INSTITUT FÜR INFORMATIK

SyncCharts in C

Reinhard von Hanxleden

Bericht Nr. 0910

May 2009, Revised[a] September 2009 **(Draft)**

Last compiled: 2009-10-07, 2:31 hrs (CET)

[a]The original, unrevised report is available at `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/report-0910-unrevised.pdf`

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
May 2009, Revised[a] September 2009 **(Draft)**
Last compiled: 2009-10-07, 2:31 hrs (CET)

e-mail:
rvh@informatik.uni-kiel.de

[a]The original, unrevised report is available at `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/report-0910-unrevised.pdf`

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

Note: An abridged version of the original report appeared elsewhere [32].

Contents

List of Figures

List of Tables

Listings, Outside Figures

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 [13], 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 [29, 34]. 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, preprocessors, 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 [3] 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 [33]. 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 [5], or coroutine [9, 15] 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 Andé [3]), 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[1].

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 [19], 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.

[1]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

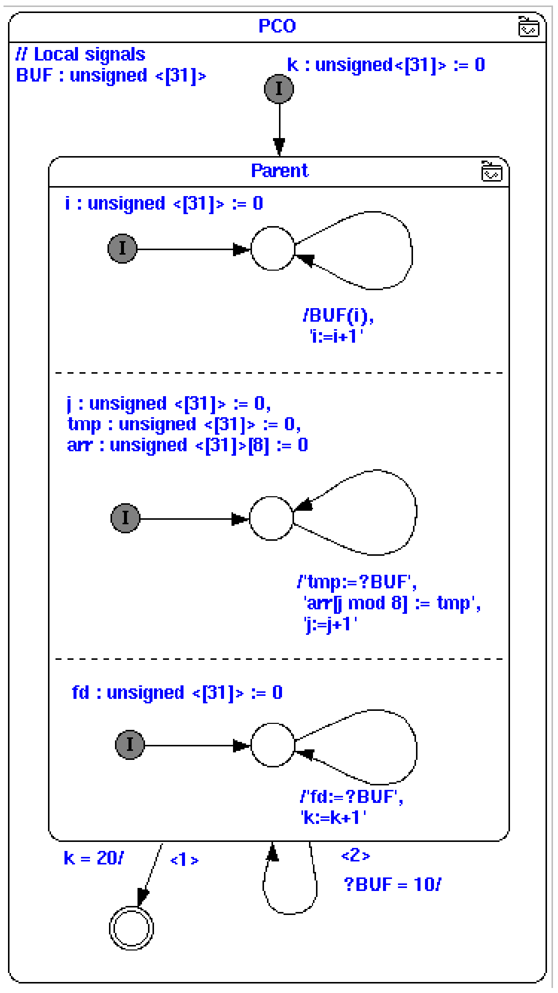
This section covers

- 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.

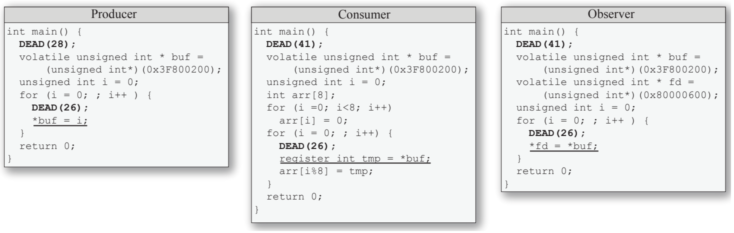
We will illustrate these points with PCO, shown in Fig. 2.1, a simple producer-consumer example with an observer, inspired by Lickly *et al.* [19]. In addition to the original example, PCO also has a parent thread that restarts production/consumption once the buffer has the value 10, and which terminates 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*)(0x80000600);
  unsigned int i = 0;
  for (i = 0; ; i++ ) {
    DEAD(26);
    *fd = *buf;
  }
  return 0;
}
```

(b) PRET version, without preemption (from [19])

```
#include "sc.h"

int BUF, fd, i, j, k = 0, tmp, arr [8];

// ====== MAIN FUNCTION ======
int main()
{
  int notDone, init = 1;

  do {
    notDone = tick( init );    // Call tick function
    //sleep(1);                  // Slow down by 1 sec
     init = 0;
  } while (notDone);
  return 0;
}

// ====== TICK FUNCTION ======
int tick ( int isInit )
{
  TICKSTART(isInit, 1);

 PCO:
  FORK(Producer, 4);
  FORK(Consumer, 2);
  FORK(Observer, 3);
  FORKE(Parent);

 Producer:
  for (i = 0; ; i++) {
    PAUSE;
    BUF = i; }

 Consumer:
  for (j = 0; j < 8; j++)
    arr [ j ] = 0;
  for (j = 0; ; j++) {
    PAUSE;
    tmp = BUF;
    arr [ j % 8] = tmp; }

 Observer:
  for ( ; ; ) {
    PAUSE;
    fd = BUF;
    k++; }

 Parent:
  while (1) {
    PAUSE;
    if (k == 20) TRANS(Done);
    if (BUF == 10) TRANS(PCO);
  }

 Done:
  TERM;
  TICKEND;
}
```

(c) Complete SC program

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

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 [19] 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 FORK/FORKE 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 FORK gives a thread its initial priority (here all 0), a starting label, and an id. FORKE 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

Mnemonic, Operands	**Notes**
TICKSTART*(*isInitial*, p)	Start (initial) tick, assign main thread priority p.
TICKEND	Finalize tick, return 1 iff there is still an enabled thread.
PAUSE^{*+}	Deactivate current thread for this tick.
PAUSEG*(l)	Shorthand for PAUSE; GOTO(l).
HALT^{*+}	Shorthand for l: PAUSE; GOTO(l).
TERM*	Terminate current thread.
ABORT(l)	Abort descendant threads.
TRANS(l)	Shorthand for ABORT; GOTO(l).
SUSPEND*(*cond*)	Suspend (pause) thread and its descendants if *cond* holds.
SUSPENDGOTO*(l)	Suspend (pause) thread and its descendants, continue at l holds.
FORK(l, p)	FORK creates a thread with start address l and priority p.
FORKE*(l)	FORKE specifies the continuation address l for the spawning thread.
JOINE^{*+}(l_{else})	If descendant threads have terminated normally, proceed; else pause, jump to l_{else}.
JOIN^{*+}	Shorthand for l_{else}: JOINE(l_{else}). Waits for descendant threads to terminated normally.
PRIO^{*+}(p)	Set current thread priority to p.
PPAUSE^{*+}(p)	Shorthand for PRIO(p); PAUSE (saves one call to dispatcher).
JPPAUSE^{*+}(p, l_{then}, l_{else})	Shorthand for JOINE(l); GOTO(l_{then}); l: PRIO(p); PAUSE; GOTO(l_{else}); l_{then}: (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. Operators marked with a plus$^+$ automatically generate continuation labels (visible in the program after macro expansion and in execution traces).

the *dispatcher*.

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), FORKE (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

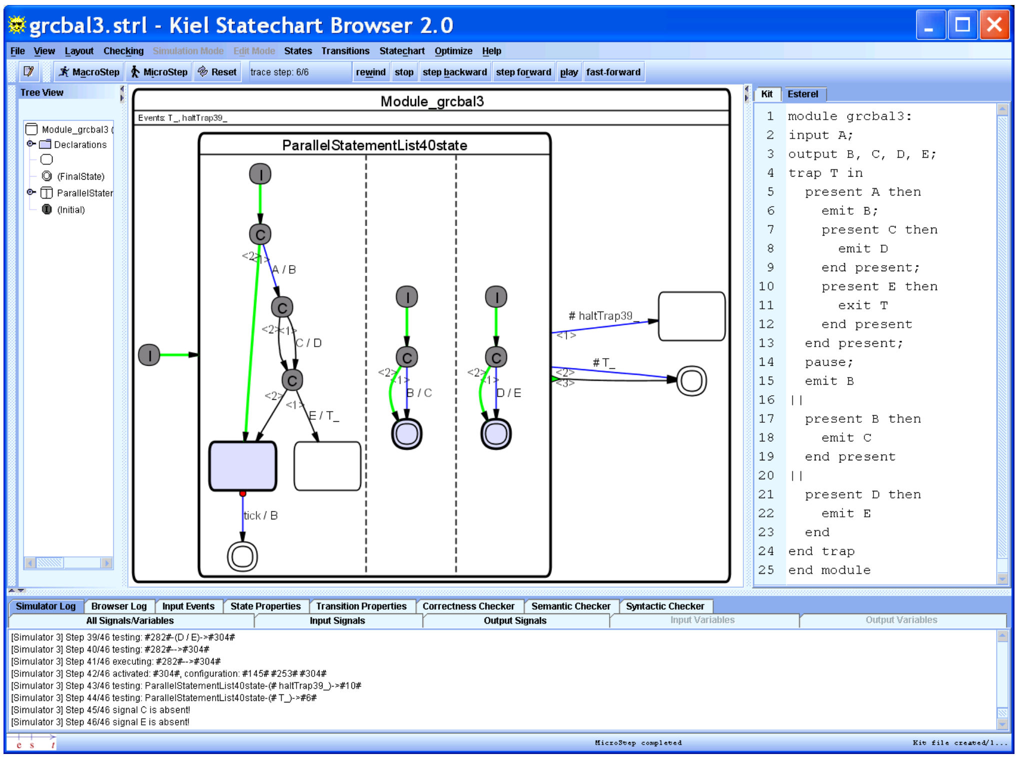
This section covers

- 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 [10]. Hence the name of the benchmark: GRC is the Graph Code intermediate representation of the CEC, BAL is the Bytcode Assembly Language of a virtual machine (VM) targeted by the CEC. The grcbal3 Esterel code has been transformed into a SyncChart using KIEL [24]. 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 $\mathsf{T}\_$. 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 superflous state, reachable via haltTrap39—which


grcbal3.strl - Kiel Statechart Browser 2.0
File View Layout Checking Simulation Mode Edit Mode States Transitions Statechart Optimize Help
MacroStep MicroStep Reset trace step: 6/6 rewind stop step backward step forward play fast-forward
Tree View
Module_grcbal3
Declarations
(FinalState)
ParallelStater
(Initial)
Module_grcbal3
Events: T_, haltTrap39_
ParallelStatementList40state
A / B
C / D
E / T_
tick / B
B / C
D / E
# haltTrap39_
# T_
Kit Esterel
Simulator Log Browser Log Input Events State Properties Transition Properties Correctness Checker Semantic Checker Syntactic Checker
All Signals/Variables Input Signals Output Signals Input Variables Output Variables
MicroStep completed
Kit file created/1...

```
module grcbal3:
input A;
output B, C, D, E;
trap T in
  present A then
    emit B;
    present C then
      emit D
    end present;
    present E then
      exit T
    end present
  end present;
  pause;
  emit B
||
  present B then
    emit C
  end present
||
  present D then
    emit E
  end
end trap
end module
```

```
[Simulator 3] Step 39/46 testing: #282#-(D / E)->#304#
[Simulator 3] Step 40/46 testing: #282#-->#304#
[Simulator 3] Step 41/46 executing: #282#-->#304#
[Simulator 3] Step 42/46 activated: #304#, configuration: #145# #253# #304#
[Simulator 3] Step 43/46 testing: ParallelStatementList40state-(# haltTrap39_)->#10#
[Simulator 3] Step 44/46 testing: ParallelStatementList40state-(# T_)->#6#
[Simulator 3] Step 45/46 signal C is absent!
[Simulator 3] Step 46/46 signal E is absent!
```

(a) Screen shot of KIEL [25], as it synthesizes a SyncChart from the original Esterel code [10]

```
 TICKSTART(isInit, 1);
 FORK(T1, 6);
 FORK(T2, 5);
 FORK(T3, 3);
 FORKE(TMain);

T1:
 PRESENTELSE(A, T1B);
 EMIT(B);
 PRIO(4);
 PRESENTEMIT(C, D);
 PRIO(2);
 PRESENTELSE(E, T1B);
 EMIT(T_);
 TERM;
T1B: PAUSE;
 EMIT(B);
 TERM;

T2:
 PRESENTEMIT(B, C);
 TERM;

T3:
 PRESENTEMIT(D, E);
 TERM;

TMain:
 PRESENTELSE(T_, TJoin);
 ABORT;
 TERM;
TJoin: JOINE(TMain);
 TICKEND;
```

(b) SC tick function (SC operators only)

```
 TICKSTART(isInit, 1);
 FORK(T1, 6);
 FORK(T2, 5);
 FORK(T3, 3);
 FORKE(TMain);

T1: if (PRESENT(A)) {
   EMIT(B);
   PRIO(4);
   if (PRESENT(C))
     EMIT(D);
   PRIO(2);
   if (PRESENT(E)) {
     EMIT(T_);
     TERM; }
 }
 PAUSE;
 EMIT(B);
 TERM;

T2: if (PRESENT(B))
   EMIT(C);
 TERM;

T3: if (PRESENT(D))
   EMIT(E);
 TERM;

TMain: if (PRESENT(T_)) {
   ABORT;
   TERM; }
 JOINE(TMain);
 TICKEND;
```

(c) SC tick function (mixed with standard C)

