INSTITUT FÜR INFORMATIK

Compiling SyncCharts to Synchronous C

Claus Traulsen, Torsten Amende,
Reinhard von Hanxleden

Bericht Nr. 1006
Juli 2010

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

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

Compiling SyncCharts to Synchronous C

Claus Traulsen, Torsten Amende, Reinhard von Hanxleden

Bericht Nr. 1006
Juli 2010

e-mail:
ctr@informatik.uni-kiel.de,
tam@informatik.uni-kiel.de,
rvh@informatik.uni-kiel.de

Technical Report

SyncCharts are a synchronous Statechart variant to model reactive systems with a precise and deterministic semantics. The simulation and software synthesis for SyncCharts usually involve the compilation into Esterel, which is then further compiled into C code. This can produce efficient code, but has two principal drawbacks: 1) the arbitrary control flow that can be expressed with SyncChart transitions cannot be mapped directly to Esterel, and 2) it is very difficult to map the resulting C code back to the original SyncChart.

This paper presents an alternative software synthesis approach for SyncCharts that compiles SyncCharts directly into Synchronous C (SC). The compilation preserves the structure of the original SyncChart, which is advantageous for validation and possibly certification. The compilation assigns thread priorities according to the data dependencies. It optimizes both the number of used threads as well as the maximal used priorities, which corresponds to fast SC code with little memory requirements.

Contents

List of Figures

1 Introduction

Reactive systems are systems that continuously interact with their environment. The execution of these systems is determined by their internal state and external stimuli. As a reaction, new stimuli and/or a new internal state are generated. The modeling of these systems requires both concurrency and preemption in a deterministic fashion.

In particular for safety-critical systems, it is also important that the behavior cannot only be understood by the programmer, but as well by experts in the application area, without further knowledge in computer science. In order to achieve this, graphical notations such as Statecharts were developed. The Statecharts formalism extends the classical formalism of finite-state machines and state transition diagrams by incorporating the notions of hierarchy, orthogonality, compound events, and a broadcast mechanism for communication between concurrent components. Statecharts provide an effective graphical notation, not only for the specification and design of reactive systems, but also for the simulation of the modeled system behavior. Since the original Statecharts proposal [7], numerous dialects of Statecharts have been developed [3] and Statecharts have also been incorporated into the UML. Today, Statecharts are supported by several commercial tools, *e. g.*, Matlab/Simulink/Stateflow from The MathWorks, IBM Rational Statemate [7] or IBM Rational Rose.

In this paper, we are particularly interested in the SyncCharts [2] dialect of Statecharts, also known as Safe State Machines (SSMs), which has a formal semantics grounded in the synchronous model of computation and is hence particularly suited for safety-critical applications [4]. A commercial tool that uses SyncCharts as design entry language is Synfora's Esterel Studio (E-Studio), which provides a range of synthesis options for generating HDL (Verilog, VHDL) or C/SystemC. These synthesis paths involve the Esterel language [6] as intermediate language, for which there is a range of compilation approaches available [9]. However, the translation from SyncCharts to Esterel is rather intricate, as the arbitrary control flow that can be expressed with state transitions (SyncCharts) must be mapped to the structured control flow operations typical for imperative programs, such as loops and conditionals (Esterel). Likewise, the translation from Esterel to C can produce efficient code, but yet again we must perform a non-trivial mapping, in this case from concurrent, preemptive code (Esterel) to sequential code (C). Ultimately, the C code resulting from a SyncChart is very difficult to map back to the original SyncChart, which makes it hard to validate and to possibly certify the design. For reference, there is a DO-178B certified code generator for Esterel Technologies' SCADE tool, which has a comparatively straightforward synthesis path from dataflow equations to code.

In this paper, we propose a compilation from SyncCharts into SC Synchronous C (SC) code. SC [15, 14], also known as SyncCharts in C, has been recently proposed

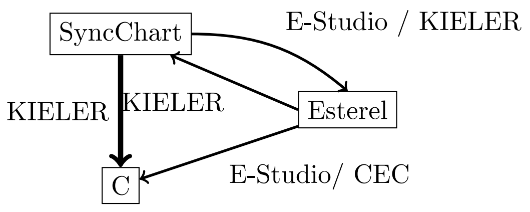


Figure 1.1: Different compilation schemes from SyncCharts to C

as a means to embed synchronous reactive control flow directly in C. SC provides a set of control and communication operators, which provide priority-based, deterministic multi-threading and a signal-based broadcasting mechanism. These SC operators can themselves be expressed in C, the freely available SC reference implementation[1] implements all operators as C macros. Hence, an SC program can be directly compiled into software with a standard C compiler. Unlike the traditional compilation from SyncCharts to C via Esterel, our compilation preserves the connection between the generated instructions and the original states and transitions is preserved. This makes the code more readable, facilitates validation, and allows a direct back annotation. As the experimental results indicate, the size and execution speed of the resulting executables are competitive with existing approaches; in terms of code compactness, it even appears superior. Compared to manually writing SC code, using SyncCharts as design entry point for SC has the advantage that the SC generator presented here automatically determines thread priorities and, if necessary, context switches that respect all data dependencies, which may not be trivial for complex applications.

A short overview over the compilation paths is given in Fig. 1.1. Our compiler is part of the KIELER (Kiel Integrated Environment for Layout Eclipse Rich-Client) tool[2]. We compare our compilation to the compilation via E-Studio and to the original compilation scheme from SyncCharts to Esterel, which is also implemented in KIELER. The Esterel code can than be further compiled using the standard Esterel compiler from Inria[3] or the Columbia Esterel compiler Columbia Esterel Compiler (CEC)[4].

In the next section we consider related work, followed by a closer look at SyncCharts and Synchronous C in Sec. 3. In Sec. 4 we describe the compilation process and give experimental results in Sec. 5. We conclude in Sec. 6.

[1]`www.informatik.uni-kiel.de/rtsys/sc/`
[2]`www.informatik.uni-kiel.de/rtsys/kieler`
[3]`www-sop.inria.fr/esterel.org/files/`
[4]`www1.cs.columbia.edu/~sedwards/cec/`

2 Related Work

While Statecharts are an appealing language to describe reactive behaviors, the generation of efficient code is not trivial. Three different methods of compiling Statecharts can be distinguished: 1) compilation into an object oriented language using the state pattern [1], 2) dynamic simulation [16], and 3) flattening into finite state machines. Since flattening can suffer from state explosion, often a combination of flattening and dynamic simulation is used. Executing SyncCharts with SC, proposed here, can be classified as a simulation based approach, where SC defines a simulator. However, the compilation from SyncCharts to SC is independent of the implementation of SC, which could also be implemented by a virtual machine or a hardware extension to a general purpose processor.

