Transaction-Level State Charts in UML and SystemC with Zero-Time Evaluation

Rainer Findenig [#1], Thomas Leitner [*2], Michael Velten [†3], Wolfgang Ecker [†4]

[#] *Upper Austrian University of Applied Sciences*
Hagenberg, Austria
[1] rainer.findenig@fh-hagenberg.at

[*] *DICE GmbH & Co KG*
Linz, Austria
[2] thomas.leitner@infineon.com

[†] *Infineon Technologies AG*
Neubiberg, Germany
[3] michael.velten@infineon.com
[4] wolfgang.ecker@infineon.com

***Abstract*—UML State Charts provide an effective and intuitive means for the design entry of hardware systems. Several approaches exist to generate executable code in a variety of languages from UML State Charts, for hardware design most notably SystemC and Verilog.**

Since State Charts are especially interesting for designing virtual prototypes for early system simulation on the transaction level, this paper provides a methodology to allow the use of transactions inside State Charts and automatically generate executable SystemC code from them. Compared to previous approaches, our translation minimizes the amount of SystemC kernel interaction to improve the simulation performance.

Based on a real-life and a best-case example we show that the speedup of code generated using the proposed approach compared to a conventional implementation averages around 2 to 30 while the generated code is still compatible with standard transaction-level models.

I. Introduction

Today's virtual prototyping design flow is often based on a top-down approach that uses transaction-level (TL) models for early hardware and software evaluation. TL modeling uses two main concepts to increase the simulation performance: communication is abstracted using method calls and timing is modeled only approximatedly. This allows several modeling styles: for hardware and architecture evaluation, timing is modeled accurately while for software testing, where simulation speed is crucial, timing may even be completely omitted.

While TL modeling allows a nearly arbitrary modeling style, the use of a formalism like State Charts it is still favorable for modeling control flow: they provide an easy and intuitive way of graphically entering complex designs for both specification and, given the necessary tool support, implementation, while still maintaining a high level of abstraction.

We propose a formalism for "transaction-level State Charts" which allows generating SystemC code from UML State Charts that can use transactions both inside states and as events and effects in transitions. Additionally, the proposed approach is developed with great focus on high simulation performance of the generated model while still maintaining interoperability with legacy transaction-level models.

A. Execution Contexts

For this paper, we define the term *execution context*, or *context* as a shorthand, to mean any SystemC construct that can serve as a vehicle for executing code; in essence, this covers both SystemC methods (`SC_METHOD`s) and SystemC threads (`SC_THREAD`s). Note that we do not consider SystemC clocked threads (`SC_CTHREAD`s) here since they need to be sensitive on a clock [1] which is not sensible for most TL environments. Additionally, as a simplification, we use the term *State Chart* not only for the graphical UML representation but also for its implementation in SystemC.

In a most basic system containing a State Chart, at least two different execution contexts are present (see Fig. 1): one for the system's controller (e. g. an instruction set simulator) and one for the State Chart[1]. As we will present in section II, traditional approaches mostly use SystemC events to trigger the State Chart whenever a transition should be executed. The notification of the event causes the SystemC kernel to schedule the State Chart's context which can then react according to the event that was generated. While this allows a straightforward implementation of the State Chart in SystemC, it causes a context switch and therefore significant overhead every time the State Chart is triggered.

B. Blocking and Non-Blocking Behavior

In this paper, we will use the term *behavior* to denote an arbitrary piece of program code that may contain calls to transactions as well as e. g. calculations. Analogously to transactions, we define a given behavior to be either non-blocking or blocking:

[1]Note that, while some translation algorithms for state charts define more than one execution context for a State Chart (e. g. [2], [3]), our approach always uses exactly one.

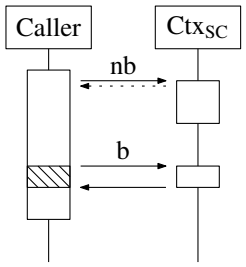

Fig. 1. A basic system consisting of a system controller and a State Chart contains at least two execution contexts. The caller (e. g. the system's controller) can use non-blocking ("nb") or blocking ("b") transactions to trigger behavior in the State Chart's context ("Ctx$_{SC}$").