```
==== TICK 0 STARTS, inputs = 01, enabled = 00
==== Inputs (id/name): 0/A
==== Enabled (id/state): <init>
FORK:    1/<init> forks 6/T1, active = 0103
FORK:    1/<init> forks 5/T2, active = 0143
FORK:    1/<init> forks 3/T3, active = 0153
FORKE:   1/<init>
PRESENT: 6/T1 determines A/0 as present
EMIT:    6/T1 emits B/1
PRIO:    6/T1 set to priority 4
PRESENT: 5/T2 determines B/1 as present
EMIT:    5/T2 emits C/2
TERM:    5/T2 terminates, enabled = 073
PRESENT: 4/_L73 determines C/2 as present
EMIT:    4/_L73 emits D/3
PRIO:    4/_L73 set to priority 2
PRESENT: 3/T3 determines D/3 as present
EMIT:    3/T3 emits E/4
TERM:    3/T3 terminates, enabled = 017
PRESENT: 2/_L76 determines E/4 as present
EMIT:    2/_L76 emits T_/5
TERM:    2/_L76 terminates, enabled = 07
PRESENT: 1/TMain determines T_/5 as present
ABORT:   1/TMain disables 070, enabled = 03
TERM:    1/TMain terminates, enabled = 03
==== TICK 0 terminates after 22 instructions.
==== Enabled (id/state): 0/_L_TICKEND
==== Resulting signals (name/id): 0/A, 1/B,
       2/C, 3/D, 4/E, 5/T_, Outputs OK.
```

(d) Example trace

Figure 2.2: The grcbal3 example.

Mnemonic, Operands	**Notes**
SIGNAL(S)	Initialize a local signal S.
EMIT(S)	Emit signal S.
SUSTAIN^{*+}(S)	Sustains signal S. Shorthand for l: EMIT(S); PAUSE; GOTO(l).
PRESENT(S)	Evaluates to 1 if S is present, else evaluates to 0.
PRESENTELSE(S, l_{else})	If S is present, proceed normally; else, jump to l_{else}. If PRESENT is available: shorthand for if (!PRESENT(s)) GOTO(l_{else}).
PRESENTEMIT(S, T)	If S is present, emit T. Shorthand for if (PRESENT(S)) EMIT(T).
AWAIT^{*+}(S)	Wait (non-immediately) for signal S. Shorthand for l_{else}: PAUSE; PRESENT(s, l_{else}).
AWAITI^{*+}(S)	Wait (immediately) for signal S. Shorthand for GOTO(l); l_{else}: PAUSE; l: PRESENT(s, 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)	Retrieve value of signal S.
VALREG(S, reg)	Retrieve value of signal S, into register/variable reg.
PRESENTPRE(S, l_{else})	Evaluates to 1 if S was present in previous tick, else evaluates to 0. If S is a signal local to thread t, consider last preceeding tick in which t was active, *i. e.*, not suspended.
PRESENTPREELSE(S, l_{else})	If S was present in previous tick, proceed normally; else, jump to l_{else}. If PRESENTPRE is available: shorthand for if (!PRESENTPRE(s)) GOTO(l_{else}).
VALPRE(S)	Retrieve value of signal S at previous tick.
VALPREREG(S, reg)	Retrieve value of signal S at previous tick, into register/variable reg.
GOTO(l)	Jump to label l.
CALL(l)	Call function l (eg, an on exit function).
RET	Return from function call.
ISAT(id, l_{state}, l)	If thread id is at state l_{state}, then proceed to next instruction (*e. g.*, an on exit function of associated with id at state l_{state}). Else, jump to label l.
ISATCALL(id, l_{state}, l_{action})	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). See Table 2.1 on the asterisk* and plus$^+$annotations.

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 [24], and a lack of a subsequent opimization in KIEL that would remove such clearly unreachable states.

Fig. 2.2c 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. 2.2d. 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, FORK, and FORKE 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, FORKE 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, 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 original code (grcbal3.c, in the SC distribution) 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 PRIO);

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 Andé [3], 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

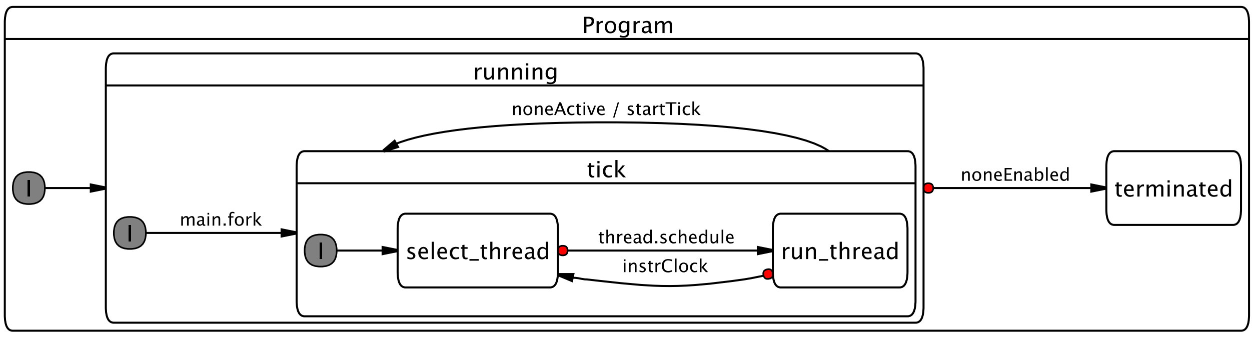


Figure 3.1: The status of the whole program (from [18]).

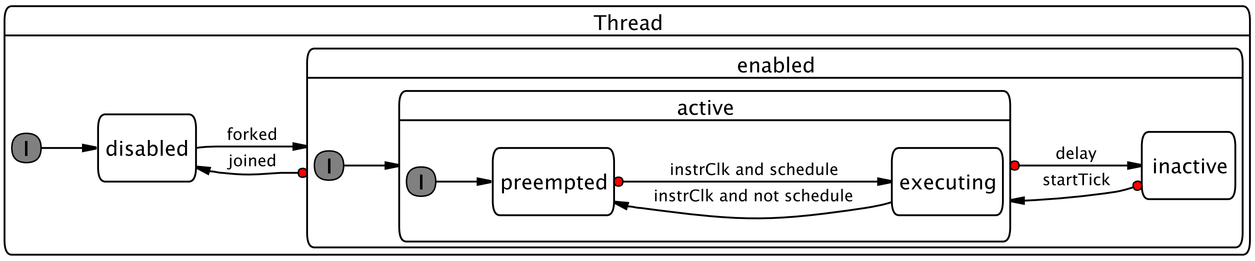


Figure 3.2: Execution status of a single thread (from [18]).

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 [18]. 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 FORK, 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 FORK). 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 [8], 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.

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: selectCid(): Computation of id of thread to be dispatched (from sc.c)

```
void selectCid () {
  int act;

  _cid = 0;
  for (act = active; act > 1; act >>= 1)
    _cid++;
}
```

The computation of cid is implemented in selectCid, 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.

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[1]. Which algorithm is actually faster depends on the application.

Listing 3.2: dispatch(): Variable definitions for the dispatcher (from sc.h)

```
#if ((defined __i386__ || defined __amd64__ || defined __x86_64__) && defined __XXGNUC__)
// Version 1: x86 + gcc available.
// Use fast Bit Scan Reverse assembler instruction.
#define dispatch()                             \
  __asm volatile ("bsrl %1,%0\n"               \
                  : "=r" (_cid)                \
                  : "c" (active)               \
                  );                           \
  goto *_pc[_cid]

#else
// Version 2: x86 + gcc not available.
// Call function, defined in sc.c.
#define dispatch()                             \
  selectCid();                                 \
  goto *_pc[_cid]
#endif
```

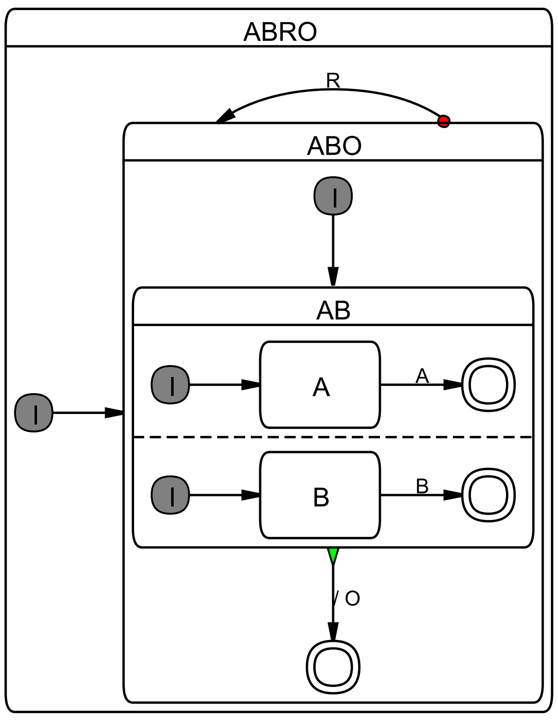
Fortunately, many processor instruction sets provide an assembler instruction that does exactly this, to detect the index of the highest set bit. The x86 does this with the Bit Scan Reverse (BSR) instruction. Using the gcc assembler escape, we can embed this instruction into C and thus obtain an even 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 3.2 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 [3, 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

[1]See eg http://graphics.stanford.edu/~seander/bithacks.html#IntegerLog



(a) SyncChart

```
TICKSTART(isInit, 4);

do {
  FORK(AB, 1);
  FORKE(ABOMain);

AB:
  FORK(WaitA, 2);
  FORK(WaitB, 3);
  FORKE(ABMain);

WaitA:
  AWAIT(A);
  TERM;

WaitB:
  AWAIT(B);
  TERM;

ABMain:
  JOIN;
  EMIT(O);
  TERM;

ABOMain:
  AWAIT(R);
  ABORT;
} while (1);

TICKEND;
```

(b) SC tick function

```
 TICKSTART(isInit, 4);

ABO:
 FORK(AB, 1);
 FORKE(ABOMain);

AB:
 FORK(WaitA, 2);
 FORK(WaitB, 3);
 FORKE(ABMain);

WaitA:
 AWAIT(A);
 TERM;

WaitB:
 AWAIT(B);
 TERM;

ABMain:
 JOIN;
 EMIT(O);
 TERM;

ABOMain:
 AWAIT(R);
 ABORT;
 TRANS(ABO);

 TICKEND;
```

(c) SC (only) tick function

Figure 3.3: The ABRO example.

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.

- 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.

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

One also distinguishes the

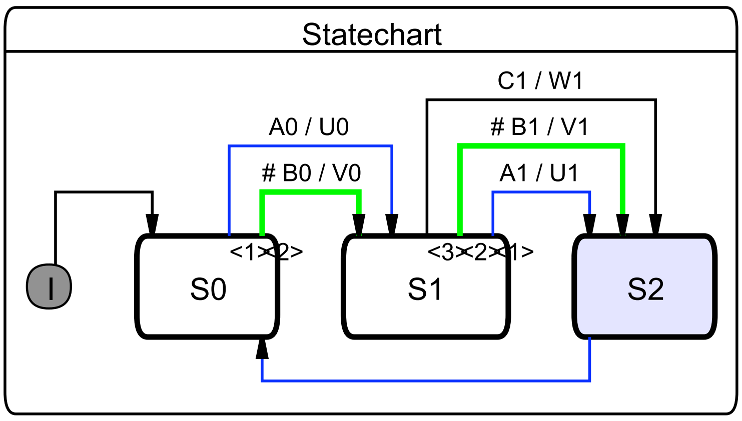
Surface of a statement, which is what is executed in the initial tick and which includes only the immediate transitions, and the

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



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

```
TICKSTART(isInit, 1);

while (1) {

  // State S0 — surface
  while (1) {
    if (PRESENT(B0)) {
      EMIT(V0);
      break; }
    PAUSE;

    // State S0 — depth
    if (PRESENT(A0)) {
      EMIT(U0);
      break; }
  }

  // State S1 — surface
  if (PRESENT(B1)) {
    EMIT(V1);
  } else {
    while (1) {
      PAUSE;

      // State S1 — depth
      if (PRESENT(A1)) {
        EMIT(U1);
        break; }
      if (PRESENT(B1)) {
        EMIT(V1);
        break; }
      if (PRESENT(C1)) {
        EMIT(W1);
        break; }
    }
  }

  // State S2
  PAUSE;
}

TICKEND;
```

(b) SC tick function

Figure 3.4: The SurfDepth example.

to minimize code size and to follow the write-things-once principle as much as possible, by sharing one $\mathsf{EMIT(V1)}_{20}$ statement between the transition tests at lines 14 and 19, with the $\mathsf{GOTO(L2)}_{15}$ 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 [18]) 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 $Op_1 = \mathsf{Main}_{24,\mathsf{PRESENT(R)}}$, and let $Op_2 = \mathsf{WaitA}_{12,\mathsf{PRESENT(A)}}$. 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 $\mathsf{Main}_{24,\mathsf{PRESENT(R)}} \succ \mathsf{WaitA}_{12,\mathsf{PRESENT(A)}}$.

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, $\mathsf{Main}_{24,\mathsf{PRESENT(R)}} \succ \mathsf{WaitA}_{12,\mathsf{PRESENT(A)}}$.

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, $\mathsf{WaitA}_{12,\mathsf{PRESENT(A)}} \succ \mathsf{AB}_{19,\mathsf{JOIN}}$.

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.*, $\mathsf{M10}_{27,\mathsf{PRESENT(A)}} \succ \mathsf{M10}_{20,\mathsf{JPPAUSE}}$; 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. 2.2c), $\mathsf{A1}_{8,\mathsf{EMIT(B)}} \succ \mathsf{A2}_{19,\mathsf{PRESENT(B)}}$.
Example: in PCO (Fig. 2.1c), $\mathsf{Prod}_{33,\mathsf{BUF=1}} \succ \mathsf{Cons}_{39,\mathsf{tmp=BUF}}$.

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 Main $\succ$ WaitA. In other words, Main should be scheduled before WaitA.

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

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

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

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

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

   This constraint is met by executing the first operation at priority 3, as induced by the $\mathsf{Main}_{2,\mathsf{FORK(3,A1,ids[A1])}}$ statement, and the second at priority 2.

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

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

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

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

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

   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*):

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

   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. 6.

Tick start and end

TICKSTART and TICKEND do some book keeping. For example, in the initial tick, TICK-START 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. 37.

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 FORK statements, followed by a FORKE statement, together form a *fork*. Each FORK creates a child thread by initializing its program counter and its priority and enabling it. FORKE 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(l_{else}).

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 FORKE. However, we may also construct a fork without FORKE, which just consists of a sequence of FORK statements that effectively create *sibling threads* of the current thread. Here the current thread simply keeps executing after the FORK statements. Since there is no FORKE, 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 FORK. 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 [18]. 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 [3]. 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 be 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 original code (PreAndSuspend.c, in the SC distribution) for how this is done.

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. 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}. 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 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 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 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 ISATCALL(id, l_{state}, l_{action}, l) as a shorthand for ISAT(id, l_{state}, l); CALL(l_{action}, l). For example, in the Exits code (Fig. 4.2b), the 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.

