Reactivity in SystemC Transaction-level Models

Frederic Doucet, R.K. Shyamasundar[1], I. H. Krueger, Saurabh Joshi[2], and Rajesh K. Gupta

University of California at San Diego
[1]IBM India Research Lab
[2]Indian Institute of Technology at Kanpur

Outline

- Introduction
- Motivating Example
- Challenges & Contributions
- Related work
- Specification of Reactive Transactions
- Verifiable Implementation in SystemC
- Verification Experiments and Results
- Summary and Future Work

Introduction

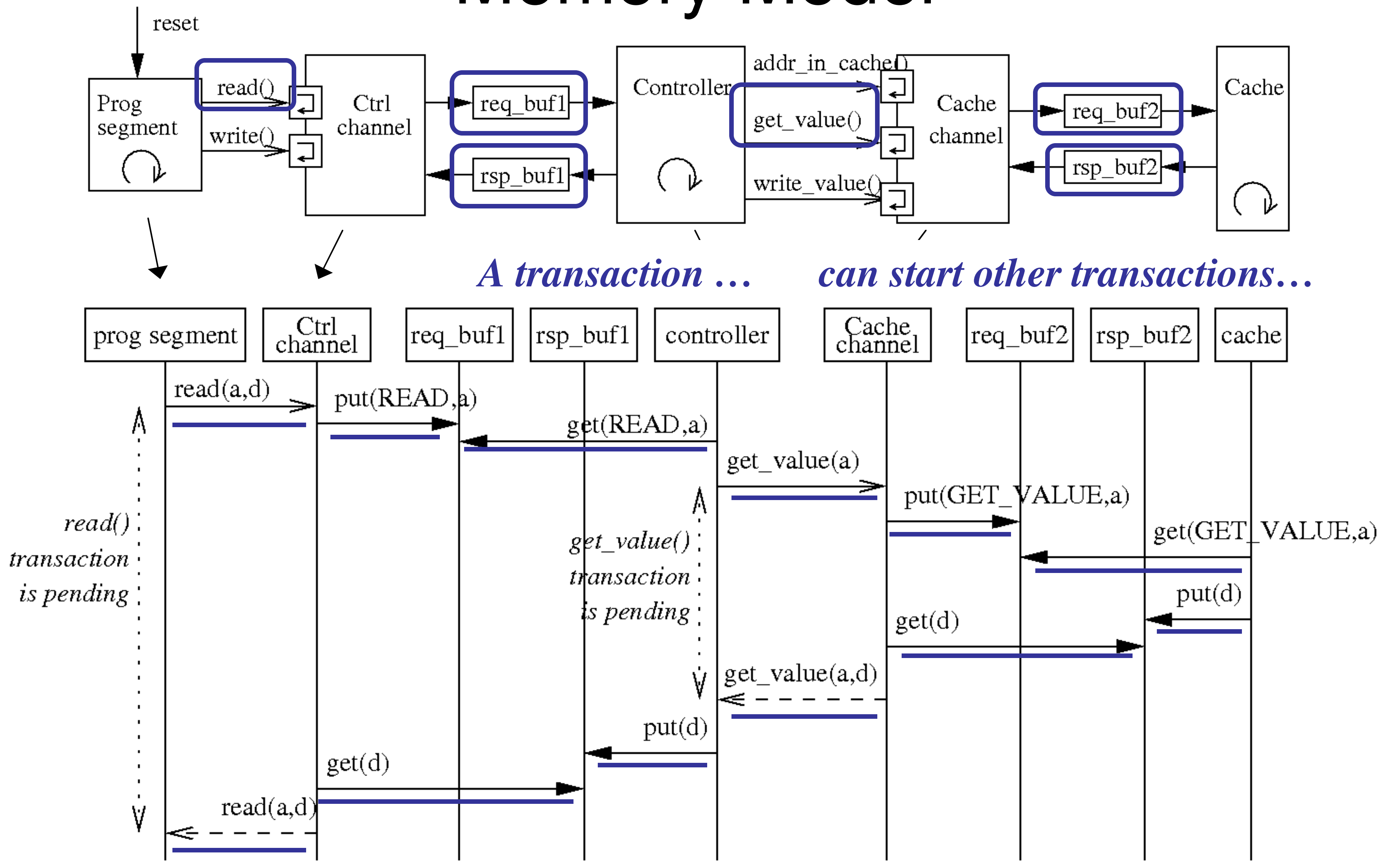
- SystemC high-level modeling of System-on-Chips
  - a set of class libraries to model hardware systems in a C++
    - processes, signals, modules, bits data types, scheduler, etc.
- Transaction-level Modeling (TLM)
  - a transaction is an abstraction of a sequence of events
    - FIFO buffer communication (carrying an abstract data type)
    - an interface method call (shared variable communication)
  - very useful to abstract low-level bus signaling details
  - provides a vast increase in simulation speed compared to RTL
    - because the model is much simpler
- Problem: no provisions for reactivity
  - found a need to extend TLM to capture *classical reactive features (reset or kill of a transaction)*
  - not possible to capture with current SystemC TLM libraries

