Synthesizing Manually Verifiable Code for Statecharts

Steven Smyth
Department of Computer Science
Kiel University
Kiel, Germany
ssm@informatik.uni-kiel.de

Christian Motika
Philotech Systementwicklung und Software GmbH
Hamburg, Germany
chris@motika.de

Reinhard von Hanxleden
Department of Computer Science
Kiel University
Kiel, Germany
rvh@informatik.uni-kiel.de

Abstract

Statecharts are an established mechanism to model reactive, state-oriented behavior of embedded systems. We here present an approach to automatically generate code from statecharts, with a particular focus on readability and ease of matching the generated code with the original model. This not only saves programming effort and reduces the error rate compared to manual coding, but it also facilitates the task of verifying that the code does what it is supposed to do. We have implemented this approach for the SCCharts language in an open-source framework. A user study confirmed that the generated code tends to be more readable than code from other code generators.

CCS Concepts • **Software and its engineering** → *Source code generation*;

Keywords statecharts, compilation, manual verification, readability, safety-critical, DO-178

ACM Reference Format:
Steven Smyth, Christian Motika, and Reinhard von Hanxleden. 2018. Synthesizing Manually Verifiable Code for Statecharts. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '18), November 4, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3281278.3281283

1 Introduction

Software for embedded reactive systems often needs to deal with system modes, which are commonly modeled with state machines. Statecharts [10], as introduced by David Harel, is a notation for visually modeling state machines with a concrete semantics. This allows for synthesizing code automatically. Statecharts combine Mealy machines with hierarchy, orthogonality, and broadcast communication. Statecharts are an established formalism for modeling *reactive systems*, which continuously react to their environment. Conceptually, time is divided into a sequence of discrete *ticks*. At the beginning of each tick, the system under development (SUD) samples its inputs from some sensors, at the end of each tick it produces outputs that control actuators, as sketched in Fig. 1. A reactive software system thus typically includes a *tick function*, which is called once per reaction, reading inputs and producing outputs.

Code generators for various statechart dialects are commonly used in industry, as maintaining large software projects manually that involve big state machines is a tedious and difficult task which is error prone and time consuming. Many applications of embedded software are in the field of safety-critical systems, such as the aerospace or automotive domain, where authorities [17] demand software companies to prove verification to high- or low-level software requirements. Some software is generated with qualified code generators, but this is still rather the exception, and verification is typically still a manual process. For automatically synthesized parts of the code, such as state machine code, verification can thus only be achieved if the generated code is readable and clearly traceable to the abstract state machine representation that implements design requirements. However, existing state machine code generators tend to focus exclusively on qualities such as code compactness, speed or predictable execution time. The underlying assumption is typically that the generated code is not looked at by humans anymore, just like one does not tend to look at the machine code generated by a compiler.

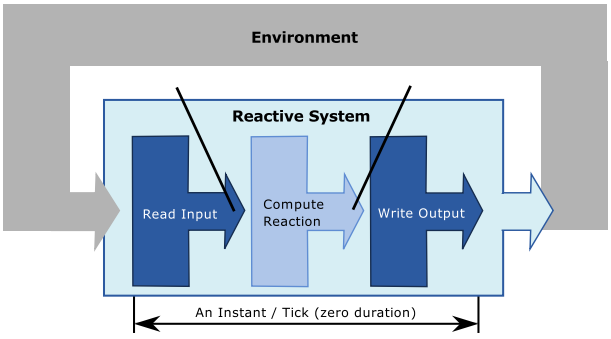


Figure 1. Reactive tick computation

REBLS '18, November 4, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6070-8/18/11…$15.00
https://doi.org/10.1145/3281278.3281283

Contribution and Outline In the next section, we present a collection of requirements for statechart code synthesis for safety-critical embedded systems. Here we distinguish *semantic requirements*, driven by the safety-critical nature of our application domains, and *coding requirements*, driven by the need for human code verification. In line with these semantic requirements, we opted for the SCCharts statechart language as a reference frame for our code generation approach; SCChart basics are covered in Sec. 3. Our core contribution, a *state-based compilation approach* to automatically generate code that meets the stated coding requirements, from statecharts that fulfill the semantic requirements, is presented in Sec. 4. To evaluate the state-based compilation approach, we conducted a user study, presented in Sec. 5. The study confirmed that our proposed new code generation approach produces code that is easier to understand, which should lead to less obscurities and errors, and ease verification compared to existing approaches. Related work is covered in Sec. 6. A conclusion and an outlook complete the paper in Sec. 7.

2 Requirements on Statecharts and Code Generation

To begin with, if one wants to derive executable code for a statechart, it must be clear what *behavior* the statechart specifies. However, while the basic idea of statecharts seems fairly straightforward, there is much room for interpretation of what their precise semantics is. Already back in 1994, von der Beeck identified 20 different variants [22], which, as it turned out, did not even include Harel's originally intended semantics. Since then, further Statechart dialects have been developed, including Stateflow [9], SyncCharts [2], SCCharts [15], or UML State Machines [6]. Even within commercially widely used UML state machines, there are various ambiguities and "semantic variation points" [7].

2.1 Semantic Requirements

A full treatment of possible statechart semantics and a detailed comparison of existing implementations is beyond the scope of this paper. However, we state at least some minimal *semantic requirements* that we consider reasonable for the realm of safety-critical systems.

Requirement R1. *The semantics shall be deterministic.*

For the aforementioned tick function, this means that the produced sequence of outputs should be fully determined by the sequence of inputs. In particular, there should be no *race conditions* due to concurrent variable accesses, and there should be no ambiguities due to multiple simultaneously enabled outgoing transitions from a state. This, for example, is not necessarily fulfilled for UML statecharts, for which the following applies: "if more than one guard evaluates to true, then the model is illformed. In such a case, the statechart semantics stipulate that only one of the transitions will be taken, but you can't predict which one it will be" [6].