```
  WaitA:
    do { goto _LL66; _L66: if (!( signals & (1 << A))) { _LL66: _pc[_cid] = &&_L66; goto _L_PAUSEG; } } while (0);
    do { goto _L_TERM; } while (0);

  WaitB:
    do { goto _LL70; _L70: if (!( signals & (1 << B))) { _LL70: _pc[_cid] = &&_L70; goto _L_PAUSEG; } } while (0);
    do { goto _L_TERM; } while (0);

  ABMain:
    do { _L74: if (((enabled & _descs[_cid]) == 0)) { goto _LL74; } _pc[_cid] = &&_L74; goto _L_PAUSEG; _LL74: (void)
          0; } while (0);
    do { signals |= (1 << O); } while (0);
    do { goto _L_TERM; } while (0);

  ABOMain:
    do { goto _LL79; _L79: if (!( signals & (1 << R))) { _LL79: _pc[_cid] = &&_L79; goto _L_PAUSEG; } } while (0);
    do { enabled &= ~_descs[_cid]; active &= ~_descs[_cid]; } while (0);
  } while (1);

  _L_TICKEND: return (enabled != (1 << 0)); _L_TERM: enabled &= ~(1 << _cid); _L_PAUSEG: active &= ~(1 << _cid);
        _L_dispatch: selectCid(); goto *_pc[_cid];
}
```

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. 3.3.4 after macro expansion. For better readability, Fig. 3.3.4 shows the same function with added comments and reformatting, and Fig. 3.3.4 shows a version with stripped empty statements (do ... while (0), (void) 0). All versions produce identical code (gcc does not require an optimization level to strip these empty statements).

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

```
int tick(int isInit )
{
  // Thread ids: AB=1, WaitB=2, WaitA=3, Main=4

  // TICKSTART(isInit);
  if ( isInit ) {
    tickCnt = 0;
    pc[TickEnd] = &&_L_TICKEND;
    pr[TickEnd] = 0;
    enabled = (1 << ids[TickEnd]);
    active = enabled;
    cid = ids[Main]; ;
    enabled |= (1 << cid);
    active |= (1 << cid);
  } else {
    active = enabled;
    goto _L_dispatch;
  };

 ABO: // PAR(0, AB, ids[AB]);
  do {
    pc[ids[AB]] = &&AB;
    pr[ids[AB]] = 0;
    enabled |= (1 << ids[AB]);
    active |= (1 << ids[AB]);
  } while (0);

  // PARE(0, ABOmain, id2b(AB) | id2b(WaitA) | id2b(WaitB));
  do {
    pc[cid] = &&ABOmain;
    pr[cid] = 0;
    descs[cid] = (1 << ids[AB]) | (1 << ids[WaitA]) | (1 << ids[WaitB]);
    goto _L_dispatch;
  } while (0);

 AB: // PAR(0, WaitA, ids[WaitA]);
  do {
    pc[ids[WaitA]] = &&WaitA;
    pr[ids[WaitA]] = 0;
    enabled |= (1 << ids[WaitA]);
    active |= (1 << ids[WaitA]);
  } while (0);

  // PAR(0, WaitB, ids[WaitB]);
  do {
    pc[ids[WaitB]] = &&WaitB;
    pr[ids[WaitB]] = 0;
    enabled |= (1 << ids[WaitB]);
    active |= (1 << ids[WaitB]);
  } while (0);

  // PARE(0, ABmain, id2b(WaitA) | id2b(WaitB));
  do {
    pc[cid] = &&ABmain;
    pr[cid] = 0;
    descs[cid] = (1 << ids[WaitA]) | (1 << ids[WaitB]);
    goto _L_dispatch;
  } while (0);

 WaitA: // AWAIT(A);
  do {
    goto _LL74;
  _L74: if (!( signals & (1 << A))) {
    _LL74: pc[cid] = &&_L74;
      goto _L_PAUSEG; }
  } while (0);

  // TERM;
  do { goto _L_TERM; } while (0);

 WaitB: // AWAIT(B);
  do {
    goto _LL77;
  _L77: if (!( signals & (1 << B))) {
    _LL77: pc[cid] = &&_L77;
      goto _L_PAUSEG; }
  } while (0);

  // TERM;
  do { goto _L_TERM; } while (0);

 ABmain: // JOIN;
  do {
  _L80: if (((enabled & descs[cid]) == 0)) {
      goto _LL80; }
    pc[cid] = &&_L80;
    goto _L_PAUSEG;
  _LL80: (void) 0;
  } while (0);

  // EMIT(O);
  do {
     signals |= (1 << O);
  } while (0);

  // TERM;
  do {
    goto _L_TERM;
  } while (0);

 ABOmain: // AWAIT(R);
  do {
    goto _LL84;
  _L84: if (!( signals & (1 << R))) {
    _LL84: pc[cid] = &&_L84;
      goto _L_PAUSEG; }
  } while (0);

  // TRANS(ABO);
  do {
    enabled &= ~descs[cid];
    active &= ~descs[cid];
    goto ABO;
  } while (0);

  // TICKEND;
 _L_TICKEND: return (enabled != (1 << ids[TickEnd]));

  // Dispatcher, part of TICKEND macro
 _L_TERM:    enabled &= ~(1 << cid);
 _L_PAUSEG:  active &= ~(1 << cid);
 _L_dispatch: __asm volatile ("bsrl %1,%0\n" : "=r" (cid) : "c" (active) );
  goto *pc[cid];
}
```

Figure 3.6: ABRO tick function after macro expansion, annotated and reformatted.

```
int tick(int isInit )
{
  // Thread ids: AB=1, WaitB=2, WaitA=3, Main=4

  // TICKSTART(isInit);
  if ( isInit ) {
    tickCnt = 0;
    pc[TickEnd] = &&_L_TICKEND;
    pr[TickEnd] = 0;
    enabled = (1 << ids[TickEnd]);
    active = enabled;
    cid = ids[Main];
    enabled |= (1 << cid);
    active |= (1 << cid);
  } else {
    active = enabled;
    goto _L_dispatch;
  };

 ABO: // PAR(0, AB, ids[AB]);
  pc[ids[AB]] = &&AB;
  pr[ids[AB]] = 0;
  enabled |= (1 << ids[AB]);
  active |= (1 << ids[AB]);

  // PARE(0, ABOmain, id2b(AB) | id2b(WaitA) | id2b(WaitB));
  pc[cid] = &&ABOmain;
  pr[cid] = 0;
  descs[cid] = (1 << ids[AB]) | (1 << ids[WaitA]) | (1 << ids[
       WaitB]);
  goto _L_dispatch;

 AB: // PAR(0, WaitA, ids[WaitA]);
  pc[ids[WaitA]] = &&WaitA;
  pr[ids[WaitA]] = 0;
  enabled |= (1 << ids[WaitA]);
  active |= (1 << ids[WaitA]);

  // PAR(0, WaitB, ids[WaitB]);
  pc[ids[WaitB]] = &&WaitB;
  pr[ids[WaitB]] = 0;
  enabled |= (1 << ids[WaitB]);
  active |= (1 << ids[WaitB]);

  // PARE(0, ABmain, id2b(WaitA) | id2b(WaitB));
  pc[cid] = &&ABmain;
  pr[cid] = 0;
  descs[cid] = (1 << ids[WaitA]) | (1 << ids[WaitB]);
  goto _L_dispatch;
 WaitA: // AWAIT(A);
    goto _LL74;
  _L74: if (!( signals & (1 << A))) {
    _LL74: pc[cid] = &&_L74;
      goto _L_PAUSEG;
    }

  // TERM;
  goto _L_TERM;

 WaitB: // AWAIT(B);
  goto _LL77;
 _L77: if (!( signals & (1 << B))) {
  _LL77: pc[cid] = &&_L77;
    goto _L_PAUSEG;
  }

  // TERM;
  goto _L_TERM;
 ABmain: // JOIN;
 _L80: if (((enabled & descs[cid]) == 0)) {
    goto _LL80;
  }
  pc[cid] = &&_L80;
  goto _L_PAUSEG;
 _LL80:

  // EMIT(O);
  signals |= (1 << O);

  // TERM;
  goto _L_TERM;

 ABOmain: // AWAIT(R);
  goto _LL84;
 _L84: if (!( signals & (1 << R))) {
  _LL84: pc[cid] = &&_L84;
    goto _L_PAUSEG;
  }

  // TRANS(ABO);
  enabled &= ~descs[cid];
  active &= ~descs[cid];
  goto ABO;

  // TICKEND;
 _L_TICKEND: return (enabled != (1 << ids[TickEnd]));

  // Dispatcher, part of TICKEND macro
 _L_TERM:    enabled &= ~(1 << cid);
 _L_PAUSEG:  active &= ~(1 << cid);
 _L_dispatch: __asm volatile ("bsrl %1,%0\n" : "=r" (cid) : "
      c" (active) );
  goto *pc[cid];
}
```

Figure 3.7: ABRO tick function after macro expansion, annotated and stripped of empty statements.

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, a dispatcher function (selectCid, 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

Further Examples

This chapter contains a selection of further examples provided by Andé [3].

4.1 Count2Suspend

Count2Suspend [3, 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 suspends (“freezes”) operation of Cnt2. The $\mathsf{Main}_{87,\mathsf{SUSPEND}}$ statement performs the according control of the execution of Cnt2, see also p. 24.

Precedence constraints

1. Strong abort and suspension on Cnt2 (*outer-inner*):

   Main $\succ$ Off0, Main $\succ$ Off1

2. Communication via a signal C0 (*writer-reader*):

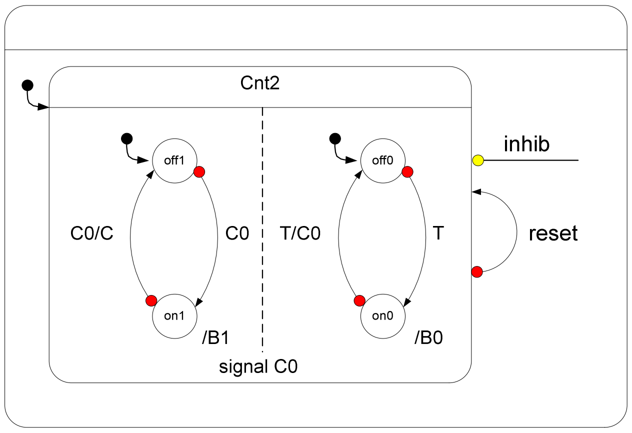
   Off0 $\succ$ Off1

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

4.2 Exits

Exits [3, 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%.



(a) SyncChart

```
 TICKSTART(isInit, 3);

Cnt2:
 FORK(Off1, 1);
 FORK(Off0, 2);
 FORKE(Cnt2Main);

Off1:
 while (1) {
   AWAIT(C0);
   do {
     EMIT(B1);
     PAUSE;
   } while (!PRESENT(C0));
   EMIT(C);
 }

Off0:
 while (1) {
   AWAIT(T);
   do {
     EMIT(B0);
     PAUSE;
   } while (!PRESENT(T));
   EMIT(C0);
 }

Cnt2Main:
 while (1) {
   PAUSE;
    if (PRESENT(reset))
     TRANS(Cnt2);
   SUSPEND(PRESENT(inhib));
 }