Motivating Example: A Transactional Memory Model

- Memory architecture to exploit multi-core architecture

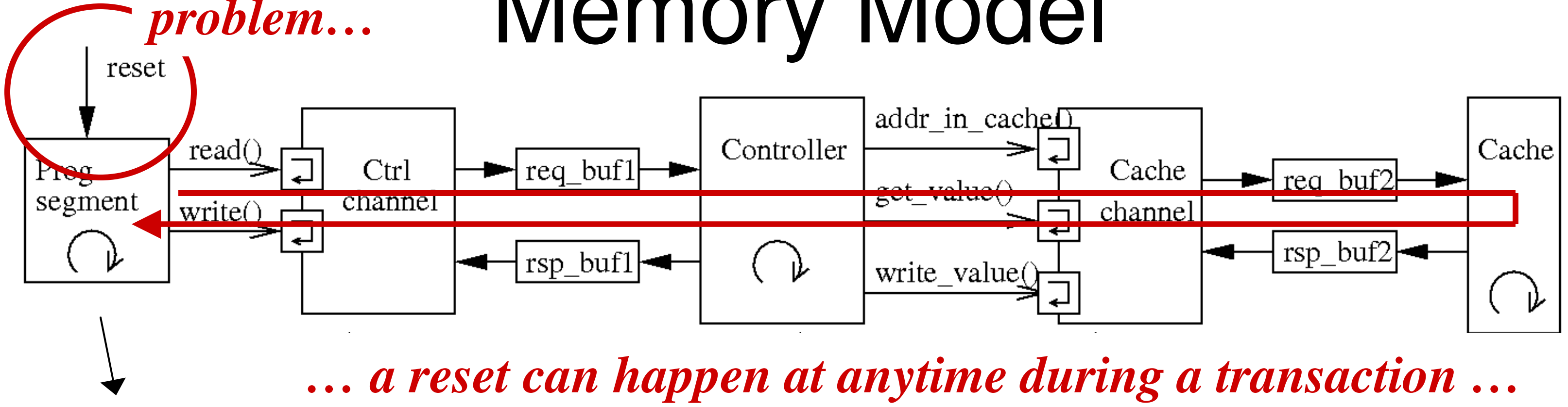
1. A program is split into many transactions
   - Program executes as phases - transactions with the memory
2. On a multi-core system - each transactions are executed concurrently and speculatively
   - Read data during the execution
     - Keep track of read-set
   - Write all data at once when done
3. When a transaction completes – conflict management
   - Writes data back to the memory
   - Other transactions listen to see if they have a data dependency
     - Is a value written to an address which is in the read-set?
     - If so, the transaction restarts

