Reactivity in SystemC Transaction-Level Models

Frederic Doucet[1], R.K. Shyamasundar[2], Ingolf H. Krüger[1], Saurabh Joshi[3], and Rajesh K. Gupta[1]

[1] University of California, San Diego
[2] IBM India Research Laboratory
[3] Indian Institute of Technology, Kanpur

Abstract. SystemC is a popular language used in modeling system-on-chip implementations. To support this task at a high level of abstraction, transaction-level modeling (TLM) libraries have been recently developped. While TLM libraries are useful, it is difficult to capture the reactive nature of certain transactions with the constructs currently available in the SystemC and TLM libraries. In this paper, we propose an approach to specify and verify reactive transactions in SystemC designs. Reactive transactions are different from TLM transactions in the sense that a transaction can be killed or reset. Our approach consists of: (1) a language to describe reactive transactions that can be translated to verification monitors, (2) an architectural pattern to implement reactive transactions, and (3) the verification support to verify that the design does not deadlock, allows only legal behaviors and is always responsive. We illustrate our approach through an example of a transactional memory system where a transaction can be killed or reset before its completion. We identify the architectural patterns for reactive transactions. Our results demonstrate the feasibility of our approach as well as support for a comprehensive verification using RuleBase/NuSMV tools.

1 Introduction

Transaction-level models are useful in SystemC [1] to understand a system by abstracting the low-level bus signaling details. In this paper, we build upon this work by extending the transactions to support reactive features that are commonly found in frameworks such as Esterel [2]. Reactivity can provide one with the capability to kill or reset a transaction before the transaction completes. This is analogous - but for transactions - to the reactive features for processes that were present in the earlier versions of SystemC through the "wait" and "watching" syntactic constructs [3]. The "watching" constructed was later dropped from SystemC due to lack of use. However, as the libraries evolve and as the role of TLM models is increasing, we believe that these constructs would find greater use and simplify the design migration to higher levels of abstraction.

This investigation of specification and verification of reactive transactions was motivated by an experiment to model and verify a transactional memory using interaction descriptions and SystemC. The fundamental difference between the transactional memory model and the typical TLM models built with SystemC is the following: in the transactional memory, a process that initiates a transaction can be reset before the transaction completes. Then, the pending transaction could be reset or not, depending on what the desired outcome is. Unfortunately, it is not possible to capture this kind of behavior with the current SystemC TLM libraries. Therefore, we had to re-think what a transaction is and what are the syntactical and architectural features that are necessary to capture the reset and kill behavior, and how to use the formal verification to guarantee the implementation of the transaction specifications.

We found three challenges for specifying implementing and verifying the reactive transactions with SystemC. The first one is to specify the transactions using the property specification languages. Because many events can potentially happen at the same time, the properties can be very tedious to specify. From our experience, as a specification can take many simultaneous events at one time, and because the properties need to describe every possible scenario, the properties become very large as one basically has to compute and write down the product of all possible event combinations for the TLM events. The second challenge is that it is difficult to implement reactive features within SystemC TLM models. This is because there are no do/watching statement we can use to capture the reactive behaviors and the necessary transaction handlers. Also, since the transaction events are atomic rendezvous in the specification and handshakes in the implementation, the implementation of the reactive transactions can be challenging as mismanaging the handshakes with the resets could easily cause synchronization problems such as deadlocks. Finally, the third challenge is to have an efficient SystemC verification framework that support the reactive transactions as a first-order construct and also includes the capability of verifying liveness properties.

In this paper, we present an approach to specify and reason about the reactive transactions by defining a language that will capture the transactions, and a tool to translate these specifications into verification monitors. While one could argue that such transactions could be specified using PSL [4], we believe the task can be slightly tedious as the properties become long and complex. This is evidenced by the continuing evolution of PSL into more elaborate higher-level design languages [5] where the specification can be a bit more high-level, making the specification easier to write. In that spirit, we use a specification language that is inspired by the process algebraic framework of CRP [6]. Our framework enables the specification of rendezvous communications a la CSP, as well as the reactive features provided by the Esterel constructs. We extend those ideas to add the features that are necessary for transactions.

The contributions described in this paper are as follows. First, we define a high-level language inspired by CRP to describe reactive transactions and their compositions as a first-order construct. Second, using the standard syntax, we provide a TLM extension in the form of an architectural pattern to capture the reactive transactions, with the cascading of resets. Third, we believe are the first to formally check TLM models with respect to transaction specifications rather than generic properties.

The rest of the paper is organized as follows. In the next section, we present the related work in monitor-based verification and SystemC verification. In Section 3, we describe a subset of the Transactional Memory example that motivated this work, and the problems and challenges of specifying and verifying reactive transactions. In Section 4, we describe how to specify reactive transactions, with the definition of the syntax and semantics of the specification language. In Section 5, we describe how to implement the reactive transactions in SystemC, and then present our experiments and results in Section 6, followed by a discussion and the conclusion.