Requirement R2. *The semantics shall be robust in that it should not depend on details of the visual representation of the model or the naming of model elements.*

The idea is that there should be no "surprises" with changes of semantics due to the visual representation of the model. For example, the positioning of transitions should not affect the semantics. This contrasts with, e. g., the "12 o'clock rule" of Simulink/Stateflow, where transitions are implicitly prioritized by their angular position relative to the source state [9]. In Yakindu, regions are scanned "either from left to right or from top to bottom"[1] (see also R3), but since regions can generally be arranged two-dimensionally, this does not always provide a clear ordering.

Requirement R3. *The semantics shall include true concurrency.*

This means that not only should the semantics permit concurrent regions that both have their own, independent state, but that these regions should also be able to react concurrently within the same tick, and they should be able to interact with each other. For example, if one region specifies that some light should turn on when a door is opened, and a concurrent region should increment an open-door counter, then both of these actions should take place when the door opens, not just one of them. This contrasts with, e. g., the "virtual concurrency" implemented in Yakindu Statecharts, where only one of these actions will take place and the choice of action depends on the arrangement of the regions (see R2). Similarly, it is debatable whether UML statecharts and their run-to-completion execution model that serializes all events is truly concurrent; at least it seems difficult to state whether, e. g., two transitions are "simultaneous" or not.

2.2 Coding Requirements

Given a statecharts language that fulfills certain semantic requirements, such as those stated in Sec. 2.1, we now formulate a number of *coding requirements* that the synthesized code should fulfill.

Requirement R4. *The code shall be self-explanatory, and it shall be easy to match the generated code with the original model and vice versa.*

This implies that the *structure* (*topology*) of model and code should match, e. g., that for each region or (super) state in the model there should be a corresponding piece of code. Likewise, the *naming* of variables and functions in the code should facilitate the mapping, e. g., a function that implements the behavior of some region should be named according to the region. Furthermore, the *order* of declarations and

[1] https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/sclang_statechart_language_reference

code segments should, as far as possible, be consistent with any order present in the model. As discussed for R2, there is not necessarily an obvious order of regions; but if there is an obvious order in the model, that order should be reflected in the code, unless there is a clear reason to change that order, e. g., to efficiently implement a scheduling requirement of the model (see Sec. 4.1). Generally, names should be rather "speaking" and not be overly abbreviated. Similarly, the generated code should include *comments* that facilitate understanding. Fulfilment of this requirement should also support debugging of the generated code, i. e., when stepping through the code, it should be clear what the code is doing even without referring to the original model.

Requirement R5. *The generated code shall be a* safe subset *of the target language.*

For example, the C language, which is still one of the most widely used languages in particular in the embedded area, has been designed with efficiency in mind, not robustness. It thus includes a number of features that are considered "unsafe" and are typically not permitted for safety-critical applications, such as *pointer arithmetic*, *complex macros*, *function pointers*, *break*, or *continue* statements [11, 14].

Requirement R6. *The generated code shall not rely on external thread support.*

Conceptually, each statechart region is a *thread*, in the sense of a concurrent flow unit. However, that concurrent control flow should be managed within the tick function, without making use of thread libraries, such as, for example, POSIX threads. Similarly, the code should not make use of thread-synchronizing constructs such as semaphores or monitors. The rationale is that one wants to make the code self-contained and in full control of scheduling decisions. In particular, one wants to avoid non-determinism (considering also R1) as introduced by POSIX threads or, for that matter, Java threads [12].

3 SCCharts

There are several statechart languages that meet the semantic requirements from Sec. 2.1, in particular the statechart variants that belong to the family of synchronous languages [3]. These include SyncCharts [2], the state machine extension of SCADE [4], and SCCharts [15, 24]. We developed our code generator for SCCharts, and will describe their basics in the following. However, we would argue that much of the code generation approach presented here could be applied to other statechart dialects as well.

Like other synchronous languages, SCCharts separate concerns of functionality and timing, and reconcile concurrency (R3) with determinism (R1). This is achieved by assuming zero reaction computation time, which implies that inputs and outputs of a tick are *synchronous* with each other. In practice, the zero computational time abstraction means that the

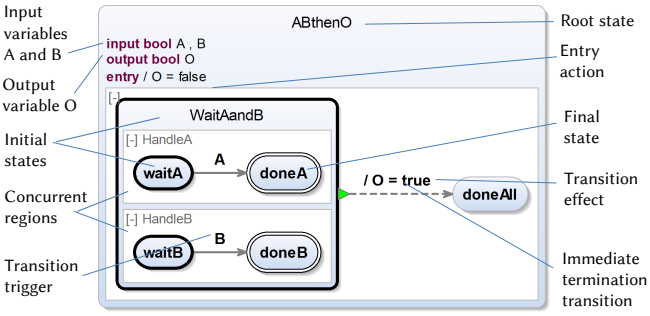


Figure 2. ABthenO SCCharts example

system reacts in each tick in time before the next tick starts. Most synchronous languages are thus rather restrictive as to what may happen within a tick, for example, by forbidding "instantaneous loops." In SCCharts, loops are permitted, which for example facilitates the mapping of C programs back to SCCharts [21]. However, if one wants to statically bind the reaction time and wants to perform worst-case reaction time (WCRT) analysis, loops must be restricted accordingly [8, 13].