A translation from SyncCharts to Esterel was proposed by André [2] together with the initial definition of SyncCharts and their semantics. This transformation, with additional unpublished optimizations, is implemented in E-Studio.

Our work is related to the extension of Esterel with GOTO by Tardieu and Edwards [13]. Since they extend the language, they have to consider all possible usages of GOTO, *e.g.*, jumping from one thread into another. Our approach could be directly used to generate efficient extended Esterel (including GOTO) from SyncCharts, since the structure of the SyncChart will always generate valid GOTOs.

Considering a non-synchronous language like plain C as alternative synthesis target, the direct generation of C code from SyncCharts might also be more efficient than the path via Esterel. This approach is taken by the SCC compiler[1]. However, it appears that the compiler generates circuit code in the spirit of Esterel and does not directly reflect the structure of the source SyncChart.[2] Our compilation is also closely related to the compilation from SyncCharts to KEP assembler [12]. However, the abortion scheme for the KEP is different, resulting in a different scheme for the priority assignment. Furthermore, for the compilation to Kiel Esterel Processor (KEP) assembler, we need to extract all complex expressions for triggers, while we can use the full power of C to express expressions within SC.

[1] `julien.boucaron.free.fr/wordpress/?page_id=6`

[2] Unfortunately, there does not appear to be documentation available on this compiler—we have contacted the author about further information.

3 SyncCharts and Synchronous C

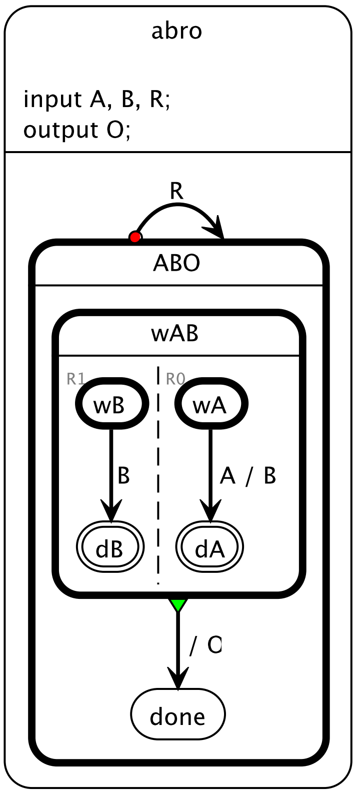
SyncCharts are a Statechart dialect with a synchronous semantics that strictly conforms to the Esterel semantics. Synchronous C is an extension of C that has been developed with the SyncCharts/Esterel model of computation in mind, while staying in the realm of sequential C. Both are explained in more detail in the following sections.

3.1 SyncCharts

A procedural definition of SyncCharts is given by André [2]. The basic object in SyncCharts is a *reactive cell*, which is a state with its outgoing transitions. Reactive cells are combined to *state-transition graphs*, which we will also refer to as *state regions*. A *macro-state* consists of one or more state-transition graphs. Fig. 3.1a shows the ABRO SyncChart, the "hello world" of synchronous programs that demonstrated deterministic concurrency, preemption, and signal-based communication. To illustrate signal dependencies, this version has been slightly modified from the original ABRO, including an emission of B on the transition from wA to dA.

States can also have *internal actions*: on entry, on exit and on inside. SyncCharts inherit the concept of signals and valued signals from Esterel. Hence a transition trigger can consist of an event, which tests for presence and absence of values, and a conditional, which may compare numerical values. Signals are by default *absent*, a signal is *present* if it is either an input signal that is set to present by the environment or if it is emitted in the current tick. The signal status and value of the previous tick can be accessed via the pre keyword. For valued signals a *combine function* can be given which combines multiple emissions of the signal within one tick, which provides determinism even if a signal is emitted multiple times with different values within one tick. Since the semantics assumes that all emits are executed at the same time and hence does not specify an order of execution of the signal emissions, the combine function must be commutative and associative. If no combine function is given, it is assumed that only one emit can occur at the same time. In contrast to the presence status of signal, the value of a signal is preserved over tick boundaries; a signal keeps its value until another emit occurs in a later tick.

A characteristic of SyncCharts are the different forms of preemption, expressed by different state transition types. *Weak* and *strong abortion* transitions as well as *suspension* can be applied to macrostates. A macrostate can either be left by an abortion, which has an explicit trigger, or by a *normal termination*, which is taken if the macrostate enters a terminal state. Analogously to Esterel, all transitions can either be immediate



(a) Original SyncChart

```
int  tick () {
  TICKSTART(4);
ABO:
  FORK(wAB, 1);
  FORKE(ABO_main);
wAB:
  FORK(wA, 3);
  FORK(wB, 2);
  FORKE(wAB_main);
wA:
  PAUSE;
  if  (PRESENT(sig_A)) {
    EMIT(sig_B);
    TERM;
  }
  GOTO(wA);
wB:
  PAUSE;
  if  (PRESENT(sig_B))
    TERM;
  GOTO(wB);
wAB_main:
  JOINELSE(wAB_Join);
  EMIT(sig_O);
  GOTO(ABO_done);
wAB_Join:
  GOTO(wAB_main);
ABO_done:
  HALT;
ABO_main:
  PAUSE;
  if  (PRESENT(sig_R)) {
    ABORT;
    GOTO(ABO);
  }
  GOTO(ABO_main);

  TICKEND;
}
```

(b) Synthesized SC tick function

Figure 3.1: The modified ABRO example: wait concurrently for the inputs A and B, if both have occurred, emit output O; the behavior is reset by the input R.

or delayed, where a delayed transitions is only taken if the source state was already active at the start of a tick. In contrast, immediate transitions may be taken as soon as the state becomes active; this enables the activation and deactivation of a state multiple times within one tick. Delayed transitions can also be *count delayed*, *i. e.*, the trigger must have been evaluated to true for a specific number of times, before the transition is enabled. When a state has more than one outgoing transition, a unique *transition number* is assigned to each of them, where lower numbers indicate higher priority. (It would be natural to refer to these numbers as "transition priorities," but this could be confused with the priorities used by SC, hence we will stick to "transition numbers"). Weak abortions must have higher transition numbers than strong abortions, and if a normal termination exists, it always has the highest transition number.

In the example in Fig. 3.1a, initially the states wA and wB are active, in regions R0 and R1. When the input A occurs, the transition to dA is taken and B is emitted. Since B is present, the transition from wB to dB is taken. Now both regions are in a final state and the normal termination to done is taken, emitting the output O. Note that all this happens within one tick.