Motivating Example: A Transactional Memory Model

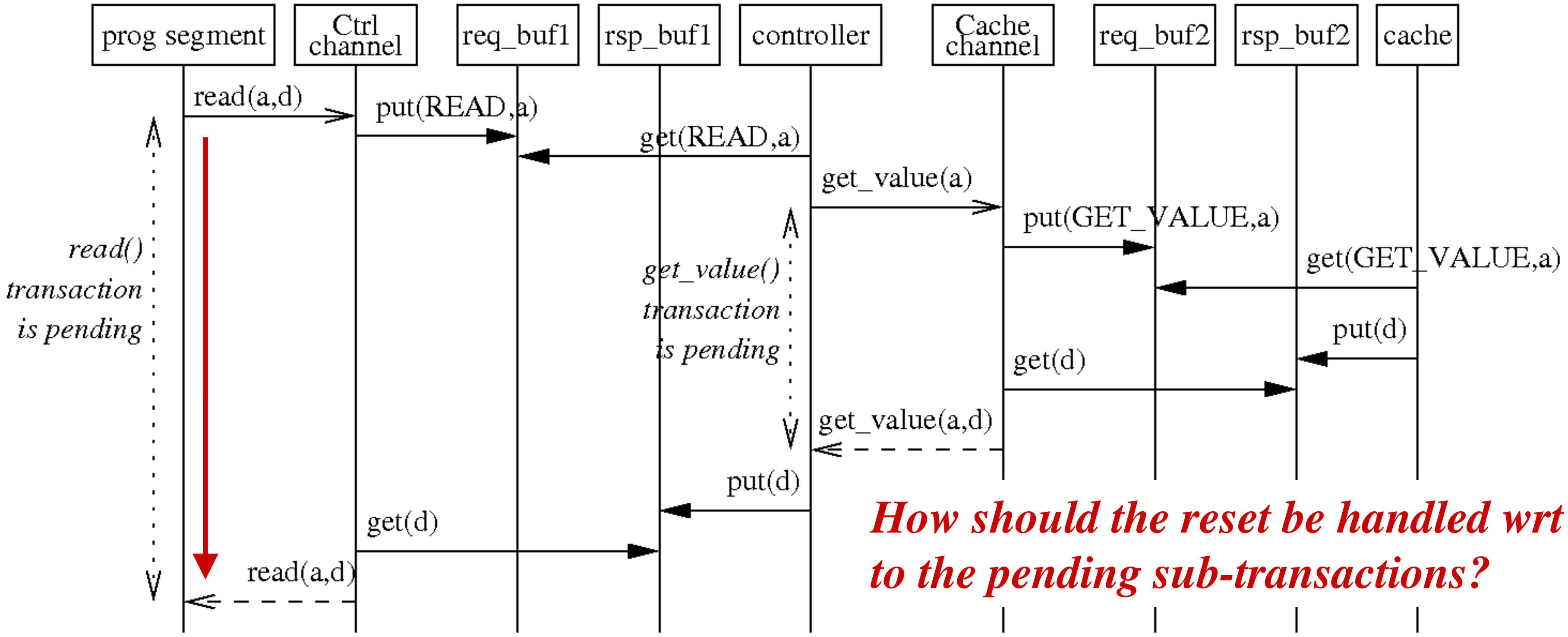


Motivating Example: A Transactional Memory Model

problem…

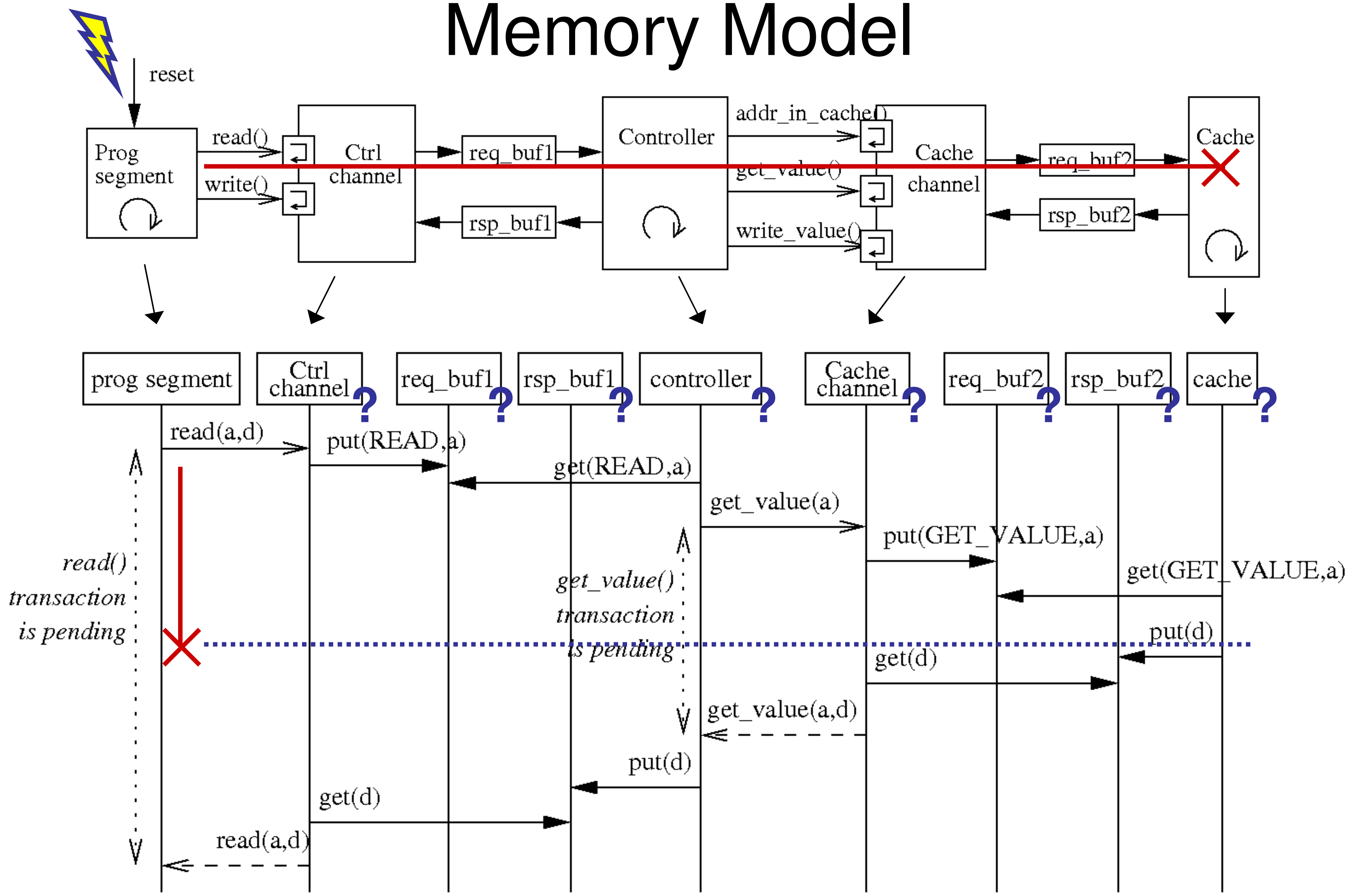


… a reset can happen at anytime during a transaction …



How should the reset be handled wrt to the pending sub-transactions?

Motivating Example: A Transactional Memory Model



Challenges

1. Specification of transactions and their compositions using property specification languages can be difficult
   - because of the semantics of SystemC TLM
   - many TLM events can happen simultaneously, makes for long and complicated properties
2. Implementation of reactive features in SystemC TLM is ad hoc
   - Reactive : respond only when events are received
   - killing/controlling the life and death of processes (do/watching statements a la Esterel to capture the reactivity)
   - atomicity of transaction events
     - in specification but not in the implementation (rendezvous vs buffered TLM communications)
3. Verification of the SystemC implementation of the transactions
   - existing approaches do not really support TLM
   - difficult to scale because → software verification

Contribution

We define an approach to specify, implement and reason about reactive transactions

- transactions that can be reset or killed before their completions
- relate atomic specification to non-atomic implementation of a transaction

Specifically, we provide:

1. A language to describe reactive transactions and their compositions as a first-order construct
2. An architectural pattern to capture reactivity and the cascading resets
3. A verification framework to verify implementation reactive transaction specifications

Outline

- Introduction
- Motivating Example
- Challenges & Contributions
- ➡ Related work
- Specification of Reactive Transactions
- Verifiable Implementation in SystemC
- Verification Experiments and Results
- Summary and Future Work

Related Work: Protocol Monitors

- A language is used to describe the communication protocol
  - automatically generate controller or a verification monitor

  - regular expression describing point-to-point communication and translation to state machines [Seawright et al - 1994] [Synopsys Protocol Compiler] [Sigmund et al. - 2002]
  - augment language with constructs for pipelines and registers – sophisticated synthesis algorithms [Oliveira et al. - 2002]
  - Language based on concurrent guarded transitions with extensive verification support [Shimizu et al. - 2002]
  - PSL and extensions used to describe module interface properties and communication protocols – efficient translations to monitors [Marschner et al. - 2002] [Balarin et al. - 2006] [IBM FoCs]

- *Protocols in this work*
  - capture the *reactive features* in the transaction and their compositions
  - use “watching” statement of CRP (Esterel + CSP)
  - SystemC TLM intricacy - possibly many events happening at an instant

Related Work : SystemC Verification

- Monitor-based approaches
  - Abstract State Machine-based [Habibi et al. - 2006]:
    - specification using PSL or MSC – translated into a monitor
    - can check for safety property using Microsoft ASML tools
  - synchronous frameworks-based
    - SIGNAL [Talpin et al. - 2003]
    - LUSTRE [Moy et al. - 2005]
- SMV-based approaches
  - predicate abstraction and other techniques scales well [Kroening et al. - 2006]
  - translation and verification of TLM subset [Shyamasundar et al. - 2007]
  - many efficient algorithms, and also includes liveness properties

