 instructions, enabled = 017.
76  ===== Resulting signals: T (0), reset (2), Outputs OK.
77
78  ===== TICK 6 STARTS, inputs = 00, enabled = 017
79  ===== Inputs: <none>
80  ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Main (3)
81  PRESENT: Main (id 3, prio 0) determines reset (2) as absent
82  PRESENT: Main (id 3, prio 0) determines inhibit (1) as absent
83  PAUSE: Main (id 3, prio 0) pauses, active = 017
84  PRESENT: Off0 (id 2, prio 0) determines T (0) as absent
85  PAUSE: Off0 (id 2, prio 0) pauses, active = 07
86  PRESENT: Off1 (id 1, prio 0) determines C0 (6) as absent
87  PAUSE: Off1 (id 1, prio 0) pauses, active = 03
88  ===== TICK 6 terminates after 7 instructions, enabled = 017.
89  ===== Resulting signals: <none>, Outputs OK.
90
91  ===== TICK 7 STARTS, inputs = 01, enabled = 017
92  ===== Inputs: T (0)
93  ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Main (3)
94  PRESENT: Main (id 3, prio 0) determines reset (2) as absent
95  PRESENT: Main (id 3, prio 0) determines inhibit (1) as absent
96  PAUSE: Main (id 3, prio 0) pauses, active = 017
97  PRESENT: Off0 (id 2, prio 0) determines T (0) as present
98  EMIT: Off0 (id 2, prio 0) emits B0 (3)
99  PAUSE: Off0 (id 2, prio 0) pauses, active = 07
100 PRESENT: Off1 (id 1, prio 0) determines C0 (6) as absent
101 PAUSE: Off1 (id 1, prio 0) pauses, active = 03
102 ===== TICK 7 terminates after 8 instructions, enabled = 017.
103 ===== Resulting signals: T (0), B0 (3), Outputs OK.
104
105  ===== TICK 8 STARTS, inputs = 03, enabled = 017
106  ===== Inputs: T (0), inhibit (1)
107  ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Main (3)
108  PRESENT: Main (id 3, prio 0) determines reset (2) as absent
109  PRESENT: Main (id 3, prio 0) determines inhibit (1) as present
110  SUSPEND: Main (id 3, prio 0) suspends itself and descendants 06
111  PAUSE: Main (id 3, prio 0) pauses, active = 011
112  ===== TICK 8 terminates after 3 instructions, enabled = 017.
113  ===== Resulting signals: T (0), inhibit (1), Outputs OK.
114
115  ===== TICK 9 STARTS, inputs = 01, enabled = 017
116  ===== Inputs: T (0)
117  ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Main (3)
118  PRESENT: Main (id 3, prio 0) determines inhibit (1) as absent
119  PAUSE: Main (id 3, prio 0) pauses, active = 017
120  PRESENT: Off0 (id 2, prio 0) determines T (0) as present
121  EMIT: Off0 (id 2, prio 0) emits C0 (6)
122  PAUSE: Off0 (id 2, prio 0) pauses, active = 07
123  PRESENT: Off1 (id 1, prio 0) determines C0 (6) as present
124  EMIT: Off1 (id 1, prio 0) emits B1 (4)
125  PAUSE: Off1 (id 1, prio 0) pauses, active = 03
126  ===== TICK 9 terminates after 9 instructions, enabled = 017.

```

```

127  ===== Resulting signals: T (0), B1 (4), C0 (6), Outputs OK.
128
129  ===== Executed tickMax = 10 ticks!
130  ##### RUN 0 terminates after 66 instructions
131
132  ##### All runs terminate, after 66 instructions

```

## B.5 Exits

Listing B.11: Exits.c

```

1 // Example from Charles Andr, Semantics of SyncCharts,
2 // ISRN 135/RR-2003-24-FR, April 2003, Figure 8-8
3 //
4 // rvh, 20 mar 2009
5
6 // This example demonstrates how to implement Exit Actions
7
8 #define USEPRIO // Select appropriate dispatcher
9 #include "sc.h"
10
11 #define RUNMAX 2 // # of runs to execute
12 #define TICKMAX 4 // # of ticks to execute
13
14 int runMax = RUNMAX; // # of runs to execute
15 int tickMax = TICKMAX; // # of ticks to execute
16
17
18 // =====
19 // Program-specific definitions
20
21 // Signals
22 typedef enum {A, B, R, X0, X2, X10, X11, Y0, Y1, Y2, Z} signaltype;
23 const char *s2signal[] = {"A", "B", "R", "X0", "X2", "X10", "X11",
24 "Y0", "Y1", "Y2", "Z"};
25
26 // Thread ids
27 int idHi = 4; // Highest thread id in use
28 typedef enum { TickEnd, M11, M10, M2, Main } idtype;
29 const int ids[] = {0, 1, 2, 3, 4};
30 const char *id2threadname[] = {"TickEnd", "M11", "M10", "M2", "Main"};
31
32 // Inputs for RUNMAX runs of TICKMAX ticks
33 signalvector inputs[RUNMAX][TICKMAX] =
34 {{0, u2b(B), u2b(R), u2b(A)},
35 {0, u2b(R), 0, u2b(A) | u2b(B)}};
36
37 // Expected outputs
38 signalvector outputs[RUNMAX][TICKMAX] =
39 {{0, u2b(Y2) | u2b(X2) | u2b(Y1) | u2b(X11),
40 u2b(Z) | u2b(Y0) | u2b(X0), u2b(Y2) | u2b(Y1) | u2b(X10)},
41 {0, u2b(Y2) | u2b(Z) | u2b(Y1) | u2b(Y0) | u2b(X0), 0, u2b(Y2) | u2b(Y1)
42 | u2b(X10)}};
43
44 void getInputs()
45 {
46 signals = inputs[runCnt][tickCnt];
47 }
48
49 // Set reference outputs and check valued signals, if there are any.
50 // Return 1 unless valued signal outputs are wrong.
51 // No valued signals here, therefore always return 1.
52 int checkOutputs(signalvector *tickOutputs)
53 {
54 *tickOutputs = outputs[runCnt][tickCnt];
55 return 1;
56 }
57
58 // No valued signals to print
59 void printVal(int id)
60 {
61 }
62
63 // Returns 1 if some thread is still active in current tick
64 // Notes:
65 // - CALL calls exit actions unconditionally
66 // - ISATCALL calls exit actions if the corresponding state is active
67 // (and now gets aborted)
68 // - At L2, we set the priority low (0) for M10main, so that
69 // it executes the \codefont{JOIN} at the end of a tick.
70 // - At M10main, we set the priority high (1) for M10depth, so that it
71 // checks for a strong abort at the beginning of a tick.
72 int tick(int isnlit)
73 {

```

```

74 // Thread ids: M11=1, M10=2, M2=3, Main=4
75 TICKSTART(isinit);
76 M0: PAR(0, M10, ids[M10]);
77 PAR(0, M11, ids[M11]);
78 PARE(1, M0main, id2b(M10) | id2b(M11) | id2b(M2));
79
80 M10: PAR(0, M2, ids[M2]);
81 PARE(0, M10main, id2b(M2));
82
83 M2: PAUSE(M2depth);
84 M2depth:PRESENT(B, M2);
85 CALL(M2exit, L1);
86 M2exit: EMIT(Y2);
87 RET;
88 L1: EMIT(X2);
89 TERM;
90
91 L2: PRIO(0, M10main);
92 M10main:JPPAUSE(1, L3, M10depth);
93 L3: ISATCALL(ids[M2], M2depth, M2exit, L4);
94 L4: CALL(M10exit, L5);
95 M10exit:EMIT(Y1);
96 RET;
97 L5: EMIT(X11);
98 TRANS(Done);
99 M10depth:PRESENT(A, L2);
100 ISATCALL(ids[M2], M2depth, M2exit, L7);
101 L7: CALL(M10exit, L8);
102 L8: EMIT(X10);
103 TRANS(Done);
104 Done: PAUSE(Done);
105
106 M11: PAUSE(M11);
107 M11exit:EMIT(Z);
108 RET;
109
110 M0main: PAUSE(L9);
111 L9: PRESENT(R, M0main);
112 ISATCALL(ids[M2], M2depth, M2exit, L10);
113 L10: ISATCALL(ids[M10], M10depth, M10exit, L11);
114 L11: CALL(M11exit, L12);
115 L12: EMIT(Y0); // Only place to call exit action of M0
116 EMIT(X0);
117 TRANS(M0);
118
119 TICKEND;
120 }
121
122 // Local Variables :
123 // compile--command: "make exits; exits"
124 // End:

```

## Listing B.12: Exits.out

```

1 ##### RUN 0 STARTS #####
2 ===== TICK 0 STARTS, inputs = 00, enabled = 00
3 ===== Inputs: <none>
4 ===== Enabled: <none>
5 PAR: Main (id 4, prio 0) forks M10 (2) with prio 0
6 PAR: Main (id 4, prio 0) forks M11 (1) with prio 0
7 PARE: Main (id 4, prio 0) has descendants 016
8 PAUSE: Main (id 4, prio 1) pauses, active = 027
9 PAR: M10 (id 2, prio 0) forks M2 (3) with prio 0
10 PARE: M10 (id 2, prio 0) has descendants 010
11 PAUSE: M2 (id 3, prio 0) pauses, active = 017
12 JPPAUSE: M10 (id 2, prio 0) does not join
13 PPAUSE: M10 (id 2, prio 0) sets prio to 1
14 PAUSE: M10 (id 2, prio 1) pauses, active = 07
15 PAUSE: M11 (id 1, prio 0) pauses, active = 03
16 ===== TICK 0 terminates after 9 instructions, enabled = 037.
17 ===== Resulting signals: <none>, Outputs OK.
18
19 ===== TICK 1 STARTS, inputs = 02, enabled = 037
20 ===== Inputs: B (1)
21 ===== Enabled: TickEnd (0), M11 (1), M10 (2), M2 (3), Main (4)
22 PRESENT: Main (id 4, prio 1) determines R (2) as absent
23 PAUSE: Main (id 4, prio 1) pauses, active = 037
24 PRESENT: M10 (id 2, prio 1) determines A (0) as absent
25 PRIO: M10 (id 2, prio 1) set to priority 0
26 PRESENT: M2 (id 3, prio 0) determines B (1) as present
27 CALL: M2 (id 3, prio 0) calls function
28 EMIT: M2 (id 3, prio 0) emits Y2 (9)
29 RET: M2 (id 3, prio 0) returns from function
30 EMIT: M2 (id 3, prio 0) emits X2 (4)
31 TERM: M2 (id 3, prio 0) terminates, enabled = 027
32 JPPAUSE: M10 (id 2, prio 0) joins
33 ISATCALL:M10 (id 2, prio 0) does_not_ call function
34 CALL: M10 (id 2, prio 0) calls function

```

```

35 EMIT: M10 (id 2, prio 0) emits Y1 (8)
36 RET: M10 (id 2, prio 0) returns from function
37 EMIT: M10 (id 2, prio 0) emits X11 (6)
38 TRANS: M10 (id 2, prio 0) transfers , enabled = 027
39 PAUSE: M10 (id 2, prio 0) pauses, active = 07
40 PAUSE: M11 (id 1, prio 0) pauses, active = 03
41 ===== TICK 1 terminates after 19 instructions, enabled = 027.
42 ===== Resulting signals: B (1), X2 (4), X11 (6), Y1 (8), Y2 (9), Outputs OK
43
44 ===== TICK 2 STARTS, inputs = 04, enabled = 027
45 ===== Inputs: R (2)
46 ===== Enabled: TickEnd (0), M11 (1), M10 (2), Main (4)
47 PRESENT: Main (id 4, prio 1) determines R (2) as present
48 ISATCALL:Main (id 4, prio 1) does_not_ call function
49 ISATCALL:Main (id 4, prio 1) does_not_ call function
50 CALL: Main (id 4, prio 1) calls function
51 EMIT: Main (id 4, prio 1) emits Z (10)
52 RET: Main (id 4, prio 1) returns from function
53 EMIT: Main (id 4, prio 1) emits Y0 (7)
54 EMIT: Main (id 4, prio 1) emits X0 (3)
55 TRANS: Main (id 4, prio 1) transfers , enabled = 021
56 PAR: Main (id 4, prio 1) forks M10 (2) with prio 0
57 PAR: Main (id 4, prio 1) forks M11 (1) with prio 0
58 PARE: Main (id 4, prio 1) has descendants 016
59 PAUSE: Main (id 4, prio 1) pauses, active = 027
60 PAR: M10 (id 2, prio 0) forks M2 (3) with prio 0
61 PARE: M10 (id 2, prio 0) has descendants 010
62 PAUSE: M2 (id 3, prio 0) pauses, active = 017
63 JPPAUSE: M10 (id 2, prio 0) does not join
64 PPAUSE: M10 (id 2, prio 0) sets prio to 1
65 PAUSE: M10 (id 2, prio 1) pauses, active = 07
66 PAUSE: M11 (id 1, prio 0) pauses, active = 03
67 ===== TICK 2 terminates after 18 instructions, enabled = 037.
68 ===== Resulting signals: R (2), X0 (3), Y0 (7), Z (10), Outputs OK.
69
70 ===== TICK 3 STARTS, inputs = 01, enabled = 037
71 ===== Inputs: A (0)
72 ===== Enabled: TickEnd (0), M11 (1), M10 (2), M2 (3), Main (4)
73 PRESENT: Main (id 4, prio 1) determines R (2) as absent
74 PAUSE: Main (id 4, prio 1) pauses, active = 037
75 PRESENT: M10 (id 2, prio 1) determines A (0) as present
76 ISATCALL:M10 (id 2, prio 1) _does_ call function
77 EMIT: M10 (id 2, prio 1) emits Y2 (9)
78 RET: M10 (id 2, prio 1) returns from function
79 CALL: M10 (id 2, prio 1) calls function
80 EMIT: M10 (id 2, prio 1) emits Y1 (8)
81 RET: M10 (id 2, prio 1) returns from function
82 EMIT: M10 (id 2, prio 1) emits X10 (5)
83 TRANS: M10 (id 2, prio 1) transfers , enabled = 027
84 PAUSE: M10 (id 2, prio 1) pauses, active = 07
85 PAUSE: M11 (id 1, prio 0) pauses, active = 03
86 ===== TICK 3 terminates after 13 instructions, enabled = 027.
87 ===== Resulting signals: A (0), X10 (5), Y1 (8), Y2 (9), Outputs OK.
88
89 ===== Executed tickMax = 4 ticks!
90 ##### RUN 0 terminates after 59 instructions
91
92 ##### RUN 1 STARTS #####
93 ===== TICK 0 STARTS, inputs = 00, enabled = 00
94 ===== Inputs: <none>
95 ===== Enabled: <none>
96 PAR: Main (id 4, prio 1) forks M10 (2) with prio 0
97 PAR: Main (id 4, prio 1) forks M11 (1) with prio 0
98 PARE: Main (id 4, prio 1) has descendants 016
99 PAUSE: Main (id 4, prio 1) pauses, active = 027
100 PAR: M10 (id 2, prio 0) forks M2 (3) with prio 0
101 PARE: M10 (id 2, prio 0) has descendants 010
102 PAUSE: M2 (id 3, prio 0) pauses, active = 017
103 JPPAUSE: M10 (id 2, prio 0) does not join
104 PPAUSE: M10 (id 2, prio 0) sets prio to 1
105 PAUSE: M10 (id 2, prio 1) pauses, active = 07
106 PAUSE: M11 (id 1, prio 0) pauses, active = 03
107 ===== TICK 0 terminates after 9 instructions, enabled = 037.
108 ===== Resulting signals: <none>, Outputs OK.
109
110 ===== TICK 1 STARTS, inputs = 04, enabled = 037
111 ===== Inputs: R (2)
112 ===== Enabled: TickEnd (0), M11 (1), M10 (2), M2 (3), Main (4)
113 PRESENT: Main (id 4, prio 1) determines R (2) as present
114 ISATCALL:Main (id 4, prio 1) _does_ call function
115 EMIT: Main (id 4, prio 1) emits Y2 (9)
116 RET: Main (id 4, prio 1) returns from function
117 ISATCALL:Main (id 4, prio 1) _does_ call function
118 EMIT: Main (id 4, prio 1) emits Y1 (8)
119 RET: Main (id 4, prio 1) returns from function
120 CALL: Main (id 4, prio 1) calls function
121 EMIT: Main (id 4, prio 1) emits Z (10)
122 RET: Main (id 4, prio 1) returns from function

```