3.1 SCCharts Language Overview by Example

Fig. 2 shows the ABthenO example that illustrates the basic SCCharts language features. ABthenO is the *root state* and declares the *interface* of the SCChart: Two boolean *inputs* A and B and a boolean *output* O. ABthenO also declares an *entry action* setting O to false initially, when computation starts. Inside ABthenO, there is just one (anonymous) *region* with two states WaitAandB and doneAll. WaitAandB's bold border indicates that it is an *initial state*. Hence, when the SCCharts starts, this will be the first *active* state. Furthermore, WaitAandB is a *superstate*, because it contains further regions and states. Inside WaitAandB, there are two *concurrent regions*: HandleA and HandleB. Both have their own initial states where control starts, waitA and waitB, respectively. Each of these states has an outgoing transition with a *trigger* that references a declared input. Whenever the input becomes true and the SCChart is in the respective state, it will take the transition to its destination state doneA or doneB. Both are *final states*, visually indicated by the double border. When a region ends up in a final state, it terminates control. If all concurrent regions of a superstate terminate control, a *termination* ("join") transition, indicated with a green triangle, can trigger (cf. transition from WaitAandB to doneAll). A dashed transition line indicates that the transition can trigger *immediate*ly (in the same tick), when the source state of the transition is entered. A solid transition line indicates a *delayed* transition, which can only trigger at the earliest in the next tick after its source state has been entered.

The behavior of the SCChart is as follows: In the initial tick, states ABthenO, WaitAandB, waitA, and waitB are entered

and the output O is set to false. Since waitA and waitB have no outgoing immediate transitions, these states cannot be left and inputs A and B are simply ignored. Then, in the following ticks, if A or B are *present*, the SCChart reacts to A and/or B by taking the respective transitions from waitA to doneA and/or waitB to doneB. As soon as both doneA and doneB have been reached, the termination transition is taken immediately, sets the output O to true, and transitions to state done. Here, the reaction stops but the SCChart will remain active (does not terminate) as doneAll is not a final state.

3.2 Extended SCCharts

ABthenO makes do with the so-called *Core SCCharts* language features. SCCharts supports additional languages features, also referred to as *Extended SCCharts*, including handling of physical time [20]. However, Motika et al. showed that every SCChart can be represented by a semantically equivalent Core SCChart [15]. Hence, it is sufficient if a code generator supports Core SCCharts to be able to compile every model. In fact, Core SCCharts can be *normalized* to even simpler, but less compact patterns to ease the downstream compilation. However, as one goal is the preservation of the statechart's topology (R4), we refrain from using the normalized variant for the state-based approach presented here. The same might apply to Extended SCCharts; while compiling away more complex features may help the compiler, it might obscure the topology of the original model for the human. For now, it remains future work to analyze which features obscure the topology, and we focus on Core SCCharts in this contribution.

3.3 Existing SCCharts Compilers

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) provides an SCCharts modeling environment, which includes the KIELER Compiler (KiCo) that provides various compilation options, including different target languages (e. g., C, Java, VHDL). The initial SCCharts paper [24] introduced two techniques, namely *netlist-based* and *priority-based* compilation, which subsequently have been covered in more detail elsewhere [15]. As these compilation approaches adhere to the SCCharts semantics, they do meet R1, R2, R3. They also fulfill R6. However, they do not meet R4, and at least the priority-based compilation based on computed gotos does not fulfill R5 either.

4 The State-Based Compilation Approach

Driven by the coding requirements from Sec. 2.2, we developed the *state-based* compilation approach, of which we now present the main principles. For full detail, we refer to our open-source implementation in the KIELER SCCharts modeling environment (see also Sec. 6).

4.1 Priority-Based Concurrency

To implement concurrency (R3) without external thread support (R6), the state-based approach adopts the concept of node *priorities* from the priority-based compilation [24] with an explicit light-weight application-level thread concept. The basic idea is that concurrent regions are scheduled according to a statically computed priority. The generated code includes a *dispatcher*, which, within a tick, determines which regions are eligible for execution. Among those, it dispatches the ones with the highest priority, and keeps doing so until no more region has any work to do in the current tick. If there are multiple eligible regions with highest priority, there should be a deterministic ordering of these regions, typically given by their order in the code. Thus the code generator may use both, priorities and the region ordering in the code, to implement scheduling constraints given by the statechart semantics.

There are various options on how to implement the dispatcher, which affect both code complexity and efficiency. We here opted for a rather straightforward approach, which first scans all regions that are ready to execute for their maximum priority, and runs them if they are ready and have that highest priority. This appears to perform reasonably well in most cases (see also Sec. 5), but for models with many regions per superstate and many priority changes, alternative schemes with additional data structures might be more efficient. In either case, this dispatching is a comparatively very light-weight application-level context switching mechanism, typically much more efficient than OS threading.

Another interesting question is how the priorities are determined. For example, if one wanted to implement the semantics of LabVIEW statecharts, then this priority could be assigned according to the lexical order of region names (notwithstanding the robustness issue discussed on R2). To implement the concurrent scheduling of Céu [19] or PRET-C [1], which execute concurrent threads sequentially according to their syntactic order in the program, the priority could be assigned according to that order. Alternatively, for both of these schemes, one could assign all regions the same priority and just generate code such that the regions appear in the correct order. However, LabVIEW, PRET-C, and CÉU preclude back-and-forth interaction of concurrent regions (R3). For these reasons, SCCharts adopt a different semantics, namely the *sequentially constructive* semantics, which schedules concurrent regions depending on how these regions interact through shared data [25], meeting R3. In the example of Fig. 3, R0 writes a variable (O2) that is read concurrently by R1. This induces a *data dependency*, indicated with a dashed arrow, and implies that R0 must be executed before R1 and, hence, receives a higher priority.

