SyncCharts in C— A Proposal for Light-Weight, Deterministic Concurrency

[Extended Abstract] *

Reinhard von Hanxleden
Department of Computer Science, Christian Albrechts Universität zu Kiel
Olshausenstr. 40, 24098 Kiel, Germany
rvh@informatik.uni-kiel.de

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 paper presents SyncCharts in C (SC), an approach to seamlessly and efficiently embed SyncCharts constructs into a conventional imperative programming language. SC offers deterministic, light-weight concurrency and preemption via a simulation of multi-threading, inspired by reactive processing.

A reference implementation of SC, based on C macros, is available as open source code. SC can be used in a number of scenarios: 1) as a regular programming language, requiring just a C compiler; 2) as an intermediate target language for synthesizing graphical SyncChart models into executable code, in a traceable manner; 3) as instruction set architecture for programming precision timed (PRET) or reactive architectures, abstracting functionality from physical timing; or 4) as a virtual machine instruction set, with a very dense encoding.

Categories and Subject Descriptors

D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features*

General Terms

Design, Languages

Keywords

SyncCharts, Statecharts, Esterel, synchronous programming, multi-threading, reactive processing, model-based design

*A full version of this paper is available on-line as technical report [27]

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. How to express this adequately in an imperative language such as C or Java is a notoriously difficult problem. Threads and associated synchronization primitives are widely used, but entail significant overhead and easily lead to non-determinism and deadlock [13].

This paper presents *SyncCharts in C* (SC), which is a light-weight approach to express reactive control flow 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.

Statecharts, SyncCharts. Statecharts [10] extend classical finite state machines by concurrency and hierarchy/preemption. These extensions allow to keep descriptions compact and avoid the classical state explosion problem. There exist numerous variants of Statecharts, including *e. g.* Simulink/Stateflow and UML Statecharts. SyncCharts [2] are variant of Statecharts that builds on the synchronous model of computation, hence it employs a formal semantics and ensures determinism.

A typical design flow employing Statecharts may start with a graphical modeling tool that 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 semantics in a C program, so it is possible for a human programmer to express Statechart behavior as a C program [24, 29]. This does not offer the visual appeal of graphical Statecharts, but has other advantages: 1) no need for a modeling tool, 2) high portability, and 3) seamless integration with a fully featured, widely used programming language, including the type system, expression handling, control flow, access to low-level I/O, pre-processors, etc.

Even if one assumes a design flow that starts at a graphical modeling tool that supports SyncCharts, it is of interest how SyncChart 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: 1) it simplifies the development of the code synthesizer of the modeling tool; 2) 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 3) it simplifies code certification for safety-critical embedded systems.

Contributions. The main contribution of this paper is a concise, light-weight embedding of deterministic concurrency primitives into the C language. The main idea of SC is to emulate multi-threading, and is inspired by reactive processing [28]. As we do not have direct access to the program counter at the C language level, we keep track of individual threads via state labels, implemented as usual C program labels. These labels can also be viewed as continuations [4], or coroutine re-entry points [11]. Precedence among transitions, respecting strong/weak abortions and hierarchy, and the adherence to signal dependencies is achieved by checking transition triggers in the proper order as well as assigning appropriate thread ids and priorities. SC is no panacea, it is still the responsibility of the SC programmer to express the order in which threads are executed according to the logic of the application; however, SC does offer a deterministic mechanism to do so via thread ids/priorities.

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

Outline. The next section relates SC to other work, followed in Sec. 3 by an example, PCO, that illustrates how SC programmers can implement reactive control. Sec. 4 discusses signal-based communication, illustrated with the grcbal3 example, and briefly elaborates on thread scheduling. We present experimental results in Sec. 5, before concluding in Sec. 6. The full version [27] of this paper presents the programming model in more detail including guidelines on finding proper schedules, covers all of the SC operators, discusses specific issues such as schizophrenia, suspension and delayed signals, documents the current implementation, and presents a range of further examples.

2. RELATED WORK

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++ [24]. As in UML Statecharts, this approach does not provide deterministic concurrency. Wagner *et al.* describe how to implement FSMs in C [29], but these are flat automata without any concurrency.

