In Proceedings of *Forum on Specification and Design Languages 2006* (FDL 2006)
Darmstadt, Germany, September 19-22, 2006, © ECSI.

Formalizing TLM with Communicating State Machines

B. Niemann
Fraunhofer Institute for Integrated Circuits
Am Wolfsmantel 33
91058 Erlangen
Germany

Ch. Haubelt
University of Erlangen-Nuremberg
Weichselgarten 3
91058 Erlangen
Germany

Abstract

Transaction Level Models are widely being used as high-level reference models during embedded systems development. High simulation speed and great modeling flexibility are the main reasons for the success of TLMs. While modeling flexibility is desirable for the TLM designer, it generates problems during analysis and verification of the model. In this paper we formalize the notion of Transaction Level Models by introducing a set of rules that allow the transformation of TLMs to a set of communicating state machines. SystemC being the most popular TLM language, we additionally present a finite state model of the SystemC scheduler. Finally, we demonstrate that using our modeling approach, a standard model checker can be employed to formally prove properties on the finite state model.

1 Introduction and Motivation

Transaction Level Modeling [RSPF05] using SystemC [OSC05] has become a standard way of implementing high-level reference models for embedded systems as it allows to implement efficient and flexible virtual prototypes. Over the last years, research activities were mainly focused on exploiting modeling flexibility and exploring different levels of communication and behavior abstraction (see e.g. [CG03]). More recent work concentrates on formalization and verification. The aim of our work is to introduce enough formalism to Transaction Level Modeling to allow the application of formal methods like model checking but to retain enough flexibility to be able to apply our methodology to a wide range of different models.

While existing work is mostly focused on the translation of C++ functionality and SystemC constructs, we aim at a strong emphasis on communication and transactions. To this end we translate a SystemC TLM to a set of communicating state machines. The automata proceed synchronously, meaning that a transition of the overall system consists of the simultaneous transition of all automata in the system. The state of a module in our approach is composed from *behavior state*, *initiator state*, *target state* and *object state*. SystemC provides *events*, *time-outs* and a *scheduler* to enable modeling of a wide variety of communication styles. Therefore, in addition to the modules itself, we provide rules for a finite state representation of scheduler, timed and un-timed events.

The result of the transformations presented in this work is a finite state model which is functionally equivalent to the original TLM and can be used as the input to a standard model checker. Contrary to other approaches, our model explicitly separates behavior from communication and transaction initiation from the target of a transaction. The advantage is a concise and standardized way of formulating properties reasoning about transactions and transaction sequences.

In the remaining part of the paper, we survey related approaches in Section 2 and formally define the terms Transaction Level Model and finite state machine as used in this work in Section 3. Section 4 is dedicated to the transformation of modules, while in Section 5, the rules for the transformation of scheduler and events are presented. Experimental results are discussed in Section 6.

2 Related Work

Related approaches are split into two closely linked fields both being subject of current research. Firstly, as we enable the formulation of properties for Transaction Level Models, work carried out in the area of assertions for TLMs has to be considered. Secondly, existing methods to apply formal verification to SystemC designs are discussed.

Regarding assertions, Ruf et al. [RHKR01] convert FLTL properties to C++ which can directly be integrated into a SystemC design. Dahan et al. [DGG^{+}05] convert PSL properties to deterministic finite automata which are converted to VHDL or SystemC. Habibi et al. [HTLM06] propose static program analysis and genetic algorithms to increase coverage of assertions specified in PSL. All these approaches require a system clock. In [NH06] the application of assertions to timed and un-timed TLMs is proposed. Every transaction is associated with a Boolean signal, so that abstract system properties can be specified using temporal assertions.

