Consistent SystemC and VHDL Code Generation from State Charts for Virtual Prototyping and RTL Synthesis

Rainer Findenig
Upper Austrian University o. A. S.
Hagenberg, Austria
rainer.findenig@fh-hagenberg.at

Thomas Leitner
DICE GmbH & Co KG
Linz, Austria
thomas.leitner@infineon.com

Vokan Esen, Wolfgang Ecker
Infineon Technologies AG
Neubiberg, Germany
{volkan.esen,wolfgang.ecker}@infineon.com

***Abstract*—In today's hardware development, SystemC code is widely used for virtual prototyping, where an abstract system model is used to do an early exploration of the hardware implementation as well as software development. The synthesis tools, on the other hand, conventionally still rely on VHDL as the entry language.**

State Charts, as for example included in UML, provide a rich graphical design entry method that can serve as both documentation and executable specification. This paper presents a method for both cycle callable SystemC and register transfer level VHDL code generation from State Chart models. The presented approach utilizes different modeling styles for SystemC and VHDL to allow simulation performance optimizations for the SystemC code and resource optimizations for the VHDL code, while still keeping their behavior consistent.

I. Introduction

To shorten today's design cycles, virtual prototyping is an essential concept to allow both faster evaluation of the architecture of hardware systems and the possibility to develop and test software for the generated hardware before actual RTL code is available. A virtual prototype (VP) is usually a SystemC design compiled from highly abstracted models (e. g. transaction level models) to ensure high simulation performance as well as cycle accurate modules where the exact timing is needed to ensure the system's function.

In a later design stage, for synthesis, these cycle callable models are converted to an HDL like VHDL or Verilog. Developing the cycle callable model in an HDL in the first place is usually not an option: While there is a free simulator for pure SystemC designs, tools that allow a cosimulation of SystemC and VHDL or Verilog have high license costs. Moreover, the conversion is a manual, time-consuming, and error-prone task: In case there are changes in either the specification or the implementation, both models need to be adapted consistently, which requires a tight interaction between the system designer and the RTL designer.

While automatically generating a synthesizable HDL design from a cycle callable SystemC design (or vice versa) is possible (e. g. [1]), this approach still requires the designer to keep the implementation consistent to the specification. In this paper, we present an approach to generate both a cycle callable SystemC model and a synthesizable VHDL model from a graphical specification given as a State Chart. State Charts provide several advantages over traditional code entry: They can directly serve as documentation, they are well-known from UML, they ease the modeling of complex reactive systems, and, because UML defines a standardized file format, their usage is more or less tool-independent.

In contrast to existing work, on generating either SystemC or HDL from State Charts, our approach ensures consistency between the SystemC and HDL implementation while employing two different modeling styles for SystemC and VHDL to focus on both a computationally efficient implementation for SystemC and a resource-efficient implementation in VHDL.

A. Introduction to State Charts

The following section presents a quick overview over the syntax and semantics of State Charts. For an in-depth explanation, refer to e. g. [2], [3], [4], [5].

State Charts are an extension to finite automata that adds hierarchical nesting of states and concurrent execution[1] [2]. Additionally, behavior can be attached to both states and transitions.

To introduce both hierarchy and concurrent execution (orthogonality), UML defines both *states* and *regions* [5]. States are divided into *simple states*, *composite states*, and *submachine states*. Since submachine states are semantically identical to composite states, we do not elaborate on those in this work. Composite states differ from simple states in the fact that they contain one or more regions: A region is an orthogonal part of a composite state [5] and consists of transitions and mutually exclusive substates, i. e. at most one of those substates can be *active* at any given time. At most one of those states can be an initial state, which is the state in which the region starts its execution[2]. A *configuration* is the set of currently active states, i. e. the set of states the State Chart is currently in.

[1]Additionally, State Charts allow broadcast communication. Due to the restriction to a single clock source for the input events, this is not relevant for this paper, though.

[2]Note that our approach requires a region to have exactly one initial state.

Note that original State Charts used a different denotation [3]: Composite states containing exactly one region correspond *OR-state* while those with more than one region correspond to *AND-states*.

One can attach behavior to states by specifying one of the following:

- an *entry action* that is executed every time the state is entered,
- a *do activity* that is executed after the entry action and while the State Chart is in the state, and
- an *exit action* that is executed every time the state is left [5].