2 Related Work

We broadly categorize the related work as being the specification of protocols and the generation of verification monitors, as well as the verification of SystemC designs. In some sense, this work bridges transaction specification with SystemC verification by using the specification of transactions for TLM verification.

2.1 Protocol Monitors

In the context of system-level design, a transaction is a concept that is a slight bit above the components. We need to capture the transactions spanning accross multiple components in the system into properties that can be used for verification. Specification languages do not clearly provide the necessary constructs, since there is no notion of global transactions.

To address this gap, there have been several attempts at describing transactions as high-level entities, at the level above the components. Seawright et al. [7] proposed an approach to describe the transaction that can happen at an interface using regular expressions. Such a protocol description can be used to generate interface monitors from the regular expressions. Siegmund et al. [8] followed this approach and showed how one can synthesize bus interface circuits from the regular expressions. The approach has the advantage that, instead of describing the producer and the consumer, the description models the protocol as a monitor that observes a set of variables. To describe the monitor, their language has four operators: “serial”, “parallel”, “repeat” and “select”. A synthesis algorithm is used to generate the state machines for both the producer and the consumer. Although it greatly simplified complex hardware design, one of the limitations is that it can be difficult to specify and synthesize the reactive features (kill/reset) with the available operators.

Several more interesting contributions followed. Oliveira et al. [9] extended monitor-based specification languages by introducing storage variables, a pipeline operator, and also improved the algorithms for generating the protocol monitor. However, one drawback of their approach is the lack of formal semantics. Shimizu [10] addressed part of the problem by using a framework of concurrent guarded transitions, and showed how to model check the descriptions for useful properties.

Interface descriptions and monitors are now widely used for both documentation and validation [4] [11]. Many engineers use the PSL language (and extensions [5]) to describe the interfaces, and several tools exist to generate protocol monitors for simulation. There exist commercial tools that generate protocol monitors from such descriptions for simulation or verification, notably FoCs [12]. In this context, we see two opportunities stemming from this body of work: (1) to easily and elegantly capture the reactive features in the transactions, and (2) to have a compositional analysis from transactions to interface specification, which challenging to achieve with the reactive features.

2.2 SystemC Verification

The goal of the Transaction-Level Modeling (TLM) with SystemC [13] is to define a model where the details of the RTL bus communications are abstracted away either (1) instead of going through signal transitions, have a component directly call the method of another component, or (2) having the components communicating through buffered FIFO communications. In both cases abstract data types can also be used to bundle low-level bus data types into one chunk of data. The benefit of using a TLM model is that the simulator does not need to spend cycles on simulating all the RTL bus synchronizations, thus the design will simulate much faster. Typically, a SystemC TLM model simulates 2-3 orders of magnitude faster than an “equivalent” RTL model. There exists a number of verification approaches for both RTL and TLM models written in SystemC. These approaches support TLM models in the sense that they support the syntactic constructs found in the models, which include function calls, access to FIFO buffers, and reading and writing signals. However, it is difficult to verify a TLM model because (1) the model can be non-deterministic due to the shared variable communications within the channels, and (2) the number of elements queued in a FIFO buffers can grow without a bound. Thus, all existing approaches impose restrictions on the input syntax to avoid these problems.

The approach defined by Habibi et al. [14] uses a specification format based on PSL sequences or basic Message Sequence Charts, augmented with clocking guards. The properties range over the signals and the buffers in the architecture. A property is translated into a monitor, which is an automaton with fail, in-progress and accept states. Similarly, a SystemC design is translated into automaton and then the design and the property automata are composed together. During the composition, the ASML composition tool will expand the state machine and check that the monitor is always asserted. Similarly, the Lussy tool suite developed by Moy et al. [15] translates SystemC modules into an intermediate automaton based on the Lustre formalism. This approach also uses synchronous observers for verification. In this work, the notion of a transaction matches the SystemC TLM definitions, where the bus transactions which are abstracted into simple function calls. These function calls are then mapped to architectural blocks that capture the TLM communication through simple minimal handshakes. The approach by Kroening [16] provides efficient SystemC verification by using by translating a SystemC model to a SMV description, using predicate abstraction and other techniques. However, here again there is no explicit notion of a transaction.

3 Motivating Example: Transactional Memory

Figure 1 shows an example of how modules, channels and buffers are connected in the transactional memory system. There are three components with their own threads: the program segment, the controller and the cache. There are also two channels, which convert a SystemC TLM method call into a request/response handshake through buffers.