Note that data dependencies may lead to scheduling conflicts that cannot be resolved; for example, if one transition guarded by a flag A sets a flag B, and concurrently another

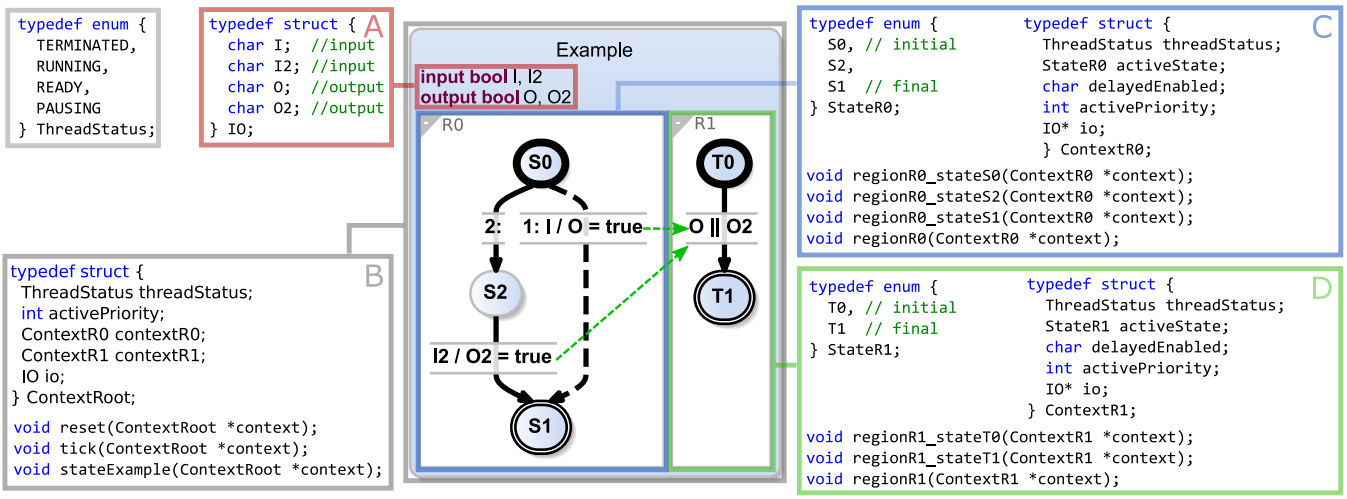


Figure 3. Topology-preserving code-generation (see Sec. 4.2). The SCChart's interface gets transformed into the interface IO struct (red A); the root state creates the main root context and the functions that the environment should call, namely reset and tick (gray B); the R0 region gets transformed into the ContextR0 struct with its accompanying functions (blue C); similarly, R1 is transformed into the ContextR1 (green D). The green dashed arrows indicate *data dependencies*, which influence the scheduling order.

transition guarded by B sets flag A. In such a case, the SCChart is *not constructive* (*not causal*) and rejected. This causality requirement is common in synchronous languages; it is also found, e. g., in functional reactive programming, where nodes are topologically sorted according to their data dependencies to avoid "glitches" [5]. However, since priorities are computed statically, constructiveness is also checked at compile time. Hence, there are no run-time surprises. More information on how priorities are actually computed can be found elsewhere [24].

Unlike earlier synchronous statechart proposals, such as SyncCharts [2], the sequentially constructive semantics of SCCharts allows variable values to change within a reaction and to modify variables after they have been written, as long as the model is still deterministic. We did not state this flexibility as a hard requirement and our code synthesis approach would be applicable just as well to SyncCharts; this is not surprising, since in a way, SCCharts can be viewed as a conservative extension of SyncCharts. However, in practice we consider this flexibility to be a significant advantage, again leading to more compact models and simpler code, which would not be possible under the more restrictive semantics of, e. g., SyncCharts.

4.2 Preserving Topology

The basic topology mapping is straightforward. Fig. 3 shows how different parts of the SCCharts model get transformed into C code. The *interface* of the SCChart is translated into a dedicated IO struct (red A). Comments are added to give additional information about inputs and outputs. The root state is transformed into a ContextRoot struct which holds the interface, a ThreadStatus, and *sub-contexts* from enclosed regions (dark gray B). The functions that can be called from the environment are reset and tick. reset initializes the state machine and is usually called at start-up. tick can then be invoked periodically to calculate exactly one discrete tick of the automaton. The regions inside of the root state are then transformed similarly (blue C and green D). Each region has a set of states, an own context struct, and a set of functions that mimic the topology of the statechart. The details of each are described in the following.

4.3 Thread Status

Every thread has a ThreadStatus member. It is an enum (light gray, Fig. 3), which indicates what status this thread has, with four different statuses, see Fig. 4. A READY thread is ready and waits for its execution. RUNNING indicates that the thread is currently executing. If a thread has finished its reaction for a tick, it sets itself to PAUSING. If it ceases to exist, it is TERMINATED and can only be re-spawned from a higher hierarchy.[2]

4.4 Superstates

As explained in Sec. 4.1, the priorities are calculated statically. The concurrent *data dependencies*, shown as green, dashed edges in Fig. 3, determine the priorities in this example. Static thread priorities are set when a superstate is

[2]These four states are a flattened and slightly refined encoding of the enabled/active flags of the original SCCharts proposal [24]: disabled threads are TERMINATED; enabled and inactive threads are PAUSING; enabled and active threads are READY or RUNNING, depending on whether they are currently dispatched or not.