Note that the presented approach does not support do activities, since their main use is to model continuous behavior, which, in a cycle callable design, needs to be refined into single steps that can be modeled, for example, using hierarchy or self loops. Do activities are therefore omitted in the further discussion.

States can be connected by transitions, that can (but are not required to) have

- a *trigger* that fires the transition,
- a *guard* that disables the transition if it evaluates to false when the trigger occurs, and
- an *effect* specifying behavior that is executed every time the transition is taken [5].

Transitions are written as $s \xrightarrow{trigger[guard]/effect} s'$ where s and s' are denoted the *source* state and the *target* state, respectively. Since our approach targets synchronous cycle-callable designs, the only allowed input event *trigger* is the positive clock edge and therefore omitted where not needed for clarity. Note that *completion transitions* pose an exception to this rule: Any transition without explicit trigger and guard starting from a composite state is taken as soon as the state is finished executing, i. e. as soon as all it's substates are in their final state.

Let $entry(s)$ and $exit(s)$, denote the entry action and exit action of a state s, respectively. If a transition $s \xrightarrow{e[guard]/effect} s'$ is executed because the event e is received while *guard* is true,

1) $exit(s)$,
2) *effect*, and
3) $entry(s')$ are executed in this order.

As an example, Fig. 1 shows a simplified State Chart. It consists of six states (State$i$, $1 \leq i \leq 6$) with State2 being a composite state containing the two regions S2a and S2b.

The State Chart starts its execution in State1 and changes to State2 when the guard start is true while an input event (i. e. a rising clock edge) is received[3]. Since State2 is decomposed into two regions that are executed concurrently, as soon as State2 is entered, the State Chart also enters the states State3 and State5. In other words, the configuration $\{\text{State1}\}$ is changed to $\{\text{State2}, \text{State3}, \text{State5}\}$ as soon as the guard start is true while a rising clock edge is received. As with every transition, as soon as State1 is left, $exit(\text{State1})$ and then $enter(\text{State2})$, $enter(\text{State3})$, and $enter(\text{State5})$ are executed.

State2 can be left in two cases:

- If abort becomes true, State2 is left for State6, regardless of the states the regions S2a and S2b are currently in.
- As soon as both regions S2a and S2b enter their final states, State2 is left for State1 through the completion transition.

[3]Note that, since the rising clock edge is the only valid input event, it is omitted in the State Chart.

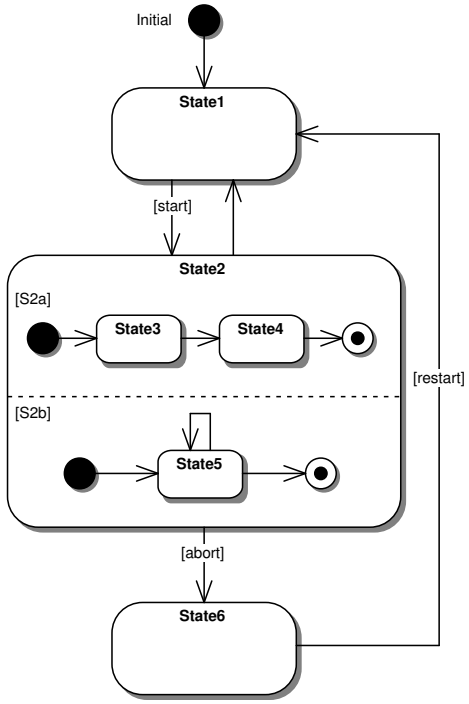

Fig. 1. A simplified State Chart.

II. Related Work

Since State Charts were first proposed by Harel [2] and several approaches to formalize their semantics were made (see, for example, [3], [4], [5], and the comparisons in [6] and [7]), much effort was dedicated to automatic generation of executable models from them or similar representations, which lead to several commercial products such as MATLAB Stateflow and IBM Rational Statemate. Other approaches are available that generate models for formal verification [8], [9] and, most related to our approach, in hardware description languages [10], [11] and SystemC [12], [13], [14].

In contrast the the approaches presented in [10] and [11], our approach generates synthesizable VHDL code (i. e. on the register transfer level) instead of behavioral code which requires an additional manual refinement step before being implementable in hardware.