3.2 Synchronous C (SC)

Synchronous C (SC) [15, 14] is an extension of C to allow concurrency and preemption in a deterministic way, adhering the synchronous hypothesis. SC was designed to express the behavior described by a SyncChart directly in C in a concise and readable fashion. Figs. 3.2 and 3.3 provide a short overview of the SC instructions that are generated by our compiler.

SC extends C by a light-weight, priority based thread model. Each thread has a program counter and an id, which also serves as its priority. (The original SC proposal distinguished thread ids and priorities, but for simplification of the programming model, these have been unified since.) The scheduler/dispatcher will always choose the active thread with the highest priority. A thread can either be *enabled*, *i. e.* it should be executed in the current tick, or *disabled*. Furthermore, an enabled thread can be either *inactive*, *i. e.*, it was already executed in the current tick, or *active* if it still needs to be executed in the current tick. A scheduler selects the next thread to execute based on the current priorities. The scheduler is called by any instruction that either changes the priority or the status of a thread. The status of all enabled threads is set to active at the start of a tick. PAUSE and HALT are tick delimiting instructions, which stop the execution of the executed thread for the current tick by setting its status to inactive.

The FORK statement initializes a new thread. A sequence of FORK's is concluded with a FORKE, which activates all threads that were initialized by the preceding FORK statements and sets the continuation of the current thread. As the priorities of the threads are also used as unique identifier, a new thread might never be initialized with the same priority as an already active thread. The priority of the currently executed thread can be changed by the PRIO instruction. Again the set priority may not be used

Operands	**Notes**
TICKSTART$^*(p)$	Start (initial) tick, assign priority p to the Main thread.
TICKEND	Finalize tick, return 1 iff there is still an enabled thread.
PAUSE^{*+}	Deactivate current thread for this tick.
HALT^{*+}	Shorthand for l: PAUSE; GOTO(l).
TERM*	Terminate current thread.
ABORT	Abort descendant threads.
SUSPEND$^*(cond)$	Suspend (pause) thread and its descendants if *cond* holds.
SUSPENDG$^*(l)$	Suspend (pause) thread and its descendants, continue at l.
FORK(l, p)	Create a thread with start address l and priority p.
FORKE$^*(l)$	Finalize FORK, resume at l.
JOINELSE$^{*+}(l_{else})$	If descendant threads have terminated normally, proceed; else pause, jump to l_{else}.
JOIN^{*+}	Shorthand for l_{else}: JOINELSE(l_{else}).
PRIO$^{*+}(p)$	Set current thread priority to p.
PPAUSE$^{*+}(p)$	Shorthand for PRIO(p); PAUSE.
GOTO(l)	Jump to label l.

Figure 3.2: 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).

by any active thread.

In addition to providing operators for reactive control flow, SC extends C by a signal-based communication mechanism that allows to broadcast information within and across threads. SC signals behave like signals in SyncCharts or Esterel. They are per default *absent* and only *present* if they are emitted in the current tick. SC itself does not distinguish between input, output and local signals. A signal can be emitted by the EMIT statement or reset by the SIGNAL statement, used to reinitialize a local signal. Valued signals can be emitted using special variants of EMIT that specify the type of the emitted value and, optionally, a combine function. For example, EMITINT(S) emits a valued signal S with an integer value, without a combine function.

Note that as a consequence of embedding SC in (sequential) C, SC trivially provides a deterministic order of execution of all statements within one tick. This differs from SyncCharts or Esterel, where all statements within a tick are considered to occur simultaneously. This determinism of SC provides some additional flexibility for signal handling compared to classical synchronous languages. For example, if multiple EMITINT(S) are executed within a tick, the result in SC is still well-defined, namely the last emitted value. Alternatively, one might decide to still forbid these cases, and to either statically (conservatively) analyze whether they might occur or to detect these situations at run time. More fundamentally, one might also relax the synchrony hypothesis with respect

Operands	**Notes**
SIGNAL(S)	Initialize a local signal S.
EMIT(S)	Emit signal S.
SUSTAIN^{*+}(S)	Shorthand for l: EMIT(S); PAUSE; GOTO(l).
PRESENT(S)	True iff S is present.
AWAIT^{*+}(S)	Shorthand for l_{else}: PAUSE; PRESENT(s, l_{else}).
AWAITI^{*+}(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.
VAL(S)	Retrieve value of signal S.
PRESENTPRE(S, l_{else})	True iff S was present in previous tick. If S is a signal local to thread t, consider last preceding tick in which t was active, *i. e.*, not suspended.
VALPRE(S)	Retrieve value of signal S at previous tick.

Figure 3.3: SC signal operators (pure signals, valued signals, and accesses to the previous tick). See Fig. 3.2 on the asterisk* and plus$^+$annotations.

to signal presence, and might allow to determine a signal to be absent and then to emit it within the same tick. This would mean to allow absence and presence of a signal within the same tick, but the result would still be deterministic. However, we here still take the conservative stance that signals should be emitted before they are tested. The SC implementation monitors adherence to this rule at run time, and raises an error if a signal that has been determined absent is emitted within the same tick. (This error checking can be disabled if one wants to use the more liberal interpretation outlined earlier.)

With both SyncCharts and Esterel one might write self-contradicting, *i. e.* non-constructive, programs, for which no execution schedule can be found. Such programs should be rejected by a compiler for SyncCharts or Esterel. In SC, this situation is somewhat different; on the one hand, the execution schedule is always implied by the sequential nature of C, but on the other hand it is easily possible to write programs that violate the synchrony hypothesis with respect to signal statuses, as discussed above. To help the programmer, run-time-checks are activated per default to check, as mentioned above, that a signal which status was already read is never emitted later within a tick. There are other consistency checks as well, *e. g.*, checking that the priority of a thread is never set to the same value as an already existing priority. For our compilation from SyncCharts to SC, the compiler requires that the SyncChart can be statically scheduled, and hence rejects non-constructive programs; also, the priority assignment algorithm ensures that priorities are unique at run time. One may still perform the aforementioned consistency checks at run time, but they should never be triggered by the synthesized code.

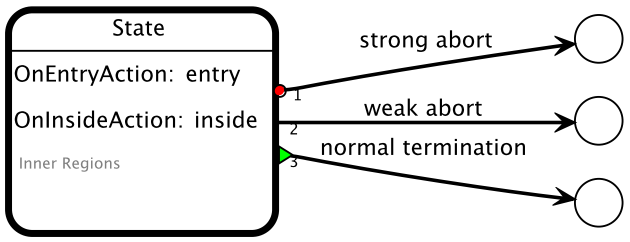
4 Compilation

Since the instructions of SC were developed to express SyncCharts naturally in C, the compilation of SyncCharts into SC is much simpler than the general compilation of SyncCharts into plain C. The main difficulty is to schedule the different threads according to their data-dependencies, which is done by computing a priority for each transition. Apart from this global priority assignment, the compilation is purely structural.