```

123 EMIT: Main (id 4, prio 1) emits Y0 (7)
124 EMIT: Main (id 4, prio 1) emits X0 (3)
125 TRANS: Main (id 4, prio 1) transfers , enabled = 021
126 PAR: Main (id 4, prio 1) forks M10 (2) with prio 0
127 PAR: Main (id 4, prio 1) forks M11 (1) with prio 0
128 PARE: Main (id 4, prio 1) has descendants 016
129 PAUSE: Main (id 4, prio 1) pauses, active = 027
130 PAR: M10 (id 2, prio 0) forks M2 (3) with prio 0
131 PARE: M10 (id 2, prio 0) has descendants 010
132 PAUSE: M2 (id 3, prio 0) pauses, active = 017
133 JPPAUSE: M10 (id 2, prio 0) does not join
134 PPAUSE: M10 (id 2, prio 0) sets prio to 1
135 PAUSE: M10 (id 2, prio 1) pauses, active = 07
136 PAUSE: M11 (id 1, prio 0) pauses, active = 03
137 ===== TICK 1 terminates after 22 instructions, enabled = 037.
138 ===== Resulting signals: R (2), X0 (3), Y0 (7), Y1 (8), Y2 (9), Z (10),
      Outputs OK.
139
140 ===== TICK 2 STARTS, inputs = 00, enabled = 037
141 ===== Inputs: <none>
142 ===== Enabled: TickEnd (0), M11 (1), M10 (2), M2 (3), Main (4)
143 PRESENT: Main (id 4, prio 1) determines R (2) as absent
144 PAUSE: Main (id 4, prio 1) pauses, active = 037
145 PRESENT: M10 (id 2, prio 1) determines A (0) as absent
146 PRIO: M10 (id 2, prio 1) set to priority 0
147 PRESENT: M2 (id 3, prio 0) determines B (1) as absent
148 PAUSE: M2 (id 3, prio 0) pauses, active = 017
149 JPPAUSE: M10 (id 2, prio 0) does not join
150 PPAUSE: M10 (id 2, prio 0) sets prio to 1
151 PAUSE: M10 (id 2, prio 1) pauses, active = 07
152 PAUSE: M11 (id 1, prio 0) pauses, active = 03
153 ===== TICK 2 terminates after 8 instructions, enabled = 037.
154 ===== Resulting signals: <none>, Outputs OK.
155
156 ===== TICK 3 STARTS, inputs = 03, enabled = 037
157 ===== Inputs: A (0), B (1)
158 ===== Enabled: TickEnd (0), M11 (1), M10 (2), M2 (3), Main (4)
159 PRESENT: Main (id 4, prio 1) determines R (2) as absent
160 PAUSE: Main (id 4, prio 1) pauses, active = 037
161 PRESENT: M10 (id 2, prio 1) determines A (0) as present
162 ISATCALL:M10 (id 2, prio 1) .does_ call function
163 EMIT: M10 (id 2, prio 1) emits Y2 (9)
164 RET: M10 (id 2, prio 1) returns from function
165 CALL: M10 (id 2, prio 1) calls function
166 EMIT: M10 (id 2, prio 1) emits Y1 (8)
167 RET: M10 (id 2, prio 1) returns from function
168 EMIT: M10 (id 2, prio 1) emits X10 (5)
169 TRANS: M10 (id 2, prio 1) transfers , enabled = 027
170 PAUSE: M10 (id 2, prio 1) pauses, active = 07
171 PAUSE: M11 (id 1, prio 0) pauses, active = 03
172 ===== TICK 3 terminates after 13 instructions, enabled = 027.
173 ===== Resulting signals: A (0), B (1), X10 (5), Y1 (8), Y2 (9), Outputs OK.
174
175 ===== Executed tickMax = 4 ticks!
176 ##### RUN 1 terminates after 52 instructions
177
178 ##### All runs terminate, after 111 instructions

```

## B.6 Exits-no-isatcall

Listing B.13: Exits-no-isatcall.c

```

1 // Example from Charles Andr, Semantics of SyncCharts,
2 // ISRN I3S/RR-2003-24-FR, April 2003, Figure 8-8
3 //
4 // rvh, 20 mar 2009
5
6 // This example demonstrates how to implement Exit Actions
7
8 #define USEPRIO // Select appropriate dispatcher
9 #include "sc.h"
10
11 #define RUNMAX 2 // # of runs to execute
12 #define TICKMAX 4 // # of ticks to execute
13
14 int runMax = RUNMAX; // # of runs to execute
15 int tickMax = TICKMAX; // # of ticks to execute
16
17
18 // =====
19 // Program-specific definitions
20
21 // Signals
22 typedef enum {A, B, R, X0, X2, X10, X11, Y0, Y1, Y2, Z} signaltype;
23 const char *s2signame[] = {"A", "B", "R", "X0", "X2", "X10", "X11",

```

```

24 "Y0", "Y1", "Y2", "Z"};
25
26 // Thread ids
27 int idHi = 4; // Highest thread id in use
28 typedef enum { TickEnd, M11, M10, M2, Main } idtype;
29 const int ids[] = {0, 1, 2, 3, 4};
30 const char *id2threadname[] = { "TickEnd", "M11", "M10", "M2", "Main" };
31
32 // Inputs for RUNMAX runs of TICKMAX ticks
33 signalvector inputs[RUNMAX][TICKMAX] =
34 { {0, u2b(B), u2b(R), u2b(A)},
35 {0, u2b(R), 0, u2b(A) | u2b(B)} };
36
37 // Expected outputs
38 signalvector outputs[RUNMAX][TICKMAX] =
39 { {0, u2b(Y2) | u2b(X2) | u2b(Y1) | u2b(X11),
40 u2b(Z) | u2b(Y0) | u2b(X0), u2b(Y2) | u2b(Y1) | u2b(X10)},
41 {0, u2b(Y2) | u2b(Z) | u2b(Y1) | u2b(X0), 0, u2b(Y2) | u2b(Y1)
42 | u2b(X10)} };
43
44 void getInputs()
45 {
46 signals = inputs[runCnt][tickCnt];
47 }
48
49 // Set reference outputs and check valued signals , if there are any.
50 // Return 1 unless valued signal outputs are wrong.
51 // No valued signals here, therefore always return 1.
52 int checkOutputs(signalvector *tickOutputs)
53 {
54 *tickOutputs = outputs[runCnt][tickCnt];
55 return 1;
56 }
57
58 // No valued signals to print
59 void printVal(int id)
60 {
61 }
62
63 // Returns 1 if some thread is still active in current tick
64 // Notes:
65 // - CALL calls exit actions unconditionally
66 // - CHKCALL calls exit actions if the corresponding state is active
67 // (and now gets aborted)
68 // - At L2, we set the priority low (0) for M10main, so that
69 // it executes the \codefont{JOIN} at the end of a tick.
70 // - At M10main, we set the priority high (1) for M10depth, so that it
71 // checks for a strong abort at the beginning of a tick.
72 int tick(int islnit)
73 {
74 // Thread ids: M11=1, M10=2, M2=3, Main=4
75 TICKSTART(islnit);
76 M0: PAR(0, M10, ids[M10]);
77 PAR(0, M11, ids[M11]);
78 PARE(1, M0main, id2b(M10) | id2b(M11) | id2b(M2));
79
80 M10: PAR(0, M2, ids[M2]);
81 PARE(0, M10main, id2b(M2));
82
83 M2: PAUSE(M2depth);
84 M2depth:PRESENT(B, M2);
85 CALL(M2exit, L1);
86 M2exit: EMIT(Y2);
87 RET;
88 L1: EMIT(X2);
89 TERM;
90
91 L2: PRIO(0, M10main);
92 M10main:JPPAUSE(1, L3, M10depth);
93 L3: ISAT(ids[M2], M2depth, L4);
94 CALL(M2exit, L4);
95 L4: CALL(M10exit, L5);
96 M10exit:EMIT(Y1);
97 RET;
98 L5: EMIT(X11);
99 TRANS(Done);
100 M10depth:PRESENT(A, L2);
101 ISAT(ids[M2], M2depth, L7);
102 CALL(M2exit, L7);
103 L7: CALL(M10exit, L8);
104 L8: EMIT(X10);
105 TRANS(Done);
106 Done: PAUSE(Done);
107
108 M11: PAUSE(M11);
109 M11exit:EMIT(Z);
110 RET;
111 M0main: PAUSE(L9);

```

```

112 L9: PRESENT(R, M0main);
113 ISAT(ids[M2], M2depth, L10);
114 CALL(M2exit, L10);
115 L10: ISAT(ids[M10], M10depth, L11);
116 CALL(M10exit, L11);
117 L11: CALL(M11exit, L12);
118 L12: EMIT(Y0);
119 EMIT(X0);
120 TRANS(M0);
121
122 TICKEND;
123 }
124
125 // Local Variables :
126 // compile-command: "make exits; exits"
127 // End:

```

## Listing B.14: Exits-no-isatcall.out

```

1 ##### RUN 0 STARTS #####
2 ===== TICK 0 STARTS, inputs = 00, enabled = 00
3 ===== Inputs: <none>
4 ===== Enabled: <none>
5 PAR: Main (id 4, prio 0) forks M10 (2) with prio 0
6 PAR: Main (id 4, prio 0) forks M11 (1) with prio 0
7 PARE: Main (id 4, prio 0) has descendants 016
8 PAUSE: Main (id 4, prio 1) pauses, active = 027
9 PAR: M10 (id 2, prio 0) forks M2 (3) with prio 0
10 PARE: M10 (id 2, prio 0) has descendants 010
11 PAUSE: M2 (id 3, prio 0) pauses, active = 017
12 JPPAUSE: M10 (id 2, prio 0) does not join
13 PPAUSE: M10 (id 2, prio 0) sets prio to 1
14 PAUSE: M10 (id 2, prio 1) pauses, active = 07
15 PAUSE: M11 (id 1, prio 0) pauses, active = 03
16 ===== TICK 0 terminates after 9 instructions, enabled = 037.
17 ===== Resulting signals: <none>, Outputs OK.
18
19 ===== TICK 1 STARTS, inputs = 02, enabled = 037
20 ===== Inputs: B (1)
21 ===== Enabled: TickEnd (0), M11 (1), M10 (2), M2 (3), Main (4)
22 PRESENT: Main (id 4, prio 1) determines R (2) as absent
23 PAUSE: Main (id 4, prio 1) pauses, active = 037
24 PRESENT: M10 (id 2, prio 1) determines A (0) as absent
25 PRIO: M10 (id 2, prio 1) set to priority 0
26 PRESENT: M2 (id 3, prio 0) determines B (1) as present
27 CALL: M2 (id 3, prio 0) calls function
28 EMIT: M2 (id 3, prio 0) emits Y2 (9)
29 RET: M2 (id 3, prio 0) returns from function
30 EMIT: M2 (id 3, prio 0) emits X2 (4)
31 TERM: M2 (id 3, prio 0) terminates, enabled = 027
32 JPPAUSE: M10 (id 2, prio 0) joins
33 ISAT: M10 (id 2, prio 0) is .not_ at probed label
34 CALL: M10 (id 2, prio 0) calls function
35 EMIT: M10 (id 2, prio 0) emits Y1 (8)
36 RET: M10 (id 2, prio 0) returns from function
37 EMIT: M10 (id 2, prio 0) emits X11 (6)
38 TRANS: M10 (id 2, prio 0) transfers, enabled = 027
39 PAUSE: M10 (id 2, prio 0) pauses, active = 07
40 PAUSE: M11 (id 1, prio 0) pauses, active = 03
41 ===== TICK 1 terminates after 19 instructions, enabled = 027.
42 ===== Resulting signals: B (1), X2 (4), X11 (6), Y1 (8), Y2 (9), Outputs OK
43
44 ===== TICK 2 STARTS, inputs = 04, enabled = 027
45 ===== Inputs: R (2)
46 ===== Enabled: TickEnd (0), M11 (1), M10 (2), Main (4)
47 PRESENT: Main (id 4, prio 1) determines R (2) as present
48 ISAT: Main (id 4, prio 1) is .not_ at probed label
49 ISAT: Main (id 4, prio 1) is .not_ at probed label
50 CALL: Main (id 4, prio 1) calls function
51 EMIT: Main (id 4, prio 1) emits Z (10)
52 RET: Main (id 4, prio 1) returns from function
53 EMIT: Main (id 4, prio 1) emits Y0 (7)
54 EMIT: Main (id 4, prio 1) emits X0 (3)
55 TRANS: Main (id 4, prio 1) transfers, enabled = 021
56 PAR: Main (id 4, prio 1) forks M10 (2) with prio 0
57 PAR: Main (id 4, prio 1) forks M11 (1) with prio 0
58 PARE: Main (id 4, prio 1) has descendants 016
59 PAUSE: Main (id 4, prio 1) pauses, active = 027
60 PAR: M10 (id 2, prio 0) forks M2 (3) with prio 0
61 PARE: M10 (id 2, prio 0) has descendants 010
62 PAUSE: M2 (id 3, prio 0) pauses, active = 017
63 JPPAUSE: M10 (id 2, prio 0) does not join
64 PPAUSE: M10 (id 2, prio 0) sets prio to 1
65 PAUSE: M10 (id 2, prio 1) pauses, active = 07
66 PAUSE: M11 (id 1, prio 0) pauses, active = 03
67 ===== TICK 2 terminates after 18 instructions, enabled = 037.
68 ===== Resulting signals: R (2), X0 (3), Y0 (7), Z (10), Outputs OK.

```