The SystemC modeling style presented in this paper is similar to the approach presented in [13] but focuses on a more computationally efficient implementation: Our implementation requires only a single `SC_METHOD` for any State Chart, regardless of its hierarchy or number of parallel regions. This,

as mentioned in [15], reduces the amount of context switches and therefore improves the simulation performance.

III. Semantics

As mentioned before, State Charts were first proposed by Harel [2] in a rather informal manner. Since then, several approaches with different semantics were developed, most notably the semantics for Statemate [3], Rhapsody [4], and UML [5]. Since there are many unobvious corner cases to be considered and our approach, in some cases, deviates from the conventional semantics, this section describes the semantics based on important distinctions previously identified in [6] and [7].

a) Perfect Synchrony Hypothesis: The perfect synchrony hypothesis asserts that the output for a given input event is computed instantaneously and therefore occurs at the same time as the input. This implies the zero-time assumption, which states that transitions always complete in zero time [7].

The State Chart variant supported in our approach does not allow arbitrary trigger events: All transitions are either directly or indirectly[4] triggered by a clock edge. Therefore, using the slightly relaxed definition that the perfect synchrony hypothesis also holds if all events are processed before the next input event (i. e. clock edge) occurs [6], which is obviously the fact in a synchronous system, the hypothesis holds in our approach. Using the same definition, the *zero-time assumption* holds in our approach, too.

Additionally, since all transitions are implicitly triggered by an event on the clock input, there is no need for a *distinction between internal and external events* or for support for the *conjunction, disjunction or negation of events*.

b) Simultaneous Events: Classical State Charts allow handling simultaneous events (i. e. events that occur at the same time) simultaneously, while the UML semantics queue the events and handle them one after the other [7]. Since, in our variant, there is only one single input event source, however, there is no need to allow simultaneous events. For the same reasons, *negated trigger events*, which are needed to determinize otherwise nondeterministic transitions and handling of inconsistencies between a transition's effect and its cause are not necessary. Additionally, the distinction between *preemptive and non-preemptive interrupts* is reduced to defining priorities between different enabled transitions [6], which will be discussed below.

c) Causality: Our approach respects causality in the sense that the State Chart cannot create events that trigger itself. This is trivially true since the only input event supported is derived the State Chart's clock.

d) Inter-Level Transitions and State Reference: Both are currently not supported in our approach. Nonetheless, since our focus is on intuitive semantics rather than compositional semantics, those features could easily be added.

[4]A completion event, which is needed to trigger a completion transition, is indirectly triggered by the clock edge that caused the corresponding region to move to its final state.

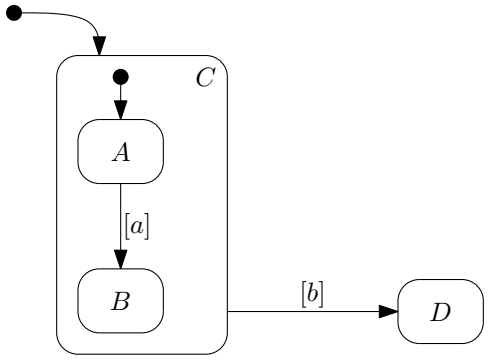

Fig. 2. Transition priorities: If both a and b evaluate to true, the State Chart will move to State D, in accordance with classical State Charts.

e) Self-start and self-termination: Both are not supported since, as mentioned before, the State Chart cannot create events that trigger itself.

f) Instantaneous States: We do not allow instantaneous states in our approach; this conforms to the UML standard and is sensible for a cycle callable implementation: The minimum time a State Chart can be in a given state is one cycle. This implies that simple states cannot have outgoing completion transitions: A simple state's entry action is guaranteed to execute in zero time, and therefore the completion transition would need to be executed immediately after entering the state [5], thus resulting in an instantaneous state. Therefore, our approach does not support completion transitions starting in simple states.