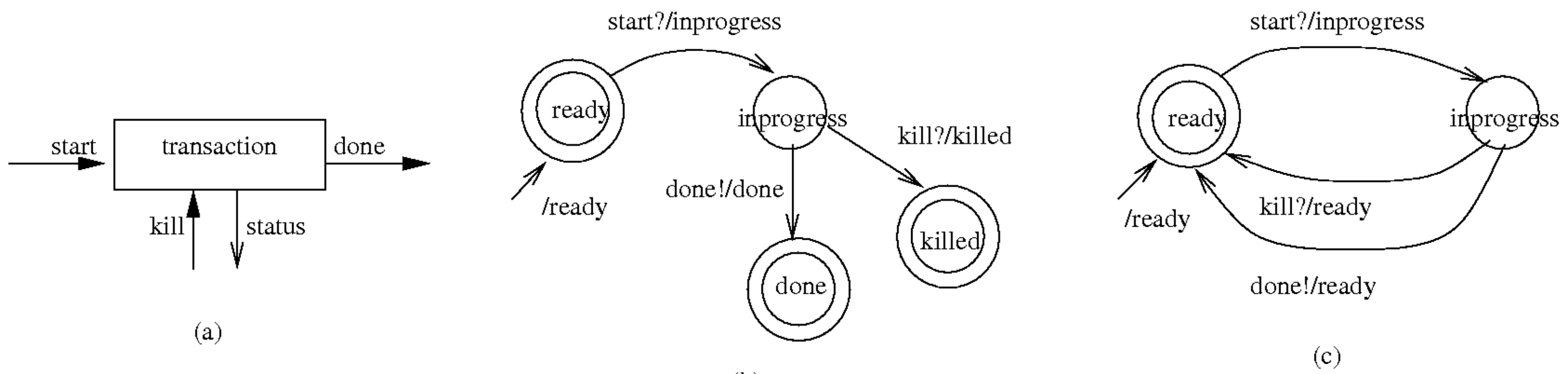
- *Verification in this work*
  - use the reactive transaction description to generate monitors
  - use an SMV-based verification engine to prove absence of deadlocks or stalls, and liveness properties.

Outline

- Introduction
- Motivating Example
- Challenges & Contributions
- Related work
- ➡ Specification of Reactive Transactions
- Verifiable Implementation in SystemC
- Verification Experiments and Results
- Summary and Future Work

Specification of Reactive Transactions

- A transaction as a first-order entity
  - provides a context and a simple interface - control signals



- Captures the control template
  - used to chain together many transactions
  - start and done can be mapped to other events
  - behavior can be distributed over many components
- Use a transaction-specific specification language

Syntax of Specification Language

Transactions are processes – a sequence of statements :

```
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                      */
```

We use the synchronous hypothesis - a la Esterel:

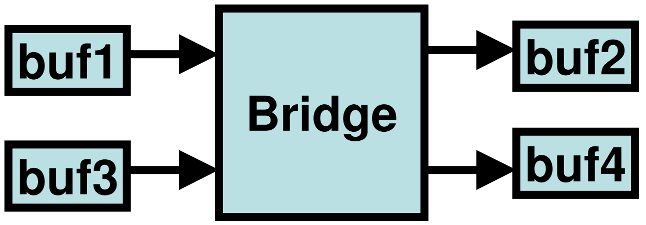
- processes can take many actions in one instant

A Note on SystemC TLM Semantics

```
SC_MODULE(Bridge) {
  sc_port<tlm_get_if<bool> > buf1;
  sc_port<tlm_put_if<bool> > buf2;
  sc_port<tlm_get_if<bool> > buf3;
  sc_port<tlm_put_if<bool> > buf4;

  SC_CTOR(Bridge) {
    SC_PROCESS(process);
  }

  void process() {
    bool val;
    while(true) {
 →    val = buf1->get();
      buf2->put(val);
      val = buf3->get();
      buf4->put(val);
    }
  }
};
```



- *Processes synchronize through the TLM buffers (FIFOs)*
- *In essence an asynchronous model*
  - Rendezvous maps to buffers
  - but core SystemC is synchronous

Many rendezvous can occur in a cycle…

```
– {buf1_get}
– {buf1_get, buf2_put}
– {buf1_get, buf2_put, buf3_get}
– {bu1_get, buf2_put, buf3_get, buf4_put}
```

… many micro steps -> one macro step

A monitor need to check for all these possible event combinations…

Semantics of Specification Language: Transition System

For each statement:

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

where:

- *stmt* : next statement at the program counter location
- σ [illegible]
- E : events in the environment
- A : set of actions in the environment
- L : pending labels in the environment
- b : flag indicating the termination of the reaction

Synchrony hypothesis: the instantaneous reaction keeps going until the b flag indicates the termination

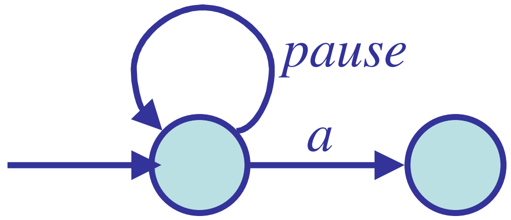
Semantics of Specification Language: Rendezvous

(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-rcv-1)

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



Both sending and receiving processes have to agree on a

At anytime, both processes can choose not to send

(rv-snd-2)

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

Semantics of Specification Language: Transactions

Transaction statements are also rendezvous

when a transaction starts, a pending transaction label is added to the environment

(exec-start-1)

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

when the transaction is done, the pending label is removed form the environment

(exec-done-1)

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

Semantics of Specification Language: Watching

Watch a process for a given condition:

(do-watching-1)

$$\frac{\sigma \not\models bexpr \qquad (\texttt{stmt1}, \sigma) \xrightarrow[\langle E,A,L\rangle]{\langle E',A',L',b\rangle} (\texttt{stmt1'}, \sigma')}{(\texttt{do \{stmt1\} watching (bexpr)}, \sigma) \xrightarrow[\langle E,A,L\rangle]{\langle E',A',L',b\rangle} (\texttt{do \{stmt1'\} watching (bexpr)}, \sigma')}$$

When the condition happens, kill all the pending transactions:

(do-watching-3)

$$\frac{\sigma \models bexpr}{(\texttt{do \{stmt1\} watching (bexpr)}, \sigma) \xrightarrow[\langle E,A,L\rangle]{\langle \forall t \in L : kill(t), \emptyset, \emptyset, 1\rangle} (\_, \sigma)}$$

Otherwise, just keep watching…

key idea: watch for the transaction kill events

Outline

- Introduction
- Motivating Example
- Challenges & Contributions
- Related work
- Specification of Reactive Transactions
- ➡ Verifiable Implementation in SystemC
- Verification Experiments and Results
- Summary and Future Work

Verifiable Implementation in SystemC

- Issues:
  1. Capture reactivity through exceptions to mimic watching statements
  2. Address the non-atomicity of rendezvous and reset handlers
  3. Provide an architectural patterns to keep track of the pending transactions

