Compiling SyncCharts to Synchronous C

Claus Traulsen and Torsten Amende and Reinhard von Hanxleden
Department of Computer Science
Christian-Albrechts-Universität zu Kiel
{ctr, tam, rvh}@informatik.uni-kiel.de

***Abstract*—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, which hampers traceability.**

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. We present a static thread-scheduling scheme that reflects data dependencies and optimizes both the number of used threads as well as the maximal used priorities. This results in SC code with competitive speed and little memory requirements.

I. 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 also by application experts. This motivated the development of graphical notations such as Statecharts.

We are here 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 [3]. A commercial tool that uses SyncCharts as design entry language is Synopsis' Esterel Studio (E-Studio), which encompasses a range of synthesis options for generating Verilog, VHDL, or C/SystemC. E-Studio's synthesis paths involve Esterel as intermediate language, for which there is a range of compilation approaches available [6]. 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 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.

Synchronous C (SC) [9], also known as SyncCharts in C, has been recently proposed 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.

Contributions, challenges and benefits. We present a compilation scheme to synthesize SyncCharts into SC code. The challenge, as in any compilation from reactive programs into a sequential language (such as C), is to make sure that the synthesized code reflects the reactive control flow according to the SyncChart semantics. The specific issues are concurrency, preemptions (weak and strong), and the surface/depth distinction. This demands a judicious ordering of transitions, the synthesis of thread priorities, and possibly dynamic priority changes (PRIO statements). 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. 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 efficient synthesis approaches based on Esterel. Compared to manually writing C 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.

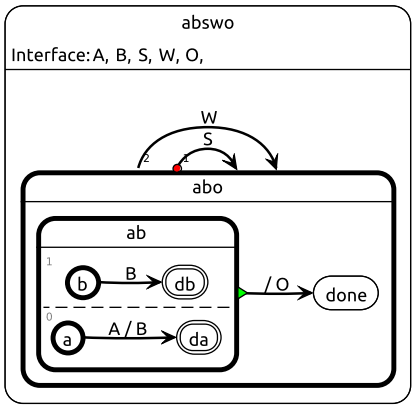
Outline. In the next section we consider related work, followed by a closer look at SyncCharts and Synchronous C in Sec. III. In Sec. IV we describe the compilation process and give experimental results in Sec. V. We conclude in Sec. VI. For a more detailed description of the compilation process than space permits here, we refer the reader to a technical report [8].

II. Related Work

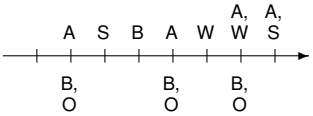
While Statecharts are an appealing language to describe reactive behaviors, the generation of efficient code is not triv-

978-3-9810801-7-9/DATE11/©2011 EDAA

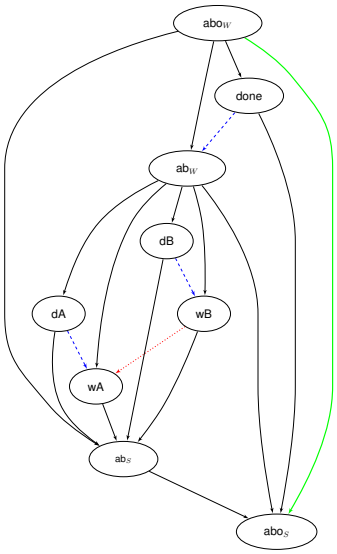
[1] www.informatik.uni-kiel.de/rtsys/sc



(a) SyncChart



(b) Sample execution trace, with inputs above and outputs below logical tick time line



(c) The priority graph. Subscripts S and W indicate strong and weak transitions. The solid (green) line from abo$_W$ to abo$_S$ is a dependency due to transition numbers. The other solid (black) lines represent hierarchical orderings, dashed (blue) lines are control flow orderings, dotted (red) lines are signal dependencies.

```
1                TICKSTART(5);
2   L_abo:       FORK(L_abo_ab, 2);
3                FORKE(L_abo_main);
4   L_abo_ab:    FORK(L_abo_ab__wA, 4);
5                FORK(L_abo_ab__wB, 3);
6                FORKE(L_abo_ab_main);
7   L_abo_ab__wA: AWAIT(sig_A);
8                EMIT(sig_B);
9                TERM;
10  L_abo_ab__wB: AWAIT(sig_B);
11               TERM;
12  L_abo_ab_main: JOIN;
13               EMIT(sig_O);
14               HALT;
15  L_abo_main:  PAUSE;
16               if (PRESENT(sig_S)) {
17                 ABORT;
18                 PRIO(5);
19                 GOTO(L_abo); }
20               PRIO (1);
21               if (PRESENT(sig_W)) {
22                 ABORT;
23                 PRIO(5);
24                 GOTO(L_abo); }
25               PRIO(5);
26               GOTO(L_abo_main);
27               TICKEND;
```