[1] http://www.informatik.uni-kiel.de/rtsys/sc/

There have been several proposals to extend traditional programming languages by synchronous constructs. Reactive C [8] 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 [5] 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) [12] 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) [22]. 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.

Lusteral, presented by Mendler and Pouzet [16], 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.

As SC expresses synchronous, control-oriented concurrency by means of an—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 [18, 7]. 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 [7], which proposes a high-level ISA that captures the Esterel semantics as closely as possible; see also Sec. 5. Another interesting approach is the dynamic list code generation [7], 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.

As discussed in Sec. 3, SC is also related to the programming model proposed for the Precision Timed Architecture

(PRET) proposed by Edwards and Lee [15]. Another related programming model is SHIM [26], 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.

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 [30], 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 [9]. Such a translation was proposed by André [3] together with the initial definition of SyncCharts and their semantics. This transformation, with additional unpublished optimizations, is implemented in Esterel Studio. The resulting Esterel program can then be translated into software or hardware [18]. As discussed in Sec. 5, 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.

One approach to synthesize SyncCharts into a textual program that does preserve the original structure is to generate code directly for a reactive processor [28], as done by the state machine to KEP compiler (smakc!) [25]. 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 [14], which implements synchronous concurrency via multi-threading. This multi-threading approach, which is also realized for example in the StarPro processor [31], 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: 1) SC provides a TRANS operator that implements an arbitrary state transition; 2) SC does not provide Esterel's exception handling via traps; 3) SC does not rely on special watcher units to implement aborts. A motivation for the KEP's watcher units was to avoid Checkabort instructions [23, 22], 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 Chkaborts, by giving parent threads the power to abort their descendants with the TRANS operator.

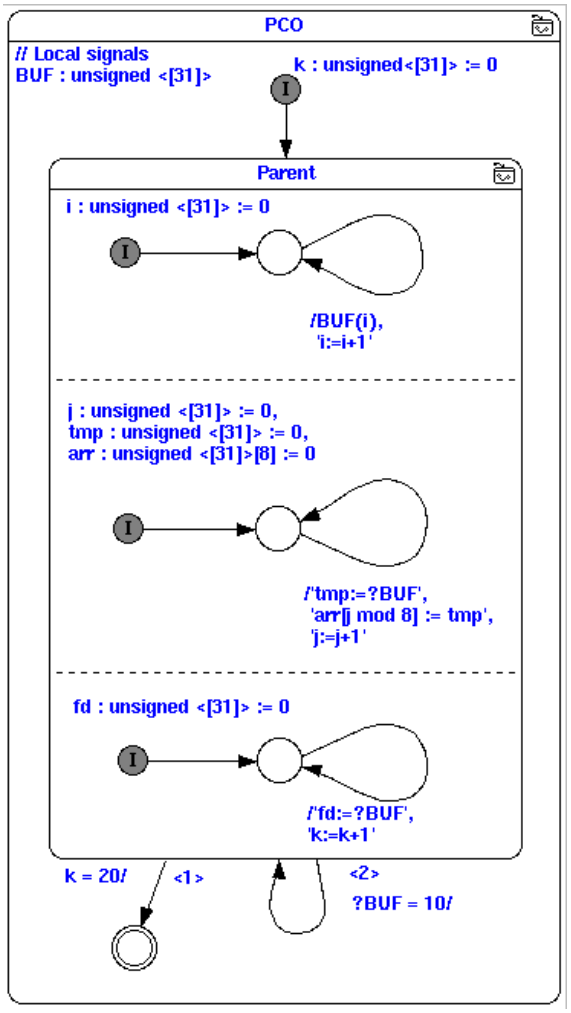
3. REACTIVE CONTROL IN SC—PCO

This section covers 1) the general structure of SC programs, 2) how SC macros are embedded in regular C code, 3) the concept of deterministic, label-based simulated multi-threading, and 4) deterministic preemptions. We will illustrate these points with PCO, shown in Fig. 1, a simple producer-consumer example with an observer. It is inspired by Lickly *et al.* [15], but to illustrate not only concurrency but also preemption, it has been augmented with a parent thread that restarts production/consumption once the buffer has the value 10 (transition at bottom right), and which terminates after 20 iterations.