Reactivity and Exceptions

1. Define a wait macro that allows for the reset behavior:

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

2. In a ctrl->write() transaction, all the waits have to check for the reset condition:

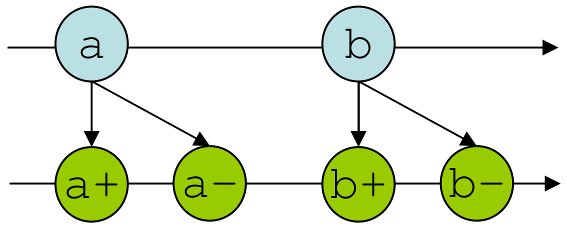
```
MYWAIT( (clk.posedge_event() | reset.posedge_event()),
        (reset.event() && reset ==1) );
```

3. Transactions are invoked in a try/catch block,
   - to propagate the reset conditions:

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

Non-atomicity Issues in Reset

- Correlation of atomic and non-atomic exchange
  - in specification, a transaction is started instantaneously
  - in the SystemC implementation, the events are not atomic
    - communication are buffered:
    - atomic events are implemented as handshakes between processes
    - example – req/ack protocol



- When a transaction reset happens
  - *the handshake needs to be cancelled*
    - similar to a CSP channel implementation

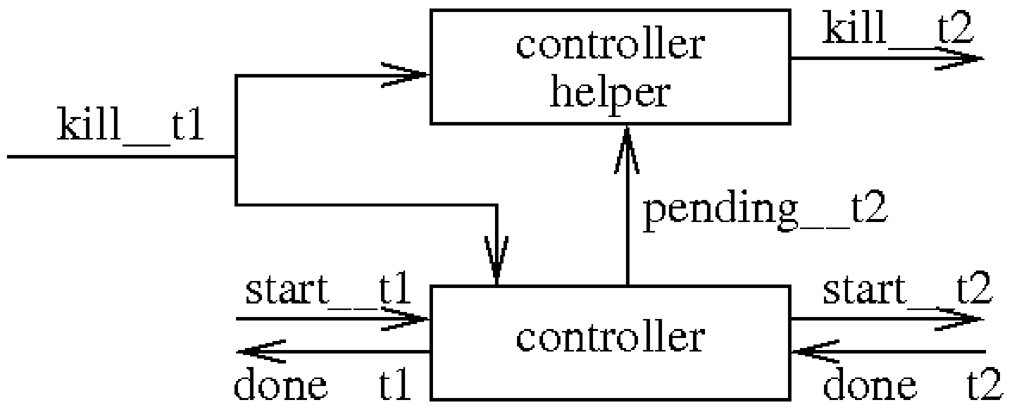
FD1

Slide 24

FD1 insert csp handshake cancel here?
Frederic Doucet, 19/10/2007

Architectural Pattern

- Provide the implementation construct and templates to keep track of transactions and handshakes
  - access the transaction status and control signals
  - monitor and reset the buffers



FD2

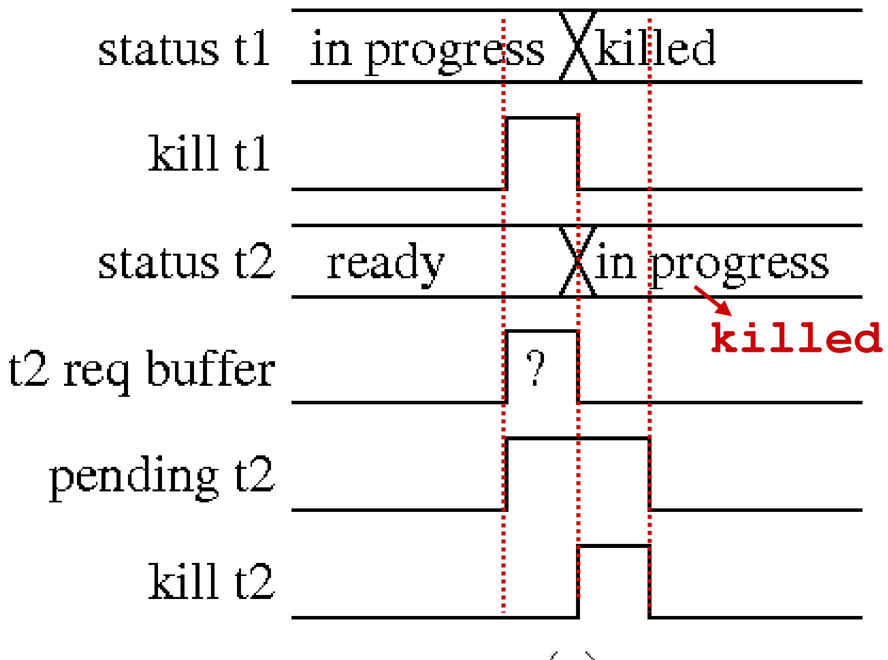
- Challenges:
  - a transaction server can process multiple transactions simultaneously
  - need to encode the product of states for all the interleavings of the concurrent transactions (with the corner cases)

Slide 25

FD2 ADD THE SDTATUS where is the status coming out from? req/rsp buffers

Frederic Doucet, 19/10/2007

Reset Scenarios #1

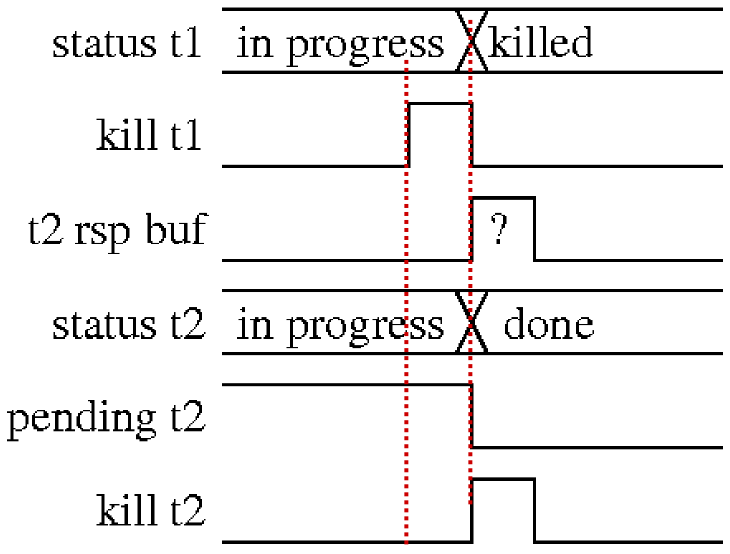


(a)