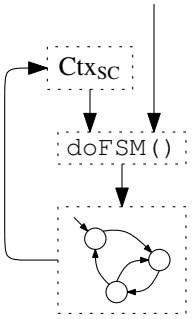

Fig. 2. Implementation of a State Chart: `doFSM()` is a simple "scheduling layer" that maintains the State Chart's execution consistency while allowing it to be called from more than one execution context. Again, "Ctx$_{SC}$" denotes the State Chart's context.

- Non-blocking behavior subsumes all calculations and calls to transactions that guarantee to complete without directly or indirectly calling `wait()`. This means that non-blocking behavior must not contain calls to blocking transactions.
- Blocking behavior, on the other hand, may call `wait()`, either directly, e. g. to approximate a calculation's timing, or indirectly, e. g. by calling a blocking transaction.

Note that this abstract definition allows our approach to be applied to all abstraction levels commonly subsumed under the term transaction level. However, the best speedup can be achieved in programmer's view implementations since they conventionally use less blocking behavior.

To increase the execution speed of the simulation model, our approach aims to reduce the number of context switches needed by executing parts of the State Chart's behavior directly in the caller's context. Since both contexts have their own notion of elapsed time, though, this is only possible for non-blocking behavior: blocking behavior would otherwise block the wrong context, i. e. the caller's instead of the State Chart's. Therefore, all non-blocking behavior can safely be executed in an arbitrary context, i. e. without a context, while blocking behavior needs to be executed in the State Chart's context and this possibly requires a context switch.

C. Scheduling Layer

To allow the caller to directly execute the State Chart, our approach provides an additional "scheduling layer" (designated `doFSM()` in Fig 2). This layer allows the State Chart to be executed both from its own context and from a foreign one while maintaining the integrity of its execution; this most notably includes the fact that while the State Chart's context is blocked, it must also not be executed in a foreign context.

II. Related Work

Since State Charts were first proposed by Harel [4] and several approaches to formalize their semantics were made (see, for example, [5], [6], [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], in hardware description languages [10], [11] and, most related to our approach, SystemC [12], [2], [3], [13], though not necessarily directly targeting transaction-level models.

Compared to the approach presented by Mura et al. [2], our approach mainly focuses on higher simulation performance: they model events as `sc_events()` and map each state to either a single SystemC thread or two SystemC methods, while our approach uses callbacks and a single SystemC thread per State Chart to significantly reduce the amount of context switches and therefore the computational overhead introduced by the scheduler. Also, they observe that SystemC threads are more computationally intensive than SystemC methods.

In further work, Mura et al. present an approach that significantly reduces the amount of concurrent execution contexts required to implement a State Chart [3]. Moreover, they limit the code generation to SystemC methods because of their previous findings regarding their performance. This, however, is not possible for our approach since standard transaction-level models rely on the SystemC statement `wait()`, which is only supported in SystemC threads, to model blocking transactions.

Apart from efforts to generate code from State Charts, the approaches presented in [14], [15] motivate the use of State Charts with the easier verification of the refinement from transaction-level to register-transfer-level code.

III. Transaction-Level State Charts

This section first outlines the basics of UML State Charts [16] and then discusses our main contribution: a translation for UML State Charts to SystemC which supports the use of transactions inside the State Chart and, at the same time, minimizes the required SystemC kernel interaction to increase the model's simulation performance.

A. UML State Charts

UML State Charts are a superset of finite automata, that, amongst other features, supports attaching behavior (function code instead of simple output symbols) both to states and transitions, making them a mixture of Mealy and Moore automata.

A State Chart[2] consists of states that can (but are not required to) specify behavior in the form of

- 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 [16].

The behavior upon entering a state is split into the entry action and the do activity because of their different semantics: actions always run to completion while activities are interruptible: an activity runs until it either finishes by itself or the state is exited, whichever comes first.

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 [16].

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

1) $\mathit{do}(s)$ is aborted (if it is still running), and then
2) $\mathit{exit}(s)$,
3) *behavior*,
4) $\mathit{entry}(s')$, and
5) $\mathit{do}(s')$ are executed in this order.

Fig. 3 shows a very basic UML Start Chart with the states `State1`, `State2`, and `State3`. `State1` defines a do activity called `do_state1` and an exit action `exit_state1`. Analogously, `State2` defines an entry action, a do activity, and an exit action while `State3` does not define any behavior.

The following presents a possible execution of the State Chart. After each step, the State Chart is in a *stable state*, i. e. it does not change the state until some event occurs.