The SyncCharts version (Fig. 1a) shows a Parent macro state, 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 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 [15] offers a DEAD instruction which guarantees a (minimal) delay before a thread proceeds. Fig. 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 synchroniza-



(a) SyncChart, designed 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 [15])

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

(c) Complete SC program

Figure 1: The PCO (Producer-Consumer-Observer) example.

Mnemonic, Operands	Notes
TICKSTART*(*isInitial*)	Start (initial) tick.
TICKEND	Finalize tick, return 1 iff there is still an enabled thread.
PAUSE*(l)	Deactivate current thread for this tick, continue next tick at address label l.
TRANS(l)	Abort descendant threads, jump to l.
SUSPEND*(l)	Suspend (pause) thread and its descendants, continue at l.
TERM*	Terminate current thread.
PAR(p, l, id)	Create a thread with an initial priority p, a start address l, and an id id.
PARE*(p, l, ids_{desc})	Specify priority p, continuation address l and descendant threads (for TRANS and JOIN).
JOIN*(l_{then}, l_{else})	If descendant threads have terminated normally, jump to l_{then}; else pause, proceed to l_{else}.
PRIO*(p, l)	Set current thread priority to p, continue at l.
PPAUSE*(p, l)	Shorthand for PRIO(p, l'); l': PAUSE(l) (saves one call to dispatcher)
JPPAUSE*(p, l_{then}, l_{else})	Shorthand for JOIN(l_{then}, l); l: PPAUSE(p, l_{else}) (saves another call to dispatcher)

Table 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.

tion 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. 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 1. The remaining SC operators are presented in Sec. 4, Table 2.

The first SC macro used in PCO, TICKSTART, performs some book keeping, depending on whether this is the initial tick or not. This is followed by a sequence of PAR/PARE macros, which fork off the children of the current thread. The current thread, started when entering tick, is the Main thread. The forked threads are Prod, Cons, and Obs. Each PAR gives a thread a priority (here all 0), a starting label, and an id. PARE specifies a priority for the current thread (again 0), a continuation label (Parent), 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. Threads are declared with the idtype enumeration type (line 4). The starting point of each thread is declared with an ordinary C label, named after the thread. This is just a convention; from a C perspective, these labels and the thread names have different name spaces and are different objects—one is a memory address, the other is an enumeration type index.

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

The dispatcher, called by PAUSE, selects a thread for resumption. In PCO the dispatcher selects from the *active* threads, which still have work to do in the current thread, the one with the highest *thread id*. The dispatcher may also consider dynamic priorities, see Sec. 4, 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 TRANS operators (lines 49 and 51). This operator transfers control to the argument label, and also aborts Main's child threads. Finally, TERM terminates the current thread (Main), and TICKEND does last book keeping before leaving tick again.

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

Modularization. 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/communication trivial, but limits its 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.

4. SIGNALS IN SC—GRCBAL3

This section covers 1) more elaborate thread scheduling via the use of dynamic thread priorities, 2) a synthesis path from Esterel to SC, 3) how SC macros alone suffice to write a tick function, and 4) signal handling. Signals are used by SyncCharts for broadcast communication among threads, and SC provides a set of operators for them; note, however, that an SC programmer may also use regular C variables, as was demonstrated in PCO. 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 [7]. Hence the name of the benchmark: GRC is the Graph Code intermediate representation of the CEC, BAL is the Bytcode Assembly Language of a VM targeted by the CEC. The grcbal3 Esterel code has been transformed into a SyncChart using KIEL [19]. Fig. 2a shows the Esterel version, on the right, with the generated SyncChart, in the midst of an animated simulation—the initial tick has just been executed, with no inputs present.

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

The SyncChart synthesized by KIEL is equivalent to the Esterel version. However, as SyncCharts do not provide traps, they have to be emulated with weak abortions. This translation is always possible, and in grcbal3 this can be done in a straightforward fashion, via a weak abort triggered by a fresh signal T_. The transition that implements this is shown in the lower right of the SyncChart, which leads to a final state (double circle).