In the area of formal verification, Grosse et al. [GD03] apply bounded model checking to SystemC designs at the Register Transfer Level (RTL). Habibi et al. [HT06] convert an AsmL model of the TLM to SystemC. A set of properties specified in PSL is converted via AsmL to C#. At the AsmL level the properties are encoded in every state of the design and can be checked on the fly during the finite state machine generation. Peranandam et al. [PWRK04] use symbolic simulation of a Message Sequence Model to verify LTL properties and obtain coverage information. Their approach does not consider event notification and event sensitivity. Karlsson et al. [KEP06] use a Petri net based approach. They translate the SystemC design to PRES+ which can be used for CTL model checking after a conversion to hybrid automata. Even though translation of TLMs is possible using this approach, there is no mechanism to easily identify a transactions. The focus is modeling of behavior and not communication. Two approaches using, similarly to our approach, a set of communicating parallel automata can be found in Kroening et al. [KS05] and Moy et al. [MMMC05]. In [KS05], Labeled Kripke Structures are used to represent a SystemC design. The work is focused on the application of abstraction techniques to the behavior. They do not take into account communication through transactions and restrict their discussion to clocked models communicating over signals. Translation of SystemC to Heterogeneous Parallel Input/Output Machines is presented in [MMMC05]. While their model of the scheduler is similar to ours, the intention of the model and the treatment of transactions differ. They propose to use translation patterns for different TLM channels, and aim at proving the validity of safety properties reasoning about implementation behavior. We suggest a pattern for the transfer of control from the initiator to the target and treat transaction functionality like any other C++ code. Our aim is to provide a formalism to reason about transactions using temporal logics with the transactions itself being atomic propositions used within the properties. This makes our model well suited for the formulation of temporal properties reasoning about transactions.

Our methodology differentiates from the related work by presenting a model for TLMs, where an easy identification of initiator, target, and transaction is possible. While previous methodologies have focused on behavior, we make a strong emphasis on communication, and preserve the separation from communication and behavior of the SystemC TLM within our formal model. Moreover, we can handle overlapping executions of the same transaction and model arbitrary primitive channels.

3 Prerequisites

Transaction Level Models are characterized by communication through *interface method calls*, i.e. calling a method implemented in one module (the target) from within another module (the initiator). Moreover, their *level of abstraction* is above RTL, even though the communication, or parts of it, may be cycle-accurate. Formally, in this work, a TLM is a six tuple $S := (M, M_I, M_T, N, T, I)$, where M is a set of *modules*, $M_I \subset M$ the set of *initiator modules* and $M_T \subset M$ the set of *target modules*. A module needs not necessarily belong to only one category, however $M_I \cup M_T = M$. N is the set of all *interface method names*. The set of transactions is described as $T \subseteq M_T \times N$ and associates target modules and interface method names. Note that the same interface method may be implemented differently in different target modules. Thus it is necessary to associate the name of the target module with the name of the interface method to uniquely identify a transaction. The function $I : T \mapsto 2^{M_I}$ maps each transaction to a set of initiators.

A finite automaton is a six tuple $A := (\Sigma, \Omega, S, \delta, \lambda, s^0)$. The *input alphabet* is given by Σ, the *output alphabet* by Ω. The *transition relation* $\delta \subseteq S \times \Sigma \times S$ describes the evolution of the state with the inputs, the *output relation* $\lambda \subseteq S \times \Sigma \times \Omega$ maps a letter of the output alphabet to each state transition. The set of initial states is given by $s^0 \subseteq S$. If the transition relation and output relation are replaced by the functions $\delta : S \times \Sigma \mapsto S$ and $\lambda : S \times \Sigma \mapsto \Omega$ and the automaton only has one initial state $s^0 \in S$, the automaton is called *deterministic*. We also use the term finite state machine (FSM) for a deterministic finite automaton. Most automata discussed in this paper are deterministic. The only exception from this rule is the automaton used for process activation in our model of the SystemC kernel (see Section 5).