1) `State1` is entered after startup and `do_state1` is executed.
2) As soon as `trigger1` occurs while `guard1` is true, `State1` is left, possibly aborting the execution of `do_state1`. Since `State1` is left, `exit_state1` is executed. Then, while the transition is executed, its effect `effect1` is run. When the State Chart enters `State2`, it begins executing `entry_state2`. As soon as this action is finished, `do_state2` is executed.
3) When the state `State2` is left because of an occurrence of `trigger2`, `exit_state2` is executed before `State3` is entered. Since it does not define an entry action or a do activity, no additional code is executed.

[2]Note that, for this paper, we do not consider advanced features of State Charts such as history or parallel states.

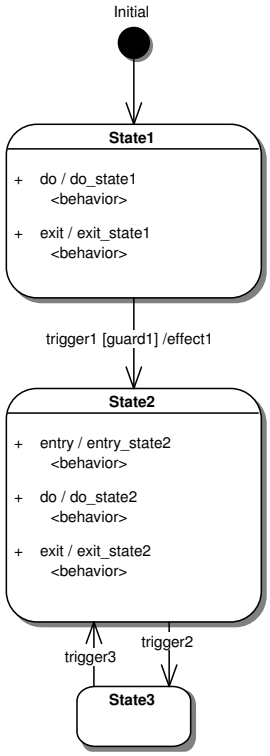

Fig. 3. A basic UML State Chart.

4) Finally, as soon as `trigger3` is received, `State2` is entered again, hence `entry_state2` and then `do_state2` are run again, and so on.

B. UML State Charts for TLM

SystemC transaction-level code generated from a UML State Chart consists of exactly one module containing a single execution context, the State Chart's implementation, and ports and exports that the State Chart can use to communicate with its surroundings via transactions. This basic structure is outlined in Fig. 4.

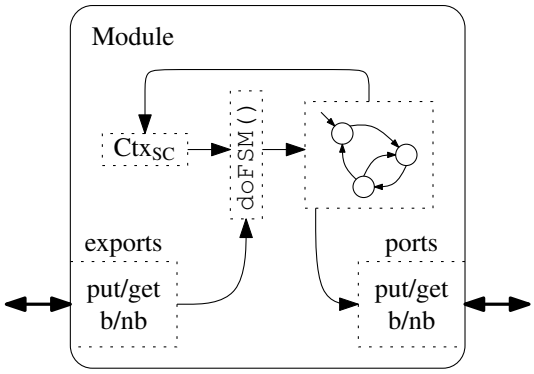

Fig. 4. A SystemC module containing a State Chart (see Fig. 2) that uses transactions to communicate with its environment. Again, "Ctx$_{SC}$" and `doFSM()` denote the State Chart's execution context and its scheduling layer, respectively.

For this paper, we will categorize the transactions supported

by the module by two distinctions:

- Blocking or non-blocking: Blocking transactions (designated "b" in Fig. 4) can block the caller (i. e., call `wait()`) while non-blocking transactions ("nb") always complete in zero time.
- Inside a port or an export: Transactions in exports can be used by the module's environment to trigger the State Chart and therefore intuitively correspond to its inputs. Transactions in ports, on the other hand, can only be called by the module itself and therefore correspond to the State Chart's outputs.

C. Transactions in Exports (Inputs)

Generally speaking, a State Chart's inputs are the set of all events that can trigger one of its transitions. With respect to the usage of transactions inside exports to trigger the State Chart, it is therefore necessary to derive such events from those transactions. The proposed approach uses a simple callback-based method to define timing points during a transaction, as shown in Fig. 5: a callback function is executed whenever the transaction reaches one of its defined timing points. Every callback function is executed in the caller's context and may directly call the the State Chart's implementation through the scheduling layer, as shown in Fig. 4, and therefore possibly cause state transitions and their associated actions without switching to the State Chart's context.

Since arbitrary timing points are supported, this approach also supports the phases as used in TLM 2.0.

```
void write (...) {
    cb_write(begin);      // define timing point 'begin'
    data = input;
    cb_write(end);        // define timing point 'end'
}
```

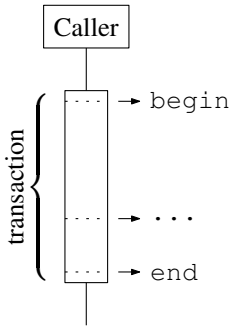

Fig. 5. To define timing points, callbacks are inserted at arbitrary locations inside a transaction. These timing points can be used as triggers inside the State Chart.