Fig. 2b shows the tick function of the SC version of grcbal3. In addition to the SC concurrency operators already introduced in Sec. 3 and Table 1, grcbal3 makes use of SC *signal operators*. An overview of these and some other, sequential control operators is given in Table 2.

To better understand this example's operation, consider also the execution trace shown in Fig. 2c. All SC macros (apart from TICKSTART and TICKEND) log their operation to stdout if instructed to do so via a preprocessor directive. The trace illustrates the operation of grcbal3 in case input signal A is present. The first line shows the input signals (A) and the enabled threads (initially none) as bit vector, in octal notation with leading 0. TICKSTART, PAR, and PARE are as explained for the PCO example (Sec. 3). One difference, however, is that threads A1, A2 and A3, which correspond to the three concurrent substates embedded in the macro state 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 only the thread id mattered to the dispatcher.

After Main has forked its children, PARE calls the dispatcher, see line 5 in the program, line 7 in the trace. This starts A1 (thread id 2), as it has the highest priority. A1 determines A as present and emits signal B. The PRIO directive lowers A1's priority to 2, specifies L0 as continuation, and calls the dispatcher. Now A2 (id 3) is started, as it has the same priority as A1, but a higher thread id. A2 determines B as present and hence emits C. Then the TERM operator *terminates* C, meaning that it is deactivated (does not resume in the current tick) and disabled (will not be resumed in the next tick). Therefore TERM calls the dispatcher, without specifying a continuation label. The set of remaining enabled threads is encoded in a bit vector, see line 13 of the trace. The vector octal 027, binary 10111, has bits 0 (rightmost bit, indicating thread TickEnd), 1 (Main), 2 (A1) and 4 (A3) set.

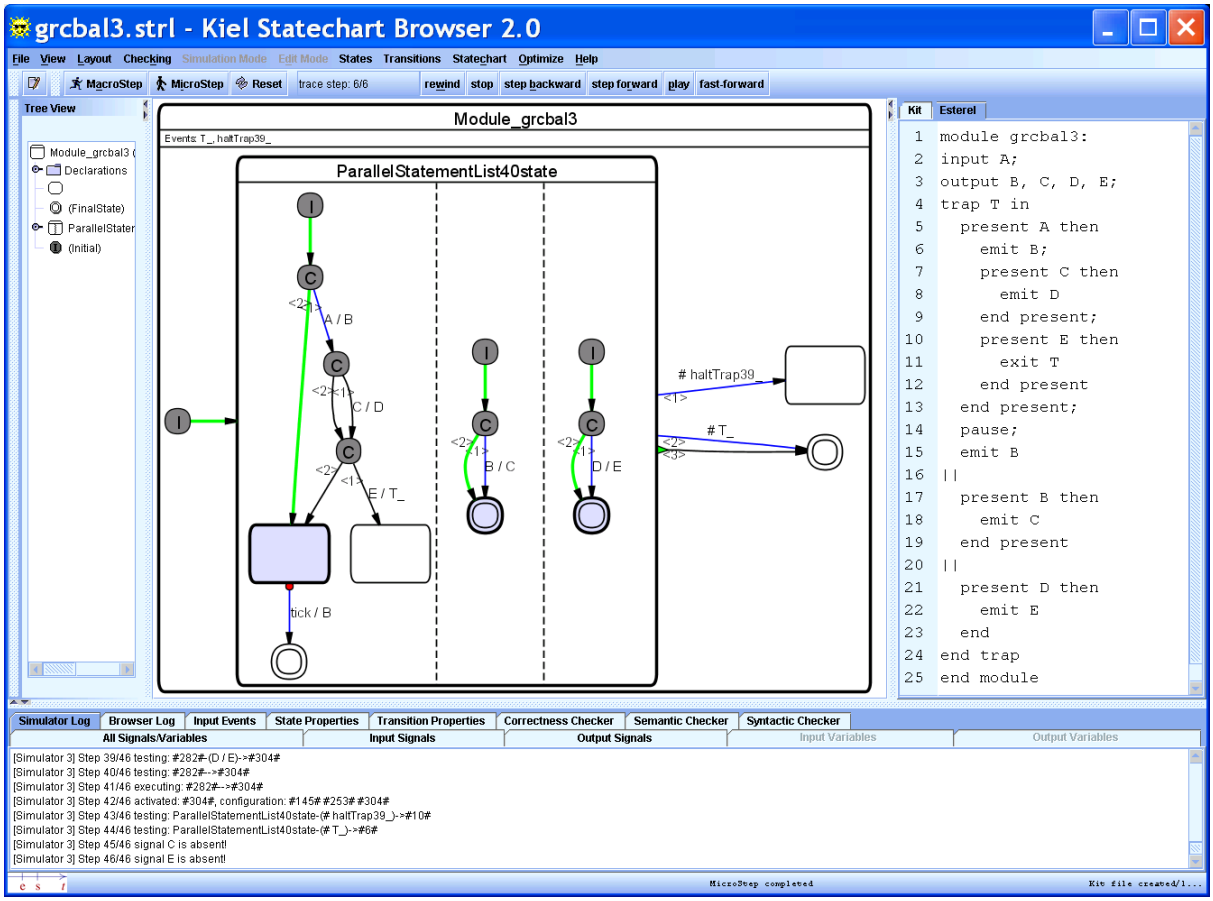
In this fashion, control is passed back and forth between Main's children until they have all have completed their tick, and the Main thread, running at priority 0, resumes; see line 24 of the trace. It determines that T_ is present, which corresponds in the original Esterel program to a thrown exception (exit T), hence the program has to terminate. This is done by first aborting Main'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 27), a total of 24 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").