Since instantaneous states are not supported, more than one transition can only fire at a single point of time if all those transitions are in mutually orthogonal regions [6]. Therefore, *parallel execution of transitions* inside a single region, *transition refinement*, and, obviously, *multiple entered or exited instantaneous states* are not supported and an *infinite number of transitions at an instant of time* cannot occur.

g) Determinism: The presented approach does not enforce determinism. The code generator nondeterministically chooses a prioritization between different transitions starting from the same state. However, the formal verification presented later in this paper will detect nondeterministic behavior.

h) Priorities for Transition Execution: We adhere to the semantics of classical State Charts regarding the priorities for transition execution, i. e. a transition $t_1 : s_1 \rightarrow s_1'$ has priority over any transition $t_2 : s_2 \rightarrow s_2'$ if s_2 is a (direct or indirect) substate of s_1 [3]. Fig. 2 shows such an example: Since C is a superstate of A, transitions originating in C have priority over those originating in A; therefore, if the State Chart is in state A and both guards (a and b) evaluate to true, the State Chart will move to state D.

Note that this prioritization does not conform to the UML standard, which defines that transitions originating from lower-level states have priority. This decision is based on the fact that, in hardware design, transitions originating from higher-level states are usually used to abort a sequence of actions or a calculation and should therefore not be blocked by lower-level

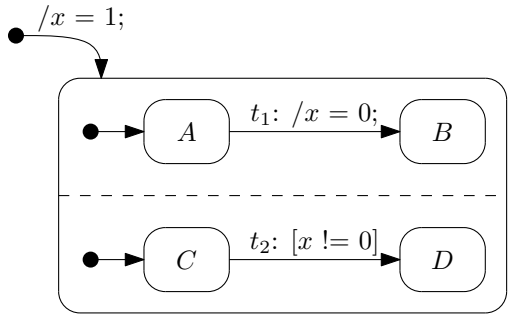

Fig. 3. Due to the variable semantics of internal storage, the execution of transition t_2 depends on the execution order.

transitions.

i) Timeout Events: Our approach supports a concept similar to timeout events: A special guard can be added to a transition to count the number of clock cycles the transition's source state was active and leave the state after a certain amount of cycles has passed.

j) Signal Semantics: For performance reasons, our approach allows the usage of storage with signal semantics only for inputs and outputs of the State Chart, and of storage with variable semantics only inside the State Chart. Therefore, our approach follows the UML specification [5] for the execution of actions, which defines that the statements of an action are executed sequentially. This is in contrast to the traditional State Chart semantics [3], which define the statements of an action to be executed in parallel.

On the other hand, this decision leads to the fact that transitions contained in orthogonal regions may nondeterministically influence each other: Consider, for example, the State Chart in Fig. 3 being is in the states A and C. The execution order of t_1 and t_2 is nondeterministic—if t_2 is evaluated before t_1, it is not enabled. If, on the other hand, t_1 is evaluated first, it changes t_2's guard and therefore, when t_2 is evaluated later, it is enabled. Therefore, the designer is required to determinize the behavior accordingly, for example with additional guards.

k) Fork and Join, History: While supporting parallel regions, our approach does not support fork and join constructs. Currently, history is not supported either.

l) Choice: In contrast to the UML standard, our approach only supports static choice, dynamic choice is not supported. Additionally, we do not support pseudostates, as initial states.

IV. Implementation

A. Design Entry

Usually, the motivation to use a code generator is to get improved code quality due to consistency with the specification and to reduce or in the best case avoid the effort of manually writing the code. Still, a suitable tool for editing the specification is required.

In our approach, we decided to use a conventional UML editor for the design entry. The editor shall support model export via XMI, which can be processed by the generator. While UML offers a variety of elements, our generator supports a subset of those elements only: This includes simple states, transitions, junctions, initial states and final states. Parallelism shall be modeled using regions while composite states enable the user to create hierarchy. Supported behavioral elements include state entry and exit actions, transition guards and effects, and time triggers. The syntax of the action language used for behavioral elements is based on C but adds features such as time triggers.

As our generator will create synchronous code, we presume that all transitions are implicitly triggered by the design's clock. As mentioned before, there is one exception from this rule: Completion transitions are triggered when the child state machines have reached their final state. Non-completion transitions are capable of interrupting the child state machine at any time.