D. Transactions in Ports (Outputs)

As mentioned before, a State Chart can call only transactions defined it its ports. Also, those transactions correspond to its outputs, or, more accurately, to its behavior. There are four possibilities to add behavior to a State Chart [16], all of which are supported in our approach: entry actions, do activities, exit actions, and transitions' effects.

Note that Crane et al. observe that the UML standard does not require a transition to complete in zero time [7]. While UML does not define the interruptibility on a transition's effect, intuitively an effect runs to completion: while the State Chart is executing a transition, other inputs are ignored and therefore cannot interrupt the transition's effect.

1) Actions: To be able to reduce the context switches needed for the execution, we impose an additional requirement on actions: since entry and exit actions always run to completion, our approach additionally requires them to complete in zero time. This effectively restricts them to only contain non-blocking behavior but allows executing them in an arbitrary context: if an event e (which, as mentioned before, corresponds to the call of a callback function) triggers a transition $s \xrightarrow{e} s'$, then the sequence *exit(s)*, *entry(s')* can be executed without switching to the State Chart's context. Therefore, no context switches are necessary to do state transitions between states that only contain entry and/or exit actions.

The given restriction does not limit the State Chart's expressiveness for entry actions: blocking behavior is supported in a state's do activity, which, as mentioned before, is started as soon as its entry action is completed. Therefore, the blocking behavior can be moved to the state's do activity. However, note that, as mentioned above, UML defines entry actions to always run to completion and do activities to be interruptible. To ensure the semantic consistency, for this special case it is therefore necessary to support non-interruptible behavior inside a do activity. To this end, we add a special code construct that allows to temporarily disable the scheduling layer from interrupting the activity.

For exit actions, on the other hand, a syntactic transformation which adds an additional state is needed to support blocking behavior. Analogously to the transformation of the blocking entry action, this transformation also needs to ensure that the converted exit activity is not interruptible.

2) Activities: Since activities can, as mentioned before, contain blocking behavior, it is not possible to execute them in an arbitrary context. Rather, as soon as a do activity is encountered during the execution of the State Chart, the current context is checked. If the State Chart is executed inside its own context, the execution can continue. If, on the other hand, the State Chart is executed in the caller's context, a context switch is necessary since the execution needs to switch to the State Chart's context to avoid blocking the caller. To this end, a conventional SystemC event is used to trigger the State Chart's own context, which then continues the execution.

In other words, if a transition $s \xrightarrow{e} s'$ is executed, while the sequence *exit*(s), *entry*(s') can be executed in the caller's context, if s' contains a do activity, it must be executed inside the State Chart's context. Therefore, after the State Chart finishes executing *entry*(s), it encounters *do*(s) and uses a SystemC event to switch to its own context and resumes the execution there.

Note that the scheduling layer needs to mark the State

Chart as being blocked while the State Chart's own context is executing blocking behavior to avoid the State Chart's function being executed from inside any other context.

As an optimization, the do activity can be statically analyzed: If it can be proven to be non-blocking, it can be appended to the entry action to avoid the otherwise necessary context switch.

3) Effects: Similarly to entry actions, non-blocking behavior inside a transition's effect can be executed inside the caller's context: for a transition $s \xrightarrow{e/behavior} s'$, the sequence *exit*$(s)$, *behavior*, *entry*$(s')$ can be executed without switching to the State Chart's context if *behavior* is non-blocking.

Blocking behavior, on the other hand, needs special care: as with activities, a context switch to the State Chart's context is necessary. We propose using a syntactic transformation to blocking behavior, as outlined in Fig. 6. The transition's effect is moved into the do activity of an additional state: Every transition $s \xrightarrow{e/behavior} s'$, where *behavior* is blocking, is modified to two transitions $s \xrightarrow{e} s''$ and $s'' \rightarrow s'$ and *behavior* is added as the do activity of s''. Therefore, the sequence of execution is as follows: *exit*(s), *do*$(s'') =$ *behavior*, *entry*(s'), but because of the do activity in s'', a context switch is inserted right before it. Note that, for the transition $s'' \rightarrow s'$, this transformation relies on the support of completion transitions as described in the following section.