 TICKEND;
```

(b) SC tick function

Figure 4.1: The Count2Suspend example.

Precedence constraints

1. Strong abort on M0 (*outer-inner*):

   Main $\succ$ M10, Main $\succ$ M2, Main $\succ$ M11

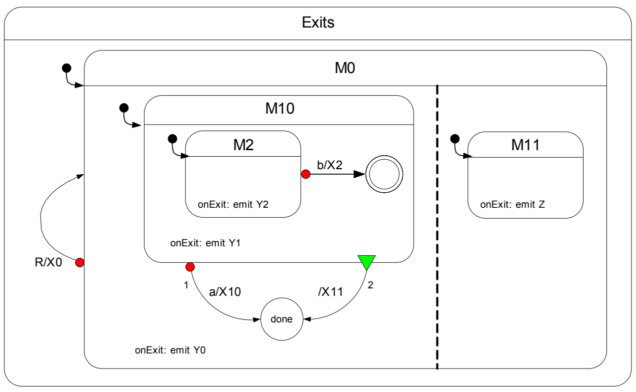
   We meet this by assigning Main the highest thread id (4). Furthermore, as thread M10, a child of Main, will eventually raise its priority to 1, Main is also assigned this priority, in $\text{Main}_{6,\text{FORKE}}$, after forking M10.

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

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

   This is met by assigning M10 priority 1, with $\text{M10}_{20,\text{JPPAUSE(1,...)}}$, 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*):



(a) SyncChart

```
 TICKSTART(isInit, 4);

M0:
 FORK(M10, 2);
 FORK(M11, 1);
 PRIO(6);
 FORKE(M0main);

M10:
 FORK(M2, 3);
 FORKE(M10main);

M2:
 PAUSE;
M2depth:
 PRESENTELSE(B, M2);
 CALL(M2exit);
 EMIT(X2);
 TERM;
M2exit:
 EMIT(Y2);
 RET;

L2:
 PRIO(2);
M10main:
 JPPAUSE(5, M10depth);
 ISATCALL(3, M2depth, M2exit);
 CALL(M10exit);
 EMIT(X11);
 TRANS(Done);
M10depth:
 PRESENTELSE(A, L2);
 ISATCALL(3, M2depth, M2exit);
 CALL(M10exit);
 EMIT(X10);
 TRANS(Done);
Done:
 HALT;
M10exit:
 EMIT(Y1);
 RET;

M11:
 HALT;
M11exit:
 EMIT(Z);
 RET;

M0main:
 AWAIT(R);
 ISATCALL(3, M2depth, M2exit);
 ISATCALL(5, M10depth, M10exit);
 CALL(M11exit);
 EMIT(Y0); // Only place to call exit action of M0
 EMIT(X0);
 TRANS(M0);

 TICKEND;
```

(b) SC tick function

Figure 4.2: The Exits example. See Fig. 4.3 for alternative tick functions.

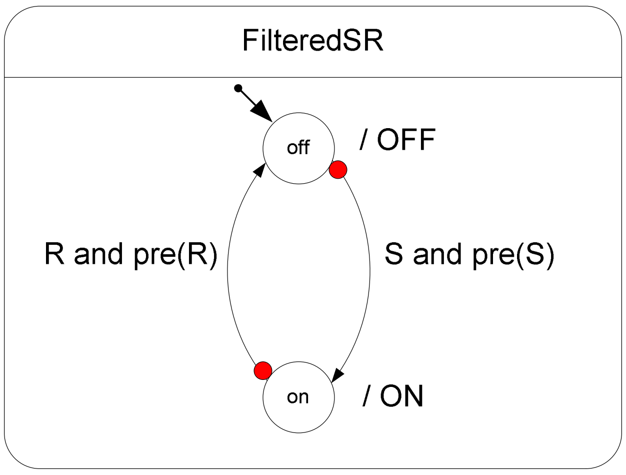
```
// Thread ids: M11=1, M10=2, M2=3, Main=4
          TICKSTART(isInit);
M0:       PAR(0, M10, ids[M10]);
          PAR(0, M11, ids[M11]);
          PARE(1, M0main, id2b(M10) | id2b(M11) | id2b(
                M2));

M10:      PAR(0, M2, ids[M2]);
          PARE(0, M10main, id2b(M2));

M2:       PAUSE;
M2depth:PRESENT(B, M2);
          CALL(M2exit, L1);
M2exit: EMIT(Y2);
          RET;
L1:       EMIT(X2);
          TERM;

L2:       PRIO(0);
M10main:JPPAUSE(1, M10depth);
          ISAT(ids[M2], M2depth, L4);
          CALL(M2exit, L4);
L4:       CALL(M10exit, L5);
M10exit:EMIT(Y1);
          RET;
L5:       EMIT(X11);
          TRANS(Done);
M10depth:PRESENT(A, L2);
          ISAT(ids[M2], M2depth, L7);
          CALL(M2exit, L7);
L7:       CALL(M10exit, L8);
L8:       EMIT(X10);
          TRANS(Done);
Done:     HALT;

M11:      HALT;
M11exit:EMIT(Z);
          RET;

M0main: AWAIT(R);
          ISAT(ids[M2], M2depth, L10);
          CALL(M2exit, L10);
L10:      ISAT(ids[M10], M10depth, L11);
          CALL(M10exit, L11);
L11:      CALL(M11exit, L12);
L12:      EMIT(Y0);
          EMIT(X0);
          TRANS(M0);

          TICKEND;
```

(a) Tick function without ISATCALL

```
// Thread ids: M11=1, M10=2, M2=3, Main=4
          TICKSTART(isInit);
M0:       PAR(0, M10, ids[M10]);
          PAR(0, M11, ids[M11]);
          PARE(1, M0main, id2b(M10) | id2b(M11) | id2b(
                M2));

M10:      PAR(0, M2, ids[M2]);
          PARE(0, M10main, id2b(M2));

M2:       PAUSE;
M2depth:PRESENT(B, M2);
          EMIT(Y2);
          EMIT(X2);
          TERM;

L2:       PRIO(0);
M10main:JPPAUSE(1, M10depth);
          ISAT(ids[M2], M2depth, L4);
          EMIT(Y2);
L4:       EMIT(Y1);
          EMIT(X11);
          TRANS(Done);
M10depth:PRESENT(A, L2);
          ISAT(ids[M2], M2depth, L7);
          EMIT(Y2);
L7:       EMIT(Y1);
          EMIT(X10);
          TRANS(Done);
Done:     HALT;

M11:      HALT;

M0main: AWAIT(R);
          ISAT(ids[M2], M2depth, L10);
          EMIT(Y2);
L10:      ISAT(ids[M10], M10depth, L11);
          EMIT(Y1);
L11:      EMIT(Z);
          EMIT(Y0);
          EMIT(X0);
          TRANS(M0);

          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).



(a) SyncChart

```
TICKSTART(isInit, 1);

while (1) {
  do {
    EMIT(OFF);
    PAUSE;
  } while (!(PRESENT(S) && PRESENTPRE(S)));

  do {
    EMIT(ON);
    PAUSE;
  } while (!(PRESENT(R) && PRESENTPRE(R)));
}

TICKEND;
```

(b) SC tick function

Figure 4.4: The FilteredSR example.

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

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 $\mathsf{M10}_{9,\mathsf{FORKE}}$, for the test in the tick when M10 is entered, and at $\mathsf{M10}_{19,PRIO(0)}$, for the test in subsequent ticks.

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

   $\mathsf{M10}_{27,\mathsf{PRESENT(A)}} \succ \mathsf{M10}_{20,\mathsf{JPPAUSE}}$

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

4.3 FilteredSR

FilteredSR [3, Fig. 8-18], shown in Fig. 4.4, illustrates the use of PRE on pure signals.

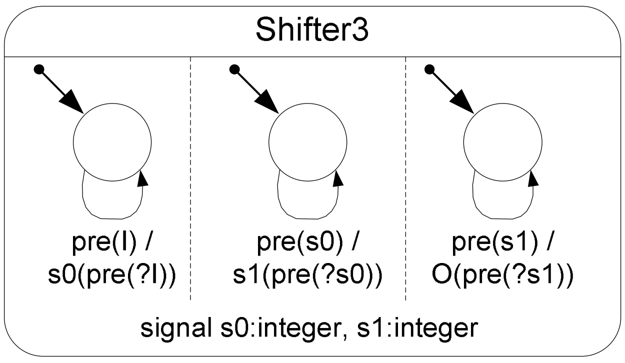
We here use *signal expressions*, in this case signal conjunction. In full regular C, such expressions can be built up the usual way with C's logical operators (!, &&, ||). If we want to restrict ourselves to plain SC operators, we can encode these expressions with control flow, as is done here.

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 [3, 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

```
 TICKSTART(isInit, 1);

 FORK(Shift1, 2);
 FORK(ShiftO, 3);

 while (1) {
   PAUSE;
   if (PRESENTPRE(I))
     EMITINT(S0, VALPRE(I));
 }

Shift1:
 while (1) {
   PAUSE;
   if (PRESENTPRE(S0))
     EMITINT(S1, VALPRE(S0));
 }

ShiftO:
 while (1) {
   PAUSE;
   if (PRESENTPRE(S1))
     EMITINT(O, VALPRE(S1));
 }

 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.

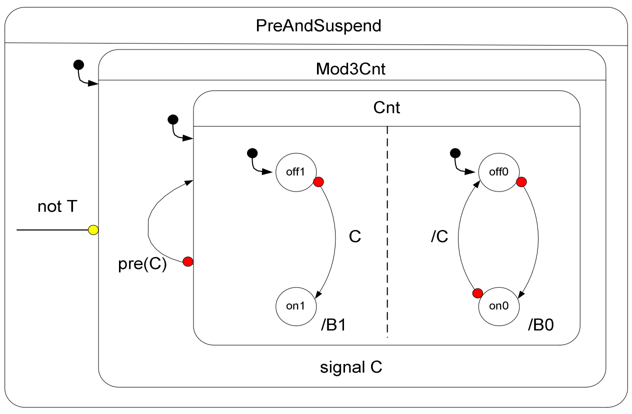
2. The Main thread is interpreted as one of the concurrent subthreads of macrostate Shifter3, say Shift0. It spawns off two concurrent threads (Shift1, ShiftO); see also the last note in Sec. 3.3.1.

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 FORKE, 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$_{13}$. Similarly for the other concurrent states.



(a) SyncChart

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

(b) Sample execution trace [3]

```
 TICKSTART(isInit, 4);

 FORK(Cnt, 3);
 FORKE(Mod3CntMain);

Cnt:
 FORK(Off1, 1);
 FORK(Off0, 2);
 FORKE(CntMain);

Off1:
 AWAIT(C);
 SUSTAIN(B1);

Off0:
 while (1) {
   PAUSE;
   EMIT(B0);
   PAUSE;
   EMIT(C);
 }

CntMain:
 while (1) {
   PAUSE;
   if (PRESENTPRE(C))
     TRANS(Cnt);
 }

Mod3CntMain:
 while (1) {
   PAUSE;
   SUSPEND(!PRESENT(T));
 }

 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

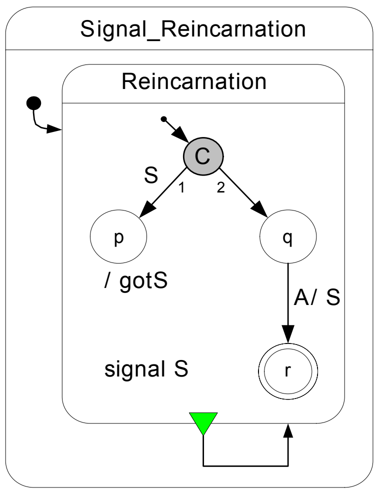
PreAndSuspend [3, Fig. 8-20], shown in Fig. 4.6, illustrates the proper handling of PRE in conjunction with suspension and local signals. See also the execution trace in Fig. 4.6b.

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



(a) SyncChart

```
TICKSTART(isInit, 1);

while (1) {
  SIGNAL(S);
   if (PRESENT(S)) {
     SUSTAIN(gotS);
   }
  AWAIT(A);
  EMIT(S);
}

TICKEND;
```

(b) SC tick function

Figure 4.7: The Reincarnation example.

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

   Cnt $\succ$ Off0, Cnt $\succ$ Off1

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

   Off0 $\succ$ Off1

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

4.6 Reincarnation

Reincarnation [3, Fig. 8-22], shown in Fig. 4.7, illustrates the SIGNAL instruction to handle signal reincarnation.

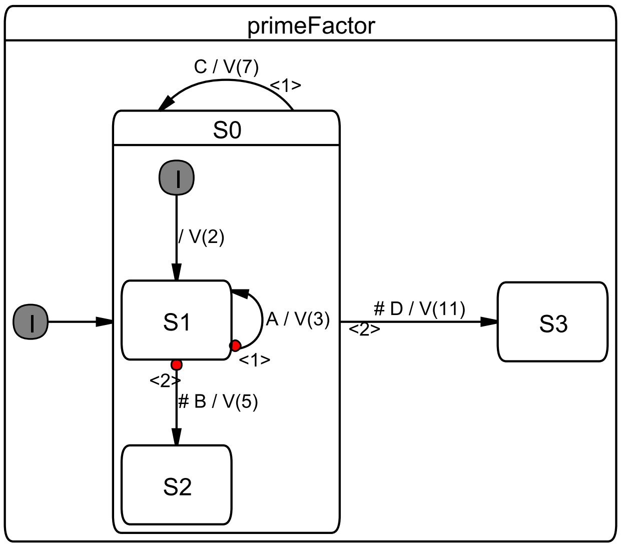
The canonical encoding in SC would have the Main thread spawn off an inner thread that computes the behavior of the Reincarnation state. However, as the macrostate Reincarnation only has a normal termination transition attached to it, all the Main thread does is to reenter itself once Reincarnation has 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 PRESENT(S, Q)$_5$.



