

# Linking Simulation with Formal Verification at a Higher Level

**Serdar Tasiran**

Koç University

**Brannon Batson**

Intel

**Yuan Yu**

Microsoft Research

*Editor's note:*

This article uses simulation to bridge the gap between specification and formal verification of high-level models and simulation of RTL models. The authors apply their practical, two-phase procedure for defining the refinement map to the Alpha 21364 multiprocessing hardware. The methodology and tools they present can improve simulation coverage.

—Carl Pixley, Synopsys

**VERIFICATION RESEARCH** has primarily focused on checking the properties of algorithm-level, executable formal specifications, with less attention on verifying the conformance of complex, highly concurrent hardware designs to higher-level descriptions—a labor- and computation-intensive process based largely on software simulation that, in practice, is never entirely completed.

RTL models for state-of-the-art designs are large, typically beyond the capacities of automatic formal tools that verify whether an implementation refines a specification. Compositional methods alleviate this difficulty, but coordinating their application by large design teams remains difficult. Consequently, verifying an implementation typically involves simulating the RTL model, using random patterns or handwritten test programs. Engineers simulate the reference models in parallel to check that their results match those of the implementation.

Simulation-based validation using informal specifications is inadequate for many reasons. Verifying whether an informal specification is consistent or satisfies certain desirable properties, such as the absence of deadlock, is often impossible. More importantly, it's difficult to quantify how well explored the different aspects

of the specification are and to provide automation for directing simulation toward unexplored areas. Recent techniques for coverage-guided simulation address the last issue to a certain extent. They define coverage metrics either on the implementation or on an abstract model tied to the implementation in an ad hoc way. Either way, the coverage

model is often a poor basis for test generation. In the former case, implementation-level models make automated test generation prohibitively expensive; in the latter, the coverage model might not contain enough information about the design or might be in a format that's unsuitable for verification and test-generation tools.

We've developed a novel coverage-guided validation technique that addresses these shortcomings. Our technique verifies that a hardware design described at the RTL is a correct implementation of an algorithm-level, executable formal specification. We use a high-level formal specification as the basis for monitoring functional correctness, measuring simulation coverage, and generating test cases. The specification we use is a protocol-level, executable model whose state variables correspond to important data structures and control finite state machines in the design. A refinement map transforms simulation steps in the implementation to state transitions at the specification level. A model checker then checks each specification-level state transition for consistency with the specification and collects coverage information using the specification states visited. When a specification violation is detect-

ed, the protocol-level state sequence produced by the map up to that point and the simulation trace help diagnose whether the error is in the implementation, the specification, or the map. Gaps in coverage and invariant violations at the specification level can be formulated as target states for the model checker, which is then used to generate traces to the target states. These traces provide useful starting points for generating implementation-level simulation runs to exercise the specification-level scenario under consideration.

## Preliminaries

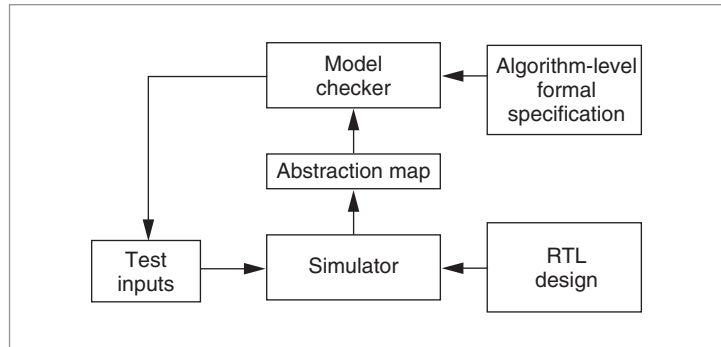
Figure 1 is a block diagram of our verification approach, which we used to verify the Alpha 21364 EV7 multiprocessing hardware. Our approach offers several benefits:

- Users can apply formal verification techniques to high-level specifications, where they're most suitable. These techniques detect and eliminate inconsistencies and property violations from the specification.<sup>1,2</sup>
- A formal verification tool checks each step of the simulation against the verified protocol-level specification. This tool distinguishes specification errors from bugs in the code that monitors the implementation for specification violations.
- A model checker detects implementation discrepancies from the specification as soon as they occur. Detecting these discrepancies early makes pinpointing their causes easier than later, when they can cause more easily observable errors such as data corruption.
- The model checker's counterexample generation facility facilitates simulation input generation to address gaps in specification coverage.

The price paid for these benefits is the work required for constructing a refinement map. To ease this task, we present an intuitive method for constructing refinement maps modularly.

## Multiprocessing hardware

The EV7 microprocessor supports glueless multiprocessing—that is, EV7 processors connected in a 2D torus function as a multiprocessor without needing extra chipsets. On-chip protocol engines collaborate with the level-2 cache controller to implement a directory-based cache-coherence protocol. The protocol engines can handle 64 outstanding protocol transactions simultaneously. The cache and memory controllers are deeply pipelined, highly optimized, complex circuits and are difficult to



**Figure 1. Simulation directed by a formal specification and a model checker.**

debug. Moreover, exercising certain aspects of the hardware for an EV7 multiprocessor system can require configurations with more than six processors. Exhaustive formal verification of such a system described at the implementation level is infeasible. Even simulating such a system is computationally demanding. The need to make better use of simulation resources was one of the motivating factors for our work.