4 A Finite State Model for TLMs

Using the FSM model presented in this section and the model of the SystemC scheduler from section 5 it is in principle possible to describe non-TLM SystemC models. This, however, is not the intention of our approach; it is much more a consequence of the need to support multiple types of TLMs. Through its explicit modeling of initiator, target, and communication channel, it is well suited to describe TLMs but will not be the optimum choice for other modeling styles. When describing e.g. a SystemC model at RT-level, there will be no need for a scheduler or explicit modeling of the

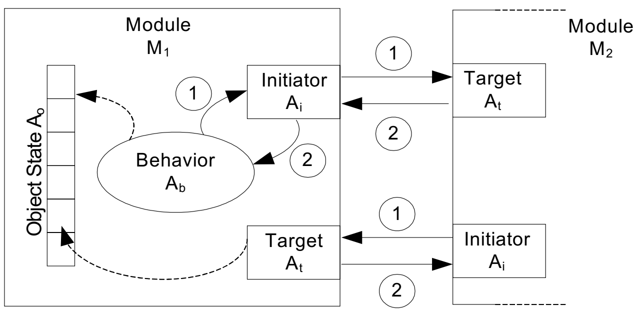
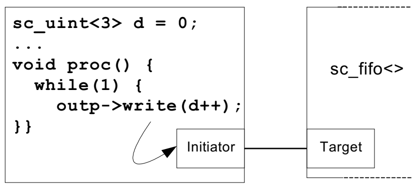


Figure 1: Model of a transaction level module composed of behavior, initiator, target, and object FSM.



```
sc_uint<3> d = 0;
...
void proc() {
  while(1) {
    outp->write(d++);
}}
```

Figure 2: A data flow source module using blocking write.

`sc_signal<>` communication channel.

A module of a Transaction Level Model is built from four basic types of FSMs: behavior, initiator, target, and object (see Figure 1). Processes are described with a Behavior FSM, A_b. An initiator FSM, A_i, is needed for each process initiating a transaction, i.e. each initiator module has at least one initiator FSM. On the other hand, each initiator FSM needs a corresponding target FSM, A_t, which resides in the respective target module. Initiator and target FSM model the communication through transactions. Using one initiator-target pair per process is important to allow multiple processes to initiate overlapping executions of the same transactions. This is, to our knowledge, not currently handled by other methodologies. The object FSM, A_o, finally represents the internal state of a module, which is composed from all the member variables of this module. Basically it is a vector of variables containing all the member variables of a module which is modified by either A_b or A_t.

Even though the current work does not explicitly deal with building hierarchical models, model checkers like NuSMV [CCG+02] have a notion of hierarchy and modules allowing to partition the above described FSMs in such a way that behavior, initiator, target, and object FSM of one SystemC module are grouped into one NuSMV module.

Transfer of Control between the FSMs Let A_1 and A_2 be two FSMs. Transfer of control from A_1 to A_2 means that A_2 is in a state s_2^i where output ω_1^l of A_1 can trigger a transition, denoted by: $s_2^i \xrightarrow{?\omega_1^l} s_2^j$. After A_1 has generated output ω_1^l by a transition from s_1^l to s_1^m, formally written as $s_1^l \xrightarrow{!\omega_1^l} s_1^m$, it is in a state s_1^m with $s_1^m \xrightarrow{?\omega_2^k} s_1^n$, where only ω_2^k from A_2 can trigger a transition.

The solid arrows in Figure 1 indicate a transfer of control from one automaton to another automaton. The numbers at the arrows indicate the order of the transfer. The behavior FSM, for example, transfers control to the initiator FSM, from where control is transfered back to the behavior FSM. It is important to note from Figure 1 that A_t of module M_2 is waiting for an appropriate input from A_i of module M_1, while A_i itself is waiting for an appropriate input from A_b of M_1. Therefore, only the behavior FSM, or the process, may finally initiate a sequence of control transfers that constitutes a transaction.

The dotted arrows in Figure 1 indicate that the object state may be changed from either the target FSM A_t or the behavior FSM A_b. Translated to SystemC this means that the member variables of a module may either be changed by a process or an interface method of this module. This requires that the member variables of the module are not accessible from the outside world.

Example - Data Flow Source Module To discuss the basic concepts and illustrate our methodology, a simple source module of a data flow system is used in the following (see Figure 2). The module issues blocking write transactions to a FIFO channel. The values written to the FIFO are incremented after each write; the usage of a three bit unsigned integer results in a wrap-around from seven to zero. The `sc_fifo<>` implementation from the SystemC reference implementation is used for the FIFO.