(a) SyncChart

```
 TICKSTART(isInit, 1);

S0:
 FORK(S1, 2);
 FORKE(S0main);

S1:
 EMITINTMUL(V, 2);
 while (1) {
    if (PRESENT(B)) {
      EMITINTMUL(V, 5);
      HALT;
    }
    PAUSE;
    if (PRESENT(A))
      EMITINTMUL(V, 3);
 }

S0main:
 while (1) {
    if (PRESENT(D)) {
      EMITINTMUL(V, 11);
      TRANS(S3);
    }
    PAUSE;
    if (PRESENT(C)) {
      EMITINTMUL(V, 7);
      TRANS(S0);
    }
 }

S3:
 HALT;
 TICKEND;
```

(b) SC tick function

Figure 4.8: The PrimeFactor example.

4.7 PrimeFactor

PrimeFactor [3, 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*):

$\mathsf{Main}_{23,\mathsf{PRESENT(C)}} \succ \mathsf{Main}_{19,\mathsf{PRESENT(D)}}$

This is met by statement ordering.

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

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

   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 [14], 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 [14], 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 [6].

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 [17]. This can be alleviated by adopting a *strictly synchronous* semantics, which precisely states how computations should proceed [7]. 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 [20] and SyncCharts [3], 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++ [29]. As in UML Statecharts, this approach does not provide deterministic concurrency. Wagner *et al.* describe how to implement FSMs in C [34], 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 [11] is an extension of C inspired by Esterel. It employs the concepts of computational instants (ticks) and preemtions, but does not provide true concurrency; Reactive C's merge operator emulates concurrency by running threads sequentially, in their textual order.

FairThreads [8] 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) [16] 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) [2, 26, 27]. 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 [21], 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 [23, 10]. 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 [10], 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 [10], 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 [19], but does not rely on low-level timing for synchronization.

Another related programming model is SHIM [31], 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 [35], 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 [12]. Such a translation was proposed by André [4] 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 [23]. 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 [33], as done by the state machine to KEP compiler (smakc!) [30]. 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 [18], which implements synchronous concurrency via multi-threading. This multi-threading approach, which is also realized for example in the StarPro processor [36], 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 [28, 26], 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

Note: the measurements have been performed with SC Release 1.2. In particular code sizes should be improved with the current release.

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. 2.2c). 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.

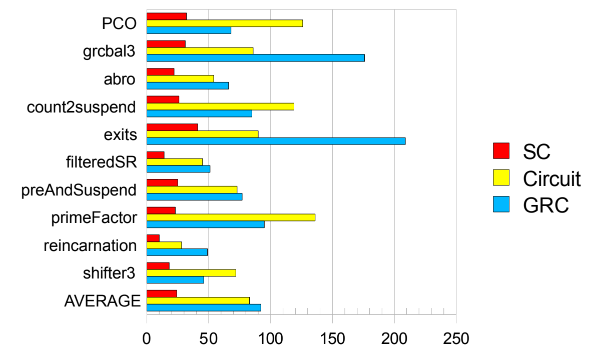
Another interesting point of comparison is the BAL VM instruction set, as it has been designed specifically to encode Esterel programs in as little memory as possible [10]. To encode grcbal3, BAL uses 74 instructions, of complexity comparable to the SC operators. The SC version makes do with 24 instructions, which—for this example—is more compact by a factor of three. Arguably, these instructions are also 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 Andé [3], 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.

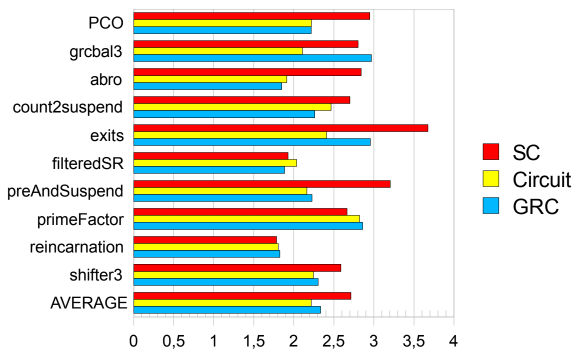
Fig. 6.1b compares sizes of object code for the tick function. Here, the SC code is larger on average, just in two cases it is slightly better than both E-Studio results. However, considering the sizes of the executable on an x86 architecture, shown in Fig. 6.1c, SC is ahead again. All results were obtained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

6.2 SC Performance

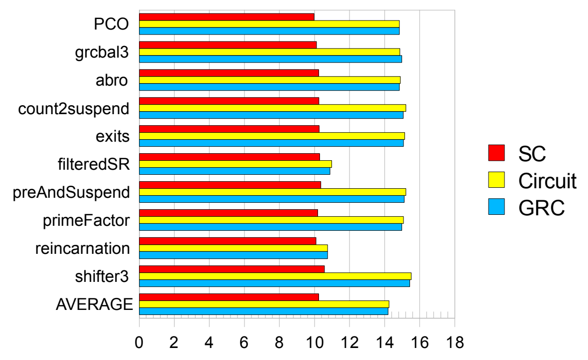
The development of SC has not been motivated primarily by performance concerns, but still it is interesting to see how it compares. On the negative side, SC basically just interprets a SyncChart, it cannot perform any global optimizations or partial evaluations at compile



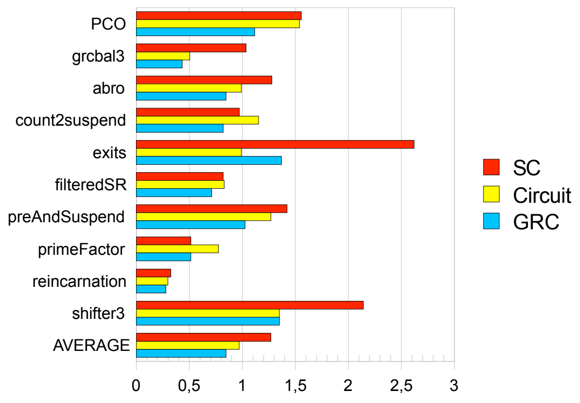
(a) Size of tick function in C source code, line count without empty lines and comments



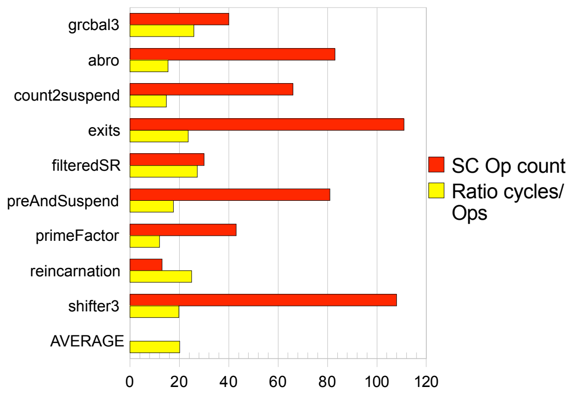
(b) Size of tick function object code, in Kbytes



(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.

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 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 μs. 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 [19] 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 [22, 26]. 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] S. Andalam, P. Roop, A. Girault, and C. Traulsen. PRET-C: A new language for programming precision timed architectures. Technical Report 6922, INRIA Grenoble Rhône-Alpes, 2009. http://hal.inria.fr/docs/00/39/16/21/PDF/rr.pdf.

[3] C. André. Semantics of SyncCharts. Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.

[4] 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.

[5] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 2007.

[6] M. v. d. Beeck. A comparison of Statecharts variants. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. Springer-Verlag, 1994.

[7] 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.

[8] F. Boussinot. Fairthreads: mixing cooperative and preemptive threads in C. *Concurrency and Computation: Practice and Experience*, 18(5):445–469, Apr. 2006.

[9] M. E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.

[10] S. A. Edwards and J. Zeng. Code generation in the Columbia Esterel Compiler. *EURASIP Journal on Embedded Systems*, Article ID 52651, 31 pages, 2007.

[11] Frederic Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, 1991.

[12] 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.

[13] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[14] 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.

[15] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pages 993–998, 1977.

[16] 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.

[17] E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[18] 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.

[19] 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.

[20] F. Maraninchi and Y. Rémond. Argos: An automaton-based synchronous language. *Computer Languages*, 27(27):61–92, 2001.

[21] M. Mendler and M. Pouzet. Uniform and modular composition of data-flow & control-flow in the lazy λ-calculus. Presentation at the International Open Workshop on Synchronous Programming (SYNCHRON'08), Aussois, France, Dec. 2008.

[22] 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.

[23] D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*. Springer, May 2007.

[24] 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.

[25] 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.

[26] P. S. Roop, S. Andalam, R. von Hanxleden, S. Yuan, and C. Traulsen. Tight WCRT analysis for synchronous C programs. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'09)*, Grenoble, France, Oct. 2009.

[27] P. S. Roop, S. Andalam, R. von Hanxleden, S. Yuan, and C. Traulsen. Tight WCRT analysis for synchronous C programs. Technical Report 0912, Christian-Albrechts-Universität Kiel, Department of Computer Science, Kiel, Germany, May 2009.

[28] 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.

[29] M. Samek. *Practical UML Statecharts in C/C++Event-Driven Programming for Embedded Systems.* Newnes, 2008.

[30] 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.

[31] 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.

[32] R. von Hanxleden. SyncCharts in C. In *Proceedings of the International Conference on Embedded Software (EMSOFT'09)*, Grenoble, France, Oct. 2009.

[33] R. von Hanxleden, X. Li, P. Roop, Z. Salcic, and L. H. Yoong. Reactive processing for reactive systems. *ERCIM News*, 67:28–29, Oct. 2006.

[34] F. Wagner, R. Schmuki, P. Wolstenholme, and T. W. Thomas. *Modeling Software with Finite State Machines: A Practical Approach.* Auerbach Publications, 2006.

[35] 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.