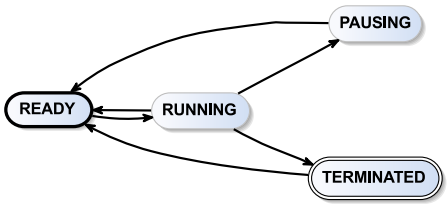


Figure 4. Thread status in priority-based concurrency

```
void reset(ContextRoot *context) {
  context->contextR0.io = &(context->io);
  context->contextR1.io = &(context->io);

  context->contextR0.activeState = S0;
  context->contextR0.delayedEnabled = 0;
  context->contextR0.activePriority = 2;
  context->contextR0.threadStatus = READY;

  context->contextR1.activeState = T0;
  context->contextR1.delayedEnabled = 0;
  context->contextR1.activePriority = 1;
  context->contextR1.threadStatus = READY;

  context->activePriority = 1;
  context->threadStatus = RUNNING;
}
```

Figure 5. Internal code structure of the reset function

entered. Note that the root state is a superstate with an interface for the environment; any superstate inside an SCCharts model is constructed the same way. For the root state, the priorities are set inside the reset function, which is called when the program is initialized. Fig. 5 shows the internals of the reset function. Besides initializing the thread context, the previously calculated priorities for R0 and R1 are set. As the dependency edges in Fig. 3 indicate, R0 must be executed before R1 and, hence, receives the higher priority.

Eventually, the root state's function stateExample is called, depicted in Fig. 6. As with every superstate, its purpose is to determine which contained region is allowed to run in which order and to adjust the priorities accordingly. Hence, at the beginning of the function, all threads that are still able to run are set to RUNNING and are invoked sequentially (see the red box A in Fig. 6). All threads that are still ready to run contribute to the new priority, which is set afterwards (blue box B). If all threads finished their execution for the active tick, the priority is set to the maximum of the paused threads for the next tick (green box C). Eventually, all threads have completed their tick and the superstate checks if it must terminate itself (gray box D). The control then returns to the caller. Note that the code can be further optimized if a superstate contains only one region or if the static priorities do not change.

4.5 Regions

Every region inside a superstate gets its own context. For each region, the context struct, an enum with the included states, and a function for every state plus one function for

```
void stateExample(ContextRoot *context) {
  int newActivePriority = 0;

  if (context->contextR0.threadStatus == READY) {
    if (context->contextR0.activePriority == context->activePriority) {
      context->contextR0.threadStatus = RUNNING;
      // Call the logic code of thread R0.
      regionR0(&context->contextR0);
    }
    if (context->contextR0.threadStatus == READY) {
      if (context->contextR0.activePriority > newActivePriority) {
        newActivePriority = context->contextR0.activePriority;
      }
    }
  }
  if (context->contextR1.threadStatus == READY) {
    if (context->contextR1.activePriority == context->activePriority) {
      context->contextR1.threadStatus = RUNNING;
      // Call the logic code of thread R1.
      regionR1(&context->contextR1);
    }
    if (context->contextR1.threadStatus == READY) {
      if (context->contextR1.activePriority > newActivePriority) {
        newActivePriority = context->contextR1.activePriority;
      }
    }
  }                                                              A

  // Set the new priority.
  context->activePriority = newActivePriority;                   B

  // Calculate the priority for the next tick if necessary.
  if (context->contextR0.threadStatus != READY &&
    context->contextR1.threadStatus != READY) {
    if ((context->contextR0.threadStatus == PAUSING) &&
      (context->contextR0.activePriority > context->activePriority)) {
      context->activePriority = context->contextR0.activePriority;
    }
    if ((context->contextR1.threadStatus == PAUSING) &&
      (context->contextR1.activePriority > context->activePriority)) {
      context->activePriority = context->contextR1.activePriority;
    }
  }                                                              C

  // Check if the root state must terminate.
  if (context->contextR0.threadStatus == TERMINATED &&
    context->contextR1.threadStatus == TERMINATED) {
    context->threadStatus = TERMINATED;
  }                                                              D
}
```

Figure 6. Internal code structure of a superstate

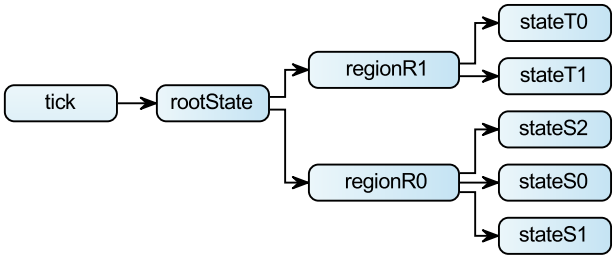


Figure 7. Hierarchical call tree of the generated code

the whole region is created. As before, the context contains a thread status, and a pointer to the interface. Additionally, it holds the active state, a flag which signals if delayed transitions are enabled, and an active priority. The functions are named appropriately and called when the corresponding element in the SCChart is active.

The call stack for these functions is constructed hierarchically, as depicted in Fig. 7. The environment sets the inputs

```
void regionR0(ContextR0 *context) {
  /* Cycle through the states of the region as long as this thread
   * is set to RUNNING. */
  while(context->threadStatus == RUNNING) {
    switch(context->activeState) {
      case S0:
        regionR0_stateS0(context);
        break;

      case S2:
        regionR0_stateS2(context);
        break;

      case S1:
        regionR0_stateS1(context);
        break;
    }
  }
}
```

Figure 8. Internal code structure generated from a region

```
void regionR0_stateS0(ContextR0 *context) {
  /* Transition 0: immediate to final state S1
   *   Trigger/Effects: I / O = 1 */
  if (context->io->I) {
    context->io->O = 1;
    context->activeState = S1;
    context->delayedEnabled = 0;
  } else if (context->delayedEnabled) {
    /* Transition 1: delayed to state S2
     *   This is the default transition, the trigger is always true. */
    context->activeState = S2;
    context->delayedEnabled = 0;
  } else {
    // Wait for next tick if no transition was taken.
    context->threadStatus = PAUSING;
  }
}
```

Figure 9. Internal code structure generated from a state

and calls the tick function to compute the reaction for one tick. The tick function then calls the root state's function stateExample. As the root state includes the two regions regionR0 and regionR1 in the example, their functions are called next. Each region function is then responsible for calling the active state functions. Note that the order in which the regions are executed depends on their priority and can be interleaved.