Each state is transferred into a PAUSE or HALT statement, according to its outgoing transitions. A new thread is generated for each parallel region in the original SyncChart, as well as for each macro state, where the outgoing transitions are checked conceptually in parallel to the content of the state.

The general translation of a state with its outgoing transitions can be seen in Fig. 4.1. Note that it might be necessary to duplicate immediate transitions; however, in most cases this can be avoided, as discussed in the next section. While inner actions are executed in the current thread, regions that are contained in the state are executed in separate threads.

The translation of a transition consists of the following sequence. 1) Check the transition's trigger predicate; if this evaluates to true: 2) execute the effects specified in the transition label, such as a signal emission, followed by 3) an ABORT if the source state is a macro state and hence has descendant threads that need to be terminated, 4) set the priority to the priority of the target state and 5) a GOTO to jump to the target state.



(a) General form of a state. Strong aborts have a red circle at the source of the transition arrow, normal terminations are indicated by a green triangle. The transition number is indicated by the tail label.

```
Label:
      [immediate Strong Transitions]
      [onEntry Actions]
      [immediate Weak Transitions]
Label_intern :
      PAUSE
      [ all Strong Transitions ]
      [ onInside Actions]
      PRIO ([weak])
      [ all Weak Transitions]
      [Normal Termination]
      PRIO ([strong])
      GOTO(Label_intern)
```

(b) Code Template

Figure 4.1: General translation of a state

Beside being immediate and delayed, transitions in SyncCharts can also have count delays: a transition can wait for the n-th tick in which its trigger is evaluated to true. While this cannot be directly expressed in SC, it can be implemented using an additional variable to count the occurrence. When the state is entered, the variable is initialized with the delay of the transitions. Whenever the trigger of the transition evaluates to true, the variable is decremented and the transition is taken as soon as zero is reached.

A common problem of the code generation from SyncCharts to Esterel and to KEP assembler is to make sure that a normal termination is not taken in case an weak abortion is also triggered in the same tick. This is not a problem for the SC code generation, where a normal termination is handled exactly like a weak abortion with the special trigger JOIN.

The translation of a region consists of the code for all states, in any order. The only requirement is that the initial state must be the first state. The ordering of the other states is still relevant for the performance, since a transition from one state to the following state can be performed without an additional GOTO instruction, therefore we order the states by a depth first search.

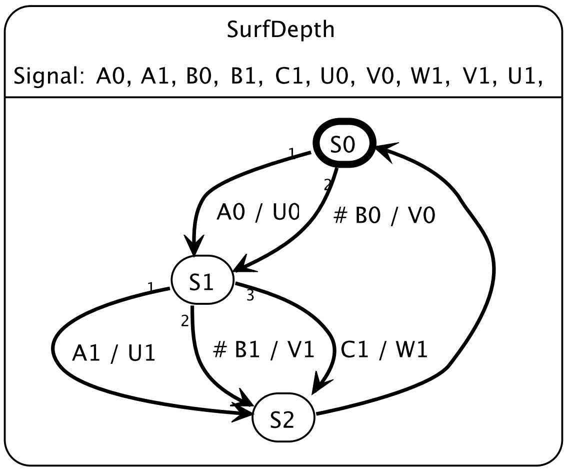
4.1 Surface and depth

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. 4.2.

One also distinguishes the *surface* of a statement, which is the code that can be executed in the tick in which the state is entered and which includes only the immediate transitions and on-entry actions, and the *depth* of a statement, which is the code that can be executed if the state is already active in the start of the tick; this includes immediate as well as delayed transitions and inner actions.

In Esterel, the distinction between surface and depth was introduced to solve schizophrenia problems due to multiple executions of the same code within one tick [11]. However, this is not an issue for the code generation to SC, since it allows the reinitialization of local signals within a tick.

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. 4.2a. The transitions from S0 to S1 can be ordered such that the priorities implied by transition numbers 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.



(a) SyncChart

```
 1  S0: if (PRESENT(sig_B0)) {
 2          EMIT(sig_V0);
 3          GOTO(S1);
 4      }
 5   S0_intern :
 6    PAUSE;
 7    if (PRESENT(sig_A0)) {
 8      EMIT(sig_U0);
 9      GOTO(S1);
10    }
11    if (PRESENT(sig_B0)) {
12      EMIT(sig_V0);
13      GOTO(S1);
14    }
15    GOTO(S0_intern);
```

(b) Naïve tick function for S0

```
 1  S0: if (PRESENT(sig_B0)) {
 2          EMIT(sig_V0);
 3          GOTO(S1);
 4      }
 5   S0_intern :
 6    PAUSE;
 7    if (PRESENT(sig_A0)) {
 8      EMIT(sig_U0);
 9      GOTO(S1);
10    }
11    GOTO(S0);
12  S1:
13    if (PRESENT(sig_B1)) {
14      EMIT(sig_V1);
15      GOTO(S2);
16    }
17   S1_intern :
18    PAUSE;
19    if (PRESENT(sig_A1)) {
20      EMIT(sig_U1);
21      GOTO(S2);
22    }
23    if (PRESENT(sig_B1)) {
24      EMIT(sig_V1);
25      GOTO(S2);
26    }
27    if (PRESENT(sig_C1)) {
28      EMIT(sig_W1);
29      GOTO(S2);
30    }
31    GOTO(S1_intern);
```

(c) Optimized tick function for S0 and S1

Figure 4.2: The SurfDepth example with non-trivial surface and depth behavior.

Fig. 4.2b shows the generated code when strictly applying the template from Fig. 4.1. Each immediate transition occurs twice in the code. However, we can avoiding the duplication of the immediate transitions by jumping back to the surface code, *i. e.*, replacing lines 11–15 in Fig. 4.2b by $\mathsf{GOTO(S0)}_{11}$, see also Fig. 4.2c. In general, this can be done if a state has no entry action and all immediate transitions have lower priority (*i. e.*, higher transition numbers) than delayed transitions.

4.2 Scheduling

The scheduling assigns priorities to code blocks, in order to meet all dependencies between threads. There are three different sources for constraints that must be met by the priorities.

1) Hierarchy: For a macrostate, all outgoing strong aborts need to be checked before the content of the state is executed, and all weak abortions must be checked after the execution. Therefore we need at least two priorities for each macrostate with both strong and weak outgoing transitions.