```

69 ===== TICK 3 STARTS, inputs = 01, enabled = 037
70 ===== Inputs: A (0)
71 ===== Enabled: TickEnd (0), M11 (1), M10 (2), M2 (3), Main (4)
72 PRESENT: Main (id 4, prio 1) determines R (2) as absent
73 PAUSE: Main (id 4, prio 1) pauses, active = 037
74 PRESENT: M10 (id 2, prio 1) determines A (0) as present
75 ISAT: M10 (id 2, prio 1) .is_ at probed label
76 CALL: M10 (id 2, prio 1) calls function
77 EMIT: M10 (id 2, prio 1) emits Y2 (9)
78 RET: M10 (id 2, prio 1) returns from function
79 CALL: M10 (id 2, prio 1) calls function
80 EMIT: M10 (id 2, prio 1) emits Y1 (8)
81 RET: M10 (id 2, prio 1) returns from function
82 EMIT: M10 (id 2, prio 1) emits X10 (5)
83 TRANS: M10 (id 2, prio 1) transfers, enabled = 027
84 PAUSE: M10 (id 2, prio 1) pauses, active = 07
85 PAUSE: M11 (id 1, prio 0) pauses, active = 03
86 ===== TICK 3 terminates after 14 instructions, enabled = 027.
87 ===== Resulting signals: A (0), X10 (5), Y1 (8), Y2 (9), Outputs OK.
88
89 ===== Executed tickMax = 4 ticks!
90 ##### RUN 0 terminates after 60 instructions
91
92 ##### RUN 1 STARTS #####
93 ===== TICK 0 STARTS, inputs = 00, enabled = 00
94 ===== Inputs: <none>
95 ===== Enabled: <none>
96 PAR: Main (id 4, prio 1) forks M10 (2) with prio 0
97 PAR: Main (id 4, prio 1) forks M11 (1) with prio 0
98 PARE: Main (id 4, prio 1) has descendants 016
99 PAUSE: Main (id 4, prio 1) pauses, active = 027
100 PAR: M10 (id 2, prio 0) forks M2 (3) with prio 0
101 PARE: M10 (id 2, prio 0) has descendants 010
102 PAUSE: M2 (id 3, prio 0) pauses, active = 017
103 JPPAUSE: M10 (id 2, prio 0) does not join
104 PPAUSE: M10 (id 2, prio 0) sets prio to 1
105 PAUSE: M10 (id 2, prio 1) pauses, active = 07
106 PAUSE: M11 (id 1, prio 0) pauses, active = 03
107 ===== TICK 0 terminates after 9 instructions, enabled = 037.
108 ===== Resulting signals: <none>, Outputs OK.
109
110 ===== TICK 1 STARTS, inputs = 04, enabled = 037
111 ===== Inputs: R (2)
112 ===== Enabled: TickEnd (0), M11 (1), M10 (2), M2 (3), Main (4)
113 PRESENT: Main (id 4, prio 1) determines R (2) as present
114 ISAT: Main (id 4, prio 1) .is_ at probed label
115 CALL: Main (id 4, prio 1) calls function
116 EMIT: Main (id 4, prio 1) emits Y2 (9)
117 RET: Main (id 4, prio 1) returns from function
118 ISAT: Main (id 4, prio 1) .is_ at probed label
119 CALL: Main (id 4, prio 1) calls function
120 EMIT: Main (id 4, prio 1) emits Y1 (8)
121 RET: Main (id 4, prio 1) returns from function
122 CALL: Main (id 4, prio 1) calls function
123 EMIT: Main (id 4, prio 1) emits Z (10)
124 RET: Main (id 4, prio 1) returns from function
125 EMIT: Main (id 4, prio 1) emits Y0 (7)
126 EMIT: Main (id 4, prio 1) emits X0 (3)
127 TRANS: Main (id 4, prio 1) transfers, enabled = 021
128 PAR: Main (id 4, prio 1) forks M10 (2) with prio 0
129 PAR: Main (id 4, prio 1) forks M11 (1) with prio 0
130 PARE: Main (id 4, prio 1) has descendants 016
131 PAUSE: Main (id 4, prio 1) pauses, active = 027
132 PAR: M10 (id 2, prio 0) forks M2 (3) with prio 0
133 PARE: M10 (id 2, prio 0) has descendants 010
134 PAUSE: M2 (id 3, prio 0) pauses, active = 017
135 JPPAUSE: M10 (id 2, prio 0) does not join
136 PPAUSE: M10 (id 2, prio 0) sets prio to 1
137 PAUSE: M10 (id 2, prio 1) pauses, active = 07
138 PAUSE: M11 (id 1, prio 0) pauses, active = 03
139 ===== TICK 1 terminates after 24 instructions, enabled = 037.
140 ===== Resulting signals: R (2), X0 (3), Y0 (7), Y1 (8), Y2 (9), Z (10),
141 Outputs OK.
142
143 ===== TICK 2 STARTS, inputs = 00, enabled = 037
144 ===== Inputs: <none>
145 ===== Enabled: TickEnd (0), M11 (1), M10 (2), M2 (3), Main (4)
146 PRESENT: Main (id 4, prio 1) determines R (2) as absent
147 PAUSE: Main (id 4, prio 1) pauses, active = 037
148 PRESENT: M10 (id 2, prio 1) determines A (0) as absent
149 PRIO: M10 (id 2, prio 1) set to priority 0
150 PRESENT: M2 (id 3, prio 0) determines B (1) as absent
151 PAUSE: M2 (id 3, prio 0) pauses, active = 017
152 JPPAUSE: M10 (id 2, prio 0) does not join
153 PPAUSE: M10 (id 2, prio 0) sets prio to 1
154 PAUSE: M10 (id 2, prio 1) pauses, active = 07
155 PAUSE: M11 (id 1, prio 0) pauses, active = 03
156 ===== TICK 2 terminates after 8 instructions, enabled = 037.

```

```

157 ===== Resulting signals: <none>, Outputs OK.
158
159 ===== TICK 3 STARTS, inputs = 03, enabled = 037
160 ===== Inputs: A (0), B (1)
161 ===== Enabled: TickEnd (0), M11 (1), M10 (2), M2 (3), Main (4)
162 PRESENT: Main (id 4, prio 1) determines R (2) as absent
163 PAUSE: Main (id 4, prio 1) pauses, active = 037
164 PRESENT: M10 (id 2, prio 1) determines A (0) as present
165 ISAT: M10 (id 2, prio 1) .is. at probed label
166 CALL: M10 (id 2, prio 1) calls function
167 EMIT: M10 (id 2, prio 1) emits Y2 (9)
168 RET: M10 (id 2, prio 1) returns from function
169 CALL: M10 (id 2, prio 1) calls function
170 EMIT: M10 (id 2, prio 1) emits Y1 (8)
171 RET: M10 (id 2, prio 1) returns from function
172 EMIT: M10 (id 2, prio 1) emits X10 (5)
173 TRANS: M10 (id 2, prio 1) transfers, enabled = 027
174 PAUSE: M10 (id 2, prio 1) pauses, active = 07
175 PAUSE: M11 (id 1, prio 0) pauses, active = 03
176 ===== TICK 3 terminates after 14 instructions, enabled = 027.
177 ===== Resulting signals: A (0), B (1), X10 (5), Y1 (8), Y2 (9), Outputs OK.
178
179 ===== Executed tickMax = 4 ticks!
180 ##### RUN 1 terminates after 55 instructions
181
182 ##### All runs terminate, after 115 instructions

```

## B.7 Exits-inlined

Listing B.15: Exits-inlined.c

```

1 // Example from Charles Andr, Semantics of SyncCharts,
2 // ISRN I35/RR-2003-24-FR, April 2003, Figure 8-8
3 //
4 // rvh, 20 mar 2009
5
6 // This example demonstrates how to implement Exit Actions
7
8 #define USEPRIO // Select appropriate dispatcher
9 #include "sc.h"
10
11 #define RUNMAX 2 // # of runs to execute
12 #define TICKMAX 4 // # of ticks to execute
13
14 int runMax = RUNMAX; // # of runs to execute
15 int tickMax = TICKMAX; // # of ticks to execute
16
17
18 // =====
19 // Program-specific definitions
20
21 // Signals
22 typedef enum {A, B, R, X0, X2, X10, X11, Y0, Y1, Y2, Z} signaltype;
23 const char *s2signame[] = {"A", "B", "R", "X0", "X2", "X10", "X11",
24 "Y0", "Y1", "Y2", "Z"};
25
26 // Thread ids
27 int idHi = 4; // Highest thread id in use
28 typedef enum { TickEnd, M11, M10, M2, Main } idtype;
29 const int ids[] = {0, 1, 2, 3, 4};
30 const char *id2threadname[] = {"TickEnd", "M11", "M10", "M2", "Main"};
31
32 // Inputs for RUNMAX runs of TICKMAX ticks
33 signalvector inputs[RUNMAX][TICKMAX] =
34 { {0, u2b(B), u2b(R), u2b(A)},
35 {0, u2b(R), 0, u2b(A) | u2b(B)} };
36
37 // Expected outputs
38 signalvector outputs[RUNMAX][TICKMAX] =
39 { {0, u2b(Y2) | u2b(X2) | u2b(Y1) | u2b(X11),
40 u2b(Z) | u2b(Y0) | u2b(X0), u2b(Y2) | u2b(Y1) | u2b(X10)},
41 {0, u2b(Y2) | u2b(Z) | u2b(Y1) | u2b(Y0) | u2b(X0), 0, u2b(Y2) | u2b(Y1)
42 | u2b(X10)} };
43
44 void getInputs()
45 {
46 signals = inputs[runCnt][tickCnt];
47 }
48
49 // Set reference outputs and check valued signals, if there are any.
50 // Return 1 unless valued signal outputs are wrong.
51 // No valued signals here, therefore always return 1.
52 int checkOutputs(signalvector *tickOutputs)
53 {
54 *tickOutputs = outputs[runCnt][tickCnt];

```

```

54 return 1;
55 }
56
57 // No valued signals to print
58 void printVal(int id)
59 {
60 }
61
62 // Returns 1 if some thread is still active in current tick
63 // Notes:
64 // - At L2, we set the priority low (0) for M10main, so that
65 // it executes the \codefont{JOIN} at the end of a tick.
66 // - At M10main, we set the priority high (1) for M10depth, so that it
67 // checks for a strong abort at the beginning of a tick.
68 // - Could save "Done: PAUSE(Done)" by changing "TRANS(Done)" to "
69 TRANS(M11)"
70 int tick(int islnit)
71 {
72 // Thread ids: M11=1, M10=2, M2=3, Main=4
73 TICKSTART(islnit);
74 M0: PAR(0, M10, ids[M10]);
75 PAR(0, M11, ids[M11]);
76 PARE(1, M0main, id2b(M10) | id2b(M11) | id2b(M2));
77
78 M10: PAR(0, M2, ids[M2]);
79 PARE(0, M10main, id2b(M2));
80
81 M2: PAUSE(M2depth);
82 M2depth:PRESENT(B, M2);
83 EMIT(Y2);
84 EMIT(X2);
85 TERM;
86
87 L2: PRIO(0, M10main);
88 M10main:JPPAUSE(1, L3, M10depth);
89 L3: ISAT(ids[M2], M2depth, L4);
90 EMIT(Y2);
91 L4: EMIT(Y1);
92 EMIT(X11);
93 TRANS(Done);
94 M10depth:PRESENT(A, L2);
95 ISAT(ids[M2], M2depth, L7);
96 EMIT(Y2);
97 L7: EMIT(Y1);
98 EMIT(X10);
99 TRANS(Done);
100 Done: PAUSE(Done);
101
102 M11: PAUSE(M11);
103
104 M0main: PAUSE(L9);
105 L9: PRESENT(R, M0main);
106 ISAT(ids[M2], M2depth, L10);
107 EMIT(Y2);
108 L10: ISAT(ids[M10], M10depth, L11);
109 EMIT(Y1);
110 L11: EMIT(Z);
111 EMIT(Y0);
112 EMIT(X0);
113 TRANS(M0);
114
115 TICKEND;
116 }
117
118 // Local Variables :
119 // compile-command: "make exits; exits"
120 // End:

```

Listing B.16: Exits-inlined.out

```

1 ##### RUN 0 STARTS #####
2 ===== TICK 0 STARTS, inputs = 00, enabled = 00
3 ===== Inputs: <none>
4 ===== Enabled: <none>
5 PAR: Main (id 4, prio 0) forks M10 (2) with prio 0
6 PAR: Main (id 4, prio 0) forks M11 (1) with prio 0
7 PARE: Main (id 4, prio 0) has descendants 016
8 PAUSE: Main (id 4, prio 1) pauses, active = 027
9 PAR: M10 (id 2, prio 0) forks M2 (3) with prio 0
10 PARE: M10 (id 2, prio 0) has descendants 010
11 PAUSE: M2 (id 3, prio 0) pauses, active = 017
12 JPPAUSE: M10 (id 2, prio 0) does not join
13 PPAUSE: M10 (id 2, prio 0) sets prio to 1
14 PAUSE: M10 (id 2, prio 1) pauses, active = 07
15 PAUSE: M11 (id 1, prio 0) pauses, active = 03
16 ===== TICK 0 terminates after 9 instructions, enabled = 037.
17 ===== Resulting signals: <none>, Outputs OK.
18

```

```

19  ===== TICK 1 STARTS, inputs = 02, enabled = 037
20  ===== Inputs: B (1)
21  ===== Enabled: TickEnd (0), M11 (1), M10 (2), M2 (3), Main (4)
22  PRESENT: Main (id 4, prio 1) determines R (2) as absent
23  PAUSE: Main (id 4, prio 1) pauses, active = 037
24  PRESENT: M10 (id 2, prio 1) determines A (0) as absent
25  PRIO: M10 (id 2, prio 1) set to priority 0
26  PRESENT: M2 (id 3, prio 0) determines B (1) as present
27  EMIT: M2 (id 3, prio 0) emits Y2 (9)
28  EMIT: M2 (id 3, prio 0) emits X2 (4)
29  TERM: M2 (id 3, prio 0) terminates, enabled = 027
30  JPPAUSE: M10 (id 2, prio 0) joins
31  ISAT: M10 (id 2, prio 0) is .not. at probed label
32  EMIT: M10 (id 2, prio 0) emits Y1 (8)
33  EMIT: M10 (id 2, prio 0) emits X11 (6)
34  TRANS: M10 (id 2, prio 0) transfers, enabled = 027
35  PAUSE: M10 (id 2, prio 0) pauses, active = 07
36  PAUSE: M11 (id 1, prio 0) pauses, active = 03
37  ===== TICK 1 terminates after 15 instructions, enabled = 027.
38  ===== Resulting signals: B (1), X2 (4), X11 (6), Y1 (8), Y2 (9), Outputs OK

39  ===== TICK 2 STARTS, inputs = 04, enabled = 027
40  ===== Inputs: R (2)
41  ===== Enabled: TickEnd (0), M11 (1), M10 (2), Main (4)
42  PRESENT: Main (id 4, prio 1) determines R (2) as present
43  ISAT: Main (id 4, prio 1) is .not. at probed label
44  ISAT: Main (id 4, prio 1) is .not. at probed label
45  EMIT: Main (id 4, prio 1) emits Z (10)
46  EMIT: Main (id 4, prio 1) emits Y0 (7)
47  EMIT: Main (id 4, prio 1) emits X0 (3)
48  TRANS: Main (id 4, prio 1) transfers, enabled = 021
49  PAR: Main (id 4, prio 1) forks M10 (2) with prio 0
50  PAR: Main (id 4, prio 1) forks M11 (1) with prio 0
51  PARE: Main (id 4, prio 1) has descendants 016
52  PAUSE: Main (id 4, prio 1) pauses, active = 027
53  PAR: M10 (id 2, prio 0) forks M2 (3) with prio 0
54  PARE: M10 (id 2, prio 0) has descendants 010
55  PAUSE: M2 (id 3, prio 0) pauses, active = 017
56  JPPAUSE: M10 (id 2, prio 0) does not join
57  PPAUSE: M10 (id 2, prio 0) sets prio to 1
58  PAUSE: M10 (id 2, prio 1) pauses, active = 07
59  PAUSE: M11 (id 1, prio 0) pauses, active = 03
60  ===== TICK 2 terminates after 16 instructions, enabled = 037.
61  ===== Resulting signals: R (2), X0 (3), Y0 (7), Z (10), Outputs OK.