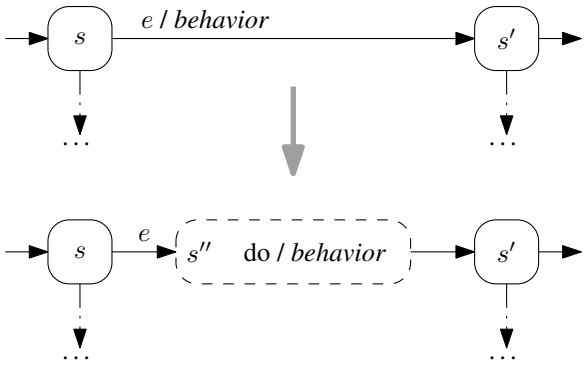

Fig. 6. Any transition containing blocking behavior inside its effect is syntactically converted to two transitions with an additional state containing the behavior as its do activity.

This also solves the following problem: If a transition's effect contains blocking behavior, the transition obviously does not complete in an indefinitely short amount of time, which entails the question of the State Chart's current state during the transition. Intuitively, the transition's source state was already left but the transition's target is not yet reached: The additional state contains neither the transitions of the source state nor those of the target state. Therefore, the State Chart does not react on any triggers of those transitions, which corresponds to the intuitive behavior while executing a transition that consumes time.

E. Support for Completion Transitions

If a transition does not define a trigger, it is called a *completion transition* (which corresponds to an ϵ-transition in a finite automaton). Completion transitions are implicitly triggered by completion events that are emitted by a state as soon as its entry action and do activity are finished. If a state does not specify an entry action or do activity, the completion event is emitted as soon as the state is entered [16].

Conventionally, such a transition introduces the need for a context switch. To avoid that, the state machine is immediately reevaluated until it is in a stable state, i.e. all possible next transitions depend on external events.

This, in essence, allows more than one transition to be done in a single step, without switching to the State Chart's context. Consider the transitions $s_0 \xrightarrow{e/behavior_0} s_1 \xrightarrow{behavior_1} s_2 \ldots$ where no state s_i and no behavior $behavior_i$ contains blocking behavior: the sequence *exit*(s_0), $behavior_0$, *entry*(s_1), *exit*(s_1), $behavior_1$, *entry*(s_2), ... can be executed without ever switching to the State Chart's context. Note that the sequence of states may obviously also contain cycles.

IV. Case Studies

This section provides three case studies to quantify the speedup that is achieved by the proposed approach.

A. AES Core

The performance of our approach was investigated using a transaction-level AES core (see Fig. 7). The core provides a simple interface that allows both decrypting and encrypting data with a configurable AES key. A simple path through the State Chart is as follows: Beginning in the state `Idle`, the State Chart waits for the completion of a `write` transaction (denoted `write'end`) where the address is set to either `KEY` or `DATA`. Setting the address to `KEY` allows changing the key, the state `Idle` is not left. If the address is set to `DATA`, the State Chart changes its state to either `Decrypt` or `Encrypt` according to the additional parameter `cmd` of the `write` transaction. In the entry action of both states, the data is processed, and the state is automatically left as soon as the entry action is finished. Finally, the end of a `read` transaction causes the State Chart to enter the state `Idle` again.

While the State Chart is in the states `Decrypt` or `Encrypt`, the begin of a `write` transaction would cause the state to be left. In this case, this is not possible, since the entry action of the states always runs to completion and, because of the ϵ-transitions to `DataRdy`, the state is left as soon as all its actions are completed. If the encryption/decryption was done inside a do activity, it could include a call to `wait()` to approximate its timing. The transitions triggered by `write'begin` could then be used to abort the calculation.

It is noteworthy that apart from the State Chart given in Fig. 7, only the definition of the transactions has to be provided to generate a fully functional SystemC model of the core.

1) Results with Entry Actions: As a comparison, the same AES core was implemented using a conventional approach that relies on SystemC events to trigger the State Chart that is only executed in its own context. Note that this implementation still, in contrast to [2] and [3], only uses a single execution context for the State Chart. Both cores were used to repeatedly encrypt