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 macro states, 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.

Thread Scheduling. The grcbal3 example is, admittedly, fairly intricate, as it has also been designed to illustrate the scheduling challenges that Esterel poses to a compiler. For



(a) Screen shot of KIEL[20], as it synthesizes a SyncChart from the original Esterel code [7]

```
1          TICKSTART(isInit);    // id 1
2          PAR(3, A1, ids[A1]);  // id 2
3          PAR(2, A2, ids[A2]);  // id 3
4          PAR(1, A3, ids[A3]);  // id 4
5          PARE(0, AMain, id2b(A1) | id2b(
               A2) | id2b(A3));
6    A1:   PRESENT(A, A1B);
7          EMIT(B);
8          PRIO(2, L0);
9    L0:   PRESENT(C, A1A);
10         EMIT(D);
11   A1A:  PRIO(1, L1);
12   L1:   PRESENT(E, A1B);
13         EMIT(T_);
14         GOTO(A1C);
15   A1B:  PAUSE(L2);
16   L2:   EMIT(B);
17   A1C:  TERM;
18   A2:   PRESENT(B, A2A);
19         EMIT(C);
20   A2A:  TERM;
21   A3:   PRESENT(D, A3A);
22         EMIT(E);
23   A3A:  TERM;
24   AMain: PRESENT(T_, AJoin);
25         TRANS(B);
26   AJoin: JOIN(B, AMain);
27   B:    TERM;
28         TICKEND;
```