[36] S. Yuan, S. Andalam, L. H. Yoong, P. S. Roop, and Z. Salcic. STARPro—a new multithreaded 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

```c
/*! \ file sc.h
 *
 * Definition of SyncChart C macros.
 *
 * See README.txt for general information.
 * See LICENSE.txt for licensing information.
 * For further information, see
 * http://www.informatik.uni-kiel.de/rtsys/sc/ .
 *
 * @author Reinhard v. Hanxleden,
 * rvh@informatik.uni-kiel.de
 */
#include <stdio.h>

// ================================
//! Instruction counting/Tracing

/*! Check whether externflags has been defined (eg from gcc command line).
 * If so, suppress tracing and instruction counting.
 * This then results in compact macro-expanded source code and executable.
 */
#ifndef externflags
// Comment the following line out to surpress detailed tracing.
#define mytrace
#define instrCnt
#endif


//! Increment/decrement SC instruction counter.

/*! Decrement is needed in some places to avoid duplicate counting.
 */
#ifdef instrCnt
  #define instrCntIncr  tickInstrCnt++;
  #define instrCntIncrc  tickInstrCnt++,
  #define instrCntDecr  tickInstrCnt--;
#else
  #define instrCntIncr
  #define instrCntIncrc
  #define instrCntDecr
#endif


//! If tracing is turned on, print trace string.
#ifdef mytrace
  #define trace0(f)                    printf(f);
  #define trace1(f, a)                 printf(f, a);
  #define trace2(f, a, b)              printf(f, a, b);
  #define trace3(f, a, b, c)           printf(f, a, b, c);
  #define trace4(f, a, b, c, d)        printf(f, a, b, c, d);
  #define trace5(f, a, b, c, d, e)     printf(f, a, b, c, d, e);
  #define trace6(f, a, b, c, d, e, g)  printf(f, a, b, c, d, e, g);
  #define trace7(f, a, b, c, d, e, g, h)  printf(f, a, b, c, d, e, g, h);
  #define trace8(f, a, b, c, d, e, g, h, i)  printf(f, a, b, c, d, e, g, h, i);
  #define trace0c(f)                   printf(f),
  #define trace1c(f, a)                printf(f, a),
  #define trace2c(f, a, b)             printf(f, a, b),
  #define trace3c(f, a, b, c)          printf(f, a, b, c),
  #define elsetrace else {
  #define elsetraceend }
#else
  #define trace0(f)
  #define trace1(f, a)
  #define trace2(f, a, b)
  #define trace3(f, a, b, c)
  #define trace4(f, a, b, c, d)
  #define trace5(f, a, b, c, d, e)
  #define trace6(f, a, b, c, d, e, g)
  #define trace7(f, a, b, c, d, e, g, h)
  #define trace8(f, a, b, c, d, e, g, h, i)
  #define trace0c(f)
  #define trace1c(f, a)
  #define trace2c(f, a, b)
  #define trace3c(f, a, b, c)
  #define elsetrace
  #define elsetraceend
#endif


//! Count instruction (optionally), print trace string prefix (optionally).

/*! Trace string prefix takes a string s (typically denoting the instruction)
 * and identifies the executing thread, both by name and thread id.
 */
#define traceThread(s)                                              \
   instrCntIncr                                                     \
  trace3("%-9s␣%d/%s␣", s, _cid, state[_cid])

#define traceThreadc(s)                                             \
   instrCntIncrc                                                    \
  trace3c("%-9s␣%d/%s␣", s, _cid, state[_cid])


//! Print trace prefix + suffix

/*! s is string denoting instruction (eg, "PAUSE:")
 * f is format string for trace suffix
 * a, b, ... are arguments for format string
 */
#define trace0t(s, f)               traceThread(s) trace0(f)
#define trace1t(s, f, a)            traceThread(s) trace1(f, a)
#define trace2t(s, f, a, b)         traceThread(s) trace2(f, a, b)
#define trace3t(s, f, a, b, c)      traceThread(s) trace3(f, a, b, c)
#define trace4t(s, f, a, b, c, d) traceThread(s) trace4(f, a, b, c, d)
#define trace5t(s, f, a, b, c, d, e) traceThread(s) trace5(f, a, b, c, d, e)
#define trace6t(s, f, a, b, c, d, e, f1) traceThread(s) trace6(f, a, b, c, d, e, f1)
#define trace7t(s, f, a, b, c, d, e, f1, g) traceThread(s) trace7(f, a, b, c, d, e, f1, g)
#define trace8t(s, f, a, b, c, d, e, f1, g, h) traceThread(s) trace7(f, a, b, c, d, e, f1, g, h)
```

```


// Variants with trailig comma insted of semicolon:
#define trace0tc(s, f)              traceThreadc(s) trace0c(f)
#define trace1tc(s, f, a)           traceThreadc(s) trace1c(f, a)
#define trace2tc(s, f, a, b)        traceThreadc(s) trace2c(f, a, b)
#define trace3tc(s, f, a, b, c)     traceThreadc(s) trace3c(f, a, b, c)


// ================================
// Type definitions

typedef void            *labeltype;     //!< Computed goto - a la gcc
typedef unsigned int    bitvector;      //!< 32 bits on IA32
typedef bitvector       signalvector;   //!< 32 signals on IA32
typedef unsigned short  threadtype;     //!< Thread id/priority
typedef bitvector       threadvector;   //!< 32 threads on IA32

// ================================
// Global variables

signalvector  signals;                  //!< Bit mask for signals
threadvector  enabled;                  //!< Bit mask for enabled threads
threadvector  active;                   //!< Bit mask for active threads

#define      _idMax 8*sizeof(threadvector) //!< Number of threads

int           runCnt;                   //!< Counts program runs
int           tickCnt;                  //!< Counts program ticks
int           tickInstrCnt;             //!< Instructions in one tick
threadtype    _cid;                     //!< Id of current thread
labeltype     _pc[_idMax];              //!< Pseudo program counters
threadvector  _descs[_idMax];           //!< Descendants of thread
threadtype    _parent[_idMax];          //!< Parent of thread
labeltype     _returnAddress;           //!< For function calls (eg Exit Actions)

#ifdef mytrace
char     *statePrev[_idMax];            //!< State where thread resumed previous tick
char     *state[_idMax];                //!< State where thread resumed current tick

#define clearPC(id)                                          \
   statePrev[id] = "<init>";                                 \
   state[id] = "<init>"

#define initPC(p, label)                                     \
   _pc[p] = &&label;                                         \
   statePrev[p] = "<init>";                                  \
   state[p] = #label

#define setPC(id, label)                                     \
   _pc[id] = &&label;                                        \
   statePrev[id] = state[id];                                \
   state[id] = #label

#else                                   // No tracing
#define clearPC(id)
#define initPC(id, label) _pc[id] = &&label
#define setPC(id, label)  _pc[id] = &&label
#endif // mytrace


// ================================
// Declarations of constants and variables

// Constants defined in <application>.c
int runMax;                 //!< # of runs to execute
int tickMax;                //!< # of ticks to execute

extern const char *s2signame[];         //!< Names of signals


// ================================
// Declarations of functions defined in <application>.c:

//! Initialize signals to inputs for one tick.
void getInputs();

//! Set reference outputs and check valued signals, if there are any.
int checkOutputs(signalvector *tickOutputs);

//! Print value of a signal, if it has one.
void printVal(int id);

//! Compute one tick.
/*! Returns 1 if some thread is still active in current tick.
 */
int tick(int isInit);

//! Functions defined in sc.c
void selectCid();


// ================================
//! Dispatcher

#if ((defined __i386__ || defined __amd64__ || defined __x86_64__) &&
      defined __XXGNUC__)
// Version 1: x86 + gcc available.
// Use fast Bit Scan Reverse assembler instruction.
#define dispatch()                                   \
  __asm volatile ("bsrl %1,%0\n"                     \
                  : "=r" (_cid)                      \
                  : "c" (active)                     \
                  );                                 \
  goto *_pc[_cid]

#else
// Version 2: x86 + gcc not available.
// Call function, defined in sc.c.
#define dispatch()                                   \
  selectCid();                                       \
  goto *_pc[_cid]
#endif


// ================================
// Low-level routines

//! Encoding of signal/thread 'u' (some non-negative int) in bitvector.
/*! This implementation is fast and simple, BUT limits the max thread
 * ID and max signal ID to the word width of the machine (eg 32).
 */
#define u2b(u)          (1 << u)


// ================================
// Keeping track of the thread status

//! Thread enabling/disabling
#define enable(id)                                   \
  enabled |= u2b(id);                                \
  active  |= u2b(id)

#define enableInit(id)                               \
  enabled = u2b(id);                                 \
  active  = enabled

#define disable(id)               enabled &= ~u2b(id)
#define disableSet(idset)         enabled &= ~idset
#define isEnabled(id)             (enabled & u2b(id))
#define isEnabledNotOnly(id)      (enabled != u2b(id))
#define isEnabledNoneOf(idset)    ((enabled & idset) == 0)

//! Thread (de-)activation
#define activate(id)              active |= u2b(id)
#define deactivate(id)            active &= ~u2b(id)
#define deactivateSet(idset)      active &= ~idset
#define isActive(id)              (active & u2b(id))


// ================================
// Tick start and end

#define _TickEnd 0                     // Priority of TickEnd thread

//! Start a tick (an instant). 'p' denotes the main thread.
/*! IF this is the initial tick ('isIni' is set),
 *   THEN initialize things and continue with following instruction,
 *   ELSE call dispatcher to resume where we left off.
 *
 * This also initializes the _TickEnd thread. Note that
 * _parent[_TickEnd] is undefined. Note also that _descs[_TickEnd]
 * does not matter, as that thread should never perform an ABORT
 * (TRANS) or JOIN.
 */
#define TICKSTART(isIni, p)                                  \
  static threadtype _pid, _ppid;                             \
  static threadvector _fdescs = 0;                           \
  static int _i;                                             \
  freezePreClear                                             \
  if (isIni) {                                               \
    tickCnt = 0;                                             \
    initPC(_TickEnd, _L_TICKEND);                            \
    enableInit(_TickEnd);                                    \
    _cid = p;                                                \
      _parent[_cid] = _TickEnd;                   \
      clearPC(_cid);                              \
      enable(_cid);                               \
      setPreInit                                  \
        setValInit                                \
        } else {                                  \
      active = enabled;                           \
      dispatch_;                                  \
    }


//! Complete a tick.
/*! Return 0 iff computation has terminated
 */
#define TICKEND                              \
  _L_TICKEND: setPre                         \
  return isEnabledNotOnly(_TickEnd);         \
  mergedDispatch


//! If inlineDispatch is defined, call dispatcher at each operator that needs it.
//! Otherwise, create shared code block for TERM/PAUSE/dispatch.
/*! This can be included by TICKEND
 */
#ifdef inlineDispatch
#define dispatch_ dispatch()
#define mergedDispatch
#else                              // Shared dispatch
#define dispatch_ goto _L_dispatch
#define mergedDispatch                                \
  _L_TERM:  disable(_cid);                            \
_L_PAUSEG:  deactivate(_cid);                         \
_L_dispatch: dispatch()

#endif




// ================================
// Pausing, suspending, aborting and terminating a thread}

//! Construct a label, of the form "_L' line in source file '".

/*! Origindally contributed by Nicolas Berthier (nicolas.berthier@imag.fr)
 *
 * To avoid label clashes, one must
 * - not generate multiple labels at the same line (this could be, eg,
 *   "PAUSE; PAUSE" in one line)
 * - not include another files within a function (this appears
 *   unlikely anyway)
 *
 * Note that goto labels have function scope, hence it is ok to have
 * identical labels in different files, as long as they belong to
 * different functions.
 *
 * Note also that the ## preprocessing macro, which concatenates
 * strings, prevents macro expansion of it arguments. Thus the
 * construction using _CONCAT and _CONCAT_helper.
 */
#define _CONCAT_helper(a, b) a ## b
#define _CONCAT(a, b)         _CONCAT_helper(a, b)
#define __LABEL__             _CONCAT(_L, __LINE__)
#define __LABELL__            _CONCAT(_LL, __LINE__)



//! Pause a thread, resume at subsequent statement.
/*! Semantically, this is the primitive operator.
 * In terms of implementation, it is built with PAUSEG.
 */
#define PAUSE                                                    \
  do {                                                           \
    trace1t("PAUSE:", "pauses,␣active␣=␣0%o\n", active)          \
    PAUSEG_(__LABEL__); __LABEL__: (void) 0;                     \
  } while (0)

//! Shorthand for 'PAUSE; GOTO(label)'.
/*! Pause a thread, resume at 'label'.
 */
#define PAUSEG(label) do {                                       \
    trace1t("PAUSEG:", "pauses,␣active␣=␣0%o\n", active)         \
    PAUSEG_(label);                                              \
  } while (0)

//! Helper function (if/else-unsafe)
#ifdef inlineDispatch
#define PAUSEG_(label)                                           \
  setPC(_cid, label);                                            \
  deactivate(_cid);                                              \
  dispatch_

#else
#define PAUSEG_(label)                                           \
    setPC(_cid, label);                                          \
    goto _L_PAUSEG
#endif


//! Shorthand for 'label: PAUSE; GOTO(label)'.
/* Halts a thread, but does not terminate it (compare with TERM).
 */
#define HALT do {                                                \
  __LABEL__: trace1t("HALT:", "pauses,␣active␣=␣0%o\n", active) \
    PAUSEG_(__LABEL__);                                          \
  } while (0)


//! Suspend current thread if 'cond' is true.
/*! Note: suspension is implemented by deactivating the current thread
 * as well as its descendants. This exploits that the PCs of the descendants
 * must reside at tick boundaries.
 */
#define SUSPEND(cond) do {                                       \
__LABEL__: if (cond) {                                           \
    trace1t("SUSPEND:", "suspends␣itself␣and␣descendants␣0%o\n", _descs[_cid]) \
    active &= ~_descs[_cid];                                     \
    freezePre                                                    \
    instrCntDecr                                                 \
    PAUSEG_(__LABEL__);                                          \
  }                                                              \
 } while (0)


//! Suspend current thread, goto 'label'.
/*! Note: suspension is implemented by deactivating the current thread
 * as well as its descendants. This exploits that the PCs of the descendants
 * must reside at tick boundaries.
 */
#define SUSPENDGOTO(label) do {                                  \
    trace1t("SUSPENDGOT:", "suspends␣itself␣and␣descendants␣0%o\n", _descs[_cid]) \
    active &= ~_descs[_cid];                                     \
    freezePre                                                    \
    instrCntDecr                                                 \
    PAUSEG_(label);                                              \
  } while (0)


//! Transition to 'label', kill descendant threads (implements abortion).
/*! Shorthand for 'ABORT; GOTO(label)'.
 */
#define TRANS(label) do {                                        \
    ABORT_;                                                      \
    trace3t("TRANS:", "disables␣0%o,␣transfers␣to␣%s,␣enabled␣=␣0%o\n", \
        _descs[_cid], #label, enabled)                           \
    goto label;                                                  \
  } while (0)


//! Abort (terminate) descendant threads.
#define ABORT do {                                               \
    ABORT_;                                                      \
    trace2t("ABORT:", "disables␣0%o,␣enabled␣=␣0%o\n",           \
        _descs[_cid], enabled)                                   \
  } while (0)


//!  Helper function (if/else-unsafe)
#define ABORT_                                                   \
    disableSet(_descs[_cid]);                                    \
    deactivateSet(_descs[_cid])


//! Terminate a thread.
/* Compare to definition of HALT, which lets a thread idle at current position.
 */
#define TERM do {                                                \
    trace1t("TERM:", "terminates,␣enabled␣=␣0%o\n", enabled)     \
    TERM_;                                                       \
  } while (0)
```

```
//! Helper function (if/else-unsafe)
#ifdef inlineDispatch
#define TERM_                                                   \
      disable(_cid);                                            \
      deactivate(_cid);                                         \
      dispatch_
#else
#define TERM_ goto _L_TERM
#endif


// ================================
// Handling concurrency

//! Spawn a thread at 'label', with priority 'p'.
/*!
 */
#define FORK(label, p) do {                                     \
      FORK_(label, p);                                          \
      trace3t("FORK:", "forks␣%d/%s,␣active␣=␣0%o\n", p, #label, active) \
   } while (0)


//! Helper function (if/else-unsafe)
#define FORK_(label, p)                                         \
   initPC(p, label);                                            \
   _parent[p] = _cid;                                           \
   _fdescs |= u2b(p);                                           \
   enable(p)


//! Denote parent thread, starting at 'label'.
/*! Must also calculate descendants.
 * Descendants are used
 * - to check for termination (with JOIN)
 * - to be disabled upon abortion (with TRANS)
 */
#define FORKE(label) do {                                       \
      trace0t("FORKE:", "\n")                                   \
      FORKE_(label);                                            \
   } while (0)


//! Helper function (if/else-unsafe)
#define FORKE_(label)                                           \
   setPC(_cid, label);                                          \
   _descs[_cid] = _fdescs;                                      \
   _fdescs = 0;                                                 \
   _pid = _cid;                                                 \
   while ((_ppid = _parent[_pid]) != _TickEnd) {                \
      _descs[_ppid] |= _descs[_pid];                            \
      _pid = _ppid;                                             \
   }                                                            \
   dispatch_

//! Join completed child threads.
/*! IF all descendants have terminated,
 *    THEN proceed after JOINE,
 *    ELSE pause, resume at 'elselabel'.
 *
 * Semantically, this is the primitive Join-operator, from which the
 * others can be derived; hence JOINE appears first, and the other
 * operators are considered shorthands.
 *
 * In terms of implementation, the operators are built from JOINEG,
 * which semantically corresponds to JOINE + GOTO.
 */
#define JOINE(elselabel) do {                                   \
   JOINEG_(__LABEL__, elselabel);                               \
   __LABEL__: (void) 0;                                         \
   } while (0)


//! Shorthand for 'JOINE(elselabel); GOTO(thenlabel)'.
/*! IF all descendants have terminated,
 *    THEN jump to 'thenlabel',
 *    ELSE pause, resume at 'elselabel'
 */
#define JOINEG(thenlabel, elselabel) do {                       \
      JOINEG_(thenlabel, elselabel);                            \
   } while (0)


//! Helper function (if/else-unsafe)
#define JOINEG_(thenlabel, elselabel)                           \
   if (isEnabledNoneOf(_descs[_cid])) {                         \
      trace1t("JOINEG:", "joins,␣transfers␣to␣%s\n", #thenlabel) \
      goto thenlabel;                                           \
   }                                                            \
   trace1t("JOINEG:", "does␣not␣join,␣pauses␣at␣%s\n", #elselabel) \
   instrCntDecr                                                 \
   PAUSEG_(elselabel)


//! Shorthand for 'elselabel: JOINE(elselabel)'. Join completed child threads.
/*! IF all descendants have terminated,
 *    THEN proceed,
 *    ELSE pause, resume at JOIN.
 */
#define JOIN do {                                               \
   __LABEL__: JOINEG_(__LABELL__, __LABEL__);                   \
   __LABELL__: (void) 0;                                        \
   } while (0)


//! Set priority of a thread.
/*! Semantically, this is the primitive operator.
 * In terms of implementation, PRIOG is the basic operator.
 */
#define PRIO(p) do {                                            \
      trace1t("PRIO:", "set␣to␣priority␣%d\n", p)               \
      PRIOG_(p, __LABEL__);                                     \
   __LABEL__: (void) 0;                                         \
   } while (0)


//! Shorthand for 'PRIO(p); GOTO(label)'.
#define PRIOG(p, label) do {                                    \
      trace2t("PRIOG:", "set␣to␣priority␣%d,␣goto␣%s\n", p, #label) \
      PRIOG_(p, label);                                         \
   } while (0)


//! Helper function (if/else-unsafe)
#define PRIO_(p)                                                \
   deactivate(_cid);                                            \
   disable(_cid);                                               \
   _descs[p] = _descs[_cid];                                    \
   _pid = _cid;                                                 \
   while ((_ppid = _parent[_pid]) != _TickEnd) {                \
      _descs[_ppid] &= ~u2b(_cid);                              \
      _descs[_ppid] |= u2b(p);                                  \
      _pid = _ppid;                                             \
   }                                                            \
   _cid = p;                                                    \
   enable(_cid);


//! Helper function (if/else-unsafe)
#define PRIOG_(p, label)                                        \
   PRIO_(p);                                                    \
   setPC(_cid, label);                                          \
   dispatch_



// ================================
// Efficient shorthands for thread handling

//! Efficient shorthand for 'PRIO(p); PAUSE; GOTO(label)'.
/*! Set a priority, then pause (this sets "prionext"), resume at 'label'.
 *
 * This shorthand avoids the context switch immediately before the PAUSE.
 */
#define PPAUSEG(p, label) do {                                  \
      trace2t("PPAUSEG:", "sets␣prio␣to␣%d,␣pauses,␣resumes␣at␣%s\n", p, #label) \
      PPAUSEG_(p, label);                                       \
   } while (0)


//! Helper function (if/else-unsafe)
#define PPAUSEG_(p, label)                                      \
   PRIO_(p);                                                    \
   instrCntDecr                                                 \
   PAUSEG_(label)


//! Efficient shorthand for 'PRIO(p); PAUSE'.
/*! Set a priority, then pause (this sets "prionext").
 *
 * This shorthand avoids the context switch immediately before the PAUSE.
 */
#define PPAUSE(p) do {                                          \
      trace1t("PPAUSE:", "sets␣prio␣to␣%d,␣pauses\n", p)        \
      PPAUSEG_(__LABEL__);                                      \
   __LABEL__: (void) 0;                                         \
    } while (0)


//! Efficient shorthand for 'JOINE(label); GOTO thenlabel; label: PRIO(p); PAUSE; GOTO(elselabel)'.
/*! IF all descendants have terminated,
 *    THEN jump to 'thenlabel',
 *    ELSE set priority , pause, and continue at ' elselabel '.
 *
 * This shorthand avoids the context switch immediately before the PAUSE.
 */
#define JPPAUSEG(p, thenlabel, elselabel) do {                         \
      trace2t ("JPPAUSEG:", "%s,␣prio␣=␣%d\n",                          \
               isEnabledNoneOf(_descs[_cid]) ? "joins" : "does␣not␣join", p) \
      JPPAUSEG_(p, thenlabel, elselabel);                              \
    } while (0)


//! Helper function (if/else-unsafe)
#define JPPAUSEG_(p, thenlabel, elselabel)                             \
    if (isEnabledNoneOf(_descs[_cid]))                                 \
      goto thenlabel;                                                  \
    instrCntDecr                                                       \
    PPAUSEG_(p, elselabel)


//! Shorthand for 'JOINE(label); GOTO thenlabel; label: PRIO(p); PAUSE; GOTO(elselabel); thenlabel:'.
/*! IF all descendants have terminated,
 *    THEN proceed,
 *    ELSE set priority , pause, and continue at ' elselabel '.
 *
 * This shorthand avoids the context switch immediately before the PAUSE.
 */
#define JPPAUSE(p, elselabel) do {                                     \
      trace2t ("JPPAUSE:", "%s,␣prio␣=␣%d\n",                           \
               isEnabledNoneOf(_descs[_cid]) ? "joins" : "does␣not␣join", p) \
      JPPAUSEG_(p, __LABEL__, elselabel);                              \
    __LABEL__: (void) 0;                                               \
    } while (0)


// ================================
// Signal initialization , emission and testing

//! Initialize a local signal (handles reincarnation )
#define SIGNAL(s) do {                                                 \
      trace2t ("SIGNAL:", "initializes␣%s/%d\n", s2signame[s], s)      \
      signals &= ~u2b(s);                                              \
    } while (0)


//! Emission of a pure signal 's'
#define EMIT(s) do {                                                   \
      trace2t ("EMIT:", "emits␣%s/%d\n", s2signame[s], s)              \
      signals |= u2b(s);                                               \
    } while (0)


//! Sustain a pure signal 's'
#define SUSTAIN(s) do {                                                \
    __LABEL__: trace2t("SUSTAIN:", "emits␣%s/%d\n", s2signame[s], s)   \
      EMIT(s); PAUSEG_(__LABEL__);                                     \
    } while (0)


//! Test for presence of signal 's'.
/*! IF 's' is present,
 *    THEN return 1,
 *    ELSE return 0.
 */
#define PRESENT(s)                                                     \
    (trace3tc ("PRESENT:", "determines␣%s/%d␣as␣%s\n",                 \
               s2signame[s], s, (signals & u2b(s)) ? "present" : "absent") \
     (signals & u2b(s)))


//! Test for presence of signal 's'.
/*! IF 's' is present,
 *    THEN proceed to next instruction,
 *    ELSE jump to 'label'
 */
#define PRESENTELSE(s, label) do {                                     \
      if (!( signals & u2b(s))) {                                      \
        trace3t ("PRESENTELSE:", "determines␣%s/%d␣as␣absent,␣transfers␣to␣%s\n", \
                 s2signame[s], s, #label)                              \
        goto label;                                                    \
      }                                                                \
      trace2t ("PRESENTELSE:", "determines␣%s/%d␣as␣present\n", s2signame[s], s) \
    } while (0)


//! If signal 's' is present, emit 't'.
/*! Shorthand for 'PRESENTELSE(s, label); EMIT(t); label:'
 */
#define PRESENTEMIT(s, t) do {                                         \
      if ( signals & u2b(s)) {                                         \
        trace4t ("PRESENTEMIT:", "determines␣%s/%d␣as␣present,␣emits␣%s/%d\n", \
                 s2signame[s], s, s2signame[t], t)                     \
        signals |= u2b(t);                                             \
      }                                                                \
      elsetrace                                                        \
        trace2t ("PRESENTEMIT:", "determines␣%s/%d␣as␣absent\n", s2signame[s], s) \
      elsetraceend                                                     \
    } while (0)


//! Await (immediately) signal 's'.
/*! IF 's' is present,
 *    THEN proceed to next instruction,
 *    ELSE pause.
 *
 * Shorthand for 'GOTO(label); elselabel: PAUSE; label: PRESENT(s, elselabel)'.
 */
#define AWAITI(s) do {                                                 \
  __LABEL__: if (!( signals & u2b(s))) {                               \
        trace2t ("AWAITI:", "determines␣%s/%d␣as␣absent,␣waits\n",     \
                 s2signame[s], s)                                      \
        PAUSEG_(__LABEL__);                                            \
      }                                                                \
      trace2t ("AWAITI:", "determines␣%s/%d␣as␣present,␣proceeds\n", s2signame[s], s) \
    } while (0)


//! Await (non-immediately) signal 's'.
/*! Pause; Then, IF 's' is present,
 *    THEN proceed to next instruction,
 *    ELSE pause.
 *
 * Shorthand for 'PAUSE; AWAITI(s)',
 * or, alternatively : ' elselabel : PAUSE; PRESENT(s, elselabel)'.
 */
#define AWAIT(s) do {                                                  \
      trace0t ("AWAIT:", "initial␣pause\n")                            \
      goto __LABELL__;                                                 \
  __LABEL__: if (!( signals & u2b(s))) {                               \
        trace2t ("AWAIT:", "determines␣%s/%d␣as␣absent,␣waits\n",      \
                 s2signame[s], s)                                      \
        __LABELL__: PAUSEG_(__LABEL__);                                \
      }                                                                \
      trace2t ("AWAIT:", "determines␣%s/%d␣as␣present,␣proceeds\n", s2signame[s], s) \
    } while (0)


// ================================
// Handling valued signals .
// The following is compiled conditionally depending on valSigIntCnt .
// <application>.c must define valSigIntCnt if valued signals are used.

#ifdef valSigIntCnt
//! At beginning of initial tick :
//! Initialize valued signals (-1 is for "undefined").
#define setValInit                                   \
    for ( _i = 0; _i < valSigIntCnt; _i++)           \
      valSigInt [_i] = -1;

#else     // #ifdef valSigIntCnt
#define setValInit
#endif

//! Emission of a valued signal 's', type integer .
#define EMITINT(s, val) do {                                           \
      valSigInt [s] = val;                                             \
      trace3t ("EMITInt:", "emits␣%s/%d,␣value␣%d\n",                  \
               s2signame[s], s, val)                                   \
      signals |= u2b(s);                                               \
    } while (0)


//! Emission of a valued signal 's', type integer , combined with * .
#define EMITINTMUL(s, val) do {                                        \
      valSigIntMult[s] *= val;                                          \
      trace4t("EMITInt*:", "emits %s/%d, value %d, result %d\n",        \
              s2signame[s], s, val, valSigIntMult[s])                   \
      signals |= u2b(s);                                                \
  } while (0)


//! Retrieve value of signal 's'.
#define VAL(s) (                                                        \
      trace3tc("VAL:", "determines value of %s/%d as %d\n",             \
              s2signame[s], s, valSigInt[s])                            \
      valSigInt[s])


//! Retrieve value of signal 's' into 'reg'.
#define VALREG(s, reg) do {                                             \
      trace3t("VALREG:", "determines value of %s/%d as %d\n",           \
              s2signame[s], s, valSigInt[s])                            \
      reg = valSigInt[s];                                               \
  } while (0)


// ================================
// Handling PRE.
// The following is compiled conditionally depending on usePRE
// <application>.c must define usePRE if PRE is used

#ifdef usePRE
signalvector sigsPre;      //!< Signals from previous tick
signalvector sigsFreeze;   //!< Signals that are frozen, due to suspension

//! At beginning of initial tick:
//!  Initialize previous signals.
#define setPreInit   \
  sigsPre = 0;       \
  setPreValInit;

//! At end of tick:
//! Copy current signals (unless frozen) to previous signals.
#define setPre                                                   \
  sigsPre = (sigsPre & sigsFreeze) | (signals & ~sigsFreeze);    \
  setPreVal

//! When suspending current thread:
//! Add signals local to current thread or its descendants
//! to list of signals to freeze.
#define freezePre          sigsFreeze |= sigsDescs[_cid];

//! At beginning of tick:
//! Clear list of signals to freeze.
#define freezePreClear   sigsFreeze = 0;

#else     // #ifdef usePRE
#define setPreInit
#define setPre
#define freezePre
#define freezePreClear
#endif    // #ifdef usePRE


//! Test for presence of signal in previous tick.
/*! IF 's' was present in previous tick,
 *   THEN return 1,
 *   ELSE return 0.
 */
#define PRESENTPRE(s)                                                   \
  (trace3tc("PRESENTPRE:", "determines %s/%d as %s\n",                  \
            s2signame[s], s, (sigsPre & u2b(s)) ? "present" : "absent") \
   (sigsPre & u2b(s)))


//! Test for presence of signal in previous tick.
/*! IF 's' was present in previous tick,
 *   THEN proceed to next instruction,
 *   ELSE jump to 'label'
 */
#define PRESENTPREELSE(s, label) do {                                   \
      if (!(sigsPre & u2b(s))) {                                        \
        trace3t("PRESENTPRE:", "determines previous %s/%d as absent, 
                transfers to %s\n", \
                s2signame[s], s, #label)                                \
        goto label;                                                     \
      }                                                                 \
      trace2t("PRESENTPRE:", "determines previous %s/%d as present\n", \
              s2signame[s], s)                                          \
  } while (0)


// ================================
// Handling valued signals in conjunction with PRE

#ifdef usePRE
#ifdef valSigIntCnt
//! At beginning of initial tick:
//!  Initialize previous signal values.
#define setPreValInit                                     \
  for (_i = 0; _i < valSigIntCnt; _i++)                   \
    valSigIntPre[_i] = -1;

//! At end of tick:
//! Copy values of current signals (unless frozen) to previous signals.
#define setPreVal                                         \
  for (_i = 0; _i < valSigIntCnt; _i++)                   \
    if (!(sigsFreeze & u2b(_i)))                          \
      valSigIntPre[_i] = valSigInt[_i];

#else     // #ifdef valSigIntCnt
#define setPreValInit
#define setPreVal
#endif    // #ifdef valSigIntCnt
#endif    // #ifdef usePRE


//! Retrieve previous value of signal 's'.
#define VALPRE(s) (                                                     \
      trace3tc("VALPRE:", "determines value of %s/%d as %d\n", \
              s2signame[s], s, valSigIntPre[s])                         \
      valSigIntPre[s])


//! Retrieve previous value of signal 's' into 'reg'.
#define VALPREREG(s, reg) do {                                          \
      trace3t("VALPREREG:", "determines value of %s/%d as %d\n", \
              s2signame[s], s, valSigIntPre[s])                         \
      reg = valSigIntPre[s];                                            \
  } while (0)


// ================================
// Control flow: jumps

//! Just a goto that also gets counted as instruction.
#define GOTO(label) do {                                                \
      trace1t("GOTO:", "transfer to %s\n",      #label)                 \
      instrCntIncr                                                      \
      goto label;                                                       \
  } while (0)


// ================================
// Support for Exit Actions

//! Test whether an Exit Action has to be performed.
/*! IF thread 'id' is active and at state 'statelabel',
 *   THEN proceed to next instruction,
 *   ELSE jump to 'label'
 */
#define ISAT(id, statelabel, label) {                                         \
  if (isEnabled(id) && (_pc[id] == &&statelabel)) {                           \
      trace1t("ISAT:", " is  at %s\n", #statelabel)                           \
  } else {                                                                    \
      trace2t("ISAT:", " is  not  at %s, transfer to %s\n",                   \
              #statelabel, #label)                                            \
      goto label;                                                             \
  }}


//! Call a function at 'label'.
/*! Use this if an Exit Action _must_ be performed.
 */
#define CALL(label) do {                                                      \
      trace1t("CALL:", " calls %s\n", #label)                                 \
        _returnAddress = &&__LABEL__;                                         \
      goto label;                                                             \
  __LABEL__: (void) 0;                                                        \
  } while (0)


//! Return from a function call
#define RET do {                                                              \
      trace0t("RET:", "returns\n")                                            \
      goto *_returnAddress;                                                   \
  } while (0)


//! Conditionally call a function.
/*! IF thread 'id' is active and at state 'statelabel',
 *   THEN call function at 'label';
 * Return to ' retlabel '
 * Use this if an Exit Action _may_ have to be performed
 * Shorthand for ISAT(id, statelabel , retlabel ); CALL(label, retlabel );
 */
#define ISATCALL(id, statelabel, label ) do {                              \
    if (isEnabled(id) && (_pc[id] == &&statelabel)) {                      \
      trace1t("ISATCALL:", "calls␣%s\n", #label)                           \
      _returnAddress = &&__LABEL__;                                        \
      goto label ;                                                         \
    }                                                                      \
    trace1t("ISATCALL:", "does␣␣not␣␣call␣%s\n", #label)                  \
    __LABEL__: (void) 0;                                                   \
  } while (0)
```