B. Code Generation

The main constraint for the code generator presented in our approach is the fact that the generated SystemC and VHDL code shall be consistent. While this consistency could be checked via cosimulation of both models, ensuring consistent code generation will save costs and resources needed for this check. While developing the concept for the code generator, several topics have been identified as issue with impact on code consistency and will be discussed below:

- different namespace concepts
- different semantics of bitfield interfaces
- different modeling styles in VHDL and SystemC

Both SystemC and VHDL generator use namespaces to partition the State Chart into distinct units for each hierarchy level and thereby avoid name clashes. Alternatively, name resolution is possible via using distinct names for each object. Nevertheless, in our approach we use namespaces, since the generated code is structured in a modular way and is easier to read and to understand.

In SystemC, each level of a state machine is implemented in its own SytemC class. Hierarchy is achieved by defining nested classes and instantiating objects of these classes within the context of the upper level state machine class. Due to leveraging C++ class scope mechanism, name clashes are avoided in a clean way.

In contrast to SystemC, classes are not supported in VHDL and therefore a different concept is needed. Nevertheless, VHDL offers packages to pack various objects like types, constants, and procedures into one common scope. Thus packages have been leveraged to implement the state machines; each level of a state machine is placed into a single packages including the relevant state record and the procedures implementing the State Charts functionality. To model hierarchy, references to the state records of child state machines are included into the state record of the parent state machine.

In SystemC, the State Chart is modeled in a single process model, whereas the VHDL implementation utilizes a conventional two-process model. The SystemC implementation is optimized for execution speed and therefore shall trigger as few context switches as possible, which can be achieved by

reducing the State Chart to one single process. Connectivity to the outside world (i.e. external bitfields, hardware signals) is achieved via SystemC ports. In the SystemC implementation, external bitfields are plain SystemC signals and therefore update their value one delta cycle after the state machine assigned the new value.

On the contrary, in the VHDL implementation external bitfields are modeled with flipflops, therefore requiring data and enable signals. These signals may not be registered, otherwise the external bitfields will get their new value one cycle after the bitfields in the state machine. Thus the VHDL implementation uses a two-process model (combinatorial and sequential process), where the combinatorial process drives the asynchronous signals and the sequential process implements the internal state.

C. Evaluation Cycle

To implement the State Chart, we generate either SystemC or VHDL code that evaluates the current state and inputs on every clock cycle. Due to the support for hierarchy, the patterns for the generated code are indirectly recursive: The code implementing a composite state will contain a copy of the code implementing a region for every region in the composite state. Similarly, the code implementing a region will, if the region contains another composite state, include a copy of the code implementing that composite state.

Fig. 4 presents the rather straight-forward way used to implement composite states in our approach: The composite state's code evaluates its contained regions, and returns the information whether all of them have entered a final state to its containing region to indicate whether a completion transition leaving this state is enabled.

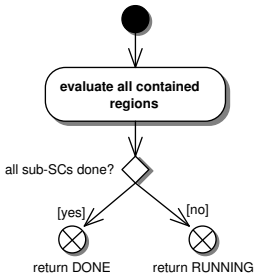

Fig. 4. Evaluation cycle for composite states.

Fig. 5, on the other hand, shows the more involved implementation of regions. When the region is evaluated, it first checks if it had already reached a final state; this is needed to synchronize several regions in one composite state: Regions running in parallel might finish at different points in time, requiring the faster ones to wait for the slower ones to finish. In this case, the faster region simply stops its evaluation immediately and returns `DONE`.

The code implementing the region is passed a parameter `force_exit` that determines if region is to be left, which would, for example, be the case if the composite state containing the region is left. In this case, all currently active contained composite states are evaluated with `force_exit` set to true, which means that only their respective exit actions are executed.

The phase "evaluate local transitions" checks whether any local transitions inside the current region are enabled (i. e. have a guard that evaluates to true). If more than one transition is enabled here, the transition that is executed depends on the order in which the transitions are checked and is therefore chosen nondeterministically[5]. If the State Chart contains active composite states, they are evaluated next (starting in the evaluation cycle for composite states, therefore making the evaluation cycle indirectly recursive). Note that local transitions are checked before completion transitions, and, therefore, have a higher priority.

If, after evaluating the local transitions and the completion transitions, no enabled transition was found and `force_exit` is false, the region returns `RUNNING` to indicate to its caller that it still running; otherwise, the current state's exit action is executed. If the region stays active, it will execute the transition, including its effect.