62  ===== TICK 3 STARTS, inputs = 01, enabled = 037
63  ===== Inputs: A (0)
64  ===== Enabled: TickEnd (0), M11 (1), M10 (2), M2 (3), Main (4)
65  PRESENT: Main (id 4, prio 1) determines R (2) as absent
66  PAUSE: Main (id 4, prio 1) pauses, active = 037
67  PRESENT: M10 (id 2, prio 1) determines A (0) as present
68  ISAT: M10 (id 2, prio 1) .is. at probed label
69  EMIT: M10 (id 2, prio 1) emits Y2 (9)
70  EMIT: M10 (id 2, prio 1) emits Y1 (8)
71  EMIT: M10 (id 2, prio 1) emits X10 (5)
72  TRANS: M10 (id 2, prio 1) transfers, enabled = 027
73  PAUSE: M10 (id 2, prio 1) pauses, active = 07
74  PAUSE: M11 (id 1, prio 0) pauses, active = 03
75  ===== TICK 3 terminates after 10 instructions, enabled = 027.
76  ===== Resulting signals: A (0), X10 (5), Y1 (8), Y2 (9), Outputs OK.

77  ===== Executed tickMax = 4 ticks!
78  ##### RUN 0 terminates after 50 instructions
79
80  ##### RUN 1 STARTS #####
81  ===== TICK 0 STARTS, inputs = 00, enabled = 00
82  ===== Inputs: <none>
83  ===== Enabled: <none>
84  PAR: Main (id 4, prio 1) forks M10 (2) with prio 0
85  PAR: Main (id 4, prio 1) forks M11 (1) with prio 0
86  PARE: Main (id 4, prio 1) has descendants 016
87  PAUSE: Main (id 4, prio 1) pauses, active = 027
88  PAR: M10 (id 2, prio 0) forks M2 (3) with prio 0
89  PARE: M10 (id 2, prio 0) has descendants 010
90  PAUSE: M2 (id 3, prio 0) pauses, active = 017
91  JPPAUSE: M10 (id 2, prio 0) does not join
92  PPAUSE: M10 (id 2, prio 0) sets prio to 1
93  PAUSE: M10 (id 2, prio 1) pauses, active = 07
94  PAUSE: M11 (id 1, prio 0) pauses, active = 03
95  ===== TICK 0 terminates after 9 instructions, enabled = 037.
96  ===== Resulting signals: <none>, Outputs OK.

97  ===== TICK 1 STARTS, inputs = 04, enabled = 037
98  ===== Inputs: R (2)
99  ===== Enabled: TickEnd (0), M11 (1), M10 (2), M2 (3), Main (4)
100 PRESENT: Main (id 4, prio 1) determines R (2) as present
101 ISAT: Main (id 4, prio 1) .is. at probed label
102 EMIT: Main (id 4, prio 1) emits Y2 (9)

```

```

107 ISAT: Main (id 4, prio 1) .is. at probed label
108 EMIT: Main (id 4, prio 1) emits Y1 (8)
109 EMIT: Main (id 4, prio 1) emits Z (10)
110 EMIT: Main (id 4, prio 1) emits Y0 (7)
111 EMIT: Main (id 4, prio 1) emits X0 (3)
112 TRANS: Main (id 4, prio 1) transfers, enabled = 021
113 PAR: Main (id 4, prio 1) forks M10 (2) with prio 0
114 PAR: Main (id 4, prio 1) forks M11 (1) with prio 0
115 PARE: Main (id 4, prio 1) has descendants 016
116 PAUSE: Main (id 4, prio 1) pauses, active = 027
117 PAR: M10 (id 2, prio 0) forks M2 (3) with prio 0
118 PARE: M10 (id 2, prio 0) has descendants 010
119 PAUSE: M2 (id 3, prio 0) pauses, active = 017
120 JPPAUSE: M10 (id 2, prio 0) does not join
121 PPAUSE: M10 (id 2, prio 0) sets prio to 1
122 PAUSE: M10 (id 2, prio 1) pauses, active = 07
123 PAUSE: M11 (id 1, prio 0) pauses, active = 03
124 ===== TICK 1 terminates after 18 instructions, enabled = 037.
125 ===== Resulting signals: R (2), X0 (3), Y0 (7), Y1 (8), Y2 (9), Z (10),
    Outputs OK.

126
127 ===== TICK 2 STARTS, inputs = 00, enabled = 037
128 ===== Inputs: <none>
129 ===== Enabled: TickEnd (0), M11 (1), M10 (2), M2 (3), Main (4)
130 PRESENT: Main (id 4, prio 1) determines R (2) as absent
131 PAUSE: Main (id 4, prio 1) pauses, active = 037
132 PRESENT: M10 (id 2, prio 1) determines A (0) as absent
133 PRIO: M10 (id 2, prio 1) set to priority 0
134 PRESENT: M2 (id 3, prio 0) determines B (1) as absent
135 PAUSE: M2 (id 3, prio 0) pauses, active = 017
136 JPPAUSE: M10 (id 2, prio 0) does not join
137 PPAUSE: M10 (id 2, prio 0) sets prio to 1
138 PAUSE: M10 (id 2, prio 1) pauses, active = 07
139 PAUSE: M11 (id 1, prio 0) pauses, active = 03
140 ===== TICK 2 terminates after 8 instructions, enabled = 037.
141 ===== Resulting signals: <none>, Outputs OK.

142
143 ===== TICK 3 STARTS, inputs = 03, enabled = 037
144 ===== Inputs: A (0), B (1)
145 ===== Enabled: TickEnd (0), M11 (1), M10 (2), M2 (3), Main (4)
146 PRESENT: Main (id 4, prio 1) determines R (2) as absent
147 PAUSE: Main (id 4, prio 1) pauses, active = 037
148 PRESENT: M10 (id 2, prio 1) determines A (0) as present
149 ISAT: M10 (id 2, prio 1) .is. at probed label
150 EMIT: M10 (id 2, prio 1) emits Y2 (9)
151 EMIT: M10 (id 2, prio 1) emits Y1 (8)
152 EMIT: M10 (id 2, prio 1) emits X10 (5)
153 TRANS: M10 (id 2, prio 1) transfers, enabled = 027
154 PAUSE: M10 (id 2, prio 1) pauses, active = 07
155 PAUSE: M11 (id 1, prio 0) pauses, active = 03
156 ===== TICK 3 terminates after 10 instructions, enabled = 027.
157 ===== Resulting signals: A (0), B (1), X10 (5), Y1 (8), Y2 (9), Outputs OK.

158
159 ===== Executed tickMax = 4 ticks!
160 ##### RUN 1 terminates after 45 instructions
161
162 ##### All runs terminate, after 95 instructions

```

## B.8 FilteredSR

### Listing B.17: FilteredSR.c

```

1 // Example from Charles Andr, Semantics of SyncCharts,
2 // ISRN I3S/RR-2003-24-FR, April 2003, Figure 8-18
3 //
4 // rvh, 25 mar 2009
5
6 // This example illustrates the use of PRE on pure signals, and also
7 // how to encode signal conjunction (eg the check for "S and pre(S)")
8 // via control flow.
9
10 // Use PRE operator - must define this before including sc.h
11 #define usePRE
12 #include "sc.h"
13
14 #define RUNMAX 1 // # of runs to execute
15 #define TICKMAX 9 // # of ticks to execute
16
17 int runMax = RUNMAX; // # of runs to execute
18 int tickMax = TICKMAX; // # of ticks to execute
19
20
21 // =====
22 // Program-specific definitions
23

```

```

24 // Signals
25 typedef enum {S, R, ON, OFF} signaltype;
26 const char *s2sname[] = {"S", "R", "ON", "OFF"};
27
28 // Thread ids
29 // Note: WaitA gets a higher id than WaitB (rather than the other way
30 // around) simply to let WaitA execute first, to make the trace match
31 // the syntactical flow of the program
32 int idHi = 1; // Highest thread id in use
33 typedef enum { TickEnd, Main } idtype;
34 const int ids[] = { 0, 1 };
35 const char *id2threadname[] = { "TickEnd", "Main" };
36
37 // Inputs for RUNMAX runs of TICKMAX ticks
38 signalvector inputs[RUNMAX][TICKMAX] =
39 { {u2b(S), u2b(S), u2b(S), u2b(S), 0, u2b(R), u2b(R), u2b(R), 0} };
40
41 // Expected outputs
42 signalvector outputs[RUNMAX][TICKMAX] =
43 { {u2b(OFF), u2b(ON), u2b(ON), u2b(ON), u2b(ON), u2b(ON), u2b(OFF),
44 u2b(OFF), u2b(OFF)} };
45
46 void getInputs()
47 {
48 signals = inputs[runCnt][tickCnt];
49 }
50
51 // Set reference outputs and check valued signals, if there are any.
52 // Return 1 unless valued signal outputs are wrong.
53 // No valued signals here, therefore always return 1.
54 int checkOutputs(signalvector *tickOutputs)
55 {
56 *tickOutputs = outputs[runCnt][tickCnt];
57 return 1;
58 }
59
60 // No valued signals to print
61 void printVal(int id)
62 {
63 }
64
65 // Returns 1 if some thread is still active in current tick
66 int tick(int islnit)
67 {
68 // Thread ids: Main=1
69 TICKSTART(islnit);
70
71 Off: EMIT(OFF);
72 PAUSE(OffDepth);
73 OffDepth:PRESENT(S, Off);
74 PRESENTPRE(S, Off);
75
76 On: EMIT(ON);
77 PAUSE(OnDepth);
78 OnDepth:PRESENT(R, On);
79 PRESENTPRE(R, On);
80
81 GOTO(Off);
82
83 TICKEND;
84 }
85
86 // Local Variables:
87 // compile--command: "make filteredSR; filteredSR"
88 // End:

```

Listing B.18: FilteredSR.out

```

1 ##### RUN 0 STARTS #####
2 ===== TICK 0 STARTS, inputs = 01, enabled = 00
3 ===== Inputs: S (0)
4 ===== Enabled: <none>
5 EMIT: Main (id 1, prio 0) emits OFF (3)
6 PAUSE: Main (id 1, prio 0) pauses, active = 03
7 ===== TICK 0 terminates after 2 instructions, enabled = 03.
8 ===== Resulting signals: S (0), OFF (3), Outputs OK.
9
10 ===== TICK 1 STARTS, inputs = 01, enabled = 03
11 ===== Inputs: S (0)
12 ===== Enabled: TickEnd (0), Main (1)
13 PRESENT: Main (id 1, prio 0) determines S (0) as present
14 PRESENTPRE:Main (id 1, prio 0) determines previous S (0) as present
15 EMIT: Main (id 1, prio 0) emits ON (2)
16 PAUSE: Main (id 1, prio 0) pauses, active = 03
17 ===== TICK 1 terminates after 4 instructions, enabled = 03.
18 ===== Resulting signals: S (0), ON (2), Outputs OK.
19
20 ===== TICK 2 STARTS, inputs = 01, enabled = 03

```

```

21 ===== Inputs: S (0)
22 ===== Enabled: TickEnd (0), Main (1)
23 PRESENT: Main (id 1, prio 0) determines R (1) as absent
24 EMIT: Main (id 1, prio 0) emits ON (2)
25 PAUSE: Main (id 1, prio 0) pauses, active = 03
26 ===== TICK 2 terminates after 3 instructions, enabled = 03.
27 ===== Resulting signals: S (0), ON (2), Outputs OK.
28
29 ===== TICK 3 STARTS, inputs = 01, enabled = 03
30 ===== Inputs: S (0)
31 ===== Enabled: TickEnd (0), Main (1)
32 PRESENT: Main (id 1, prio 0) determines R (1) as absent
33 EMIT: Main (id 1, prio 0) emits ON (2)
34 PAUSE: Main (id 1, prio 0) pauses, active = 03
35 ===== TICK 3 terminates after 3 instructions, enabled = 03.
36 ===== Resulting signals: S (0), ON (2), Outputs OK.
37
38 ===== TICK 4 STARTS, inputs = 00, enabled = 03
39 ===== Inputs: <none>
40 ===== Enabled: TickEnd (0), Main (1)
41 PRESENT: Main (id 1, prio 0) determines R (1) as absent
42 EMIT: Main (id 1, prio 0) emits ON (2)
43 PAUSE: Main (id 1, prio 0) pauses, active = 03
44 ===== TICK 4 terminates after 3 instructions, enabled = 03.
45 ===== Resulting signals: ON (2), Outputs OK.
46
47 ===== TICK 5 STARTS, inputs = 02, enabled = 03
48 ===== Inputs: R (1)
49 ===== Enabled: TickEnd (0), Main (1)
50 PRESENT: Main (id 1, prio 0) determines R (1) as present
51 PRESENTPRE:Main (id 1, prio 0) determines previous R (1) as absent
52 EMIT: Main (id 1, prio 0) emits ON (2)
53 PAUSE: Main (id 1, prio 0) pauses, active = 03
54 ===== TICK 5 terminates after 4 instructions, enabled = 03.
55 ===== Resulting signals: R (1), ON (2), Outputs OK.
56
57 ===== TICK 6 STARTS, inputs = 02, enabled = 03
58 ===== Inputs: R (1)
59 ===== Enabled: TickEnd (0), Main (1)
60 PRESENT: Main (id 1, prio 0) determines R (1) as present
61 PRESENTPRE:Main (id 1, prio 0) determines previous R (1) as present
62 EMIT: Main (id 1, prio 0) emits OFF (3)
63 PAUSE: Main (id 1, prio 0) pauses, active = 03
64 ===== TICK 6 terminates after 5 instructions, enabled = 03.
65 ===== Resulting signals: R (1), OFF (3), Outputs OK.
66
67 ===== TICK 7 STARTS, inputs = 02, enabled = 03
68 ===== Inputs: R (1)
69 ===== Enabled: TickEnd (0), Main (1)
70 PRESENT: Main (id 1, prio 0) determines S (0) as absent
71 EMIT: Main (id 1, prio 0) emits OFF (3)
72 PAUSE: Main (id 1, prio 0) pauses, active = 03
73 ===== TICK 7 terminates after 3 instructions, enabled = 03.
74 ===== Resulting signals: R (1), OFF (3), Outputs OK.
75
76 ===== TICK 8 STARTS, inputs = 00, enabled = 03
77 ===== Inputs: <none>
78 ===== Enabled: TickEnd (0), Main (1)
79 PRESENT: Main (id 1, prio 0) determines S (0) as absent
80 EMIT: Main (id 1, prio 0) emits OFF (3)
81 PAUSE: Main (id 1, prio 0) pauses, active = 03
82 ===== TICK 8 terminates after 3 instructions, enabled = 03.
83 ===== Resulting signals: OFF (3), Outputs OK.
84
85 ===== Executed tickMax = 9 ticks!
86 ##### RUN 0 terminates after 30 instructions
87
88 ##### All runs terminate, after 30 instructions

```

## B.9 PreAndSuspend

Listing B.19: PreAndSuspend.c