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

```
/*! \ file sc.c
 *
 * Main file for using SyncChart C macros
 *
 * See README.txt for general information.
 * See LICENSE.txt for licensing information .
 * For further information , see
 * http://www.informatik.uni−kiel.de/rtsys/sc/ .
 *
 * @author Reinhard v. Hanxleden,
 * rvh@informatik. uni−kiel.de
 */

#include "sc.h"


// ================================
//! Computing the id of next thread to be dispatched.
/*! Uses obvious algorithm, run time linear in position of highest bit .
 * Note that there are also alternatives that run logarithmic to bit vector size .
 * See eg http://graphics . stanford .edu/~seander/bithacks.html#IntegerLog .
 * Which is actually faster depends on application .
 */
void selectCid () {
  int act;

  _cid = 0;
  for (act = active; act > 1; act >>= 1)
    _cid++;
}


// ================================
//! Tracing routines
void vec2names(char *prefix, char* suffix , bitvector ids , const char * names[])
{
#ifdef mytrace
  int id = 0;
  int first = 1;

  printf ("%s", prefix );
  while ( ids ) {
    if ( ids & 1) {
      if ( first ) {
        first = 0;
      } else {
        printf (",␣");
      }
      printf ("␣%d/", id);
      if (names) {
        printf ("%s", names[id]);
        printVal (id);
      } else {
        printf ("%s", state[id]);
      }
    }
    ids >>= 1;
    id++;
  }

  if ( first ) {
    printf ("<init>");
  }
  printf ("%s", suffix );
#endif
}



// ================================
//! The main program
/*! Returns 0 iff outputs generated by program match reference trace
 */
int main()
{
  int runInstrCnt;              // Instructions in one run
  int runsInstrCnt = 0;         // Instructions accumulated over all runs
  int outputsOK = 1;            // Outputs of simulation correct?
  int notDone;                  // Current run not done yet?
  int init ;                    // Is initial tick?
  signalvector tickInputs ;     // Input values for a tick
  signalvector tickOutputs;     // Reference output values for a tick
  signalvector tickSignals ;    // Reference signal values for a tick

  // Execute all runs
  for (runCnt = 0; (runCnt < runMax) && outputsOK; runCnt++) {
    printf ("####␣RUN␣%d␣STARTS␣#############\n", runCnt);

    runInstrCnt = 0;
    tickCnt = 0;
    init = 1;
    enabled = 0;

    do {                                        // Execute all ticks of one run
      tickInstrCnt = 0;
      getInputs();
      tickInputs = signals;

      trace3("====␣TICK␣%d␣STARTS,␣inputs␣=␣0%o,␣enabled␣=␣0%o\n",
             tickCnt, tickInputs , enabled);
      vec2names("====␣Inputs␣(id/name):␣", "\n", tickInputs, s2signame);
      vec2names("====␣Enabled␣(id/state):␣", "\n", enabled, 0);

      notDone = tick(init);  // Call automaton function
      init = 0;

      runInstrCnt += tickInstrCnt;
      trace2("====␣TICK␣%d␣terminates␣after␣%d␣instructions.\n",
             tickCnt, tickInstrCnt );
      vec2names("====␣Enabled␣(id/state):␣", "\n", enabled, 0);
      vec2names("====␣Resulting␣signals␣(name/id):␣", "", signals,
             s2signame);
      outputsOK = checkOutputs(&tickOutputs);
      if (outputsOK) {
        tickSignals = tickInputs | tickOutputs;
        if ( signals == tickSignals) {
          trace0(",␣Outputs␣OK.\n\n");
        } else {
          vec2names(",␣Outputs␣NOT␣OK␣−␣expected␣signals␣", "!!\n\n",
                    tickSignals , s2signame);
          outputsOK = 0;
        }
      } else {
        notDone = 0;
      }

      tickCnt++;
      if (tickCnt >= tickMax) {
        printf ("====␣Executed␣tickMax␣=␣%d␣ticks,␣terminate.\n",
                tickMax);
        notDone = 0;
      }

    } while (notDone && outputsOK);

    printf ("####␣RUN␣%d␣terminates␣after␣%d␣instructions\n\n",
            runCnt, runInstrCnt);
    runsInstrCnt += runInstrCnt;
  };

  printf ("####␣All␣runs␣terminate,␣after␣%d␣instructions\n\n",
          runsInstrCnt);
  return !outputsOK;
}
```