Behavior and Internal State Processes are modeled with the behavior FSM A_b. Figure 3 shows an example state diagram of A_b for the module of Figure 2. To correctly model a SystemC process, it has to remain idle until control is diverted to the process by the SystemC scheduler.[1] This is modeled by the initial state I from which a transition is only possible on input *run*. The rest of the behavior FSM depends on the SystemC process to be modeled. In this example, in state WR output wr is generated and control is transfered to the initiator state machine; A_b will remain in state WR until the initiator FSM signals the end of communication with *end_com*. The initiator FSM A_i asserts output *end_com* whenever a transaction has been completed, thereby signalling to A_b that

[1]For a detailed explanation of the scheduler see Section 2.

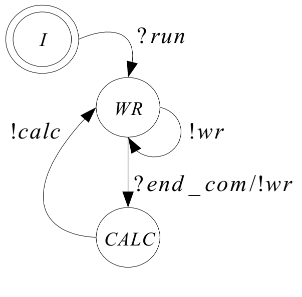


Figure 3: State diagram for the behavior FSM of the data flow module from Figure 2.

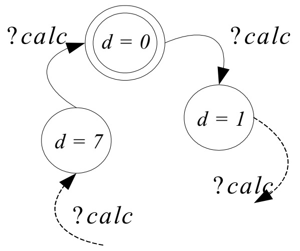


Figure 4: State diagram for the object FSM of the data flow model from Figure 2.

it can continue. After *end_com* has been received, state $CALC$ is entered from where an unconditional transition to WR is initiated. This transition generates an output *calc*.

A transition of the object FSM A_o incrementing the value of d, see Figure 4, is initiated at every occurrence of *calc*. The internal state of a module may only be changed by a process or by calling a transaction implemented inside the module. Formally, this means that $\Sigma_o \subset \{\Omega_b \cup \Omega_t\}$, i.e. the input alphabet of A_o is a subset of union of the output alphabets of A_b and A_t.

Communication Communication between the modules in a Transaction Level Model is modeled with an initiator FSM A_i and a target FSM A_t. We use $\langle T_i$ to denote the initiating state of a transaction T_i and $T_i\rangle$ for the terminating state of the transaction. The initiator FSM for the data flow example is shown in Figure 5. From its initial state NOP it steps to the transaction initiator state $\langle WR$ if wr is asserted by A_b. In the transaction initiator state, the output $\langle wr$ is generated. The initiator FSM remains in $\langle WR$ until the end of the transaction is signaled from the target FSM A_t by $wr\rangle$. Finally, on leaving the transaction initiator state and returning to the initial state, end of communication is signaled to the calling process by output *end_com*. As each process may only request the initiation of one transaction at a time, it is sufficient to use one output *end_com* to indicate the end of all transactions that may be initiated by A_i.

The target FSM A_t corresponding to the initiator FSM of Figure 5 is shown in Figure 6. It remains in the initial state NOP until the execution of a transaction, is requested by $\langle wr$. If the FIFO is full (f is *true*), the automaton proceeds to the blocking state W_E_r, modeling the blocking behavior of the write transaction. Note that the corresponding SystemC code to state W_E_r is a `wait()` statement, in which the process having called the write transaction is suspended until a *read event* e_r occurs, i.e. until at least one sample has been read from the FIFO. Any `wait()` statement can be modeled following the pattern described for state W_E_r. After having entered the blocking state W_E_r, output w_e_r is generated, signaling to the scheduler that the corresponding process should be suspended and only reactivated upon the occurrence of e_r. The automaton remains in state W_E_r until *run* is asserted by the scheduler. The output *run* is generated by the scheduler after event e_r has been triggered and the process has been selected for execution (see Section 5). The end of transaction state, $WR\rangle$, can be reached in two ways. Either by leaving the blocking state W_E_r on the occurrence of *run* or directly from the initial state NOP if the FIFO is not full. Upon transition from $WR\rangle$ to NOP, the end of transaction output, $wr\rangle$, is generated. Therefore, this transition initiates a transfer of control back to the initiator FSM.