(b) SC tick function

```
1  ==== TICK 0 STARTS, inputs = 01, enabled = 00
2  ==== Inputs (id): A (0)
3  ==== Enabled (id, state): <none>
4  PAR:    Main  (id 1, prio 0, <null> -> <null>) forks A1 (2) with prio 3, startlabel A1
5  PAR:    Main  (id 1, prio 0, <null> -> <null>) forks A2 (3) with prio 2, startlabel A2
6  PAR:    Main  (id 1, prio 0, <null> -> <null>) forks A3 (4) with prio 1, startlabel A3
7  PARE:   Main  (id 1, prio 0, <null> -> AMain) has descendants 034
8  PRESENT: A1   (id 2, prio 3, <null> -> A1) determines A (0) as present
9  EMIT:   A1    (id 2, prio 3, <null> -> A1) emits B (1)
10 PRIO:   A1    (id 2, prio 3, A1 -> L0) set to priority 2
11 PRESENT: A2   (id 3, prio 2, <null> -> A2) determines B (1) as present
12 EMIT:   A2    (id 3, prio 2, <null> -> A2) emits C (2)
13 TERM:   A2    (id 3, prio 2, <null> -> A2) terminates, enabled = 027
14 PRESENT: A1   (id 2, prio 2, A1 -> L0) determines C (2) as present
15 EMIT:   A1    (id 2, prio 2, A1 -> L0) emits D (3)
16 PRIO:   A1    (id 2, prio 2, L0 -> L1) set to priority 1
17 PRESENT: A3   (id 4, prio 1, <null> -> A3) determines D (3) as present
18 EMIT:   A3    (id 4, prio 1, <null> -> A3) emits E (4)
19 TERM:   A3    (id 4, prio 1, <null> -> A3) terminates, enabled = 07
20 PRESENT: A1   (id 2, prio 1, L0 -> L1) determines E (4) as present
21 EMIT:   A1    (id 2, prio 1, L0 -> L1) emits T_ (5)
22 GOTO:   A1    (id 2, prio 1, L0 -> L1) transfer to A1C
23 TERM:   A1    (id 2, prio 1, L0 -> L1) terminates, enabled = 03
24 PRESENT: Main (id 1, prio 0, <null> -> AMain) determines T_ (5) as present
25 TRANS:  Main  (id 1, prio 0, <null> -> AMain) disables 034, transfers to B, enabled = 03
26 TERM:   Main  (id 1, prio 0, <null> -> AMain) terminates, enabled = 01
27 ==== TICK 0 terminates after 24 instructions.
28 ==== Enabled (id, state): TickEnd (0, TickEndLabel)
29 ==== Resulting signals (id): A (0), B (1), C (2), D (3), E (4), T_ (5), Outputs OK.
```

(c) Example trace; l_0 -> l_1 indicate the previous/current continuation of a thread

Figure 2: The grcbal3 example.

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

Table 2: SC signal operators (pure signals, valued signals, and accesses to the previous tick) and SC sequential control operators (jumps and exit actions).

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; the full paper describes a systematic approach to do this. There are several possible alternatives to unchecked manual priority/id assignment: 1) one might relegate thread id and priority assignment to a separate analysis pass, using for example the assignment algorithm of the KEP Esterel compiler [14] (feasible, but it would require a separate compilation step); 2) 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, 3) 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: 1) programs that require dynamic scheduling of threads, which entails run-time alterations of thread priorities (via PRIO); 2) programs that require just static scheduling, which can be handled with thread id assignment; and 3) programs that do not impose scheduling constraints at all. From the 10 benchmarks reported on in the next section, only grcbal3 and exits belong to the first category.

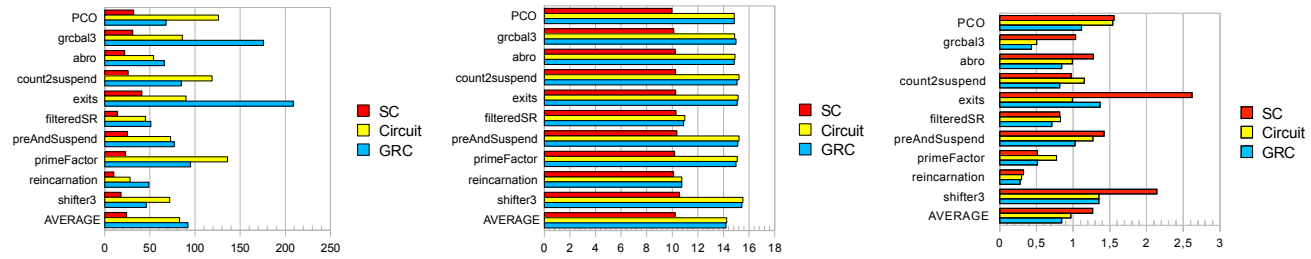
5. EXPERIMENTAL RESULTS

The main intention of designing 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. 2a); in SC, it takes 28 lines (Fig. 2b). This indicates a comparable level of conciseness, which is remarkable in that the SC operators are embedded in the imperative, sequential programming model of C. Another interesting point of comparison is the BAL VM instruction set, as it has been designed specifically to encode Esterel programs in as little memory as possible [7]. To encode grcbal3, BAL uses 74 instructions, of complexity comparable to the SC operators. The SC version makes do with 28 instructions, and these are also arguably easier to relate to an Esterel program or a SyncChart than the BAL assembler. This makes SC an attractive alternative candidate for a VM instruction set.