```

1 // Example from Charles Andr., Semantics of SyncCharts,
2 // ISBN 135/RR-2003-24-FR, April 2003, Figure 8-20
3 //
4 // rvh, 5 mar 2009
5
6 // This example illustrates the use of valued signals and PRE
7
8 // Must define the following before including sc.h
9 #define usePRE // Use PRE operator
10
11 #include "sc.h"
12

```



```

13 // =====
14 // Program-specific definitions
15
16 #define RUNMAX 1 // # of runs to execute
17 int runMax = RUNMAX;
18
19 #define TICKMAX 13 // # of ticks to execute
20 int tickMax = TICKMAX;
21
22 // Signals
23 // Valued signals come first, as their index is used to index value arrays
24 // If multiple types are used, can use appropriate offset for indexing
   arrays
25 typedef enum { T, B0, B1, C } signaltype;
26 const char *s2signal[] = { "T", "B0", "B1", "C" };
27
28 // Thread ids
29 int idHi = 4; // Highest thread id in use
30 typedef enum { TickEnd, Off1, Off0, Cnt, Main } idtype;
31 const int ids[] = { 0, 1, 2, 3, 4 };
32 const char *id2threadname[] = { "TickEnd", "Off1", "Off0", "Cnt", "Main" };
33
34 // Locally declared signals - to handle suspension properly
35 // Here, indicate that C is declared locally to Main (state Mod3Cnt in
   Andr03)
36 signalvector sigsDescs[] = { 0, 0, 0, 0, u2b(C) };
37
38 // Inputs for RUNMAX runs of TICKMAX ticks
39 // See Table 8-2 from C. Andr
40 signalvector inputs[RUNMAX][TICKMAX] =
41 { {0, u2b(T), 0, u2b(T), 0,
42 u2b(T), 0, 0, u2b(T), 0,
43 u2b(T), u2b(T), 0} };
44
45 // Expected outputs
46 signalvector outputs[RUNMAX][TICKMAX] =
47 { {0, u2b(B0), 0, u2b(B1) | u2b(C), 0,
48 0, 0, 0, u2b(B0), 0,
49 u2b(B1) | u2b(C), 0, 0} };
50
51 void getInputs()
52 {
53 signals = inputs[runCnt][tickCnt];
54 }
55
56 // Set reference outputs and check valued signals, if there are any
57 // Return 1 unless valued signal outputs are wrong
58 int checkOutputs(signalvector *tickOutputs)
59 {
60 *tickOutputs = outputs[runCnt][tickCnt];
61 return 1;
62 }
63
64 // Print value of a signal, if it has one
65 void printVal(int id)
66 {
67 }
68
69
70 // Returns 1 if some thread is still active in current tick
71 int tick(int islnit)
72 {
73 // Thread ids: Off1=1, Off0=2, Cnt=3, Main=4
74 TICKSTART(islnit);
75
76 PAR(0, Cnt, ids[Cnt]);
77 PARE(0, Mod3CntMain, id2b(Cnt) | id2b(Off1) | id2b(Off0));
78
79 Cnt: PAR(0, Off1, ids[Off1]);
80 PAR(0, Off0, ids[Off0]);
81 PARE(0, CntMain, id2b(Off1) | id2b(Off0));
82
83 Off1: PAUSE(L0);
84 L0: PRESENT(C, Off1);
85
86 On1: EMIT(B1);
87 PAUSE(On1);
88
89 L1: EMIT(C);
90 Off0: PAUSE(On0);
91
92 On0: EMIT(B0);
93 PAUSE(L1);
94
95 CntDepth:PRESENTPRE(C, CntMain);
96 TRANS(Cnt);
97 CntMain:PAUSE(CntDepth);
98
99 Mod3CntMain:PAUSE(Mod3CntDepth);

```

```

100 Mod3CntDepth:PRESENT(T, L2);
101 GOTO(Mod3CntMain);
102 L2: SUSPEND(Mod3CntDepth);
103
104 TICKEND;
105 }
106
107 // Local Variables :
108 // compile-command: "make preAndSuspend; preAndSuspend"
109 // End:

```

## Listing B.20: PreAndSuspend.out

```

1 ##### RUN 0 STARTS #####
2 ===== TICK 0 STARTS, inputs = 00, enabled = 00
3 ===== Inputs: <none>
4 ===== Enabled: <none>
5 PAR: Main (id 4, prio 0) forks Cnt (3) with prio 0
6 PARE: Main (id 4, prio 0) has descendants 016
7 PAUSE: Main (id 4, prio 0) pauses, active = 031
8 PAR: Cnt (id 3, prio 0) forks Off1 (1) with prio 0
9 PAR: Cnt (id 3, prio 0) forks Off0 (2) with prio 0
10 PARE: Cnt (id 3, prio 0) has descendants 06
11 PAUSE: Cnt (id 3, prio 0) pauses, active = 017
12 PAUSE: Off0 (id 2, prio 0) pauses, active = 07
13 PAUSE: Off1 (id 1, prio 0) pauses, active = 03
14 ===== TICK 0 terminates after 9 instructions, enabled = 037.
15 ===== Resulting signals: <none>, Outputs OK.
16
17 ===== TICK 1 STARTS, inputs = 01, enabled = 037
18 ===== Inputs: T (0)
19 ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Cnt (3), Main (4)
20 PRESENT: Main (id 4, prio 0) determines T (0) as present
21 PAUSE: Main (id 4, prio 0) pauses, active = 037
22 PRESENTPRE:Cnt (id 3, prio 0) determines previous C (3) as absent
23 PAUSE: Cnt (id 3, prio 0) pauses, active = 017
24 EMIT: Off0 (id 2, prio 0) emits B0 (1)
25 PAUSE: Off0 (id 2, prio 0) pauses, active = 07
26 PRESENT: Off1 (id 1, prio 0) determines C (3) as absent
27 PAUSE: Off1 (id 1, prio 0) pauses, active = 03
28 ===== TICK 1 terminates after 9 instructions, enabled = 037.
29 ===== Resulting signals: T (0), B0 (1), Outputs OK.
30
31 ===== TICK 2 STARTS, inputs = 00, enabled = 037
32 ===== Inputs: <none>
33 ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Cnt (3), Main (4)
34 PRESENT: Main (id 4, prio 0) determines T (0) as absent
35 SUSPEND: Main (id 4, prio 0) suspends itself and descendants 016
36 PAUSE: Main (id 4, prio 0) pauses, active = 021
37 ===== TICK 2 terminates after 2 instructions, enabled = 037.
38 ===== Resulting signals: <none>, Outputs OK.
39
40 ===== TICK 3 STARTS, inputs = 01, enabled = 037
41 ===== Inputs: T (0)
42 ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Cnt (3), Main (4)
43 PRESENT: Main (id 4, prio 0) determines T (0) as present
44 PAUSE: Main (id 4, prio 0) pauses, active = 037
45 PRESENTPRE:Cnt (id 3, prio 0) determines previous C (3) as absent
46 PAUSE: Cnt (id 3, prio 0) pauses, active = 017
47 EMIT: Off0 (id 2, prio 0) emits C (3)
48 PAUSE: Off0 (id 2, prio 0) pauses, active = 07
49 PRESENT: Off1 (id 1, prio 0) determines C (3) as present
50 EMIT: Off1 (id 1, prio 0) emits B1 (2)
51 PAUSE: Off1 (id 1, prio 0) pauses, active = 03
52 ===== TICK 3 terminates after 10 instructions, enabled = 037.
53 ===== Resulting signals: T (0), B1 (2), C (3), Outputs OK.
54
55 ===== TICK 4 STARTS, inputs = 00, enabled = 037
56 ===== Inputs: <none>
57 ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Cnt (3), Main (4)
58 PRESENT: Main (id 4, prio 0) determines T (0) as absent
59 SUSPEND: Main (id 4, prio 0) suspends itself and descendants 016
60 PAUSE: Main (id 4, prio 0) pauses, active = 021
61 ===== TICK 4 terminates after 2 instructions, enabled = 037.
62 ===== Resulting signals: <none>, Outputs OK.
63
64 ===== TICK 5 STARTS, inputs = 01, enabled = 037
65 ===== Inputs: T (0)
66 ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Cnt (3), Main (4)
67 PRESENT: Main (id 4, prio 0) determines T (0) as present
68 PAUSE: Main (id 4, prio 0) pauses, active = 037
69 PRESENTPRE:Cnt (id 3, prio 0) determines previous C (3) as present
70 TRANS: Cnt (id 3, prio 0) transfers, enabled = 031
71 PAR: Cnt (id 3, prio 0) forks Off1 (1) with prio 0
72 PAR: Cnt (id 3, prio 0) forks Off0 (2) with prio 0
73 PARE: Cnt (id 3, prio 0) has descendants 06
74 PAUSE: Cnt (id 3, prio 0) pauses, active = 017
75 PAUSE: Off0 (id 2, prio 0) pauses, active = 07

```

## B.10 PrimeFactor

### Listing B.21: PrimeFactor.c

```

76 PAUSE: Off1 (id 1, prio 0) pauses, active = 03
77 ===== TICK 5 terminates after 11 instructions, enabled = 037.
78 ===== Resulting signals: T (0), Outputs OK.
79
80 ===== TICK 6 STARTS, inputs = 00, enabled = 037
81 ===== Inputs: <none>
82 ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Cnt (3), Main (4)
83 PRESENT: Main (id 4, prio 0) determines T (0) as absent
84 SUSPEND: Main (id 4, prio 0) suspends itself and descendants 016
85 PAUSE: Main (id 4, prio 0) pauses, active = 021
86 ===== TICK 6 terminates after 2 instructions, enabled = 037.
87 ===== Resulting signals: <none>, Outputs OK.
88
89 ===== TICK 7 STARTS, inputs = 00, enabled = 037
90 ===== Inputs: <none>
91 ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Cnt (3), Main (4)
92 PRESENT: Main (id 4, prio 0) determines T (0) as absent
93 SUSPEND: Main (id 4, prio 0) suspends itself and descendants 016
94 PAUSE: Main (id 4, prio 0) pauses, active = 021
95 ===== TICK 7 terminates after 2 instructions, enabled = 037.
96 ===== Resulting signals: <none>, Outputs OK.
97
98 ===== TICK 8 STARTS, inputs = 01, enabled = 037
99 ===== Inputs: T (0)
100 ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Cnt (3), Main (4)
101 PRESENT: Main (id 4, prio 0) determines T (0) as present
102 PAUSE: Main (id 4, prio 0) pauses, active = 037
103 PRESENTPRE:Cnt (id 3, prio 0) determines previous C (3) as absent
104 PAUSE: Cnt (id 3, prio 0) pauses, active = 017
105 EMIT: Off0 (id 2, prio 0) emits B0 (1)
106 PAUSE: Off0 (id 2, prio 0) pauses, active = 07
107 PRESENT: Off1 (id 1, prio 0) determines C (3) as absent
108 PAUSE: Off1 (id 1, prio 0) pauses, active = 03
109 ===== TICK 8 terminates after 9 instructions, enabled = 037.
110 ===== Resulting signals: T (0), B0 (1), Outputs OK.
111
112 ===== TICK 9 STARTS, inputs = 00, enabled = 037
113 ===== Inputs: <none>
114 ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Cnt (3), Main (4)
115 PRESENT: Main (id 4, prio 0) determines T (0) as absent
116 SUSPEND: Main (id 4, prio 0) suspends itself and descendants 016
117 PAUSE: Main (id 4, prio 0) pauses, active = 021
118 ===== TICK 9 terminates after 2 instructions, enabled = 037.
119 ===== Resulting signals: <none>, Outputs OK.
120
121 ===== TICK 10 STARTS, inputs = 01, enabled = 037
122 ===== Inputs: T (0)
123 ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Cnt (3), Main (4)
124 PRESENT: Main (id 4, prio 0) determines T (0) as present
125 PAUSE: Main (id 4, prio 0) pauses, active = 037
126 PRESENTPRE:Cnt (id 3, prio 0) determines previous C (3) as absent
127 PAUSE: Cnt (id 3, prio 0) pauses, active = 017
128 EMIT: Off0 (id 2, prio 0) emits C (3)
129 PAUSE: Off0 (id 2, prio 0) pauses, active = 07
130 PRESENT: Off1 (id 1, prio 0) determines C (3) as present
131 EMIT: Off1 (id 1, prio 0) emits B1 (2)
132 PAUSE: Off1 (id 1, prio 0) pauses, active = 03
133 ===== TICK 10 terminates after 10 instructions, enabled = 037.
134 ===== Resulting signals: T (0), B1 (2), C (3), Outputs OK.
135
136 ===== TICK 11 STARTS, inputs = 01, enabled = 037
137 ===== Inputs: T (0)
138 ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Cnt (3), Main (4)
139 PRESENT: Main (id 4, prio 0) determines T (0) as present
140 PAUSE: Main (id 4, prio 0) pauses, active = 037
141 PRESENTPRE:Cnt (id 3, prio 0) determines previous C (3) as present
142 TRANS: Cnt (id 3, prio 0) transfers, enabled = 031
143 PAR: Cnt (id 3, prio 0) forks Off1 (1) with prio 0
144 PAR: Cnt (id 3, prio 0) forks Off0 (2) with prio 0
145 PARE: Cnt (id 3, prio 0) has descendants 06
146 PAUSE: Cnt (id 3, prio 0) pauses, active = 017
147 PAUSE: Off0 (id 2, prio 0) pauses, active = 07
148 PAUSE: Off1 (id 1, prio 0) pauses, active = 03
149 ===== TICK 11 terminates after 11 instructions, enabled = 037.
150 ===== Resulting signals: T (0), Outputs OK.
151
152 ===== TICK 12 STARTS, inputs = 00, enabled = 037
153 ===== Inputs: <none>
154 ===== Enabled: TickEnd (0), Off1 (1), Off0 (2), Cnt (3), Main (4)
155 PRESENT: Main (id 4, prio 0) determines T (0) as absent
156 SUSPEND: Main (id 4, prio 0) suspends itself and descendants 016
157 PAUSE: Main (id 4, prio 0) pauses, active = 021
158 ===== TICK 12 terminates after 2 instructions, enabled = 037.
159 ===== Resulting signals: <none>, Outputs OK.
160
161 ===== Executed tickMax = 13 ticks!
162 ##### RUN 0 terminates after 81 instructions
163
164 ##### All runs terminate, after 81 instructions

```