5 Scheduling

To preserve the *co-operative multitasking* simulation semantics of SystemC [OSC05] in the finite state model, a scheduler has to be included into our modeling effort. The scheduler in SystemC maintains a list of *runnable* processes and selects one of them non-deterministically to be *running*. This process is executed without interruption until either its end (*method processes*) or the occurrence of a `wait()` statement (*thread processes*). As every method process can be replaced by a functionally equivalent thread process, we will restrict our discussion to thread processes without loss of generality.

A process that has reached a `wait()` statement returns control to the scheduler, is removed from the runnable processes and is said to be *suspended*. Now, the scheduler selects another process from the list of runnable processes until the list is empty. The process of executing one runnable process after the other is called *evaluation phase*.

Once all processes are suspended, the scheduler calls the `update()` method of all *primitive channels* that have requested an update. A primitive channel is a channel that does not have its own processes or ports.

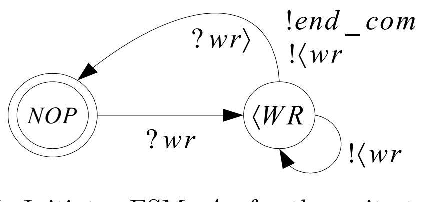


Figure 5: Initiator FSM, A_i, for the write transaction of the data flow example from Figure 2.

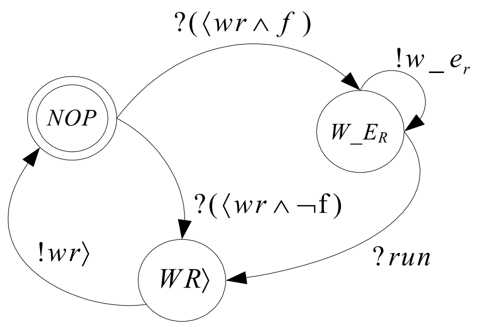


Figure 6: Target FSM, A_t, corresponding to the initiator FSM from Figure 5. The target FSM implements the blocking write transaction requested by the initiator FSM.

Examples for primitive channels are hardware signals or FIFOs.

The next step of the scheduler is to check for *events* that have been *notified* during the evaluation phase and to add processes that are sensitive[2] to one of the notified events to the list of runnable processes. This phase is called *delta notification phase*.

The cycle of evaluation phase, update phase and delta notification phase is called a *delta cycle*. If no processes are runnable at the end of a delta cycle, simulation time is advanced. Currently, we do not maintain a global simulation time, however waiting for a specified amount of time and timed event notification is supported. Note that we deliberately do not support immediate notification, as it introduces non-determinism into the design.

Modeling the Scheduler Our model of the scheduler differs from the one presented in [KS05] by adding a notion of time and supporting dynamic sensitivity. In [MMMC05] no clear separation between update phase and delta notification phase is made. The Petri net based scheduler of [KEP06] is closest to our model, however, they use a subscription scheme to associate processes and event notifications where we handle this association in the scheduler FSM.

A scheduler for N processes $P_0, P_1, ..., P_{N-1}$ is represented by an automaton A_s. This automaton maintains a *phase selector state* $\phi \in \{evaluate, update, delta\_ntfy, timed\_ntfy\}$, a *process selector state* $\sigma \in \{none, P_0, ..., P_{N-1}\}$ and, for each process P_i, a *process state*, $\pi_i \in \{runnable, running, suspended\}$.

The scheduler is initialized to $s_s^0 = (\phi = evaluate, \sigma = none, (\forall i \in \{0...N-1\} : \pi_i = runnable))$, meaning that the scheduler starts in the evaluation phase with no process being run but all processes being runnable. Then, an arbitrary runnable process P_j is selected for execution, i.e. the process selector is set to $\sigma = P_j$ and the process state is changed to $\pi_j = running$. The process is executed until the occurrence of a wait statement after which it is suspended. The process state is changed to $\pi_j = suspended$ and the process selector is set to $\sigma = none$. This procedure is repeated until no process remains runnable, meaning until $\forall i \in \{0...N-1\} : \pi_i = suspend$ and $\sigma = none$.