- t1 start t2,
  - t1 is killed before t2 actually starts
- Assumes that the server for t2 will:
  1. will eventually pick up the request
  2. will notice the kill__t2 is asserted
  3. will discard the request

```
if (pending__t2 and status__t2 == ready and req_buf__t2.full()) {
        kill__t2 = 1;
        wait until (req_buf__t2.empty());
        kill__t2 = 0;
        wait until (status__t2 == killed);
}
```

Reset Scenarios #2

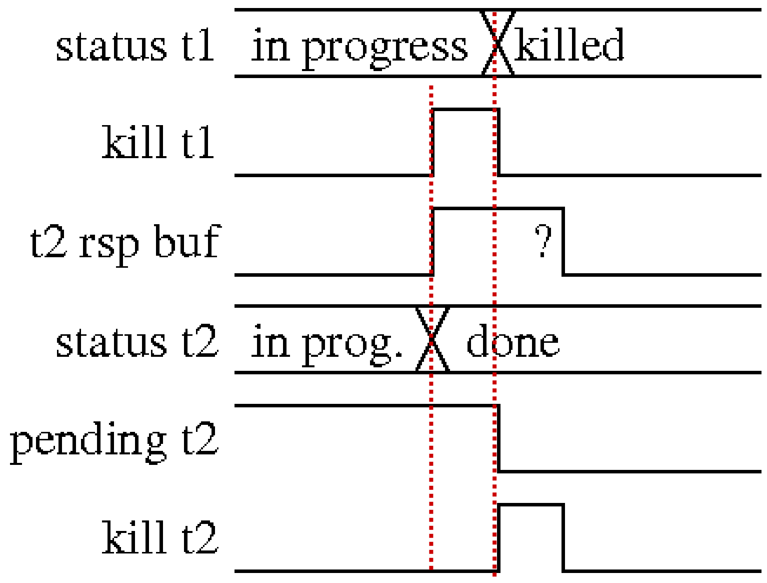


(b)

- t1 start t2,
  - t1 is killed at the same time as t2 completes
  - the handler might need to pick up and discard the response

```
if (pending__t2 and status__t2 == in_progress) {
        kill__t2 = 1;
        wait until ( status__t2 == killed r status__t2 == done );
        kill_t2 = 0;
        if (rsp_buf__t2.full())
                rsp_buf__t2.get();
}
```

Reset Scenarios #3



(c)

- t1 start t2
  - t1 is killed after t2 is done, but t1 has not yet picked up the response
  - the handler has to pick up an discard the response from t2

```
if (pending__t2 and status__t2 == done and req_buf__t2.full()) {
        assert (rsp_buf__t2);
        rsp_buf__t2.get();
        assert(status__t2 == done);
}
```

Implementation and Verification Issues

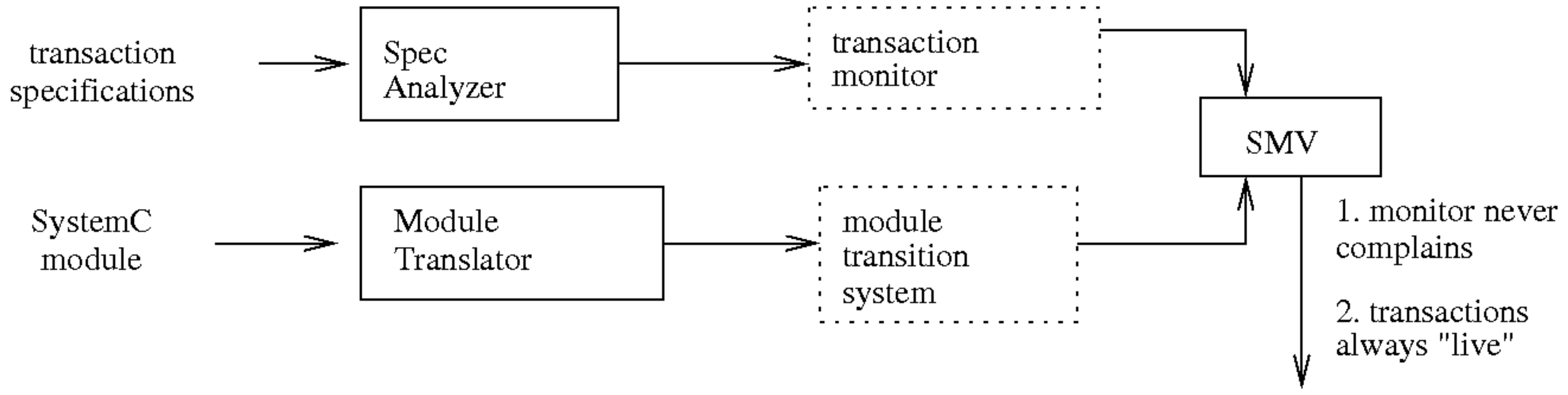
- Reset handling
  - architectural patterns provide the guideline
  - macros provide the extra statements
  - it is the responsibility of the designer to build the reset event handlers
    - we do not yet provide an algorithm to synthesize the controllers
- It is not easy to build such handlers
  - *the value of the verification framework*

Outline

- Introduction
- Motivating Example
- Challenges & Contributions
- Related work
- Specification of Reactive Transactions
- Verifiable Implementation in SystemC
- ➡ Verification Experiments and Results
- Summary and Future Work

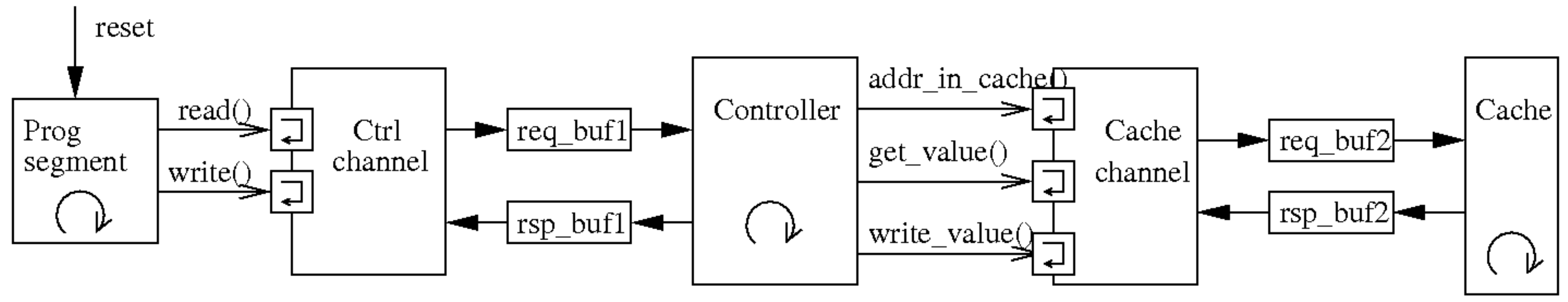
Verification Experiments

- Verify a simplified transactional memory controller
  - Automatic generation of transaction monitors
  - Automatic translation of SystemC modules into SMV modules