```

1 // Example from Charles Andr., Semantics of SyncCharts,
2 // ISRN I3S/RR-2003-24-FR, April 2003, Figure 8-25
3 //
4 // rvh, 5 mar 2009
5
6 // This example illustrates the use of valued signals and the proper
7 // handling of reincarnation /schizophrenia.
8 //
9 // Remarkably, all scheduling constraints are handled by proper
10 // ordering of the transition predicate tests, and by the fact that
11 // the id of the inner state (S0, id 1) is higher than the priority of
12 // the surrounding root thread.
13
14 #include "sc.h"
15
16 // =====
17 // Program-specific definitions
18
19 #define RUNMAX 2 // # of runs to execute
20 #define TICKMAX 2 // # of ticks to execute
21
22 int runMax = RUNMAX; // # of runs to execute
23 int tickMax = TICKMAX; // # of ticks to execute
24
25 // Signals
26 // Valued signals come first, as their index is used to index value arrays
27 // If multiple types are used, can use appropriate offset for indexing
   arrays
28 typedef enum {V, A, B, C, D} signaltype;
29 const char *s2signame[] = {"V", "A", "B", "C", "D"};
30
31 // Define valued int signals, combined with * (signal "V")
32 #define valSigIntMultCnt 1 // Number of signals
33 int valSigIntMult [valSigIntMultCnt]; // Values
34
35 // Thread ids
36 int idHi = 2; // Highest thread id in use
37 typedef enum { TickEnd, Main, S1 } idtype;
38 const int ids [] = { 0, 1, 2 };
39 const char *id2threadname[] = {"TickEnd", "Main", "S1"};
40
41 // Inputs for RUNMAX runs of TICKMAX ticks
42 signalvector inputs [RUNMAX][TICKMAX] =
43   {{0, u2b(B)},
44    {0, u2b(A)u2b(B)u2b(C)u2b(D)}};
45
46 // Expected outputs
47 signalvector outputs [RUNMAX][TICKMAX] =
48   {{u2b(V), u2b(V)},
49    {u2b(V), u2b(V)}};
50
51 int outputValues_V [RUNMAX][TICKMAX] =
52   {{2, 5},
53    {2, 11550}};
54
55 void getInputs ()
56 {
57   signals = inputs [runCnt][tickCnt];
58   valSigIntMult [0] = 1;
59 }
60
61 // Set reference outputs and check valued signals, if there are any
62 // Return 1 unless valued signal outputs are wrong
63 int checkOutputs (signalvector *tickOutputs)
64 {
65   int isOk;
66
67   *tickOutputs = outputs [runCnt][tickCnt];
68   isOk = (valSigIntMult [V] == outputValues_V [runCnt][tickCnt]);
69
70   if (!isOk)
71     printf ("\nERROR: Value of V is %d, should be %d!\n",
72            valSigIntMult [V], outputValues_V [runCnt][tickCnt]);
73
74   return isOk;
75 }
76
77 // No valued signals to print
78 void printVal (int id)
79 {
80 }
81
82

```

```

83 // Returns 1 if some thread is still active in current tick
84 // Notes:
85 // - S0 needs no join, as it never terminates normally
86 // - "S2: PAUSE(S2)" encodes final, but non-terminating state (HALT)
87 int tick(int islnit)
88 {
89 // Thread ids: Main=1, S1=2
90 TICKSTART(islnit);
91
92 S0: PAR(0, S1, ids[S1]);
93     PARE(0, S0main, id2b(S1));
94
95 S1: EMITINTMUL(V, 2);
96
97 S1surf: PRESENT(B, S1depth);
98     EMITINTMUL(V, 5);
99     GOTO(S2);
100 S1depth: PAUSE(L0);
101 L0: PRESENT(A, S1surf);
102     EMITINTMUL(V, 3);
103     GOTO(S1surf);
104
105 S2: PAUSE(S2);
106
107 S0main: PRESENT(D, S0depth);
108     EMITINTMUL(V, 11);
109     TRANS(S3);
110 S0depth: PAUSE(L1);
111 L1: PRESENT(C, S0main);
112     EMITINTMUL(V, 7);
113     TRANS(S0);
114
115 S3: PAUSE(S3);
116     TICKEND;
117 }
118
119 // Local Variables :
120 // compile-command: "make PrimeFactor; PrimeFactor"
121 // End:

```

### Listing B.22: PrimeFactor.out

```

1 ##### RUN 0 STARTS #####
2 ===== TICK 0 STARTS, inputs = 00, enabled = 00
3 ===== Inputs: <none>
4 ===== Enabled: <none>
5 PAR: Main (id 1, prio 0) forks S1 (2) with prio 0
6 PARE: Main (id 1, prio 0) has descendants 04
7 EMITInt*:S1 (id 2, prio 0) emits V (0), value 2, result 2
8 PRESENT: S1 (id 2, prio 0) determines B (2) as absent
9 PAUSE: S1 (id 2, prio 0) pauses, active = 07
10 PRESENT: Main (id 1, prio 0) determines D (4) as absent
11 PAUSE: Main (id 1, prio 0) pauses, active = 03
12 ===== TICK 0 terminates after 7 instructions, enabled = 07.
13 ===== Resulting signals: V (0), Outputs OK.
14
15 ===== TICK 1 STARTS, inputs = 04, enabled = 07
16 ===== Inputs: B (2)
17 ===== Enabled: TickEnd (0), Main (1), S1 (2)
18 PRESENT: S1 (id 2, prio 0) determines A (1) as absent
19 PRESENT: S1 (id 2, prio 0) determines B (2) as present
20 EMITInt*:S1 (id 2, prio 0) emits V (0), value 5, result 5
21 PAUSE: S1 (id 2, prio 0) pauses, active = 07
22 PRESENT: Main (id 1, prio 0) determines C (3) as absent
23 PRESENT: Main (id 1, prio 0) determines D (4) as absent
24 PAUSE: Main (id 1, prio 0) pauses, active = 03
25 ===== TICK 1 terminates after 8 instructions, enabled = 07.
26 ===== Resulting signals: V (0), B (2), Outputs OK.
27
28 ===== Executed tickMax = 2 ticks!
29 ##### RUN 0 terminates after 15 instructions
30
31 ##### RUN 1 STARTS #####
32 ===== TICK 0 STARTS, inputs = 00, enabled = 00
33 ===== Inputs: <none>
34 ===== Enabled: <none>
35 PAR: Main (id 1, prio 0) forks S1 (2) with prio 0
36 PARE: Main (id 1, prio 0) has descendants 04
37 EMITInt*:S1 (id 2, prio 0) emits V (0), value 2, result 2
38 PRESENT: S1 (id 2, prio 0) determines B (2) as absent
39 PAUSE: S1 (id 2, prio 0) pauses, active = 07
40 PRESENT: Main (id 1, prio 0) determines D (4) as absent
41 PAUSE: Main (id 1, prio 0) pauses, active = 03
42 ===== TICK 0 terminates after 7 instructions, enabled = 07.
43 ===== Resulting signals: V (0), Outputs OK.
44
45 ===== TICK 1 STARTS, inputs = 036, enabled = 07
46 ===== Inputs: A (1), B (2), C (3), D (4)

```

```

47 ===== Enabled: TickEnd (0), Main (1), S1 (2)
48 PRESENT: S1 (id 2, prio 0) determines A (1) as present
49 EMITInt*:S1 (id 2, prio 0) emits V (0), value 3, result 3
50 PRESENT: S1 (id 2, prio 0) determines B (2) as present
51 EMITInt*:S1 (id 2, prio 0) emits V (0), value 5, result 15
52 PAUSE: S1 (id 2, prio 0) pauses, active = 07
53 PRESENT: Main (id 1, prio 0) determines C (3) as present
54 EMITInt*:Main (id 1, prio 0) emits V (0), value 7, result 105
55 TRANS: Main (id 1, prio 0) transfers, enabled = 03
56 PAR: Main (id 1, prio 0) forks S1 (2) with prio 0
57 PARE: Main (id 1, prio 0) has descendants 04
58 EMITInt*:S1 (id 2, prio 0) emits V (0), value 2, result 210
59 PRESENT: S1 (id 2, prio 0) determines B (2) as present
60 EMITInt*:S1 (id 2, prio 0) emits V (0), value 5, result 1050
61 PAUSE: S1 (id 2, prio 0) pauses, active = 07
62 PRESENT: Main (id 1, prio 0) determines D (4) as present
63 EMITInt*:Main (id 1, prio 0) emits V (0), value 11, result 11550
64 TRANS: Main (id 1, prio 0) transfers, enabled = 03
65 PAUSE: Main (id 1, prio 0) pauses, active = 03
66 ===== TICK 1 terminates after 21 instructions, enabled = 03.
67 ===== Resulting signals: V (0), A (1), B (2), C (3), D (4), Outputs OK.
68
69 ===== Executed tickMax = 2 ticks!
70 ##### RUN 1 terminates after 28 instructions
71
72 ##### All runs terminate, after 43 instructions

```

## B.11 Reincarnation

### Listing B.23: Reincarnation.c

```

1 // Example from Charles Andr, Semantics of SyncCharts,
2 // ISRN 135/RR-2003-24-FR, April 2003, Figure 8-22
3 //
4 // rvh, 20 mar 2009
5
6 // This program illustrates the SIGNAL instruction to handle signal
7 // reincarnation .
8
9 #include "sc.h"
10
11 #define RUNMAX 1 // # of runs to execute
12 #define TICKMAX 4 // # of ticks to execute
13
14 int runMax = RUNMAX; // # of runs to execute
15 int tickMax = TICKMAX; // # of ticks to execute
16
17
18 // =====
19 // Program-specific definitions
20
21 // Signals
22 typedef enum {A, gotS, S} signaltype;
23 const char *s2sname[] = {"A", "gotS", "S"};
24
25 // Thread ids
26 int idHi = 1; // Highest thread id in use
27 typedef enum { TickEnd, Main } idtype;
28 const int ids[] = { 0, 1 };
29 const char *id2threadname[] = { "TickEnd", "Main" };
30
31 // Inputs for RUNMAX runs of TICKMAX ticks
32 signalvector inputs[RUNMAX][TICKMAX] =
33     {{u2b(A), 0, u2b(A), 0}};
34
35 // Expected outputs
36 signalvector outputs[RUNMAX][TICKMAX] =
37     {{0, 0, 0, 0}};
38
39
40 void getInputs()
41 {
42     signals = inputs[runCnt][tickCnt];
43 }
44
45 // Set reference outputs and check valued signals, if there are any.
46 // Return 1 unless valued signal outputs are wrong.
47 // No valued signals here, therefore always return 1.
48 int checkOutputs(signalvector *tickOutputs)
49 {
50     *tickOutputs = outputs[runCnt][tickCnt];
51     return 1;
52 }
53
54 // No valued signals to print

```

```

55 void printVal(int id)
56 {
57 }
58
59 // Returns 1 if some thread is still active in current tick
60 int tick(int islnit)
61 {
62 // Thread ids: Main=1
63     TICKSTART(islnit);
64
65     Reinc: SIGNAL(S);
66           PRESENT(S, Q);
67     P:     EMIT(gotS);
68           PAUSE(P);
69     Q:     PAUSE(L0);
70     L0:    PRESENT(A, Q);
71           EMIT(S);
72           GOTO(Reinc);
73
74     TICKEND;
75 }
76
77 // Local Variables:
78 // compile-command: "make reincarnation; reincarnation"
79 // End:

```

Listing B.24: Reincarnation.out

```

1  ##### RUN 0 STARTS #####
2  ===== TICK 0 STARTS, inputs = 01, enabled = 00
3  ===== Inputs: A (0)
4  ===== Enabled: <none>
5  SIGNAL: Main (id 1, prio 0) initializes S (2)
6  PRESENT: Main (id 1, prio 0) determines S (2) as absent
7  PAUSE: Main (id 1, prio 0) pauses, active = 03
8  ===== TICK 0 terminates after 3 instructions, enabled = 03.
9  ===== Resulting signals: A (0), Outputs OK.
10
11 ===== TICK 1 STARTS, inputs = 00, enabled = 03
12 ===== Inputs: <none>
13 ===== Enabled: TickEnd (0), Main (1)
14 PRESENT: Main (id 1, prio 0) determines A (0) as absent
15 PAUSE: Main (id 1, prio 0) pauses, active = 03
16 ===== TICK 1 terminates after 2 instructions, enabled = 03.
17 ===== Resulting signals: <none>, Outputs OK.
18
19 ===== TICK 2 STARTS, inputs = 01, enabled = 03
20 ===== Inputs: A (0)
21 ===== Enabled: TickEnd (0), Main (1)
22 PRESENT: Main (id 1, prio 0) determines A (0) as present
23 EMIT: Main (id 1, prio 0) emits S (2)
24 SIGNAL: Main (id 1, prio 0) initializes S (2)
25 PRESENT: Main (id 1, prio 0) determines S (2) as absent
26 PAUSE: Main (id 1, prio 0) pauses, active = 03
27 ===== TICK 2 terminates after 6 instructions, enabled = 03.
28 ===== Resulting signals: A (0), Outputs OK.
29
30 ===== TICK 3 STARTS, inputs = 00, enabled = 03
31 ===== Inputs: <none>
32 ===== Enabled: TickEnd (0), Main (1)
33 PRESENT: Main (id 1, prio 0) determines A (0) as absent
34 PAUSE: Main (id 1, prio 0) pauses, active = 03
35 ===== TICK 3 terminates after 2 instructions, enabled = 03.
36 ===== Resulting signals: <none>, Outputs OK.
37
38 ===== Executed tickMax = 4 ticks!
39 ##### RUN 0 terminates after 13 instructions
40
41 ##### All runs terminate, after 13 instructions

```

## B.12 Shifter3

Listing B.25: Shifter3.c