(d) Synthesized SC tick function

Figure 1. The ABSWO example: wait concurrently for the inputs A and B, if both have occurred, emit output O. The behavior is reset by the inputs S and W, which trigger a strong and weak abort of abo, respectively.

ial. Three different methods of compiling Statecharts can be distinguished: 1) compilation into an object oriented language using the state pattern [1], 2) dynamic simulation [10], and 3) flattening into finite state machines. Executing SyncCharts with SC, proposed here, can be classified as a simulation-based approach, where SC defines a simulator.

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 [7]. Since they extend the language, they have to consider all possible usages of GOTO, *e. g.*, jumping from one thread into another.

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[2]. However, this compiler generates circuit code in the spirit of Esterel and does not directly reflect the structure of the source SyncChart.

III. SyncCharts and Synchronous C

SyncCharts: SyncCharts are a Statechart dialect with a semantics that adheres to the *synchrony hypothesis* [4]. This implies that the execution is divided into discrete *ticks*, and computations within a tick do not take time. Fig. 1a shows the ABSWO SyncChart, which demonstrates deterministic concurrency, preemption, and signal-based communication. A possible execution trace is given in Fig. 1b.

SyncCharts inherit the concept of signals and valued signals from Esterel. 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 synchrony hypothesis implies that each signal has a unique status within a tick.

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.)

For further explanations of the language features of SyncCharts, the reader may refer to the overview given by André, which also contains numerous illustrative examples [2].

Synchronous C (SC): SC is an extension of C to allow concurrency and preemption in a deterministic way. It was designed to express the behavior described by a SyncChart directly in C in a concise and readable fashion. Fig. 2 provides 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 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

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

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.
FORK(l, p)	Create a thread with start address l and priority p.
FORKE$^*(l)$	Finalize FORK, resume at l.
JOIN^{*+}	Proceed if descendant threads have terminated normally
PRIO$^{*+}(p)$	Set current thread priority to p.
GOTO(l)	Jump to label l.
EMIT(S)	Emit signal S.
PRESENT(S)	True iff S is present.
AWAIT$^{*+}(S)$	Shorthand for l_{else}: PAUSE; PRESENT(s, l_{else}).

Figure 2. Main SC operators. 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).

inactive, *i. e.*, it was already executed in the current tick, or *active* if it still needs to be executed in the current tick.

As the priorities of the threads are also used as unique identifier, the priority of a thread might never be set to the same priority as an already active thread.

With SyncCharts as with Esterel, one might write self-contradicting, *i. e.*, non-constructive, programs, for which no execution schedule can be found. In SC, this situation is somewhat different. On the one hand, the execution schedule is always implied by the sequential nature of C, hence we have determinism. 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 that a signal whose status has already been tested is not emitted later within the same tick (no "write after read"). For our compilation from SyncCharts to SC, the compiler requires that the SyncChart can be statically scheduled, and hence rejects non-constructive programs.

IV. SC Synthesis

Since the instructions of SC were developed to express SyncCharts naturally in C, the compilation of SyncCharts into SC is more straightforward than the general compilation of SyncCharts into plain C. 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 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.

The remaining difficulty is to schedule the different threads such that the execution conforms to the SyncChart semantics, which is done by computing a priority for each transition There are different types of *ordering constraints*, alternatively also referred to as *dependencies*, that must be obeyed in thread scheduling. According to these constraints, the compiler creates a *priority graph*, from which scheduling priorities of thread (segments) are computed by a topological sort. The priority graph for ABSWO, which illustrates most of the ordering constraints, is shown in Fig. 1c.

Hierarchy Order: For a macro-state, all outgoing strong aborts need to be checked before the content of the state is executed, and all weak abortions and normal terminations must be checked after the execution. This is reflected in the priority graph by duplicating state nodes. For a macro state M, the priority graph contains nodes M_S and M_W that correspond to the strong and weak abortions, respectively, that leave M. For a region S within M, M_S must be scheduled before S, and M_W must be scheduled after S. In ABSWO, for example, state done is ordered after abo_S and before abo_W. This is achieved by reducing the thread priority from 5 to 1 in Line 20 in the generated SC code (Fig. 1d) and back to 5 in Line 23.