2) Transition Order: The outgoing transitions must be handled according to the priorities indicated by their transition numbers. Following the rules laid down by SyncCharts, we assume that strong abortions have higher priorities than weak abortions and that a normal termination has lowest priority if one exists. However, there are no requirements with respect to immediate and delayed transitions, hence we might need to duplicate the code for immediate transitions. In most cases, the transition order can be handled by ordering the code that checks the triggers accordingly. However, since a thread can only lower its priority within a tick, the dependencies of transitions with low priority affect also the execution priority of all transitions with higher priority for the same state.

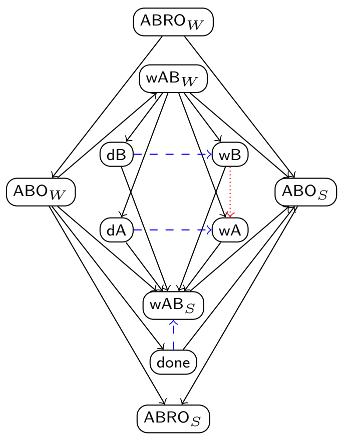
3) Data dependencies: We also need the priorities to assure that all possible writers of a signal are executed before its possible readers. Note that not only the directly outgoing transitions of a state must be considered, but *all* transitions that are immediately reachable, either through immediate abortions or normal terminations. Data dependencies are also the only source of non-constructiveness [5]; for a non-constructive SyncChart no possible schedule can be found, hence the compiler will reject the SyncChart. Due to the data dependencies, it might be necessary to change the priority between two transitions.

Remember that within a tick a thread can only lower its own priority, but for the next tick it can also raise its priority. Hence, we also must consider control flow dependencies, which require that a state must have a priority equal to or lower than all states from which it can be reached via immediate transitions. An important design choice for the compiler is the granularity of the priority computation. The most detailed granularity would be to compute a priority for each SC instruction; however, this is not necessary, since the priority will never change inside an action, for example. It is sufficient to compute a priority for each state and transition.

Consider the modified ABRO in Fig. 3.1. This example includes all discussed dependencies. For example state wB is hierarchically dependent of state ABO and state wAB, and its transition is signal dependent of state wA. Furthermore we have control flow dependencies, *e. g.*, between wA and wB. With this information we can create a *dependency tree*, which includes all dependencies of a given SyncChart. For each hierarchical state we compute two priorities for checking strong and weak transitions.

Fig. 4.3a represents the resulting dependency tree of the modified ABRO example. The priorities for strong and weak transitions are indicated with subscripts S and W. Black continuous lines represent hierarchical dependencies, blue dashed lines are control flow dependencies, and red dotted lines are signal dependencies. Because the dependency tree is acyclic, we can perform a topological sort to order the resulting threads in the SC code.

The resulting order need not be unique, any topological sort yields an adequate priority



(a) Dependency tree of the modified ABRO example

Thread	Prio.	Opt.
ABRO	10	4
wAB	4	1
wB	8	3
wA	6	2

(b) Thread priorities resulting from topological order, and after compaction (optimization) done by the compiler

Figure 4.3: Computation of the priorities from the dependency tree

assignment. For our example a possible order could be the following:

$$\mathsf{ABRO}_W \longmapsto \mathsf{ABO}_W \longmapsto \mathsf{done} \longmapsto \mathsf{wAB}_W \longmapsto \mathsf{dB} \longmapsto \mathsf{wB} \\ \longmapsto \mathsf{dA} \longmapsto \mathsf{wA} \longmapsto \mathsf{wAB}_S \longmapsto \mathsf{ABO}_S \longmapsto \mathsf{ABRO}_S$$

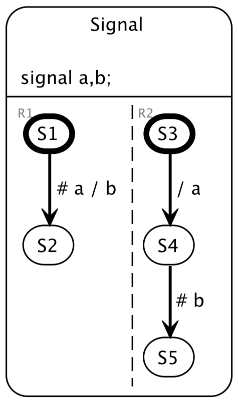
As we can see, all inner states of the hierarchical state wAB are between wAB$_W$ and wAB$_S$. Thus the thread priorities of the two threads inside this states are greater than wAB when weak abortions are checked and smaller than wAB when strong abortions are checked. Furthermore the thread waiting for B gets a lower priority than the thread waiting for A because of the topological sorting above. Hence the producer thread for signal B executes before its consumer.

For the generated SC-code, shown in Fig. 3.1b, this results in the priority assignment in Fig. 4.3b, directly transcribed from the topological order above. Note that to simplify the reading, we polished the generated code by introducing and renaming labels.

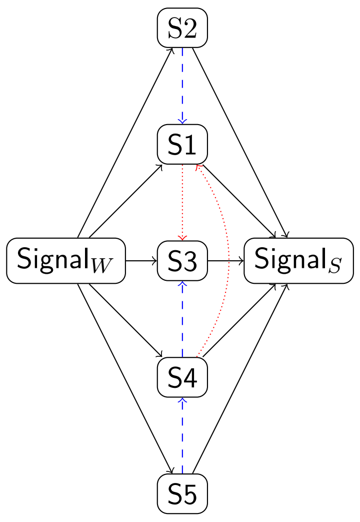
This static assignment of thread priorities is insufficient if signal dependencies reverse between two threads during one tick. Fig. 4.4a shows a simple SyncChart where the threads communicate back and forth within one tick, hence we need to change the priority after the initialization. From its dependency tree (Fig. 4.4b), a topological sort yields the following order:

$$\mathsf{Signal}_W \mapsto \mathsf{S5} \mapsto \mathsf{S4} \mapsto \mathsf{S2} \mapsto \mathsf{S1} \mapsto \mathsf{S3} \mapsto \mathsf{Signal}_S$$

The resulting thread priorities are 1 for the Signal (main) thread and 4 respectively 5 for the two concurrent threads corresponding to regions R1 and R2. Without changing the priority of the threads, the R2 thread would start with S3, emit a, recognize signal b as absent and finish in state S4 for this tick. After this, thread R1 would start with S1, recognize a as present, emit b and finish in state S2. This execution violates the SyncCharts semantics, as b was tested (and determined absent) before it was emitted.