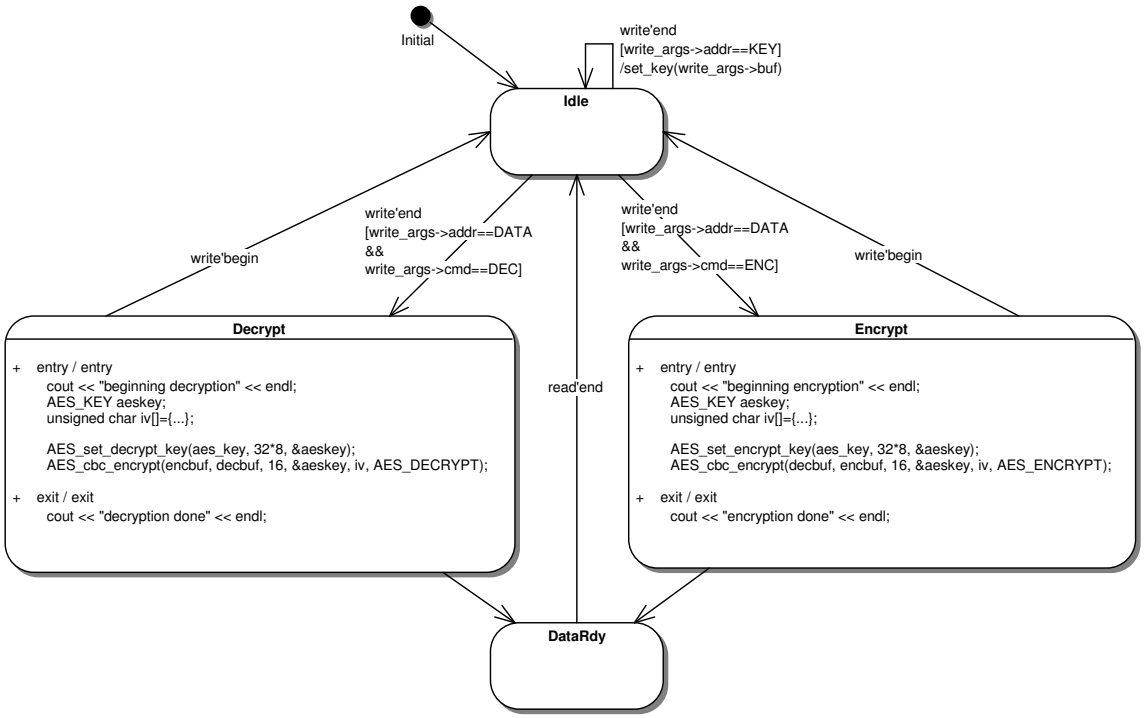

Fig. 7. A State Chart implementing a transaction-level AES core.

TABLE I
AES CORE: SIMULATION TIME OF BOTH THE CONVENTIONAL AND THE PROPOSED APPROACH WHEN USING ENTRY ACTIONS FOR THE ENCRYPTION.

	Run Time $[s]$		
Iterations	Proposed (entry)	Conventional	Speedup
1000000	2.90	6.44	2.22
2000000	5.67	12.72	2.24
5000000	14.38	31.52	2.19
10000000	28.91	62.96	2.18
Average Speedup			2.21

and decrypt data. Tbl. I and Fig. 8 show that the speedup averaged around 2.2.

2) Results with Do Activities: Since the State Chart as given in Fig. 7 does not contain blocking behavior, no context switches at all are performed during the simulation. To determine the impact of blocking behavior with our approach, the State Chart was modified to use do activities instead of entry actions. As stated before, the proposed approach inserts a context switch before the do activity to ensure that the State Chart's context is blocked. This means that less context switches can be omitted and therefore the average speedup decreases, as shown in Tbl. II and Fig. 8, to around 1.4.

B. Best-Case Example

The speedup of our proposed approach is obviously highly dependent on the application. In the AES core example, most of the simulation time is consumed by the AES encryption and decryption, thereby reducing the relative impact of the context switches' overhead. To quantify the maximum speedup, a simple State Chart without complex computations, as shown in Fig. 9, was implemented in both the proposed and a conventional approach.

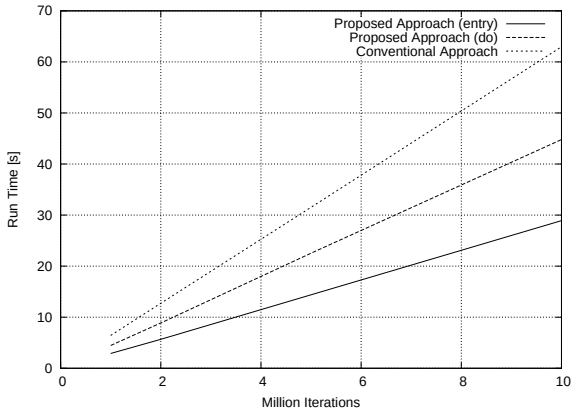

Fig. 8. AES core: the proposed approach improves the simulation performance by a factor of approx. 2.2 when using entry actions and 1.4 when using do activities.