As stated before, the superstate function calls the functions responsible for the regions in the correct order. Such a function is straightforward, as shown in Fig. 8. The region code is looped as long as the thread status is RUNNING. The contained switch block selects the correct function for the actual state and calls it. A reaction inside of a state function can set the thread status to READY, PAUSING, or TERMINATED to yield, so that the region while loop will be left. READY means that the thread releases its control so that another thread can continue. This happens when the priority of a thread changes. PAUSING signals that the thread finished its reaction for this tick. Paused threads will be set to READY at the beginning of the next tick. TERMINATED indicates that the region has reached a final state and is terminated until invoked again from a higher hierarchy.

Table 1. Lines of code for the different trials: netlist-based, priority-based, state-based without comments, and state-based with comments

Lines of code	Netlist	Prio	State w/o	State w/
Header	34	39	55	129
Source	52	104	154	226
Overall	86	143	209	355

4.6 States

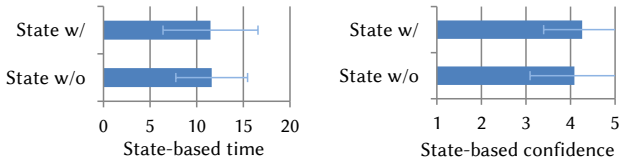
The reactions happen inside the state functions. Fig. 9 depicts the source code of the state S0. Here, for every outgoing transition it is checked whether or not the transition is eligible to fire. Therefore, the *delay status* and the *trigger* of all transitions are checked in order of their transition priorities. S has two such transitions: One immediate transition with index 1 to S1 with I as trigger, and one delayed transition with index 2 without trigger to S2. According to the transition index, the function first checks if I is true. If so, the reaction of that transition is executed: The output O is set to true and the control is handed over to S1 while setting the delayed flag to false. If the transition cannot fire, it is checked if the delayed transition is eligible to run. If so, the control is transferred to S2. If no transition can be taken, the thread is set to PAUSING to signal the reaction's end for this tick. Note that extended information of the transitions and their reactions are annotated as comments by the code generator. These comments are also added to the header file where the function declarations are stored. While it is still necessary to validate the source code inside the source files, the transition reactions can also be seen inside the header files to support readability.

5 Evaluation

The goal of the presented approach was to create well-readable code, which should ease verification. To validate this, we conducted a user study that compares the code generation to other approaches. The participants were given code from three different code generators to compare in different ways. Further, they were asked to reverse-engineer the original statechart from it, explained further in Sec. 5.1. Results of the study are presented in Sec. 5.2, followed by executable sizes and execution times in Sec. 5.3.

5.1 Set-Up

24 students participated in the study. All students were given a short 5 min introduction to the semantics of SCCharts, but without information on how the different code generators work or what kind of source they produce. For every code generation approach, the participants should inspect the automatically generated source code without having seen the original source model. They should then draw the model from which the model was generated. Here, the students



(a) Mean time (min) of the state-based trials

(b) State-based mean confidence (1=low, 5=high)

Figure 10. Mean time and confidence of the state-based compilation trials with vs. without comments regardless of trial ordering

were asked to draw as many characteristics of the SCCharts as possible, which were states, regions, transitions, the interface, and the labels of the states and regions.

There was a time limit of 20 min per trial, but a trial could be finished prematurely. For each trial, the SCCharts were similar to the example shown in Fig. 3. There were four trials that were presented in different orderings to each participant: Netlist-based compilation, priority-based compilation, state-based compilation without comments, and state-based compilation with auto-generated comments. The different sizes in lines of code, including comments, between the trials can be seen in Tab. 1. The overall program code that needs to be understood by the study's participants grows with each trial with 86 lines for the netlist-based, 143 lines for the priority-based, and 209 and resp. 355 lines for the state-based approach. The drawn SCCharts were checked for their correctness. Additionally, we measured the participant's time needed and asked them to give a confidence rating between 1 (very unsure) to 5 (very sure) for each trial.

5.2 Case Study Results

As the participants got all trials in different order, we first wanted to know if the order in which they got the two state-based trials matters. As Fig. 10 shows, the mean time (Fig. 10a) and the mean confidence (Fig. 10b) of the two state-based trials are nearly identical even though the participants got the two trials in different ordering. Hence, for the comparison of the different compilation approaches, we do not differentiate between the state-based approaches with or without comments, but whichever the participant worked on first (indicated by State I and State II respectively).

Time & Confidence Fig. 11 depicts the overall time and confidence results of all trials. Despite shorter programs w.r.t. lines of code as shown in Tab. 1, the trials with the netlist-based and priority-based generated codes almost always needed the full amount of time (see Fig. 11a). Even the first state-based trial only needs 14 minutes in the mean although the code is up to four times larger. Once accustomed to the structure of the code, the second trial of the state-based code-generation can be done in under 10 minutes mean.

Of course it is also possible get a training effect with the netlist- or priority-based compilation, but we argue that it is inherently more difficult and with larger models or in some other cases, e. g., strongly optimized netlists, maybe even impossible to reach same result in time and confidence compared to the state-based compilation.