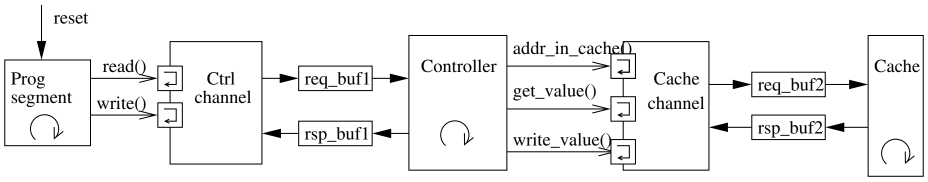

Fig. 1. Simplified example architecture for the Transactional Memory model

The program segment starts a `read()` or a `write()` transaction with a method call to the channel. Figure 2 depicts a scenario for the interactions for a read transaction. The channel converts the call to a request which it places on th `req_buf1 buffer`. The controller will pick up the request, and if it is a read request it is going to check if the address is in the cache by calling the `addr_in_cache` method of the cache channel. If it, then it will get the value by calling the `get_value` method. If it is a write request, it will just call the write value method. The methods of the cache channel will generate a request to the `req_buf2` buffer. The cache will then process the request and place the response on the response buffer `rsp_buf2`, and the response will eventually make its way to the program segment that will eventually pick it up through the value returned by the original method call. Note that for a read or write transaction, there is at least one sub-transaction that will be called the controller and the cache.

The `reset` signal is used to reset the program segment when there is a conflict on the cache. The program segment can be reset at anytime while a transaction is in progress. The main challenge in this example is the following: when the program segment is reset, what happens with the pending transactions? Should they complete or be killed? It is the responsibility of the designer (or the synthesis tool) to decide what the desirable outcomes in such situations are. However, in the current SystemC TLM standard, there is no support to handle these situations. Therefore, defining the required control signals and communication protocols to support these situations, both for specification and verification, is the central problem we are addressing in this paper.

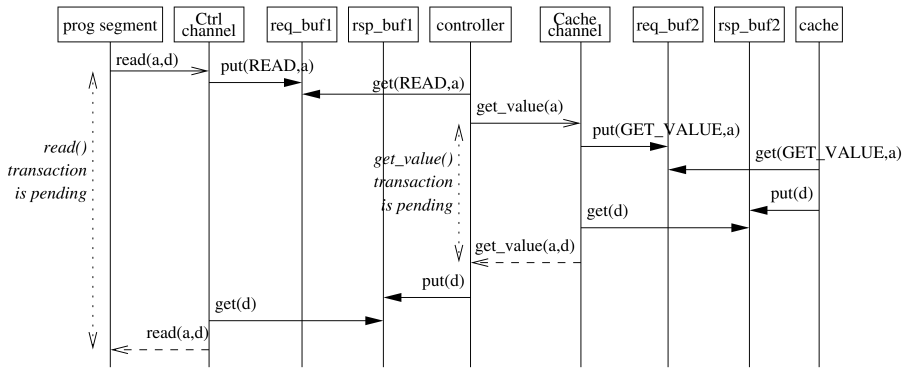

Fig. 2. Message exchange and scope for a read transaction

4 Specification of Reactive Transactions

We capture a transaction as a first-order entity, in the sense that it can be specified, it has a context, control signals and that it can describe behavior which can be distributed over many components in an architecture. Figure 3(a) depicts the "abstract" interface of a transaction: there is a `start` and a `done` signal, both being used as the normal and entry and exit event of the transaction. As its name indicates, the `kill` signal is used to terminate a transaction. The `status` is used by other components to observe the status of the transaction. The possible statuses are "ready", "done", "in progress" and "killed". Figure 3(b) and Figure 3(c) shows the abstract interface behavior for a terminating and a reactive transaction. The start and done signals are abstract in the sense that they can be mapped to given events in the system, such as specific reading a value for a buffer. In between and start and the completion of a transaction can be events, operations, and sub-transactions.

The specification language we use to capture transactions is rooted in CRP, but we augment it to capture the transactions as first-order constructs. The

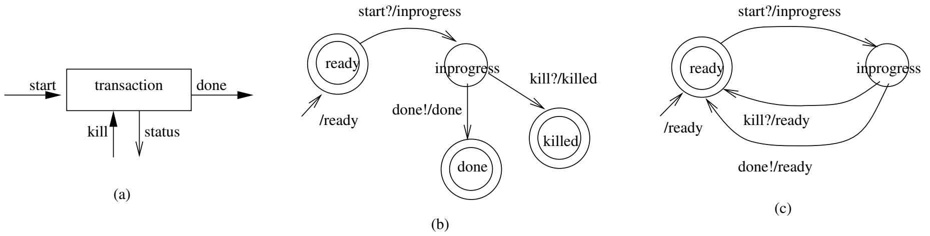

Fig. 3. Interface for Reactive Transaction: (a) control and status signals (b) normal transaction (c) reactive transaction

formal foundation of CRP [6] is composed of CSP [17], where we borrow the rendezvous communication, with the synchronous foundation in Esterel with its reactive features [2]. In the same fashion, we will define the semantics of our specification language with a semantics domain composed of an environment, which is a set of events, rendezvous actions, pending labels and status flag, and a set of state variables. The semantics of a specification description is defined through a transition system which is induced by operational semantics rule of the form:

$$(\langle stmt\rangle, \sigma) \xrightarrow[\langle E,A,L\rangle]{\langle E',A',L',b\rangle} (\langle stmt'\rangle, \sigma')$$

where:

– *stmt* is a specification statement, meaning the location of the program counter for the specification, and $stmt'$ is the program text with the location of the program counter after the transition,
– σ and σ' are the states before and after the reaction respectively,
– E is the set of events in the environment before taking the transition
– L is the set of pending labels in the environment before taking the transition,
– E' is the set of events emitted by this transition,
– A' is the set of rendezvous labels agreed for this transition,
– L' is a set of pending labels, containing the pending labels after taking the transactions,
– b is a boolean flag indicating if the taken transition terminates (blocks) the instantaneous reaction or not.