Should the transition lead to a final state, the region returns `DONE` to indicate that it finished processing and the containing composite state might be able to execute a completion transition. If, on the other hand, the transition leads to a normal state, the state is reset (which, for a composite state, includes setting its current state to its initial state) and, finally, the new state's enter action is executed.

V. Ensuring Consistency

As the goal of our generation approach is to automatically generate consistent design implementations in different design languages, a methodology to ensure consistency was developed. In Fig. 6, the framework for checking consistency is shown. Note that the methods presented here are checks to increase the confidence in the generator: As soon as the generator has matured enough, the generated models could also be assumed to be correct by construction.

A common way to check consistency between two different models is to run a cosimulation. Both models are instantiated in one common testbench, their inputs are fed with identical input symbols and the output symbols are compared. By leveraging constrained-random techniques, the ability to detect mismatches in corner cases is improved and coverage is increased.

Nevertheless, the result of a cosimulation is only valid if one of the models can be assumed to be correct, i. e. to be the golden model. This means, that for one model another check against the specification needs to take place. One way to verify a model against a specification is property checking. Design properties, derived from the design specification are verified by means of formal tools. Moreover, these properties could also be reused for simulation-based verification.

In our case, the generator is capable of transforming the specification into SystemVerilog assertions. Currently, each

[5]As mentioned before, formal verification based on generated assertions will be able to identify such cases.

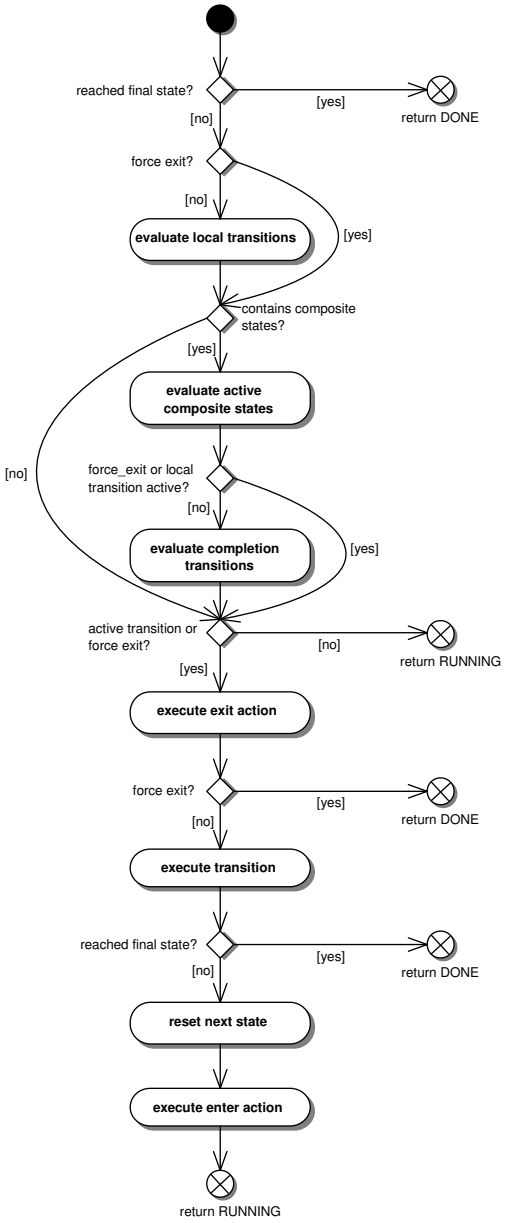

Fig. 5. Evaluation cycle for regions.

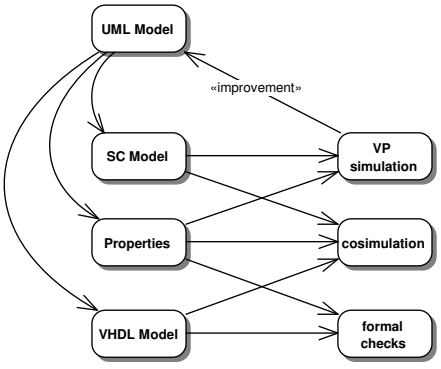

Fig. 6. A framework to ensure consistency of generated models.