[2]Here we mean both, static sensitivity (the event is in the sensitivity list) and dynamic sensitivity (the process waits for an event at a `wait()` statement embedded in the code).

Next, the scheduler switches to the update phase, $\phi = update$. All primitive channels receive an update signal in this phase. Each channel has the responsibility to decide whether it wants to execute its update method. Only after all updates have been carried out, the scheduler proceeds and changes the phase selector state to $\phi = delta\_ntfy$.

In the delta notification phase, events that have been notified during the evaluation phase are triggered. All processes that are sensitive (either statically or dynamically) to one of the triggered events, change their state from *suspended* to *runnable*.

Now, the phase selector changes to $\phi = evaluate$ and the next delta cycle is started. If no processes become runnable at the end of a delta cycle, the phase selector is set to $\phi = timed\_ntfy$ and simulation time is advanced.

In Figure 7 the transitions of the process state for the example from Figure 2 are shown. Once the process is selected for execution ($\sigma = P_0$), the process state changes from *runnable* to *running*. The process remains in this state until the `wait()` statement in the blocking write transaction triggers a transition to *suspended* by asserting w_e_r. Upon occurrence of t_e_r, the process state is changed to *runnable* again. The input t_e_r corresponds to triggering event e_r. See Paragraph *Modeling Events and Time-Out* for details.

Modeling Channel Updates The request-

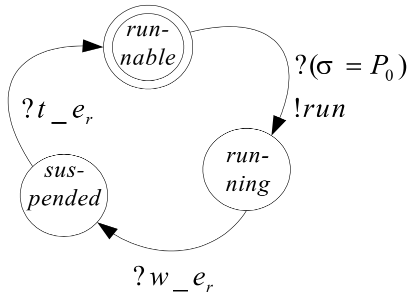


Figure 7: Process state for the example shown in Figure 2.

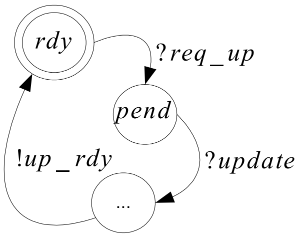


Figure 8: Generic state diagram for the update FSM.

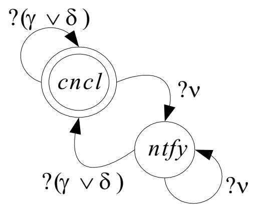


Figure 9: Event automaton.

update mechanism for primitive channels requires an additional automaton A_u (see Figure 8). The automaton starts in state *rdy*. On an update request req_{up} from within the channel, the automaton changes its state to *pend*. This happens during the evaluation phase of the scheduler. During the update phase of the scheduler, the automaton receives an input *update* that initiates execution of the update functionality if the automaton is in state *pend*. The update functionality depends on the primitive channel being modeled. After having carried out the update, the automaton returns to its initial state, thereby signaling the end of the update.

Modeling Events and Time-Out Events are modeled by means of an automaton A_e (see Figure 9). It has two states, $S_e = \{ntfy, cncl\}$, an input alphabet $\Sigma_e = \{\nu, \gamma, \delta\}$, an output alphabet $\Omega_e = \{nil, trig\}$ and an initial state $s_e^0 = cncl$. The occurrence of an event notification (a `notify` statement) is modeled by an input ν, a cancellation (a `cancel` statement) by γ. On a notification, the state is set to $ntfy$, on a cancellation to *cncl*. When the scheduler is in the delta notification phase and has added all processes sensitive to a notified event to the list of runnable processes, the corresponding event automaton is reset by input δ. This assures that the same event notification does not trigger a process twice. The output of A_e is set to *trig* if it is in state $ntfy$ and to *nil* if it is in state *cncl*.