Figure 4 shows the main statements in the language, and Table 1 and 2 show the functions defining the semantics for the transaction and reactive statements respectively. The statements to specify the behaviors of transactions are `exec_start` and `exec_done`, where `exec_start t` will denotes the beginning of a transaction `t`, synchronizing on rendezvous start(t), and posting a label pending(t) in the environment (to remember that transaction `t` is pending). Similarly, statement `exec_done t` denotes the completion of `t`, and synchronizes on rendezvous done(t), also removing the pending label `t` from the environment. The `exec_start t` and `exec_done t` statements are meant to be paired with `rv_rcv t` and `rv_snd t` rendezvous statements. The exec statements are to be used by the master process (the one initiating the transaction), and rendezvous statements are used by the slave process (the one receiving the transaction). The only difference between the exec and the rendezvous statement is the exec statements post and remove a transaction labels in the environment.

The rendezvous statements work like CSP rendezvous, with a slight variation to accommodate the synchrony hypothesis. The synchrony hypothesis, a concept defined in Esterel [2], is that at a given instant, all processes synchronously execute a sequence of statements instantaneously (up until the next pause). The `rv_snd a` and `rv_rcv a` statements synchronize on the shared action `a` only if that action is not in the preceding environment, and is present only in the output environment. This is to avoid the possibility of a rendezvous being taken twice during a synchronous reaction (synchronous as in synchrony hypothesis).

The watching statement is used to monitor events which will interrupt statement `stmt` when `bexpr` evaluate to true. When the condition evaluates to false, the watch computation will keep proceeding along `stmt` and its derivative (`stmt` can be a complex statement) until the termination of `stmt`. If the condition is true, then the computation of `stmt` will terminate immediately, and all the pending transactions will be killed, and L' will be empty. In other words, during a watch condition, if there is a pending transaction label, the transaction will be killed - including all transactions started by `stmt`. This ability to keep track of what transactions has been started, and be able to kill them in the event to a watch statement is the main feature of the reactive transaction specification language. This is the same as the hidden signals that are found in the composition operators (such as prefix) in process algebras, and used to simplify specifications. Note that it is possible to define scopes for the set L of pending labels to follow the hierarchical structure of the specification. But this leads to much more complicated semantic rules, which we will omit for the sake of space and simplicity.

The rest of the language borrows heavily from CRP, with the wait, emit, rendezvous, sequencing, choice, guarded actions and pause statements. The emit statement posts an event `e` into the set E', while a wait expression is evaluated in the incoming environment E. The pause statement terminate an instantaneous reaction. The language also has constructs for parallel compositions, arithmetic and Boolean expressions, and usual control flow statement etc. The syntax semantics of these and other statements in the language follow from the definitions Esterel and CSP with the addition of the transformation for the synchrony hypothesis, but are out of the scope of this paper.

```
stmt ::=
   exec_start t         /* start transaction t                    */
 | exec_done t          /* wait for transaction t to be done      */
 | rv_snd a             /* rendezvous at a (can send data)        */
 | rv_rcv a             /* rendezvous at a (can receive data)     */

 | do { stmt } watching bexpr         /* do/watching stmt         */
 | G(bexpr) {stmt} [] G(bexpr) {stmt}  /* guarded selection        */
 | stmt |C| stmt        /* choice                                 */
 | stmt ; stmt          /* sequence                               */

 | emit e               /* emit event e                           */
 | wait bexpr           /* wait for given boolean expression      */
 | pause                /* wait for a moment                      */
```

Fig. 4. Syntax of the specification language

Table 1. Semantics for the Transaction Statements

(rv-snd-1)
$$\frac{a \notin A}{(\texttt{rv\_snd a}, \sigma) \xrightarrow[\langle E,A,L\rangle]{\langle \emptyset,a,L,1\rangle} (\_, \sigma)}$$

(rv-snd-2)
$$(\texttt{rv\_snd a}, \sigma) \xrightarrow[\langle E,A,L\rangle]{\langle \emptyset,\emptyset,L,0\rangle} (\texttt{rv\_snd a}, \sigma)$$

(rv-rcv-1)
$$\frac{a \notin A}{(\texttt{rv\_rcv a}, \sigma) \xrightarrow[\langle E,A,L\rangle]{\langle \emptyset,a,L,1\rangle} (\_, \sigma)}$$