```

1 // Example from Charles Andr, Semantics of SyncCharts,
2 // ISRN I3S/RR-2003-24-FR, April 2003, Figure 8-19
3 //
4 // rvh, 5 mar 2009
5
6 // This example illustrates the use of valued signals and PRE
7
8 // Must define the following before including sc.h
9 #define usePRE // Use PRE operator
10 #define valSigCnt 4 // Number of valued signals

```

```

11
12 #include "sc.h"
13
14 // =====
15 // Program-specific definitions
16
17 #define RUNMAX 1 // # of runs to execute
18 int runMax = RUNMAX;
19
20 #define TICKMAX 12 // # of ticks to execute
21 int tickMax = TICKMAX;
22
23 // Signals
24 // Valued signals come first, as their index is used to index value arrays
25 // If multiple types are used, can use appropriate offset for indexing
26 // arrays
27 typedef enum { S0, S1, O, I } signaltype;
28 const char *s2signame[] = { "S0", "S1", "O", "I" };
29
30 // Define valued int signals
31 int valSigInt [valSigCnt]; // Values
32 int valSigIntPre [valSigCnt]; // Previous values
33
34 // Thread ids
35 int idHi = 3; // Highest thread id in use
36 typedef enum { TickEnd, Main, Shift1, ShiftO } idtype;
37 const int ids[] = { 0, 1, 2, 3 };
38 const char *id2threadname[] = { "TickEnd", "Main", "Shift1", "ShiftO" };
39
40 // Inputs for RUNMAX runs of TICKMAX ticks
41 signalvector inputs[RUNMAX][TICKMAX] =
42 { { 0, u2b(1), 0, u2b(1),
43 0, u2b(1), 0, u2b(1),
44 u2b(1), u2b(1), u2b(1), u2b(1) } };
45
46 // Expected outputs
47 signalvector outputs[RUNMAX][TICKMAX] =
48 { { 0, 0, u2b(S0), u2b(S1),
49 u2b(S0) | u2b(O), u2b(S1), u2b(S0) | u2b(O), u2b(S1),
50 u2b(S0) | u2b(O), u2b(S0) | u2b(S1), u2b(S0) | u2b(S1) | u2b(O), u2b(S0)
51 | u2b(S1) | u2b(O) } };
52
53 // Expected values for signals S0, S1, O
54 // Recall that valued sigs preserve value even if absent
55 int outputValues[RUNMAX][TICKMAX] =
56 { { {-1, -1, 1, 1,
57 3, 3, 5, 5,
58 7, 8, 9, 10} },
59 { {-1, -1, -1, 1,
60 1, 3, 3, 5,
61 5, 7, 8, 9} },
62 { {-1, -1, -1, -1,
63 1, 1, 3, 3,
64 5, 5, 7, 8} } };
65
66 void getInputs()
67 {
68     signals = inputs[runCnt][tickCnt];
69     valSigInt [1] = tickCnt;
70 }
71
72 // Set reference outputs and check valued signals, if there are any
73 // Return 1 unless valued signal outputs are wrong
74 int checkOutputs(signalvector *tickOutputs)
75 {
76     int s;
77
78     *tickOutputs = outputs[runCnt][tickCnt];
79
80     for (s = S0; s <= O; s++)
81         if (valSigInt [s] != outputValues[s][runCnt][tickCnt]) {
82             printf("\nERROR: Value of %s is %d, should be %d!\n",
83                 s2signame[s], valSigInt [s], outputValues[s][runCnt][tickCnt]);
84             return 0;
85         }
86     }
87
88     return 1;
89 }
90
91 // Print value of a signal, if it has one
92 void printVal(int id)
93 {
94     if (id < valSigCnt)
95         printf("=%d", valSigInt[id]);
96 }
97
98 // Returns 1 if some thread is still active in current tick
99 // Notes:

```

```

98 // - As the top-level thread has nothing to do after spawning the
99 // subthreads, it takes on the role of one of the concurrent
100 // subthreads. Or, put another way, the Main thread is one of the
101 // concurrent subthreads (Shift0), and performs two PAR statements to
102 // spawn off the concurrent subthreads.
103 // - Starting with state Shift0 Shift0depth allows to save final GOTO
104 // by folding it into PAUSE
105 int tick(int islnit)
106 {
107 // Thread ids: Main=1, Shift1=2, ShiftO=3
108 int reg0;
109
110 TICKSTART(islnit);
111
112 PAR(0, Shift1, ids [Shift1]);
113 PAR(0, ShiftO, ids [ShiftO]);
114 GOTO(Shift0);
115
116 Shift0depth:PRESENTPRE(I, Shift0);
117 VALPRE(I, reg0);
118 EMITINT(S0, reg0);
119 Shift0: PAUSE(Shift0depth);
120
121 Shift1depth:PRESENTPRE(S0, Shift1);
122 VALPRE(S0, reg0);
123 EMITINT(S1, reg0);
124 Shift1: PAUSE(Shift1depth);
125
126 ShiftOdepth:PRESENTPRE(S1, ShiftO);
127 VALPRE(S1, reg0);
128 EMITINT(O, reg0);
129 ShiftO: PAUSE(ShiftOdepth);
130
131 TICKEND;
132 }
133 // Local Variables:
134 // compile-command: "make shifter3; shifter3"
135 // End:

```

## Listing B.26: Shifter3.out

```

1 ##### RUN 0 STARTS #####
2 ===== TICK 0 STARTS, inputs = 00, enabled = 00
3 ===== Inputs: <none>
4 ===== Enabled: <none>
5 PAR: Main (id 1, prio 0) forks Shift1 (2) with prio 0
6 PAR: Main (id 1, prio 0) forks ShiftO (3) with prio 0
7 PAUSE: Main (id 1, prio 0) pauses, active = 017
8 PAUSE: ShiftO (id 3, prio 0) pauses, active = 015
9 PAUSE: Shift1 (id 2, prio 0) pauses, active = 05
10 ===== TICK 0 terminates after 6 instructions, enabled = 017.
11 ===== Resulting signals: <none>, Outputs OK.
12
13 ===== TICK 1 STARTS, inputs = 010, enabled = 017
14 ===== Inputs: I=1 (3)
15 ===== Enabled: TickEnd (0), Main (1), Shift1 (2), ShiftO (3)
16 PRESENTPRE:ShiftO (id 3, prio 0) determines previous S1 (1) as absent
17 PAUSE: ShiftO (id 3, prio 0) pauses, active = 017
18 PRESENTPRE:Shift1 (id 2, prio 0) determines previous S0 (0) as absent
19 PAUSE: Shift1 (id 2, prio 0) pauses, active = 07
20 PRESENTPRE:Main (id 1, prio 0) determines previous I (3) as absent
21 PAUSE: Main (id 1, prio 0) pauses, active = 03
22 ===== TICK 1 terminates after 6 instructions, enabled = 017.
23 ===== Resulting signals: I=1 (3), Outputs OK.
24
25 ===== TICK 2 STARTS, inputs = 00, enabled = 017
26 ===== Inputs: <none>
27 ===== Enabled: TickEnd (0), Main (1), Shift1 (2), ShiftO (3)
28 PRESENTPRE:ShiftO (id 3, prio 0) determines previous S1 (1) as absent
29 PAUSE: ShiftO (id 3, prio 0) pauses, active = 017
30 PRESENTPRE:Shift1 (id 2, prio 0) determines previous S0 (0) as absent
31 PAUSE: Shift1 (id 2, prio 0) pauses, active = 07
32 PRESENTPRE:Main (id 1, prio 0) determines previous I (3) as present
33 VALPRE: Main (id 1, prio 0) determines value of I (3) as 1
34 EMITInt: Main (id 1, prio 0) emits S0 (0), value 1
35 PAUSE: Main (id 1, prio 0) pauses, active = 03
36 ===== TICK 2 terminates after 8 instructions, enabled = 017.
37 ===== Resulting signals: S0=1 (0), Outputs OK.
38
39 ===== TICK 3 STARTS, inputs = 010, enabled = 017
40 ===== Inputs: I=3 (3)
41 ===== Enabled: TickEnd (0), Main (1), Shift1 (2), ShiftO (3)
42 PRESENTPRE:ShiftO (id 3, prio 0) determines previous S1 (1) as absent
43 PAUSE: ShiftO (id 3, prio 0) pauses, active = 017
44 PRESENTPRE:Shift1 (id 2, prio 0) determines previous S0 (0) as present
45 VALPRE: Shift1 (id 2, prio 0) determines value of S0 (0) as 1

```

```

46 EMITInt: Shift1 (id 2, prio 0) emits S1 (1), value 1
47 PAUSE: Shift1 (id 2, prio 0) pauses, active = 07
48 PRESENTPRE:Main (id 1, prio 0) determines previous I (3) as absent
49 PAUSE: Main (id 1, prio 0) pauses, active = 03
50 ===== TICK 3 terminates after 8 instructions, enabled = 017.
51 ===== Resulting signals: S1=1 (1), I=3 (3), Outputs OK.
52
53 ===== TICK 4 STARTS, inputs = 00, enabled = 017
54 ===== Inputs: <none>
55 ===== Enabled: TickEnd (0), Main (1), Shift1 (2), ShiftO (3)
56 PRESENTPRE:ShiftO (id 3, prio 0) determines previous S1 (1) as present
57 VALPRE: ShiftO (id 3, prio 0) determines value of S1 (1) as 1
58 EMITInt: ShiftO (id 3, prio 0) emits O (2), value 1
59 PAUSE: ShiftO (id 3, prio 0) pauses, active = 017
60 PRESENTPRE:Shift1 (id 2, prio 0) determines previous S0 (0) as absent
61 PAUSE: Shift1 (id 2, prio 0) pauses, active = 07
62 PRESENTPRE:Main (id 1, prio 0) determines previous I (3) as present
63 VALPRE: Main (id 1, prio 0) determines value of I (3) as 3
64 EMITInt: Main (id 1, prio 0) emits S0 (0), value 3
65 PAUSE: Main (id 1, prio 0) pauses, active = 03
66 ===== TICK 4 terminates after 10 instructions, enabled = 017.
67 ===== Resulting signals: S0=3 (0), O=1 (2), Outputs OK.
68
69 ===== TICK 5 STARTS, inputs = 010, enabled = 017
70 ===== Inputs: I=5 (3)
71 ===== Enabled: TickEnd (0), Main (1), Shift1 (2), ShiftO (3)
72 PRESENTPRE:ShiftO (id 3, prio 0) determines previous S1 (1) as absent
73 PAUSE: ShiftO (id 3, prio 0) pauses, active = 017
74 PRESENTPRE:Shift1 (id 2, prio 0) determines previous S0 (0) as present
75 VALPRE: Shift1 (id 2, prio 0) determines value of S0 (0) as 3
76 EMITInt: Shift1 (id 2, prio 0) emits S1 (1), value 3
77 PAUSE: Shift1 (id 2, prio 0) pauses, active = 07
78 PRESENTPRE:Main (id 1, prio 0) determines previous I (3) as absent
79 PAUSE: Main (id 1, prio 0) pauses, active = 03
80 ===== TICK 5 terminates after 8 instructions, enabled = 017.
81 ===== Resulting signals: S1=3 (1), I=5 (3), Outputs OK.
82
83 ===== TICK 6 STARTS, inputs = 00, enabled = 017
84 ===== Inputs: <none>
85 ===== Enabled: TickEnd (0), Main (1), Shift1 (2), ShiftO (3)
86 PRESENTPRE:ShiftO (id 3, prio 0) determines previous S1 (1) as present
87 VALPRE: ShiftO (id 3, prio 0) determines value of S1 (1) as 3
88 EMITInt: ShiftO (id 3, prio 0) emits O (2), value 3
89 PAUSE: ShiftO (id 3, prio 0) pauses, active = 017
90 PRESENTPRE:Shift1 (id 2, prio 0) determines previous S0 (0) as absent
91 PAUSE: Shift1 (id 2, prio 0) pauses, active = 07
92 PRESENTPRE:Main (id 1, prio 0) determines previous I (3) as present
93 VALPRE: Main (id 1, prio 0) determines value of I (3) as 5
94 EMITInt: Main (id 1, prio 0) emits S0 (0), value 5
95 PAUSE: Main (id 1, prio 0) pauses, active = 03
96 ===== TICK 6 terminates after 10 instructions, enabled = 017.
97 ===== Resulting signals: S0=5 (0), O=3 (2), Outputs OK.
98
99 ===== TICK 7 STARTS, inputs = 010, enabled = 017
100 ===== Inputs: I=7 (3)
101 ===== Enabled: TickEnd (0), Main (1), Shift1 (2), ShiftO (3)
102 PRESENTPRE:ShiftO (id 3, prio 0) determines previous S1 (1) as absent
103 PAUSE: ShiftO (id 3, prio 0) pauses, active = 017
104 PRESENTPRE:Shift1 (id 2, prio 0) determines previous S0 (0) as present
105 VALPRE: Shift1 (id 2, prio 0) determines value of S0 (0) as 5
106 EMITInt: Shift1 (id 2, prio 0) emits S1 (1), value 5
107 PAUSE: Shift1 (id 2, prio 0) pauses, active = 07
108 PRESENTPRE:Main (id 1, prio 0) determines previous I (3) as absent
109 PAUSE: Main (id 1, prio 0) pauses, active = 03
110 ===== TICK 7 terminates after 8 instructions, enabled = 017.
111 ===== Resulting signals: S1=5 (1), I=7 (3), Outputs OK.
112
113 ===== TICK 8 STARTS, inputs = 010, enabled = 017
114 ===== Inputs: I=8 (3)
115 ===== Enabled: TickEnd (0), Main (1), Shift1 (2), ShiftO (3)
116 PRESENTPRE:ShiftO (id 3, prio 0) determines previous S1 (1) as present
117 VALPRE: ShiftO (id 3, prio 0) determines value of S1 (1) as 5
118 EMITInt: ShiftO (id 3, prio 0) emits O (2), value 5
119 PAUSE: ShiftO (id 3, prio 0) pauses, active = 017
120 PRESENTPRE:Shift1 (id 2, prio 0) determines previous S0 (0) as absent
121 PAUSE: Shift1 (id 2, prio 0) pauses, active = 07
122 PRESENTPRE:Main (id 1, prio 0) determines previous I (3) as present
123 VALPRE: Main (id 1, prio 0) determines value of I (3) as 7
124 EMITInt: Main (id 1, prio 0) emits S0 (0), value 7
125 PAUSE: Main (id 1, prio 0) pauses, active = 03
126 ===== TICK 8 terminates after 10 instructions, enabled = 017.
127 ===== Resulting signals: S0=7 (0), O=5 (2), I=8 (3), Outputs OK.
128
129 ===== TICK 9 STARTS, inputs = 010, enabled = 017
130 ===== Inputs: I=9 (3)
131 ===== Enabled: TickEnd (0), Main (1), Shift1 (2), ShiftO (3)
132 PRESENTPRE:ShiftO (id 3, prio 0) determines previous S1 (1) as absent
133 PAUSE: ShiftO (id 3, prio 0) pauses, active = 017
134 PRESENTPRE:Shift1 (id 2, prio 0) determines previous S0 (0) as present

```