(a) Instantaneous communication between threads



(b) Generated dependency tree

```
 1  int tick () {
 2    TICKSTART(1);
 3
 4  L_Signal :
 5    FORK(L_Signal_R2_S3,5);
 6    FORK(L_Signal_R1_S1,4);
 7    FORKE(L_Signal_main);
 8
 9  L_Signal_R2_S3:
10    PAUSE;
11    EMIT(sig_a);
12    PRIO(2);
13    GOTO(L_Signal_R2_S4);
14    PRIO(5);
15
16  L_Signal_R2_S4:
17    if (PRESENT(sig_b)) {
18      GOTO(L_Signal_R2_S5);
19    }
```

(c) Generated SC code

Figure 4.4: The Signal example illustrating alternating signal dependencies between regions R1 and R2

To alleviate this, we need to change the execution context from the R2 thread to the R1 thread during this tick, which is done by lowering the priority of R2 below that of R1. Line 12 in the code snippet in Fig. 4.4c contains the PRIO operation executed by the R2 thread to lower its own thread priority and to defer control to the concurrent R1 thread. The priority of this assignment is directly derived from the topological order. To lower the own priority it is necessary that the taken transition has at least one signal-dependent state in a concurrent thread. To initiate the context switch, the R2 thread must lower its priority from S3's priority (5) to S4's priority, which the compiler sets equal to S5's priority (2) after recognizing that no further intervening context switch is necessary. At the end of generated code for a state with a priority change we restore the former priority. In the example, this is the PRIO(5) statement in line 14, which turns out to be dead code and could be eliminated in a subsequent optimization.

4.3 Optimizations

As mentioned before, our compilation scheme is structural. However, we apply some optimizations in order to generated faster and more compact code. There are three objectives for optimization.

Dynamic priority changes: To implement abortions, threads must keep track of their descendants, via their thread ids. Hence changing the priority of a thread, done by the PRIO operator, requires a fair bit of internal book keeping (see the SC implementation for details). As a consequence, PRIO is the most expensive instruction in SC, both in terms of execution time and (if inlined) code size. To reduce the number of generated PRIO instructions, we try to avoid changing priorities for states without signal dependencies.

Threads: Each thread is associated with some state, such as its list of descendant threads; furthermore, the run time of some operations (notably PRIO, and possibly also the dispatcher) depends on the maximal thread id. Hence the performance and resource requirements of the generated SC code depends on the maximal number of threads. We need one thread for each region, both for macro-states and for parallel regions. We perform some simple optimizations, *e. g.*, states without outgoing transitions do not require an extra thread. It is common in SyncCharts to have hierarchical states without outgoing transitions, in this case no extra thread is needed. Further threads could be saved by a static analysis which merges concurrent regions into one thread, *e. g.*, if they are executed in lock step.

Code size: We want to express the behavior with as few instructions as possible. The code size is already positively effected by the optimizations mentioned before, in particular by the removing of PRIO instructions. Furthermore, in many realistic examples surface and depth can be folded to avoid code duplication. We generate one code block whenever the priority of all immediate transitions is higher than the priority of all delayed transitions.

4.4 Restrictions

Our compilation cannot handle all SyncCharts. Some requirements, like enforcing that strong abortions have higher priorities than weak abortions, are no problem in practice and also applied by other modeling tools like Esterel Studio. Valued signals are currently restricted to simple types (integers, floats, booleans); more complex types, such as strings, should be straightforward to implement. Similarly, suspension is currently not implemented as it seems to be rarely used in practice, but should be not difficult to add; SC already provides a SUSPEND operator for this.

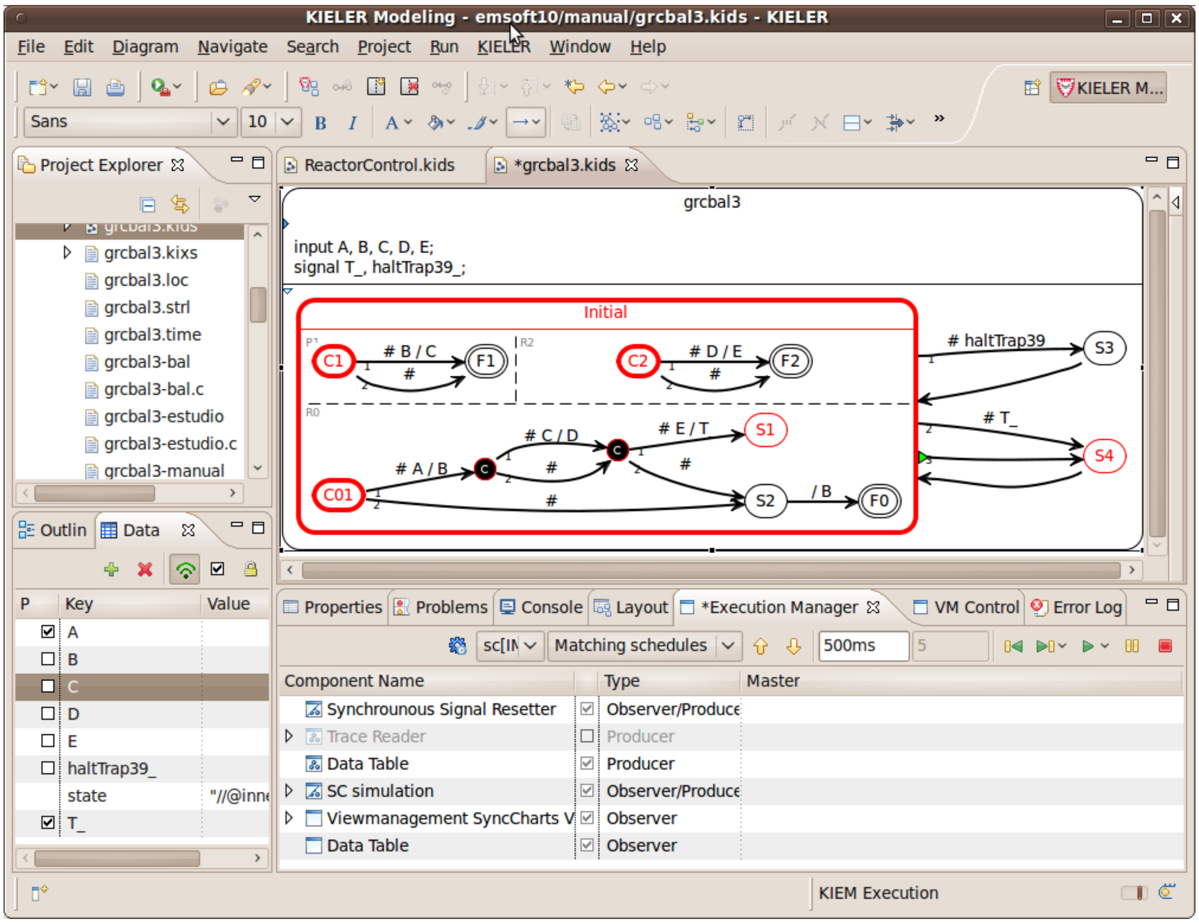
Less trivial to add would be on exit actions, which break the normal flow of control. An on exit action is executed whenever the state it is associated with is left, regardless how. In particular the state could be left by a strong abort on any outer hierarchy level, but the on exit action still needs to be executed. Note that on exit actions are not handled by the original transformation for SyncCharts to Esterel. For the current implementation of on exit actions in Esterel Studio, Esterel itself was augmented by a finalize statement, which has the same semantics as on exit. While this simplifies the translation to Esterel, it makes the translation from Esterel into C code much harder. We propose to handle on exit action by a pre-processing step, where each SyncChart that contains on exit action is transformed into an equivalent SyncChart without on exit actions. This can be done by adding the on exit action to each transition that may force the state to leave. In practice, the on exit action most often contains a single call to an host function in order to release some reserved resource. We need some additional bookkeeping to check whether the state was really active when the transitions is taken, and that the on exit action is only executed once. To support on exit actions, SC already provides the ISAT(id, l) operator that tests whether thread id is at label l.