state s in the model is taken into account, and for every outgoing transition $s \xrightarrow{[g]} s'$ of the state a property

$$(s \in \textit{Configuration}) \wedge g \ \mid\!=\!> \ (s' \in \textit{Configuration})$$

is created. Additional properties, especially concerning the output behavior of the State Chart (i. e., checking exit and entry actions as well as transitions' effects) as well as special cases such as variable accesses and time-triggered transitions will be included in future versions of the generator. The resulting set of properties is used to formally prove the correct state transitions in the VHDL model. As stated before, the SVAs can also be used for functional verification. Moreover, if they are generated in an assertion language for SystemC (as, e. g., presented in [16]), they can also be used in the SystemC virtual prototype simulation.

Additionally, the generator is able to create a common SystemC testbench, prepared to be run with either the OSCI reference simulator or a HDL simulator capable of mixed-language simulation. As the VHDL model is proven formally, it acts as the golden model in this cosimulation to ensure the consistency of the SystemC model.

VI. Results

During pilot testing of the methodology, we noted a major impact on design productivity of the design team. Taking into account, that a manual implementation requires

1) knowledge transfer between concept engineering and design teams,
2) implementation of the State Chart in HDL, and
3) verification to detect implementation bugs and/or misinterpretation of design intent,

an automatic generation flow offers a high potential to decrease the pressure of tight project plans on the design teams. According to feedback from designers a productivity gain of up to one man week for an average State Chart (of course depending on the State Chart's complexity) can be achieved by rendering VHDL implementation automatically rather than manually writing it.

The methodology has been applied to three different State Charts:

- a small test design (proof of concept),
- a minor industrial example (industrial example #1), and
- an average industrial example (industrial example #2).

For each of those examples, the SystemC and VHDL implementations as well as the SystemVerilog assertions were generated.

For each of the generated properties the formal tool used reported a runtime between 0.05 sec and 0.10 sec. Nevertheless, the overall runtime for the formal checks includes preparation steps as well (e.g. memory allocation). Therefore we measured the total runtime using the TCL command "time". The checks have been run with the first two State Charts, but could not be run with the average industrial example since not all features of the State Charts are currently supported by the SVA generator. Table I shows the results of the performance measurements.

The two small examples with 10 properties each, have an average runtime of 5.26 seconds. We assume the methodology to scale linearly with the number of properties, which would lead to an estimated runtime of around 28.40 seconds for industrial example #2. While runtime of formal tools usually increases exponentially with design complexity, the very short runtimes given in Table I indicate that also more complex examples should be verifiable with the given approach.

TABLE I
RUNTIME PERFORMANCE OF FORMAL CHECKS

SC	# of properties	runtime (sec.)
proof of concept (POC)	10	5.62
industrial example #1 (IE1)	10	4.90
industrial example #2 (IE2)	54	28.40 [6]

With regard to consistency, especially a comparison of the simulation runtime of the different modeling styles is of interest. Table II shows the results from the runtime measurements. First, the runtime of the simulation using the OSCI simulator ($t_{SC@OSCI}$) and the HDL simulator ($t_{SC@HDL}$) has been measured. Second, the simulation runtime of the same testbench instantiating the VHDL model using a HDL simulator has been measured ($t_{VHDL@HDL}$). The speed-up factor is defined to be $S = \frac{t_{VHDL@HDL}}{t_{SC@OSCI}}$. It can be seen from the table, that, depending on the application, the SystemC implementation leads to a small speed-up compared to the VHDL implementation. Additionally, it allows the usage of the freely available OSCI simulator, which showed the best simulation performance in our tests. Moreover, the SystemC model can also be reused in high-complexity system simulations, where the performance of the HDL simulator is expected to drop further.

TABLE II
SIMULATION PERFORMANCE

SC	$t_{SC@OSCI}$ (sec.)	$t_{SC@HDL}$ (sec.)	$t_{VHDL@HDL}$ (sec.)	speed-up
POC	0.25	0.739	0.569	2.27
IE1	0.40	0.490	0.550	1.38
IE2	0.46	0.590	1.53	3.33

It is noteworthy that the second presented industrial example (IE2) is already silicon proven in a 2G/3G transceiver and will also be included in an upcoming LTE transceiver.