Listing A.3: The Makefile

```
sc−progs := ABRO Count2Suspend FilteredSR grcbal3 \
        PreAndSuspend PrimeFactor Reincarnation Shifter3 SurfDepth

progs := $(sc−progs) $(sc−progs:=−sc) \
        ABRO−expanded ABRO−expanded−annotated ABRO−expanded−stripped \
        Exits IfElseSafe PCO

```

```
downloads := LICENSE.txt Makefile README.txt doxygen.conf sc.c sc.h selectCid.c \
	$(progs:=.c) $(progs:=.out)

# CCFLAGS := -Wall -D inlineDispatch
CCFLAGS := -O3
# -Wall complains about unused label _L_TERM
# CCFLAGS := -Wall -O3

all: $(progs:=.out)

make.trace:
	(time make) >& $@

allprogs: $(progs)

PCO: PCO.c sc.h Makefile
	gcc $(CCFLAGS) PCO.c selectCid.c -o PCO

PCO2: PCO2.c sc.h Makefile
	gcc $(CCFLAGS) PCO2.c selectCid.c -o PCO2

# Want to compile ABRO-C without tracing, to not interfere with kbd input
ABRO-C: ABRO-C.c sc.c sc.h Makefile
	gcc $(CCFLAGS) -D externflags -D instrCnt ABRO-C.c sc.c -o $@

%.out2: %.c sc.c sc.h Makefile
	gcc $(CCFLAGS) $*.c sc.c -o $*
	./$* > $@

%.out: % %.c
	./$* > $@


# Preprocessing
%-expanded.c: %.c sc.h Makefile
	gcc $(CCFLAGS) -D externflags -E $*.c > $@

%-expanded-flags.c: %.c sc.h Makefile
	gcc $(CCFLAGS) -E $*.c > $@


# Assembler
%.s: %.c sc.h Makefile
	gcc $(CCFLAGS) -D externflags -S $*.c

%-unopt.s: %.c sc.h Makefile
	gcc $(CCFLAGS) -o $@ -D externflags -S $*.c

%.o: %.c sc.h Makefile
	gcc $(CCFLAGS) -D externflags -c -o $*.o $*.c

%.asm: %.o
#	objdump -d $*.o > $@
	otool -tv $*.o > $@

%-linked.asm: %.c sc.c sc.h Makefile
	gcc $(CCFLAGS) -D externflags $*.c sc.c -o $*
#	objdump -d $*.exe > $@
	otool -tv $* > $@


# Default rule for executable
%: %.c sc.c sc.h Makefile
	gcc $(CCFLAGS) $*.c sc.c -o $*

# Statistics
all%.wc: $(progs:=%)
	echo $@, CCFLAGS = $(CCFLAGS) > $@
	date >> $@
	wc $^ >> $@

wc: all.c.wc all-expanded.c.wc all.o.wc
	cat $^


# Example to match either or expression:
# grep -E 'trace|\"' sc.h
%.stats:
	echo "Line␣count␣of␣$*:"
	wc $*
	echo "Comment␣line␣count␣of␣$*:"
	grep -c "^␣*//" $*
	echo "Empty␣line␣count␣of␣$*:"
	grep -c "^$$" $*
	echo "Trace␣related␣line␣count␣of␣$*,␣discouting␣multi-line␣trace␣commands␣(9),␣counting␣again␣comments␣(4):"
	grep -c "trace" $*


# Documentation, Publishing
doxy:
	doxygen doxygen.conf

doxyv: doxy
	open doc/html/sc_8h.html

sc.tar.gz: $(downloads)
	tar -cf sc.tar $(downloads)
	gzip -f sc.tar
	ls -l $@

BIBLIO_REMOTEPATH=biblio@rtsys.informatik.uni-kiel.de:/home/biblio/public_html/downloads/sc

%p: %
	scp $^ $(BIBLIO_REMOTEPATH)/

pub: doxy sc.tar.gz
	scp -r sc.h sc.c selectCid.c sc.tar.gz LICENSE.txt README.txt doc/html $(BIBLIO_REMOTEPATH)/


# Cleaning up
clean:
	-rm *~ *-expanded.c *.stackdump *.o

realclean: clean
	-rm *.exe *.out
```