Suspending a process for a certain amount of time, i.e. the `wait(<time>)` statement is supported with the help of a counter. At the occurrence of a time-out, the process state is changed from *running* to *suspended*. Moreover, a counter is incremented each time the scheduler enters a timed notification phase, $\phi = timed\_ntfy$. If the counter has reached the value specified as an argument to the `wait(<time>)` statement, the process state is changed from *suspended* to *runnable* and the counter is reset to zero.

6 Experimental Results

In the following, we will demonstrate the applicability of our approach using several examples. All experiments were carried out using NuSMV [CCG+02] running on a 3.2 GHz Linux PC. For property specification, Computation-Tree Logic (CTL) has been used.

Writing Properties The advantage of the explicit modeling of initiator FSM, A_i, and target FSM, A_t, is the easy identification of transactions. A transaction T_i, for example, lasts as long as the output $\langle T_i$ of A_i is asserted. The end of a transaction T_i is marked with $T_i\rangle$. Stating in CTL that transactions T_1 and T_2 should never interfere is as simple as $AG(\ \neg(\ \langle T_1 \wedge \langle T_2\ )\ )$. A CTL property that checks if a condition *cond* in the target module holds after transaction T_1 has been executed can be formulated as $AG(\ T_1\rangle \rightarrow AX(cond)\ )$. As a last example, specifying that a blocking transaction T_1 initiated under a certain condition *cond* should block until that condition is false, looks like $AG((\ \langle T_1 \wedge cond\ ) \rightarrow A[\ \neg T_1\rangle\ U\ \neg cond\ ])$.

Traffic Light System A relatively simple example of a TLM is the traffic light system presented in [NH06]. A controller changes the status of two traffic lights by using transactions. We have modified the original model to have an additional red-yellow state. We have proven the usual safety and liveness properties:

P1 If one traffic light is green, the other one has to be red.

P2 Each traffic light has to be green infinitely often.

P3-P6 Verify the correct sequence of transaction calls, e.g. after a red light, red-yellow has to occur.

All properties could be verified within a fraction of a second.

Vending Machine Another simple example is the vending machine presented in [PWRK04]. A user can insert a coin and select coffee or tea. This can be done in arbitrary order. After both transactions, inserting a coin and selecting a product, have been committed, the vending machine responds with delivering either coffee or tea to the user. The following properties were verified:

P1 After first inserting money and then selecting a product, the vending machine has to deliver the product.

P2 After first selecting a product and then inserting a coin, the vending machine has to deliver the product.

P3/P4 No product must be delivered without having inserted a coin / having selected a product.

Table 1: Run times needed to check properties of the un-timed data flow system. All times are measured in seconds. The size of the FIFOs is given in brackets, [].

		P1	P2	P3	P4
SRC-SNK	[64]	2	1	1	<1
SRC-SNK	[256]	366	62	11	11
SRC-FT-SNK	[16]	2	2	<1	<1
SRC-FT-SNK	[64]	1176	1452	4	4
SRC-FT-FT-SNK	[4]	95	115	2	2

Again, all properties could be verified within a fraction of a second.

Un-Timed Data Flow System As a more complex example, we have implemented a simple data flow system. In its least complex version it consists of a source (SRC) connected to a sink (SNK) through a FIFO channel. In other versions, we have inserted a different number of feedthrough (FT) models between source and sink. The feedthrough models read data from the input FIFO and write it back to an output FIFO.

Source, sink, and feedthrough modules are implemented with an `SC_THREAD` process using blocking write and read access to the FIFO. The FIFO implementation we used is the `sc_fifo<>` primitive channel from the SystemC distribution. As we are mainly interested in communication, we have only modeled the number of samples stored into the FIFO and have abstracted from their values. The following properties were checked using different FIFO sizes:

P1/P2 The blocking write/read of the source always finally returns.

P3 A write to a full FIFO always blocks.

P4 A read from an empty FIFO always blocks.

The run-times needed to check those properties for different numbers of feedthrough modules and different FIFO sizes are shown in Table 1.