(rv-rcv-2)
$$(\texttt{rv\_rcv a}, \sigma) \xrightarrow[\langle E,A,L\rangle]{\langle \emptyset,\emptyset,L,0\rangle} (\texttt{rv\_rcv a}, \sigma)$$

(exec-start-1)
$$\frac{start(t) \notin A}{(\texttt{exec\_start t}, \sigma) \xrightarrow[\langle E,A,L\rangle]{\langle \emptyset,start(t),\{L \cup pending(t)\},1\rangle} (\_, \sigma)}$$

(exec-done-1)
$$\frac{done(t) \notin A}{(\texttt{exec\_done t}, \sigma) \xrightarrow[\langle E,A,L\rangle]{\langle \emptyset,done(t),\{L \setminus pending(t)\},1\rangle} (\_, \sigma)}$$

(exec-start-2)
$$(\texttt{exec\_start t}, \sigma) \xrightarrow[\langle E,A,L\rangle]{\langle \emptyset,\emptyset,L,0\rangle} (\texttt{exec\_start t}, \sigma)$$

(exec-done-2)
$$(\texttt{exec\_done t}, \sigma) \xrightarrow[\langle E,A,L\rangle]{\langle \emptyset,\emptyset,L,0\rangle} (\texttt{exec\_done t}, \sigma)$$

5 Verifiable Implementation in SystemC

In this section we discuss the following challenges in the verifiable implementation of reactive transactions:

1. How to have an implementation of reactivity through a simple architectural pattern that is generalizable for reactive transactions, and
2. How to correlate the atomic events in a transaction specification to the non-atomic handshakes in the SystemC code.

5.1 Reactivity Through Exceptions and Architectural Patterns

To implement reactivity within TLM models, we need to use the reactive features that were removed from SystemC a short time ago. These watching-and-waiting statements have been using exceptions to throw special conditions designating reset conditions [3]. For this purpose, we follow a similar pattern and we introduce a new wait macro, which we call MYWAIT :

```
#define MYWAIT(event_expr, reset_cond) \
    wait(event_expr); \
    if (reset_cond) \
      throw 1;
```

Table 2. Semantics for the Reactive Statements

(do-watching-1)

$$\frac{\sigma \not\models \mathit{bexpr} \qquad (\texttt{stmt}, \sigma) \xrightarrow[\langle E,A,L\rangle]{\langle E',A',L',b\rangle} (\texttt{stmt'}, \sigma')}{(\texttt{do \{stmt\} watching (bexpr)}, \sigma) \xrightarrow[\langle E,A,L\rangle]{\langle E',A',L',b\rangle} (\texttt{do \{stmt'\} watching (bexpr)}, \sigma')}$$

(do-watching-2)

$$\frac{\sigma \not\models \mathit{bexpr} \qquad (\texttt{stmt}, \sigma) \xrightarrow[\langle E,A,L\rangle]{\langle E',A',L',b\rangle} (\_, \sigma')}{(\texttt{do \{stmt\} watching (bexpr)}, \sigma) \xrightarrow[\langle E,A,L\rangle]{\langle E',A',L',b\rangle} (\_, \sigma')}$$

(do-watching-3)

$$\frac{\sigma \models \mathit{bexpr}}{(\texttt{do \{stmt\} watching (bexpr)}, \sigma) \xrightarrow[\langle E,A,L\rangle]{\langle \forall t \in L: kill(t), \emptyset, \emptyset, \rangle} (\_, \sigma)}$$

The macro defines a wait statement, which will wait on a given list of events. This will be a regular transaction event, or a reset event. The second parameter is the reset condition, and it identifies which event condition means the transaction has been reset, and if this condition holds the macro will throw an exception (here just an integer). An example of how to use this macro is as follows:

```
MYWAIT( (clk.posedge_event() | reset.posedge_event()),
        (reset.event() && reset==1) );
```

where the event expression is either a clock up-tick or a reset up-tick, and the reset expression a reset event and the reset signal to one. The MYWAIT macro is meant to be used inside a try/catch statement. Here is an example of a process which invokes a `write` transaction on a `ctrl` channel:

```
try {
  ctrl->write(1,1)
} catch (int reset_code) {
  ctrl->kill__write();
}
```

When a reset event occurs, the exception will be thrown from inside the `write()` method implementing the transaction inside the ctrl channel. The exception will be caught by the outer handler– not in the channel but in the component. In this case, the process can choose to kill the transaction in the server by calling the `kill__write()` method on the channel to send the kill signal to the server.

In this case, the handling of the reactivity can be done inside the component, but in general an architectural pattern with a transaction controller and a separate controller helper to handle the reactivity can be used. Figure 5 shows the architectural pattern to use to separate the reset conditions from the regular TLM processing. A controller processes transactions and dispatches sub-transactions.