- We use the transaction specification language
  - Specify the global transaction specifications
  - Manually derive local component specification

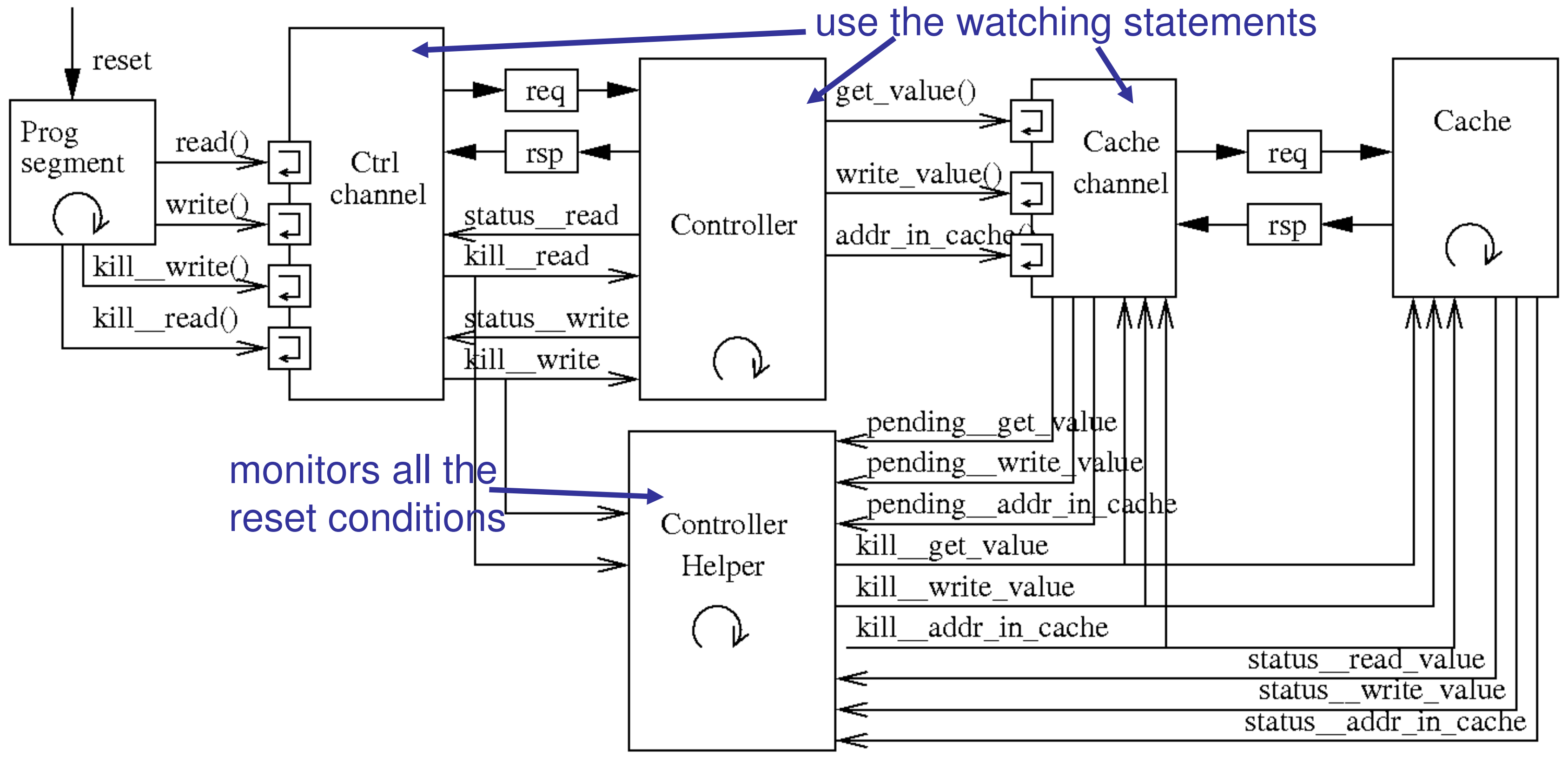
Specification view: conceptual model of the transactions



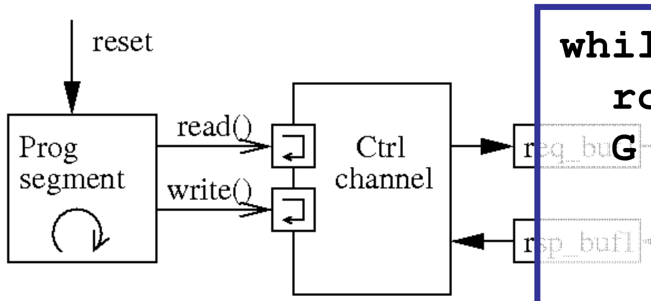
Implementation view: with the reset handlers