There are also some corner cases where the semantics of SyncCharts are not clearly defined, *e. g.*, whether on entry actions are executed when a state is left immediate by a strong abort. There are two views for actions, either they are really part of the state, or they are just an abbreviation for adding the action to all incoming transitions. In such cases, we refer to E-Studio as a reference implementation for SyncCharts and mimic its behavior. Hence, in the above case, the action is not executed.

5 Experimental Results

The compiler from SyncCharts to SC is part of the KIELER tool. When the user generates SC code for a SyncChart A, three files are generated: 1) A.c contains the definition of the tick function, 2) A.h contains the declaration of the functions from A.c, as well as the declaration of the signal types, and 3) A_data.c contains a simple main function to call the tick function. This function can be used for performance evaluation.

The compiler is also used for simulation, as shown in Fig. 5.1. In the simulation mode, the code is augmented with additional information for the active state and the active transitions, and a file is generated with wrapper functions that communicate with the



P	Key	Value
☑	A	
☐	B	
☐	C	
☐	D	
☐	E	
☐	haltTrap39_	
	state	"//@inne
☑	T_	

sc[IN Matching schedules 500ms 5

Component Name		Type	Master
Synchrounous Signal Resetter	☑	Observer/Produce	
Trace Reader	☐	Producer	
Data Table	☑	Producer	
SC simulation	☑	Observer/Produce	
Viewmanagement SyncCharts V	☑	Observer	
Data Table	☑	Observer	

Figure 5.1: Simulation of the grcbal3 example within KIELER, using the generated SC code.

execution engine using JSON[1] strings. The generated tick function and the wrapper are then compiled in the background and the generated executable is started, communicating with KIELER via standard input/output.

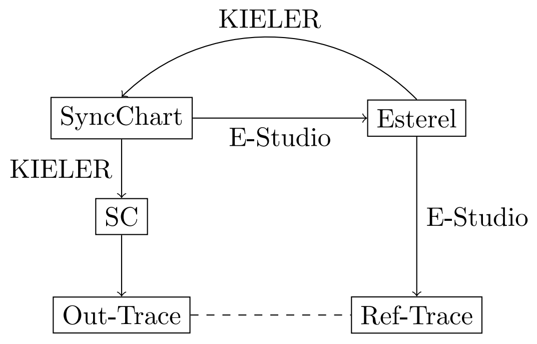


Figure 5.2: Validation of the compiler

For the validation of the compiler we compare the generated code to the behavior of SyncCharts in E-Studio. Fig. 5 gives an overview of the validation process. We have to two start points: 1) A set of SyncCharts modeled in KIELER and in Esterel Studio. 2) Another set of SyncCharts synthesized from Esterel programs [10]. In both cases we use Esterel Studio to generate reference traces. Inputs of these traces are executed by the generated SC code and the outputs are compared. The outputs match for all given traces. Note that the generated code contains some additional instructions for the exchange with KIELER.

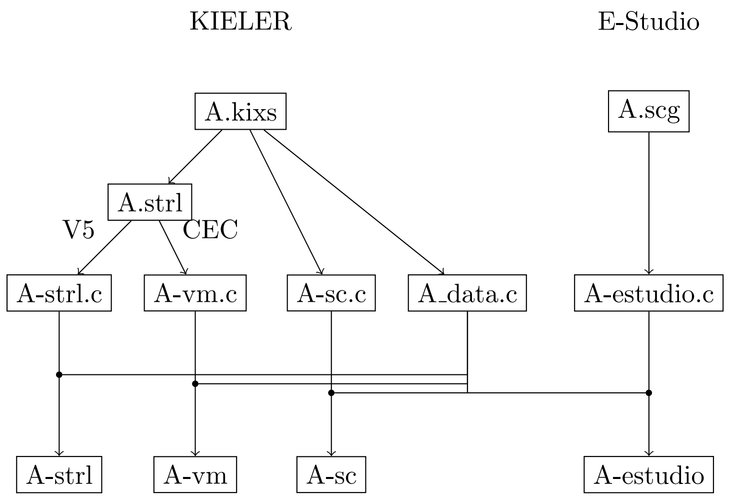


Figure 5.3: Alternative code synthesis paths

For the performance evaluation, we compile the SC code without the code for the

[1] www.json.org

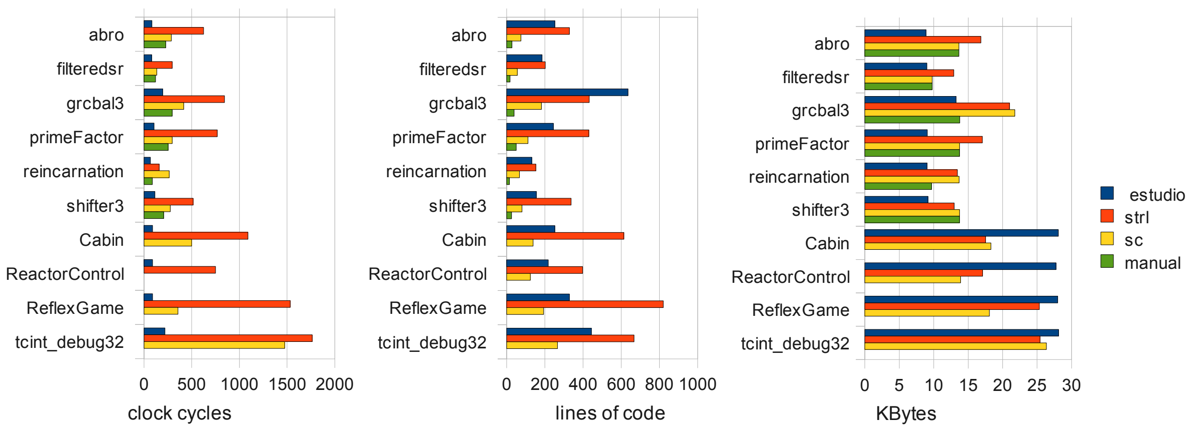
communication, implementing the reactive interface defined for Esterel [9]. Hence we can use the same wrapper to run the Esterel code and the generated SC code. First we compare our generated code to the hand-written code from the SC report [15]. These are small examples suited to show the full range of SC language features. While we do not really expect an improvement over the handwritten code, it shows how close the generated code can get to the optimal code. We also applied the compiler to medium size SyncCharts where manually writing the SC code is not practical, in particular because the data-dependencies are more complex and hence the priority assignment is hard to get correct for a human programmer. The Cabin and ReflexGame charts are taken from the Esterel Studio examples, while the ReactorControl and tcint_debug32 examples were Esterel programs which were transformed into SyncCharts.