```

135 VALPRE: Shift1 (id 2, prio 0) determines value of S0 (0) as 7
136 EMITInt: Shift1 (id 2, prio 0) emits S1 (1), value 7
137 PAUSE: Shift1 (id 2, prio 0) pauses, active = 07
138 PRESENTPRE:Main (id 1, prio 0) determines previous I (3) as present
139 VALPRE: Main (id 1, prio 0) determines value of I (3) as 8
140 EMITInt: Main (id 1, prio 0) emits S0 (0), value 8
141 PAUSE: Main (id 1, prio 0) pauses, active = 03
142 ===== TICK 9 terminates after 10 instructions, enabled = 017.
143 ===== Resulting signals: S0=8 (0), S1=7 (1), I=9 (3), Outputs OK.
144
145 ===== TICK 10 STARTS, inputs = 010, enabled = 017
146 ===== Inputs: I=10 (3)
147 ===== Enabled: TickEnd (0), Main (1), Shift1 (2), ShiftO (3)
148 PRESENTPRE:ShiftO (id 3, prio 0) determines previous S1 (1) as present
149 VALPRE: ShiftO (id 3, prio 0) determines value of S1 (1) as 7
150 EMITInt: ShiftO (id 3, prio 0) emits O (2), value 7
151 PAUSE: ShiftO (id 3, prio 0) pauses, active = 017
152 PRESENTPRE:Shift1 (id 2, prio 0) determines previous S0 (0) as present
153 VALPRE: Shift1 (id 2, prio 0) determines value of S0 (0) as 8
154 EMITInt: Shift1 (id 2, prio 0) emits S1 (1), value 8
155 PAUSE: Shift1 (id 2, prio 0) pauses, active = 07
156 PRESENTPRE:Main (id 1, prio 0) determines previous I (3) as present
157 VALPRE: Main (id 1, prio 0) determines value of I (3) as 9
158 EMITInt: Main (id 1, prio 0) emits S0 (0), value 9
159 PAUSE: Main (id 1, prio 0) pauses, active = 03
160 ===== TICK 10 terminates after 12 instructions, enabled = 017.
161 ===== Resulting signals: S0=9 (0), S1=8 (1), O=7 (2), I=10 (3), Outputs OK.
162
163 ===== TICK 11 STARTS, inputs = 010, enabled = 017
164 ===== Inputs: I=11 (3)
165 ===== Enabled: TickEnd (0), Main (1), Shift1 (2), ShiftO (3)
166 PRESENTPRE:ShiftO (id 3, prio 0) determines previous S1 (1) as present
167 VALPRE: ShiftO (id 3, prio 0) determines value of S1 (1) as 8
168 EMITInt: ShiftO (id 3, prio 0) emits O (2), value 8
169 PAUSE: ShiftO (id 3, prio 0) pauses, active = 017
170 PRESENTPRE:Shift1 (id 2, prio 0) determines previous S0 (0) as present
171 VALPRE: Shift1 (id 2, prio 0) determines value of S0 (0) as 9
172 EMITInt: Shift1 (id 2, prio 0) emits S1 (1), value 9
173 PAUSE: Shift1 (id 2, prio 0) pauses, active = 07
174 PRESENTPRE:Main (id 1, prio 0) determines previous I (3) as present
175 VALPRE: Main (id 1, prio 0) determines value of I (3) as 10
176 EMITInt: Main (id 1, prio 0) emits S0 (0), value 10
177 PAUSE: Main (id 1, prio 0) pauses, active = 03
178 ===== TICK 11 terminates after 12 instructions, enabled = 017.
179 ===== Resulting signals: S0=10 (0), S1=9 (1), O=8 (2), I=11 (3), Outputs
      OK.
180
181 ===== Executed tickMax = 12 ticks!
182 ##### RUN 0 terminates after 108 instructions
183 ##### All runs terminate, after 108 instructions

```

## B.13 SurfDepth

Listing B.27: SurfDepth.c

```

1 // Example from Charles Andr, Semantics of SyncCharts,
2 // ISRN I3S/RR-2003-24-FR, April 2003, Figure 8-5
3 //
4 // rvh, 19 mar 2009
5 #include "sc.h"
6
7 #define RUNMAX 2 // # of runs to execute
8 #define TICKMAX 5 // # of ticks to execute
9
10 int runMax = RUNMAX; // # of runs to execute
11 int tickMax = TICKMAX; // # of ticks to execute
12
13
14 // =====
15 // Program-specific definitions
16
17 // Signals
18 typedef enum {A0, A1, B0, B1, C1, U0, U1, V0, V1, W1} signaltype;
19 const char *s2signame[] = {"A0", "A1", "B0", "B1", "C1", "U0", "U1", "V0",
      "V1", "W1"};
20
21 // Thread ids
22 int idHi = 1; // Highest thread id in use
23 typedef enum { TickEnd, Main } idtype;
24 const int ids[] = { 0, 1 };
25 const char *id2threadname[] = { "TickEnd", "Main" };
26
27 // Inputs for RUNMAX runs of TICKMAX ticks
28 signalvector inputs[RUNMAX][TICKMAX] =

```

```

29 { { u2b(A0) | u2b(A1), u2b(A0) | u2b(A1), u2b(A0) | u2b(A1), u2b(A0) |
      u2b(A1), u2b(A0) | u2b(A1) },
30 { u2b(B0) | u2b(B1), u2b(B0) | u2b(B1), u2b(B0) | u2b(B1) | u2b(B0) | u2b
      (C1), u2b(B0) | u2b(C1) } };
31
32 // Expected outputsS
33 signalvector outputs[RUNMAX][TICKMAX] =
34 { { 0, u2b(U0), u2b(U1), 0, u2b(U0) },
35 { u2b(V0) | u2b(V1), u2b(V0) | u2b(V1), u2b(V0), u2b(W1), u2b(V0) } };
36
37 void getInputs()
38 {
39 signals = inputs[runCnt][tickCnt];
40 }
41
42 // Set reference outputs and check valued signals, if there are any.
43 // Return 1 unless valued signal outputs are wrong.
44 // No valued signals here, therefore always return 1.
45 int checkOutputs(signalvector *tickOutputs)
46 {
47 *tickOutputs = outputs[runCnt][tickCnt];
48 return 1;
49 }
50
51 // No valued signals to print
52 void printVal(int id)
53 {
54 }
55
56 // Returns 1 if some thread is still active in current tick
57 int tick(int isInit)
58 {
59 // Thread ids: Main=1
60 TICKSTART(isInit);
61
62 GOTO(S0surf);
63
64 S0depth:PRESENT(A0, S0surf);
65 EMIT(U0);
66 GOTO(S1surf);
67 S0surf: PRESENT(B0, L0);
68 EMIT(V0);
69 GOTO(S1surf);
70 L0: PAUSE(S0depth);
71
72 S1surf: PRESENT(B1, L4);
73 GOTO(L2);
74 S1depth:PRESENT(A1, L1);
75 EMIT(U1);
76 GOTO(S2);
77 L1: PRESENT(B1, L3);
78 L2: EMIT(V1);
79 GOTO(S2);
80 L3: PRESENT(C1, L4);
81 EMIT(W1);
82 GOTO(S2);
83 L4: PAUSE(S1depth);
84
85 S2: PAUSE(S0surf);
86
87 TICKEND;
88 }
89
90 // Local Variables :
91 // compile-command: "make SurfDepth; SurfDepth"
92 // End:

```

Listing B.28: SurfDepth.out

```

1 ##### RUN 0 STARTS #####
2 ===== TICK 0 STARTS, inputs = 03, enabled = 00
3 ===== Inputs: A0 (0), A1 (1)
4 ===== Enabled: <none>
5 PRESENT: Main (id 1, prio 0) determines B0 (2) as absent
6 PAUSE: Main (id 1, prio 0) pauses, active = 03
7 ===== TICK 0 terminates after 3 instructions, enabled = 03.
8 ===== Resulting signals: A0 (0), A1 (1), Outputs OK.
9
10 ===== TICK 1 STARTS, inputs = 03, enabled = 03
11 ===== Inputs: A0 (0), A1 (1)
12 ===== Enabled: TickEnd (0), Main (1)
13 PRESENT: Main (id 1, prio 0) determines A0 (0) as present
14 EMIT: Main (id 1, prio 0) emits U0 (5)
15 PRESENT: Main (id 1, prio 0) determines B1 (3) as absent
16 PAUSE: Main (id 1, prio 0) pauses, active = 03
17 ===== TICK 1 terminates after 5 instructions, enabled = 03.
18 ===== Resulting signals: A0 (0), A1 (1), U0 (5), Outputs OK.
19

```

```

20  ===== TICK 2 STARTS, inputs = 03, enabled = 03
21  ===== Inputs: A0 (0), A1 (1)
22  ===== Enabled: TickEnd (0), Main (1)
23  PRESENT: Main (id 1, prio 0) determines A1 (1) as present
24  EMIT:   Main (id 1, prio 0) emits U1 (6)
25  PAUSE:  Main (id 1, prio 0) pauses, active = 03
26  ===== TICK 2 terminates after 4 instructions, enabled = 03.
27  ===== Resulting signals: A0 (0), A1 (1), U1 (6), Outputs OK.
28
29  ===== TICK 3 STARTS, inputs = 03, enabled = 03
30  ===== Inputs: A0 (0), A1 (1)
31  ===== Enabled: TickEnd (0), Main (1)
32  PRESENT: Main (id 1, prio 0) determines B0 (2) as absent
33  PAUSE:  Main (id 1, prio 0) pauses, active = 03
34  ===== TICK 3 terminates after 2 instructions, enabled = 03.
35  ===== Resulting signals: A0 (0), A1 (1), Outputs OK.
36
37  ===== TICK 4 STARTS, inputs = 03, enabled = 03
38  ===== Inputs: A0 (0), A1 (1)
39  ===== Enabled: TickEnd (0), Main (1)
40  PRESENT: Main (id 1, prio 0) determines A0 (0) as present
41  EMIT:   Main (id 1, prio 0) emits U0 (5)
42  PRESENT: Main (id 1, prio 0) determines B1 (3) as absent
43  PAUSE:  Main (id 1, prio 0) pauses, active = 03
44  ===== TICK 4 terminates after 5 instructions, enabled = 03.
45  ===== Resulting signals: A0 (0), A1 (1), U0 (5), Outputs OK.
46
47  ===== Executed tickMax = 5 ticks!
48  ##### RUN 0 terminates after 19 instructions
49
50  ##### RUN 1 STARTS #####
51  ===== TICK 0 STARTS, inputs = 014, enabled = 00
52  ===== Inputs: B0 (2), B1 (3)
53  ===== Enabled: <none>
54  PRESENT: Main (id 1, prio 0) determines B0 (2) as present
55  EMIT:   Main (id 1, prio 0) emits V0 (7)
56  PRESENT: Main (id 1, prio 0) determines B1 (3) as present
57  EMIT:   Main (id 1, prio 0) emits V1 (8)
58  PAUSE:  Main (id 1, prio 0) pauses, active = 03
59  ===== TICK 0 terminates after 9 instructions, enabled = 03.
60  ===== Resulting signals: B0 (2), B1 (3), V0 (7), V1 (8), Outputs OK.
61
62  ===== TICK 1 STARTS, inputs = 014, enabled = 03
63  ===== Inputs: B0 (2), B1 (3)
64  ===== Enabled: TickEnd (0), Main (1)
65  PRESENT: Main (id 1, prio 0) determines B0 (2) as present
66  EMIT:   Main (id 1, prio 0) emits V0 (7)
67  PRESENT: Main (id 1, prio 0) determines B1 (3) as present
68  EMIT:   Main (id 1, prio 0) emits V1 (8)
69  PAUSE:  Main (id 1, prio 0) pauses, active = 03
70  ===== TICK 1 terminates after 8 instructions, enabled = 03.
71  ===== Resulting signals: B0 (2), B1 (3), V0 (7), V1 (8), Outputs OK.
72
73  ===== TICK 2 STARTS, inputs = 024, enabled = 03
74  ===== Inputs: B0 (2), C1 (4)
75  ===== Enabled: TickEnd (0), Main (1)
76  PRESENT: Main (id 1, prio 0) determines B0 (2) as present
77  EMIT:   Main (id 1, prio 0) emits V0 (7)
78  PRESENT: Main (id 1, prio 0) determines B1 (3) as absent
79  PAUSE:  Main (id 1, prio 0) pauses, active = 03
80  ===== TICK 2 terminates after 5 instructions, enabled = 03.
81  ===== Resulting signals: B0 (2), C1 (4), V0 (7), Outputs OK.
82
83  ===== TICK 3 STARTS, inputs = 024, enabled = 03
84  ===== Inputs: B0 (2), C1 (4)
85  ===== Enabled: TickEnd (0), Main (1)
86  PRESENT: Main (id 1, prio 0) determines A1 (1) as absent
87  PRESENT: Main (id 1, prio 0) determines B1 (3) as absent
88  PRESENT: Main (id 1, prio 0) determines C1 (4) as present
89  EMIT:   Main (id 1, prio 0) emits W1 (9)
90  PAUSE:  Main (id 1, prio 0) pauses, active = 03
91  ===== TICK 3 terminates after 6 instructions, enabled = 03.
92  ===== Resulting signals: B0 (2), C1 (4), W1 (9), Outputs OK.
93
94  ===== TICK 4 STARTS, inputs = 024, enabled = 03
95  ===== Inputs: B0 (2), C1 (4)
96  ===== Enabled: TickEnd (0), Main (1)
97  PRESENT: Main (id 1, prio 0) determines B0 (2) as present
98  EMIT:   Main (id 1, prio 0) emits V0 (7)
99  PRESENT: Main (id 1, prio 0) determines B1 (3) as absent
100 PAUSE:  Main (id 1, prio 0) pauses, active = 03
101 ===== TICK 4 terminates after 5 instructions, enabled = 03.
102 ===== Resulting signals: B0 (2), C1 (4), V0 (7), Outputs OK.
103
104 ===== Executed tickMax = 5 ticks!
105 ##### RUN 1 terminates after 33 instructions
106
107 ##### All runs terminate, after 52 instructions

```