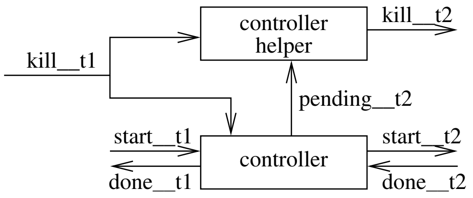

Fig. 5. Architectural pattern to propagate the transaction kills

With the architecture on the figure, assume a situation where a transaction `t1` is started, followed transaction `t2` being started by the controller, `t2` being necessary to complete `t1`. When `t2` is started, signal `pending__t2` is sent to the controller to tell that `t2` is pending. If `t1` is killed while `t2` is pending, then from `pending__t2` the controller helper will go ahead and kill `t2`.

This pattern is useful when a controller helper can process and keep track of all the simultaneous transactions. Then, the controller it does not have to be concerned about keeping track of which sub-transactions to kill, matching the idea the designer has when using the transaction description algebra. An underlying question is how to implement this controller helper.

5.2 Non-atomicity of Rendezvous and Kill Handlers

In the specification, a rendezvous is atomic. However, in the SystemC TLM implementation, a rendezvous is not atomic but a handshake. The master component synchronizes with the slave (also called the transaction server - or just server) through a method call that leads to an exchange using a TLM FIFO buffer. Until the slave has picked up the data from the buffer, the exchange cannot be considered done, but only in progress. In that sense, the challenge here is implementing the transactions with the reactive features is to manage the kills that occur during the handshakes that are in progress. When a kill occurs during that time, there has to be special conditions to correlate the non-atomic exchange to the atomic exchange in the specification.

Figure 6 shows the scenarios that can occur when kill happens during a handshake. Each of these scenarios requires a specific handling strategy which will make sure the buffers are emptied and the transaction in the slave is cleanly killed. The first case, illustrated Figure 6(a), is when t2 gets killed before it started; this assumes that the server of t2 will be able to eventually unblock and pick up the request from the buffer, see the `kill__t2` signal to be asserted, and then thus discard the request:

```
if ( pending__t2 and ready__t2 and full(t2_req_buf) ) {
  kill__t2 = 1;
  wait_until empty(t2_req_buf);
  kill__t2 = 0;
}
```

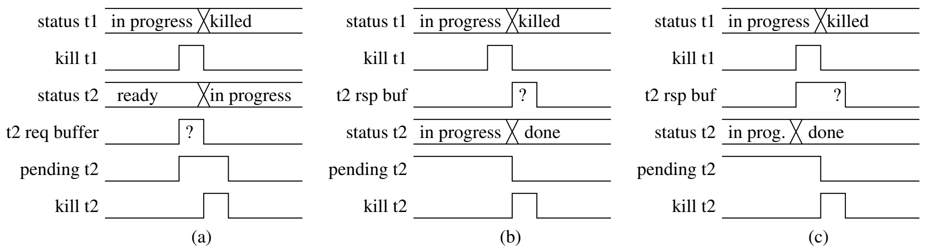

Fig. 6. Scenarios for handshakes with kill: (a) request posted but slave has not picked up yet, (b) slave is processing transaction, (c) slave is done but master has not picked up response yet

The second scenario is when `t2` terminates at the same time it gets killed. In that case, the handler might need to pick up and discard the response:

```
if (pending__t2 and in_progress__t2_) {
  kill__t2 = 1;
  wait_until ready(t2) or kill(t2) or done(t2);
  kill__t2 = 0;
  if (full(t2_rsp_buf))
    get(rsp_buf)
}
```

The third scenario is when `t2` is done serving the transaction, but the master has not yet picked up the response from the buffer. Then, the handler just picks up and discards the response from the buffer:

```
if (pending__t2 and ready__t2 and not full(t2_rep_buf)) {
  assert( full(t2_rsp_buf) );
  get(t2_rsp_buf);
}
```

We believe that, the transaction interfaces defined in our reactive transaction framework gives the tools to implement the handling strategies for reactive transactions. However, it is the *responsibility of the designer* to make sure there are no deadlocks and de-synchronization situations in the design. While we provide the signals and the patterns, correctly implementating the transaction controllers can be a challenging task. In that context, it is very valuable to have the verification support to be able to formally prove the correctness of the implementation.

6 Experiments and Results

Our verification setup is to use monitor-based model checking, where a monitor will check a SystemC component for any unallowed behaviors. Furthermore, we also use temporal logic formulas to ensure no deadlocks or stalls are reached. To convert a specification description into a verification monitor, we designed and implemented a Spec Analyzer tool. The conversion from specification to monitor directly follows the operational semantics rules, with the addition of the conversion for the synchrony hypothesis. The pass about the synchrony hypothesis is used to reduce a sequence of micro-transitions into one synchronous macro-transition, by following the termination flag (the b in the semantic rules). The Spec Analyzer generates a verification monitor which has a an error state which denotes a problem in the design, as well as a special state to handle the environment assumptions (whether we are "in-transaction" or not). Furthermore, using the Module Analyzer tool we previously presented [18], we convert the SystemC to a transition system described in an SMV file.