As another point of reference, Fig. 3a compares the size of the SC tick functions for a number of benchmarks 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 than the synthesized C code. Fig. 3b compares sizes of the executable on an x86 architecture, compiled with gcc. Here the difference is less remarkable, but SC is still ahead.

The development of SC has not been motivated primarily by performance concerns, but still it is interesting to see how it compares. On the negative side, SC basically just interprets a SyncChart, it cannot perform any global optimizations or partial evaluations at compile time, as do for example the EsterelStudio synthesis tools. On the positive side, SC code has no scalability problems, neither in terms of code size (unlike 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. 3), and 2) the dispatcher is fast; on the x86, in case of static thread schedules and a maximal degree of concurrency smaller than the native word width, SC does thread selection with a single bsr (Bit Scan Reverse) assembler instruction on the active thread vector. (See the report [27] for a further discussion of dispatching complexity.) 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. The benchmarks considered here, mostly taken from André [2], mainly serve to demonstrate the proper coverage of the full SyncChart



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

(b) Size of executable, in Kbytes

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

Figure 3: Experimental results.

language. Still, they provide some indication of SC's performance characteristics. Figure 3c compares the run times of the tick functions, on an Intel Xeon 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 3 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, SC programs are dominated by regular C operations, not the SC operators.

6. 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 SyncChart language; the full report details non-trivial issues not addressed here, like the handling of exit actions or the proper interaction of pre, suspend and local signals [27]. 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 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 unproblematic; see again the full version of the paper [27].

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 1.3.3, 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 609 lines long, of which 171 lines are comments, 62 lines are related to tracing, and 132 lines are empty. This leaves 243 lines of C code that explain what the 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. For example, the current thread continuation handling is based on computed gotos, as provided by the gcc; an alternative, which would probably be slightly less efficient, but does not rely on this, would be to use switch/case logic instead. Also, as already mentioned, SC seems a viable candidate as an intermediate language when synthesizing visual SyncCharts into software, especially if traceability is required; as discussed in Sec. 4, this would also alleviate the user from the burden of scheduling possibly intricate thread interdependencies. SC might also be used as input language for PRET architectures, specifying thread orderings without relying on specific timing characteristics. It would also be an interesting exercise to add something like a DEAD timing primitive [15] to SC. Unlike PRET architectures, traditional architectures probably cannot do this cycle-accurate; however, using something like the x86 rtsc instruction or the Timing Constraint Violation Exceptions of the Open Macro Library[2] [6], 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. 1c, line 15). A related issue is the WCRT analysis for SC, which could build on earlier work [17, 21]. Another question not addressed at all so far is how the SC

[2] http://oml.sourceforge.net

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.

7. REFERENCES

[1] J. Ali and J. Tanaka. Converting Statecharts into Java code. In *Proceedings of the Fourth World Conference on Integrated Design and Process Technology (IDPT '99)*, Dallas, Texas, June 2000. Society for Design and Process Science (SDPS).

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

[3] C. André. Computing SyncCharts reactions. In *SLAP 2003: Synchronous Languages, Applications and Programming, A Satellite Workshop of ECRST 2003*, volume 88, pages 3 – 19, 2004.

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

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

[6] T. Cucinotta, D. Faggioli, and A. Evangelista. Exception-based management of timing constraints violations for soft real-time applications. In *Proceedings of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'09)*, Dublin, Ireland, June 2009.

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

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

[9] 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.

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

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

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

[25] 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.

[26] 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.

[27] R. von Hanxleden. SyncCharts in C. Technical Report 0910, Christian-Albrechts-Universität Kiel, Department of Computer Science, May 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/report-0910.pdf.

[28] R. von Hanxleden, X. Li, P. Roop, Z. Salcic, and L. H. Yoong. Reactive processing for reactive systems. *ERCIM News*, 66:28–29, Oct. 2006.

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

[30] 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.

[31] 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.