VII. CONCLUSION

This paper presented an approach to generate both cycle callable SystemC and register transfer level VHDL code from a graphical specification given as a State Chart. Differences in the generated models' respective use cases are identified and addressed: The SystemC model uses a single-process modeling style to improve the simulation speed for virtual prototyping over existing solutions while the VHDL model uses a conventional two-process modeling style comprising a combinatorial and a sequential process to improve the resource usage for RTL synthesis. While their implementation differs, both models behave consistently. Models generated with the presented approach are already silicon proven.

As the set of generated SystemVerilog assertions is not yet complete, the generator will be extended to create a complete set of properties. Further work will also include aligning a transaction-level modeling style, e. g. as presented in [15], to the cycle callable modeling style presented in this paper to facilitate a semi-automatic refinement from the transaction level to the register transfer level. Moreover, it is planned to introduce a sequence generator derived from the UML model, which allows to stimulate all possible transition paths through the State Chart model in a controlled manner.

Finally, it is noteworthy that due to the standardization of State Charts and their XML representation, the generator is independent of the State Chart input tool, avoiding vendor lock-in and ensuring future support for the given approach. Also, the presented approach is based on a clearly defined meta model and therefore readily extensible to other code generation applications, such as Verilog, SystemVerilog, or even firmware in C/C++.

[6] Estimated, refer to text.

REFERENCES

[1] N. Bombieri, G. Di Guglielmo, L. Di Guglielmo, M. Ferrari, F. Fummi, G. Pravadelli, F. Stefanni, and A. Venturelli, "HIFSuite: Tools for HDL code conversion and manipulation," in *Proc. of HLDVT 2010*, Jun. 2010, pp. 40–41.

[2] D. Harel, "Statecharts: a Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.

[3] D. Harel and A. Naamad, "The STATEMATE semantics of statecharts," *ACM transactions on software engineering and*, vol. 5, pp. 293–333, 1996.

[4] D. Harel and H. Kugler, "The RHAPSODY Semantics of Statecharts (or, On the Executable Core of the UML)," in *Integration of Software Specification Techniques for Application in Engineering*, ser. Lect. Notes in Comp. Sci., vol. 3147. Springer-Verlag, 2004, pp. 325–354.

[5] *OMG Unified Modeling Language: Superstructure, Version 2.2*. Object Management Group, 2009.

[6] M. Von Der Beeck, "A comparison of statecharts variants," *Lecture Notes in Computer Science*, pp. 1–25, 1994.

[7] M. L. Crane and J. Dingel, "UML vs. classical vs. rhapsody statecharts: not all models are created equal," *Software & Systems Modeling*, vol. 6, pp. 415–435, January 2007.

[8] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann, "Implementing statecharts in PROMELA/SPIN," pp. 90–101, 1998.

[9] D. Latella, I. Majzik, and M. Massink, "Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker," *Formal Aspects of Computing*, vol. 11, pp. 637–664, December 1999.

[10] S. Qin and W.-N. Chin, *Mapping Statecharts to Verilog for Hardware/-Software Co-specification*, 2003, pp. 282–300.

[11] V.-A. V. Tran, S. Qin, and W. N. Chin, "An Automatic Mapping from Statecharts to Verilog," *Theoretical Aspects of Computing - ICTAC 2004*, pp. 187–203, 2005.

[12] M. Mura, M. Paolieri, L. Negri, and M. G. Sami, "StateCharts to systemc: a high level hardware simulation approach," *Proceedings of the 17th ACM Great Lakes symposium*, pp. 505–508, 2007.

[13] M. Mura and M. Paolieri, "SC2 StateCharts to SystemC: Automatic Executable Models Generation," in *Embedded Systems Specification and Design Languages*, 2008, pp. 227–239.

[14] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio, "A model-driven design environment for embedded systems." San Francisco, CA, USA: ACM, 2006, Conference proceedings (article), pp. 915–918.

[15] R. Findenig, T. Leitner, M. Velten, and W. Ecker, "Transaction-level State Charts in UML and SystemC with zero-time evaluation," in *Proceedings of DVCon 2010*, February 2010, pp. 13–19.

[16] V. Esen, "A new assertion language covering multiple levels of abstraction," Ph.D. dissertation, Technische Universität München, 2008.