TABLE II
AES CORE: SIMULATION TIME OF BOTH THE CONVENTIONAL AND THE PROPOSED APPROACH WHEN USING DO ACTIVITIES FOR THE ENCRYPTION.

	Run Time $[s]$		
Iterations	Proposed (do)	Conventional	Speedup
1000000	4.47	6.44	1.44
2000000	8.88	12.72	1.43
5000000	22.53	31.52	1.40
10000000	44.79	62.96	1.41
Average Speedup			1.42

The State Chart models a very simple counter that changes

its state on every occurrence of `write'begin` and increments a variable `cnt` every time `State3` is entered. This simple computation was chosen to maximize the impact of the overhead introduced by the context switches.

Since this State Chart does not contain any elements requiring a context switch, the proposed approach allows the implementation to run without any SystemC kernel interaction. As presented in Tbl. III, in this case the speedup averages around 31, indicating that the overhead introduced by the context switches is considerable.

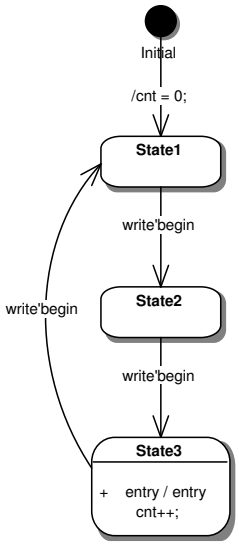

Fig. 9. A "best-case" State Chart for measuring the maximal speedup of the proposed approach.

TABLE III
"BEST-CASE" STATE CHART: SIMULATION TIME OF BOTH THE PROPOSED AND A CONVENTIONAL APPROACH.

	Run Time $[s]$		
Iterations	Proposed	Conventional	Speedup
10000000	0.092	2.704	29.39
50000000	0.429	13.428	31.30
100000000	0.843	27.024	32.06
500000000	4.205	136.567	32.48
Average Speedup			31.31

V. CONCLUSION

In this paper we presented an approach to translate UML State Charts to transaction-level State Charts. As a new contribution, we identified which constructs require a context switch between the caller and the State Chart's module and showed that all non-blocking behavior can be executed in the caller's context without SystemC kernel interaction.

Finally, we showed that the speedup achieved by the proposed approach is highly dependent on the application as well as the implementation: In a real-life example, the speedup averages around 2.2 for a State Chart that can be executed completely in the caller's context and decreases to 1.4 if the State Chart contains blocking behavior that requires a context switch.

To quantify the overhead introduced by the context switches, we presented a simple best-case example in which the simulation time is nearly only spent on transitions that can be done inside the caller's context when using the proposed approach. In this case, a speedup of more than 30 was achieved.

Future work will include further improving the simulation time by optimizing the code generator. Additional focus will be on supporting additional State Charts elements such as history, parallel states, and compound states that allow instantiating a State Chart inside a state.

Moreover, we we will investigate applying temporal decoupling to our approach. Instead of really executing `wait()` statements, the times can be accumulated and then accounted for in using a single `wait()` statement. Since this, in effect, reduces the number of blocking behaviors, it allows to decrease the amount of context switches even in applications with a high amount of blocking behavior.

Finally, a future topic will be a transformation that addresses the limitation that exit actions cannot contain blocking behavior, as outlined before.

REFERENCES

[1] D. C. Black and J. Donovan, *SystemC: from the ground up*. Springer Science+Business Media, LLC, 2004.

[2] 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.

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

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

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

[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] K. D. Nguyen, "Model-Driven SoC Design via Executable UML to SystemC," *25th IEEE International Real-Time Systems Symposium*, pp. 459–468, 2004.

[13] 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.

[14] R. Findenig, M. Velten, V. Esen, W. Ecker, and R. Schwencker, "Applying transaction-level modeling on different abstraction levels," IEEE International High Level Design Validation and Test Workshop, Nov. 2009.

[15] R. Findenig and W. Ecker, "A systemc design pattern for the cosimulation of transaction-level and refined cycle-callable models," in *Proceedings Austrochip 2009*, Graz, Austria, October 2009, pp. 123–128.

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