Timed Data Flow System As an example of a timed, but un-clocked system, we have modeled a different version of the data flow system using non-blocking read and write transactions. The source issues two consecutive non-blocking write accesses to the FIFO and then waits for T_0, while the sink always only issues one non-blocking read attempt before waiting for T_1. We assume that a non successful read or write attempt results in a loss of data. Therefore it is important to check whether every read or write attempt was successful. The properties that have been checked are summarized below:

Table 2: Run times needed to check properties of the timed data flow system. All properties were verified for a FIFO size of 16. All times are measured in seconds.

	P1	P2	P3	P4
$T_0 = 40, T_1 = 20$	<1	<1	<1	<1
$T_0 = 40, T_1 = 21$	13	5	5	5

P1/P2 Every write/read attempt was successful.

P3/P4 If the FIFO is full/empty, a write/read attempt fails.

Table 2 shows the run-times measured for checking the different properties. Note, that property P1 for $T_0 = 40$ and $T_1 = 21$ fails because the source issues its write transactions with a higher frequency than the sink issues its read transactions.

7 Conclusions

The methodology we have presented in this paper has three main advantages: (1) It is a native finite state model that does not need an additional translation step before being able to apply model checking. (2) The emphasis is on the communication aspect, allowing convenient identification of transactions. (3) It can handle timed and un-timed models at levels of abstraction above RTL. We have shown the applicability of this approach with several examples.

References

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Proceeding of the 14th International Conference on Computer-Aided Verification (CAV'2002)*, 2002.

[CG03] Lukai Cai and Daniel Gajski. Transaction Level Modeling: An Overview. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, New York, NY, USA, 2003. ACM Press.

[DGG+05] A. Dahan, D. Geist, L. Gluhkovsky, D. Pidan, G. Shapir, Y. Wolfsthal, L. Benalycherif, R. Kamdem, and Y. Lahbib. Combining System Level Modeling with Assertion Based Verification. In *Proceedings of the Sixth International Symposium on Quality Electronic Design (ISQED'05)*, 2005.

[GD03] Daniel Große and Rolf Drechsler. Formal Verification of LTL Formulas for SystemC Designs. In *IEEE International Symopsium on Circuits and Systems (ISCAS'03)*, 2003.

[HT06] A. Habibi and S. Tahar. Design and Verification of SystemC Transaction-Level Models. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14:57–68, 2006.

[HTLM06] A. Habibi, S. Tahar, D. Li, and O. A. Mohamed. Efficient Assertion Based Verification Using TLM. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'06*, 2006.

[KEP06] D. Karlsson, P. Eles, and Z. Peng. Formal Verification of SystemC Designs Using a Petri-Net Based Representation. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'06)*, 2006.

[KS05] D. Kroening and N. Sharygina. Formal Verification of SystemC by Automatic Hardware/Software Partitioning. In *Proceedings of MEMOCODE 2005*, pages 101–110. IEEE, 2005.

[MMMC05] M. Moy, F. Maraninchi, and L. Maillet-Contoz. LusSy: A Toolbox for the Analysis of Systems-on-a-Chip at the Transaction Level. In *Proceedings of the Fifth International Conference on Application of Concurrency to System Design (ACSD'05)*, 2005.

[NH06] B. Niemann and C. Haubelt. Assertion-Based Verification of Transaction Level Models. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, 2006.

[OSC05] Open SystemC Initiative OSCI. Draft Standard SystemC Language Reference Manual.
`http://www.systemc.org`, 2005.

[PWRK04] Prakash M. Peranandam, Roland J. Weiss, Juergen Ruf, and Thomas Kropf. Transaction Level Verification and Coverage Metrics by Means of Symbolic Simulation. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 260–269, Aachen, 2004. Shaker Verlag.

[RHKR01] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-Guided Property Checking Based on Multi-Valued AR-Automata. In *Proceedings of Design Automation and Test in Europe (DATE)*, Munich, March 2001.

[RSPF05] A. Rose, S. Swan, J. Pierce, and J.-M. Fernandez. Transaction Level Modeling in SystemC. OSCI TLM Working Group, 2005.
`http://www.systemc.org`.