use the watching statements

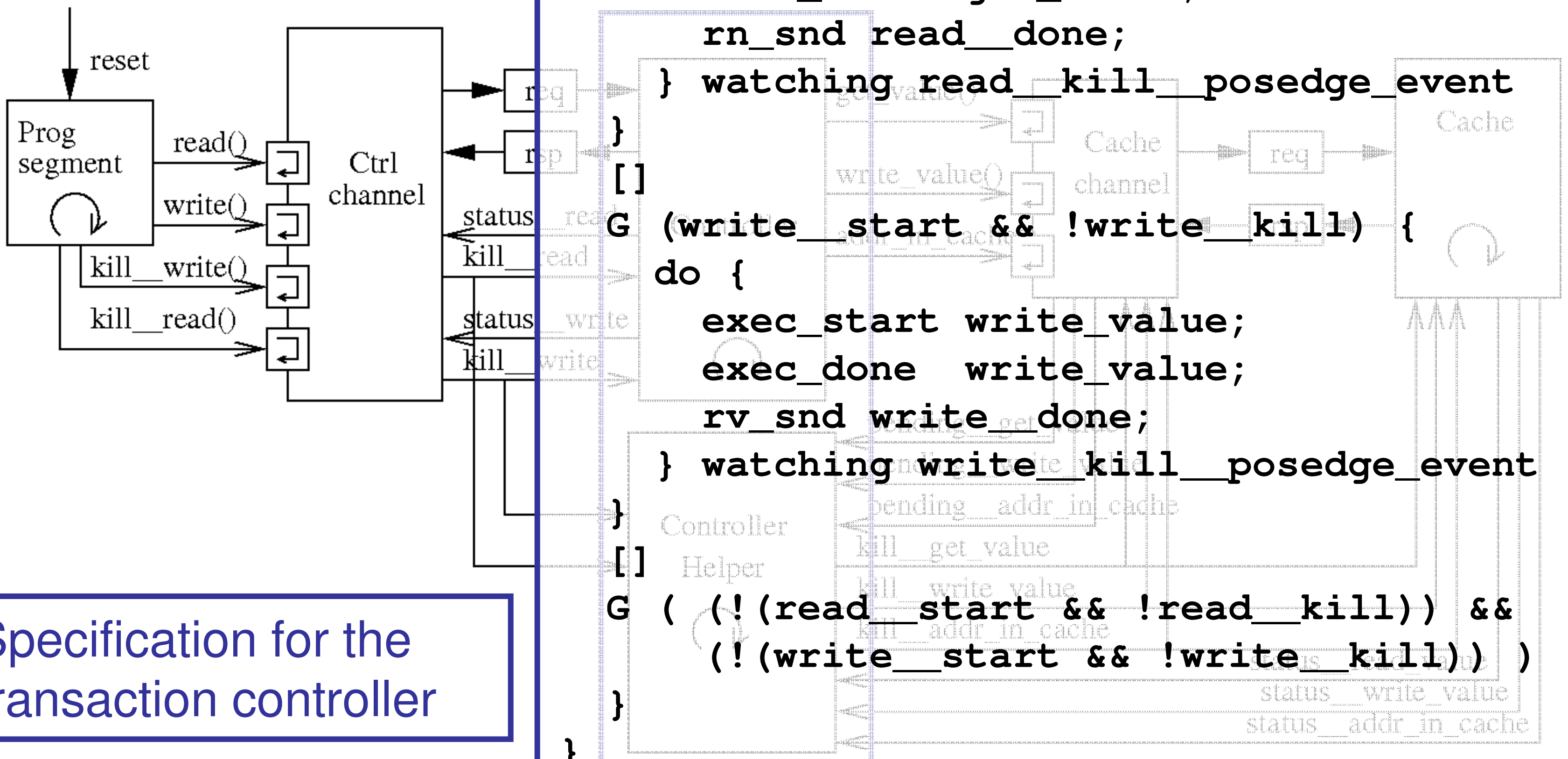
monitors all the reset conditions



Specification view: conceptual model of the transactions



Implementation view:

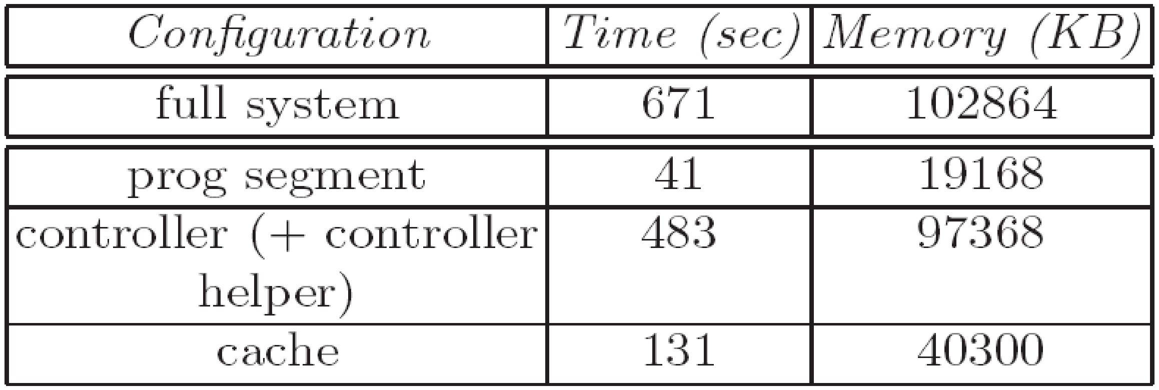


```
while (true) {
  rc_rcv read__start |C| rv_rcv write__start;
  G (read__start && !read__kill) {
    do {
      exec_start addr_in_cache;
      exec_done  addr_in_cache;
      exec_start get_value;
      exec_done  get_value;
      rn_snd read__done;
    } watching read__kill__posedge_event
  }
  []
  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)) ) {
  }
}
```

Specification for the transaction controller

Verification Results

Table 2. 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

- Properties are monitor assertions, C++ assertions, liveness assertions
  - the verification times are compounded in the table entries
- Found many bugs -
  - deadlock caused by buffers not being properly reset
  - in concurrent transactions – bad encoding of interleavings

Limitations

- Architectural pattern can be challenging to implement
  - User needs to keep track of many concurrent transactions
- User needs to write the top-level SMV file
  - with the environment fairness constraints
- Language-level limitations
  - support TLM buffers of size one only.
  - other constructs close the RTL subset
- Verification performance
  - is function of the efficiency of the SystemC translation
  - can be optimized further

Summary and Future Work

- Problem:
  - Need for reactive features in TLM models
- Contributions:
  - a specification language for reactive transactions
  - an architectural template to implement the reactive transactions
  - implementation constructs for reset/kill of transactions in TLM
  - a verification framework, including a tool for the generation of the transaction verification monitors
- Future work
  - Be able to generate the controller and reset channels
  - Automatic check for the composability of the specification
    - Address the issues in mixing asynchrony and synchrony
  - Improve verification performance

THE END

Thank you for listening