For the example, we implemented a simplified version of the transactional memory in SystemC with the reactive transactions library. Figure 7 shows the structure of the system we implemented and verified. The structure is the same as the one in Figure 1 with the addition of the reactive features. The program segment implements reactivity with a try/catch and takes care of the pending transactions in its catch handler. For the controller, we use the pattern with the controller helper as described in the previous section.

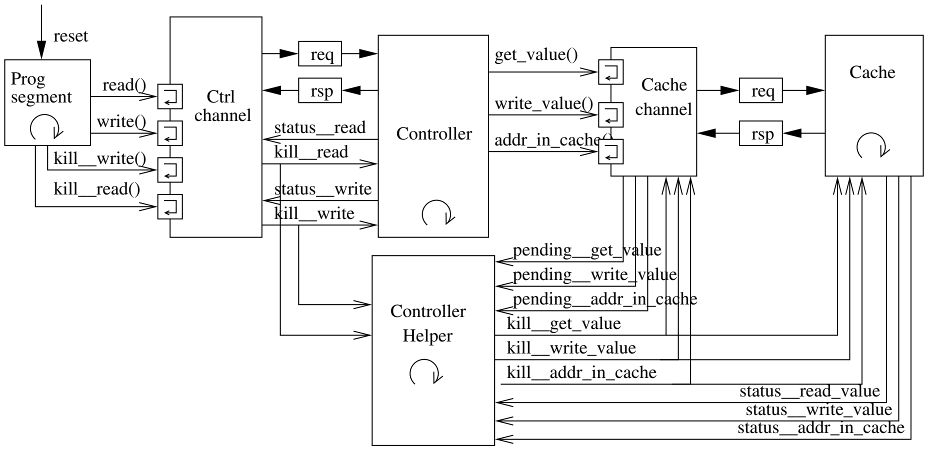

Fig. 7. Reactive architecture for the simplified Transactional Memory model

We have verified the design both at the system-level and at the component level. At the system-level, the global specification is an infinite sequence of read or write transactions that can be reset and restarted. We derived local component specifications from the global transactions for component-level verification.

Figure 8 lists the specification for the controller, which reads as follows: the controller will first wait for a rendezvous on either a `read_start` or `write_start` transaction. These transactions are to be initiated by the program segment using exec statements. Then, if controller picks up a read or a

```
 while (true) {
        rv_rcv read__start |C| rv_rcv write__start ;
        G( read__start && !read__kill ) {
                do {
                        exec_start addr_in_cache;
                        exec_done  addr_in_cache;
                        exec_start get_value;
                        exec_done  get_value;
                        rv_snd read__done
                } watching read__kill__posedge_event
        } [] G( write__start && !write__kill ) {
                do {
                        exec_start write_value;
                        exec_done  write_value;
                        rv_snd write__done
                } watching write__kill__posedge_event
        } [] G( (!(read__start && !read__kill )) &&
                (!(write__start && !write__kill)) ) {
                // other guards are false
        };
}
```

Fig. 8. Specification for the Controller

write transaction and it was not killed at the same instant, it will proceed on it; if the transaction was killed it will discard the request go back to the rendezvous. When the controller processes the read transaction, it will execute two sub-transactions and complete with a done rendezvous - all this while watching the kill read condition. If a kill read occurs, the controller shall return to the initial rendezvous, and the pending transactions will get killed by the combination of exec and watching statements. A write transaction works similarly. Note that the controller can wait for an arbitrary amount of time between the rendezvous.

Table 3 lists the verification results for the example using NuSMV 2.4.3. For each run, the property we verify are the following:

1. *Monitor assertions:* `AG !(monitor.state==ERROR)`
   This guarantees all the behaviors of the implementations are permitted by the transaction specification;
2. *C++ assertions:* `AG !(component.ERR)`
   When an assertion inside a C++/SystemC module fails, it will set the ERR flag. This is often used to monitor the conditions inside the modules.
3. *Liveness assertions:* `AG AF trans_starts` or `AG EF trans_starts`
   The liveness property specifies that we can always eventually start a new transaction, or there always is a path leading to the start of a new transaction, depending on how strong the property has to be. This will prove absence of stalls or deadlocks with respect to those events and the branching conditions.

The verification times for all the properties are compounded in the entries of Table 3. The transaction channels are inlined inside the SystemC components. To keep track of transactions implemented through method calls, start and done events are added at the boundary of methods calls. We also currently limit the sizes of the TLM buffers to one unit only. The verifiation times are reasonable, and in line with the verification times for other SystemC verification frameworks. However, we cannot fairly compare our numbers with the numbers from other verification frameworks because the other frameworks do not capture the reactive transactions as we do, thus the specification is different. As for the numbers in the table, one can notice that the verification of the Controller + Controller Helper takes significantly more time and space than for the other components. This is because the controller interacts with all components - thus all buffers are there - and its environment model has many constraints.