We also compiled the SyncCharts to Esterel and then further to C code, using two different approaches: 1) The standard esterel V5 compiler, where the generated code simulates an equivalent circuit. 2) The CEC to compile code for a virtual machine. This was not always applicable, because the CEC cannot handle Esterel's pre operator and the virtual machine is limited to 255 signals, which is easily reached, since the compilation to Esterel introduces one signal for each state. Both compilations depend on the Esterel code that is generated for the SyncCharts in KIELER. We also modeled the examples in Esterel Studio to compare to its optimized compilation to Esterel, and compiled the generated Esterel code within Esterel-Studio to C using the fast graph code [9] approach. All code is linked to the same data file which contains the main function to set the inputs and measure the execution time. For the example SyncCharts from the SC introduction, we also compared the results to the manually written code.

All programs are executed for 1 million ticks with random, but identical, input traces, and the execution time for each tick in clock cycles is measured. As a consistency check, we also compare how often each output signal is emitted. Beside the execution time, also the code size itself is a relevant factor for embedded software. Therefore we measured both the size of the generated and linked object code as well as the lines of code (loc) of the generated c code. For SC, the loc count gives an estimation of the code size if it where executed on a virtual machine or a processor with an adapted instruction set. We also measure the size of the generated executable after linking. All compilations of C code are performed using the gcc with default optimizations (O2). All experiments were performed on an Intel Xeon running with 3 GHz and 6 MB cache.

The results of the comparison are shown in Fig. 5.4. In general, the performance of the generated code (sc) is almost as good as for the manually written code (manual). However, the generated code is significantly larger. This is primarily due to code duplications that could be avoided, *i. e.*, by folding surface and depth.

Compared to the compilation via Esterel, the compilation via SC is both faster and smaller than using the Esterel V5 compiler (strl). However, it must be noticed that the generated Esterel code is not optimized. Such optimizations are performed by the Esterel Studio compiler, which outperforms our compiler. In particular, the compiler performs a static analysis of the possible behavior of a SyncChart. Still, the size of the generated code is similar for most examples. For all examples that contain numerical



(a) Reaction time (b) Size of generated C code (c) Size of executable

Figure 5.4: Comparison between code generated by Esterel Studio via Esterel (estudio), code generated by KIELER via Esterel and the Esterel V5 compiler (strl), SC code generated by KIELER using the approach presented here (sc), and manually written SC code (manual).

computations, the code generated by Esterel Studio must be linked to an additional library, which significantly increases the size of the executables.

6 Conclusion and Outlook

We presented a compilation from SyncCharts into Synchronous C (SC). Since SC can directly express the control flow of SyncCharts, the code generation allows easy traceability between the source code and the generated code. The compiler is implemented in the KIELER tool and also used for simulation of SyncCharts within the tool.

The first results for small and medium-sized examples show that the generated code has almost the same performance as code generated via Esterel. This is encouraging and even somewhat surprising, as the underlying execution approach is basically a simulation, following directly the SyncChart structure, rather than the extensive compile-time analysis and optimization performed by the Esterel-based synthesis approach. Compared to the existing compilation approaches for SyncCharts, we consider the synthesis path via SC as rather straightforward and light-weight. This also refers to the implementation effort, which was only a couple of man months both for the SyncCharts-to-SC compiler presented here and for SC itself.

In principle this compilation approach should scale well to larger applications, both in terms of performance and code size, and thus be superior *e. g.* to the circuit-based or automata-based compilation approaches; however, this yet has to be validated. In particular, it remains to be seen (and is likely to be application specific) how frequent and expensive dynamic priority changes are in larger applications.

There is still room for improvement of the priority assignments, since the compiler might assign a priority to a state but in a later stage realize that the priority is not needed at all, in this case the priority is not used. Other optimizations would also be worth considering, such as dead-code elimination.

Due to the close relationship between SyncCharts and SC, all computations within the compiler are performed directly on the SyncChart and a derived dependency tree. However, an intermediate structure which more closely resembles the control flow of SC, in the spirit of the concurrent KEP assembler graph [8], might allow a more fine grained optimization of the generated code.

We are currently also investigating the multi-core distribution of SC code, which might improve performance for larger applications; the question is how much thread synchronization requirements will offset performance improvements from parallelization.

Bibliography

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

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

[3] 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 *LNCS*, pages 128–148. Springer-Verlag, 1994.

[4] 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.

[5] G. Berry. *The Constructive Semantics of Pure Esterel.* Draft Book, 1999. `ftp://ftp-sop.inria.fr/esterel/pub/papers/constructiveness3.ps`.

[6] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[7] 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.

[8] X. Li and R. von Hanxleden. Multi-threaded reactive programming—the Kiel Esterel Processor. *IEEE Transactions on Computers*, accepted 2010.

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

[10] 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.

[11] K. Schneider and M. Wenz. A new method for compiling schizophrenic synchronous programs. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 49–58, Atlanta, Georgia, USA, Nov. 2001. ACM.

[12] 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.

[13] O. Tardieu and S. A. Edwards. Instanteneous transitions in esterel. In *Proceedings of Model Driven High-Level Programming of Embedded Systems (SLA++P'07)*, Braga, Portugal, Mar. 2007.

[14] R. von Hanxleden. SyncCharts in C. Technical Report 0910, Christian-Albrechts-Universität Kiel, Department of Computer Science, May 2009.

[15] R. von Hanxleden. SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. In *Proceedings of the International Conference on Embedded Software (EMSOFT'09)*, Grenoble, France, Oct. 2009.

[16] 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.