Analogously to the time ratings, the confidence rises with the state-based approach. While the netlist-based code seems to be difficult to understand, the priority-based compilation, being closer bound to the structure of the program, scores better. The state-based compilation, which maps the topology of the statechart nearly one-to-one, always scores a mean *sure* or better rating, even though the participants did see that kind of code before.

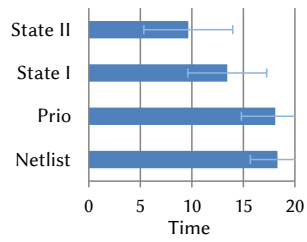
Correctness As stated in Sec. 5.1, we also checked for the correctness of the drawn SCCharts. The results are shown in Fig. 11. In the evaluation, we split the ratings into *functional correctness* (Fig. 11c), which says if the model behaves semantically correct, and *correctness of appearance* (Fig. 11d), which rates if the model's graphical appearance resembles the original model. In the case study, this mainly concerns state and region labels. While superfluous syntax, e. g., transient states, is not per se incorrect, it can impair the overview of the model. Hence, we added a second rating to Fig. 11d with deductions for superfluous syntax.

The functional correctness results almost mirror the confidence ratings. The participant's models for the netlist-based compilation were 40% correct. For the priority-based compilation they reached 70%, and the state-based compilations were with 90% almost correct without prior knowledge of the SCChart.

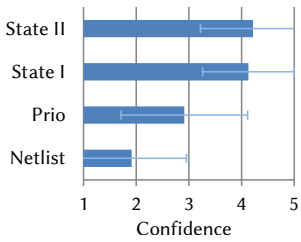
The correctness of appearance results are similar. Many of the graphical elements, which include for example labels of regions, states, and variables, are transformed and partially lost during the netlist-based and priority-based compilation. In the second evaluation with the point deductions, the priority-based approach score worse, because the participants drew many transient states that mimic the program control flow which are not necessary and did not resemble the original model even if the model is semantically correct. The topology-preserving state-based compilation keeps most of these elements with their names in the final code and also uses them to auto-generate appropriate comments.

The result suggests that there is some room for improvement even with the state-based approach. The participant's unfamiliarity to the subject may play a role here, because a commented header (cf. Fig. 3) is arguably enough to score a 100% correctness of appearance rating. Hence, we will investigate this further.

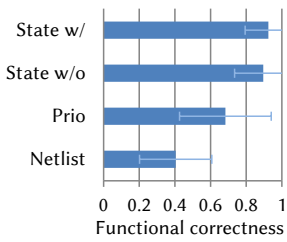
Note that in practice w.r.t. verification, the source statechart is most-likely known. We argue that close to perfect correctness is feasible (at least for small model sizes), because the untrained participants were able to score a 90%



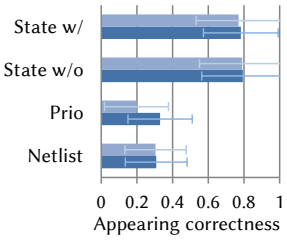
(a) Mean time (min) of all trials



(b) Mean confidence of all trials (1=low, 5=high)



(c) Mean functional correctness of all trials



(d) Mean correctness of appearance (light: deductions)

Figure 11. Mean time and confidence of all trials.

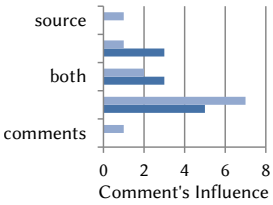


Figure 12. Comment's influence rating

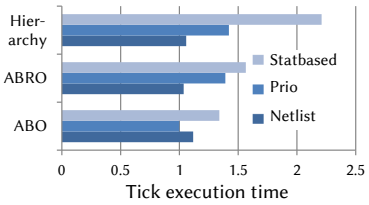


Figure 13. Mean tick execution time (µs)

(resp. 80%) rating without knowledge of the SCChart. We plan to conduct another study that solely focuses on verification to confirm this. The second study should also include more complex models.

Code Comments In the trial on the state-based approach with comments, we further asked the participants to rate the influence of the comments towards their drawings (see Fig. 12) from 1 ("I looked only at the comments") to 5 ("I looked only at the source code"). Again, we separated the results in two groups. The results of the group who got the state-based pattern without comments first are depicted in dark blue. The results of the group with the comments first are shown in light blue. While there is a peak in both groups at the "I looked more on the comments" rating, we feel that there is not enough data to support that claim yet, especially, as the results in Fig. 10 and Fig. 11 do not show significant distinctions between the two state-based trials. Overall, the variant with the comments got a slightly better confidence and functional correctness rating, whereas the appearance rating is slightly worse. However, the survey sample size is too small to conclude from this slight differences.

5.3 Run-Time Results

Fig. 13 shows a preliminary runtime evaluation for small models. While the state-based approach is only marginally slower at models with little concurrent communication (see ABO and ABRO), a slow down is measurable for models with more concurrency (see Hierarchy with five communicating concurrent regions) due to the traversal through the call stack. We will further investigate this on possible optimizations, such as an optimized call stack traversal or simplified code generation for superstates.

6 Related Work

As discussed in Sec. 2, there exist numerous statechart dialects, and most of them can be synthesized into code, but typically not such that it meets our stated requirements. For example, SyncCharts [2] can by synthesized into Esterel, for which in turn various approaches exist to generate C code [16]. These have also inspired the original code synthesis approaches for SCCharts, but like them suffer from poor traceability back to the original model. One aim of the state-based synthesis approach is to generate code in such a way as a programmer would have written it manually. The generated code has similarities to the state-machine pattern advocated when programming directly in C/C++/Java. Samek discusses how to code UML Statechart in C/C++ [18], von Hanxleden presents an approach to implement SyncCharts in C [23]. For SCADE [4], a synchronous modeling language that also includes statecharts, there exists a qualified code generator. The emphasis there is qualification of the tool itself, not of the generated code. However, we could envision that the traceability offered by the state-based code synthesis approach presented here would also be an enabler for qualification of the compiler and additionally eases debugging on the target.