Table 3. Verification results (with NuSMV)

Configuration	*Time (sec)*	*Memory (KB)*
full system	671	102864
prog segment	41	19168
controller (+ controller helper)	483	97368
cache	131	40300

Note that we did not prove the compositionality of the specification, and this is outside the scope of this paper. The system-level verification is important to prove that the reset of nonatomic rendezvous avoids all integration problems. In our case, we found several integration bugs and this lead us of to formulate those conditions. One of the next steps is to generalize those conditions and elaborate a proof structure to avoid having to do the system-level verification.

7 Summary and Future Work

In this paper, we have presented an approach to specify, implement and verify reactive transactions in SystemC. To specify the transactions, We defined a language that implicitely keeps track of pending transaction and a watching statement is used to abstract away the bookeeping details of propagating the reactivity to sub-transactions (propagating the reset and kill events).

Many of the implementation efforts are spent on explicitely instantiating these signals in an verifiable implementation pattern. Indeed, we provide the sketch of an architectural pattern to implement the reactive transactions in SystemC, as well as an outline of the conditions to correlate the non-atomic SystemC implementation of atomic transaction events. Our third contribution is the verification path, currently supported by SMV-based model checkers.

One of the broader goal of this work is to exploit the compositionality in the transaction specifications, as well as, when possible, its reflection in the architecture and proof structure. We believe that the style of specification we have developed will be amenable to the decomposition and consistency checks that are necessary for to support this example. In that context we are also investigating using equivalence checking techniques to address the verification problem more directly.

Acknowledgments

The authors would like to acknowledge the anonymous reviewers for their excellent suggestions.

References

1. Groetker, T., Liao, S., Martin, G., Swan, S.: System Design with SystemC. Kluwer Academic Publishers, Dordrecht (2002)
2. Berry, G.: The Foundations of Esterel. MIT Press, Cambridge (2000)
3. Liao, S., Tjiang, S., Gupta, R.: An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment. In: Proc. of the Design Automation Conf. (1997)
4. Marschner, E., Deadman, B., Martin, G.: IP Reuse Hardening via Embedded Sugar Assertions. In: Proc. of the Int. Workshop on IP SOC Design (2002)
5. Balarin, F., Passerone, R.: Functional Verification Methodology Based on Formal Interface Specification and Transactor Generation. In: Proc. Design Automation and Test in Europe Conf. (2006)
6. Berry, G., Ramesh, S., Shyamasundar, R.K.: Communicating Reactive Processes. In: Proc. of the Symposium on Principles of Programming Languages (1993)
7. Seawright, A., Brewer, F.: Clairvoyant: A Synthesis System for Production-based Specifications. IEEE Trans. on Very Large Scale Integration (VLSI) Systems 2, 172–185 (1994)
8. Siegmund, R., Muller, D.: Automatic Synthesis of Communication Controller Hardware from Protocol Specification. IEEE Design and Test of Computer 19, 84–95 (2002)
9. Oliveira, M., Hu, A.: High-level Specification and Automatic Generation of IP Interface Monitors. In: Proc. of the Design Automation Conf. (2002)
10. Shimizu, K.: Writing, Verifying, and Exploiting Formal Specifications for Hardware Designs. PhD thesis, Stanford University (2002)
11. Zhu, Q., Oishi, R., Hasegawa, T., Nakata, T.: System-on-Chip Validation using UML and CWL. In: Proc. of the Int. Conf. on Hardware-Software Codesign and System Synthesis (2004)
12. Abarbanel, Y., Beer, I., Glushovsky, L., Keidar, S., Wolfsthal, Y.: FoCs: Automatic Generation of Simulation Checkers from Formal Specifications. In: Proc. of the Int. Conf. on Computer Aided Verification. pp. 538–542 (2000)
13. Cai, L., Gajski, D.: Transaction-level Modeling: an Overview. In: Proc. of the Int. Conf. on Hardware/Software Codesign and System Synthesis, pp. 19–24. ACM Press, New York (2003)

14. Habibi, A., Tahar, S.: Design and Verification of SystemC Transaction-level Models. IEEE Transactions on Very Large Scale Integration Systems 14, 57–68 (2006)
15. Moy, M., Maraninchi, F., Maillet-Contoz, L.: LusSy: A Toolbox for the Analysis of System-on-a-Chip at the Transactional Level. In: Proc. of the Int. Conf. on Application of Concurrency to System Design (2005)
16. Kroening, D., Sharygina, N.: Formal Verification of SystemC by Automatic Hardware/Software Partitioning. In: Proc. of the Int. Conf. on Formal Methods and Models for Codesign (2007)
17. Hoare, C.A.R.: Communicating Sequential Processes. Series in Computer Science. Prentice-Hall International, Englewood Cliffs (1985)
18. Shyamasundar, R., Doucet, F., Gupta, R., Krüger, I.H.: Compositional Reactive Semantics of SystemC and Verification in RuleBase. In: Proc. of the Workshop on Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems (2007)