Transition Order: The outgoing transitions must be handled according to the priorities indicated by their transition numbers. In most cases, the transition order can be handled by ordering the code that checks the triggers accordingly. In ABSWO there is only one transition order dependency due to the two outgoing transitions of abo.

Data Dependency Order: We also utilize thread 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. In ABSWO, state wB depends, via signal B, on state wA.

Control Flow Order: A state must have a priority equal to or lower than all states from which it can be reached via immediate transitions.

V. Experimental Results

The compiler from SyncCharts to SC has been implemented as extension of the open-source KIELER (Kiel Integrated Environment for Layout Eclipse Rich-Client) tool[3] [5], which also uses the compiler for simulation (see Fig. 3).

For the validation of the compiler we compare the behavior synthesized code to the behavior of SyncCharts in E-Studio. For the performance evaluation, we implemented the reactive interface defined for Esterel [6]. Hence we can use the same wrapper to run the Esterel code and the synthesized SC code. First we compare our synthesized code to the hand-written SC examples that are part of the SC distribution. While we

[3]www.informatik.uni-kiel.de/rtsys/kieler

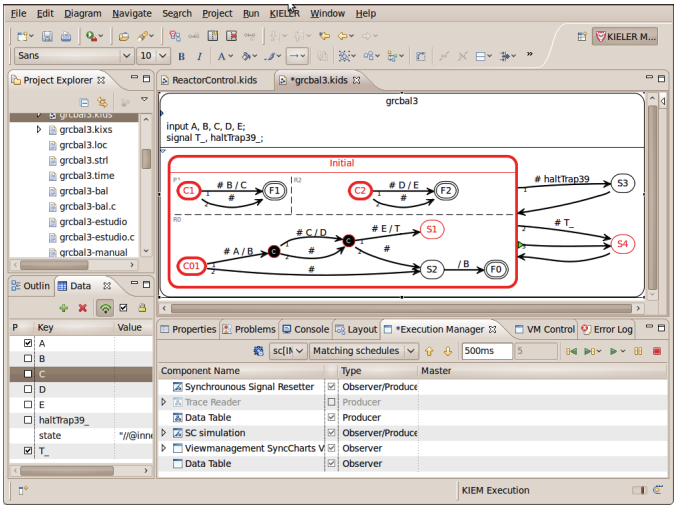
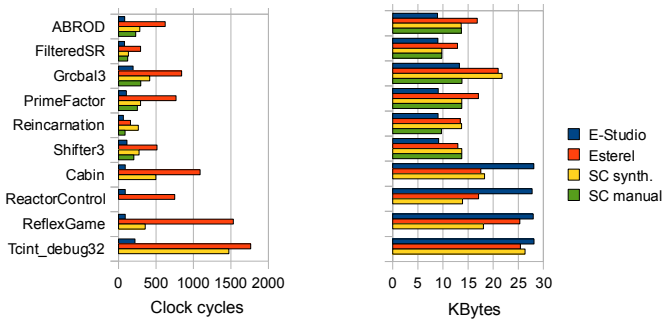


Figure 3. Simulation of the Grcbal3 example within KIELER, using the synthesized SC code.



(a) Reaction time (b) Size of executable

Figure 4. Comparison between code generated by Esterel Studio (E-Studio), code generated by KIELER via Esterel and the Esterel V5 compiler (Esterel), SC code synthesized by KIELER using the approach presented here (SC synth.), and manually written SC code (SC manual).

do not really expect an improvement, it shows how close the synthesized 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. These were compiled to Esterel and further to C code using the fast graph code [6] approach. Further compilation of C code is done with gcc and default optimizations (O2). All experiments were performed on an Intel Xeon running with 3 GHz and 6 MB cache. All programs are executed for 1 million ticks with random, but identical, input traces. Fig. 4a shows the average execution time for each tick in clock cycles. The executable sizes are shown in Fig. 4b.

In general, the performance of the synthesized SC is almost as good as for the manually written SC. However, the synthesized 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. However, it must be noted that the Esterel code generated by KIELER, which directly follows André [2], is not as optimized as the Esterel Studio compiler. Still, the size of the synthesized code is similar for most examples.

VI. 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 synthesized 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 synthesized 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.

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.

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] 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.

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

[5] H. Fuhrmann and R. von Hanxleden. Taming graphical modeling. In *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, LNCS, Oslo, Norway, Oct. 2010. Springer.

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

[7] 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.

[8] C. Traulsen, T. Amende, and R. von Hanxleden. Compiling SyncCharts to Synchronous C. Technical Report 1006, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Kiel, Germany, July 2010.

[9] 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.

[10] 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.