7 Conclusion and Outlook

We have presented a new code synthesis approach for statecharts, the *state-based* approach, with the aim of producing code that lends itself to human inspection and verification. This should facilitate model-driven software development for safety-critical applications, by minimizing the time from specification to qualified code. One guiding principle was to create code as a human programmer would have written it, including self-explanatory names and sufficient commenting. We have presented our code generation approach fulfilling requirements R1-R6 in the setting of SCCharts, which possess properties such as determinism, true concurrency, and

the possibility of sequentially evolving variable values within a reaction.

Based on the presented results, we plan to conduct a second study to evaluate the suitability of the state-based approach w.r.t. concrete verification steps. The second study should include more complex models. Moreover, we want to improve efficiency of the generated code. Further, we plan to study extended SCCharts features w.r.t. their impact to obscure topology and enhance them accordingly. We also consider the generation of truly parallel code for multi-core processing, although at this point it is not clear how well that will lend itself to qualification for safety-critical applications.

Acknowledgment

This work was supported by the German Science Foundation (DFG HA 4407/6-2 and ME 1427/6-2) as part of the PRETSY project.

References

[1] Sidharta Andalam, Partha S. Roop, and Alain Girault. 2010. Deterministic, predictable and light-weight multithreading using PRET-C. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'10)*. Dresden, Germany, 1653–1656.

[2] Charles André. 2004. Computing SyncCharts Reactions. *Electr. Notes Theor. Comput. Sci.* 88 (2004), 3–19.

[3] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The Synchronous Languages Twelve Years Later. In *Proc. IEEE, Special Issue on Embedded Systems*, Vol. 91. IEEE, Piscataway, NJ, USA, 64–83.

[4] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2017. SCADE 6: A formal language for embedded critical software development (invited paper). In *11th International Symposium on Theoretical Aspects of Software Engineering TASE*. Sophia Antipolis, France, 1–11.

[5] Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding Dynamic Dataflow in a Call-by-Value Language. In *15th European Symposium on Programming (ESOP '06)*. Vienna, Austria, 294–308.

[6] Bruce Powel Douglass. 1999. UML Statecharts. *Embedded Systems Programming* (Jan. 1999), 22–42.

[7] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P. de Roever. 2005. 29 New Unclarities in the Semantics of UML 2.0 State Machines. In *ICFEM (LNCS)*, Vol. 3785. Springer, 52–65.

[8] Insa Fuhrmann, David Broman, Reinhard von Hanxleden, and Alexander Schulz-Rosengarten. 2016. Time for Reactive System Modeling: Interactive Timing Analysis with Hotspot Highlighting. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems (RTNS '16)*. ACM, New York, NY, USA, 289–298. https://doi.org/10.1145/2997465.2997467

[9] Grégoire Hamon. 2005. A denotational semantics for Stateflow. In *EMSOFT'05: Proceedings of the 5th ACM International Conference on Embedded Software*. ACM Press, New York, NY, USA, 164–172.

[10] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 3 (June 1987), 231–274.

[11] Les Hatton. 1995. *Safer C: Developing Software for in High-Integrity and Safety-Critical Systems*. McGraw-Hill, Inc.

[12] Edward A. Lee. 2006. The Problem with Threads. *IEEE Computer* 39, 5 (2006), 33–42.

[13] Michael Mendler, Reinhard von Hanxleden, and Claus Traulsen. 2009. WCRT Algebra and Interfaces for Esterel-Style Synchronous Processing. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE '09)*. Nice, France.

[14] MISRA. 2013. *MISRA C:2012: Guidelines for the Use of the C Language in Critical Systems*. Motor Industry Research Association.

[15] Christian Motika. 2017. *SCCharts—Language and Interactive Incremental Implementation*. Number 2017/2 in Kiel Computer Science Series. Department of Computer Science. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel.

[16] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. 2007. *Compiling Esterel*. Springer.

[17] Leanna Rierson. 2013. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. Taylor & Francis Inc.

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

[19] Francisco Sant'Anna, Roberto Ierusalimschy, Noemi de La Rocque Rodriguez, Silvana Rossetto, and Adriano Branco. 2017. The Design and Implementation of the Synchronous Language CÉU. *ACM Trans. Embedded Comput. Syst.* 16, 4 (2017), 98:1–98:26.

[20] Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Frédéric Mallet, Robert de Simone, and Julien Deantoni. 2018. Time in SCCharts. In *Proc. Forum on Specification and Design Languages (FDL '18)*. Munich, Germany.

[21] Steven Smyth, Stephan Lenga, and Reinhard von Hanxleden. 2016. Model Extraction for Legacy C Programs with SCCharts. In *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '16), Doctoral Symposium (Electronic Communications of the EASST)*, Vol. 74. Corfu, Greece. With accompanying poster.

[22] Michael von der Beeck. 1994. A Comparison of Statecharts Variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (LNCS)*, H. Langmaack, W. P. de Roever, and J. Vytopil (Eds.), Vol. 863. Springer-Verlag, 128–148.

[23] Reinhard von Hanxleden. 2009. SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. In *Proc. Int'l Conference on Embedded Software (EMSOFT '09)*. ACM, Grenoble, France, 225–234.

[24] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. 2014. SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, Edinburgh, UK, 372–383.

[25] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. 2014. Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation. *ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design* 13, 4s (July 2014), 144:1–144:26.