## Formal specification

Designers wrote the EV7 processor specification in TLA+, a language for writing high-level formal specifications of concurrent and reactive systems based on the temporal logic of actions (TLA).<sup>3</sup> TLA+ supports constructs often needed for readable high-level specifications, such as sets, queues, records, and tuples. The language is well-suited for writing executable specifications that are essentially algorithm- or protocol-level models of a system. This latter feature distinguishes TLA+ from verification input languages aimed at writing properties at the RTL, such as the Process Specification Language (PSL)/Sugar<sup>4</sup> or OpenVera Assertions (<http://www.open-vera.com/>). Digital, Compaq, Hewlett-Packard, and Intel have used TLA+ and related tools to specify and verify concurrent designs.<sup>1,2,5</sup> A detailed presentation of TLA+ is beyond the scope of this article, but is available elsewhere.<sup>3</sup>

Figure 2 informally illustrates the features of TLA+ relevant to our purposes on a load-store unit example. The TLA+ **LoadStore** module is the high-level description for a hardware unit that controls load and store operations between a register file and the memory. Requests from a processor stay in a queue (**reqQ**) and are processed in order. The **TypeInvariant** formula states that

- variable **mem**, representing the memory, maps each address to a data value;

```

1 module LoadStore
2
3 extends Sequences
4 constant Req, Data, Addr, RegIndex
5 variables reqQ, reg, mem
6
7 TypeInvariant  $\triangleq$   $\wedge$  mem  $\in$  [ Addr  $\rightarrow$  Data ]
8    $\wedge$  reg  $\in$  [ RegIndex  $\rightarrow$  Data ]
9    $\wedge$  reqQ  $\in$  Seq([ reg : Reg,
10     addr: Addr  $\cup$  {NotAddr},
11     data : Data  $\cup$  {NotData},
12     regInd : RegIndex])
13
14 EntryReady(e) =  $\wedge$  e.addr  $\neq$  NotAddr
15    $\wedge$  e.data  $\neq$  NotData
16
17 ProcessLoadEntry  $\triangleq$  LETentry  $\triangleq$  Head(reqQ)
18   IN  $\wedge$  Len(reqQ) > 0
19    $\wedge$  EntryReady(entry)
20    $\wedge$  entry.req = "Load"
21    $\wedge$  reg' = [reg EXCEPT ![entry.regInd] = entry.data]
22    $\wedge$  reqQ' = Tail (reqQ)
23    $\wedge$  UNCHANGED mem
24
25 ProcessStoreEntry  $\triangleq$  LET entry  $\triangleq$  Head(reqQ)
26   IN  $\wedge$  Len(reqQ) > 0
27    $\wedge$  EntryReady(entry)
28    $\wedge$  entry.req = "Store"
29    $\wedge$  mem' = [mem EXCEPT ![entry.addr] = entry.data]
30    $\wedge$  reqQ' = Tail (reqQ)
31    $\wedge$  UNCHANGED reg
32
33 IssueLoadRequest(regInd, addr)  $\triangleq$ 
34    $\wedge$  reqQ' = Append(reqQ, [req  $\rightarrow$  "Load," addr  $\rightarrow$ 
35     addr, data  $\rightarrow$  NotData, regInd  $\rightarrow$  regInd])
36    $\wedge$  UNCHANGED .mem, reqQ
37
38 IssueStoreRequest (regInd, addr)  $\triangleq$ 
39    $\wedge$  reqQ' = Append(reqQ, [req  $\rightarrow$  "Store," addr  $\rightarrow$ 
40     addr, data  $\rightarrow$  reg[regInd].data, regInd  $\rightarrow$  regInd])
41    $\wedge$  UNCHANGED .mem, req
42
43 Next  $\triangleq$   $\vee$  ProcessLoadEntry
44    $\vee$  ProcessStoreEntry
45    $\vee$   $\exists$  regInd  $\in$  RegIndex :  $\exists$  addr  $\in$  Addr :
46     IssueLoadRequest(regInd, addr)
47    $\vee$   $\exists$  regInd  $\in$  RegIndex :  $\exists$  addr  $\in$  Addr :
48     IssueStoreRequest (regInd, addr)
49    $\vee$   $\exists$  rqInd  $\in$  Nats : MemToLS(rqInd)

```

**Figure 2. TLA+ code fragment for the load-and-store example. The TLA+ language is based on the temporal logic of actions (TLA) and is used for writing high-level formal specifications of concurrent and reactive systems.**

- variable **reg**, representing the register file, maps register indexes to data values; and
- request queue (**reqQ**) is a finite sequence of records, each having four fields: request type (**req**), address (**addr**), data (**data**), and register index (**regInd**).

The **NotData** and **NotAddr** values represent invalid data and address fields.

TLA+ specifications consist of state variables whose evolution is described by atomically executed actions:

- **ProcessLoadEntry** and **ProcessStoreEntry** describe the atomic processing of load and store requests at the head of the queue.
- **IssueLoadRequest** and **IssueStoreRequest** model the processor and atomically add a request to the tail of the request queue.
- **MemToLS** (not shown in the figure) transfers data from the memory to the **rqInd**th entry (a load) in the request queue.
- **Next** states that the unit can at any time execute one of these actions atomically—for instance, it can add an arbitrary request to the request queue or transfer data from the memory to some entry in the request queue.

The **ProcessStoreEntry** action requires that **EntryReady(entry)** hold (that is, the **.addr** and **.data** fields are properly filled) and that the request be a **Store**. It writes **entry.data** to **mem[addr]** (line 29 in Figure 2) and removes the top entry from the queue (line 30).

#### Cache-coherence protocol

We applied our technique to the EV7 implementation verification at a stage at which both the implementation and specification were fairly mature, with the intention of performing a feasibility study.

Architects, in consultation with formal verification researchers, wrote the EV7 cache-coherence protocol specification in TLA+. The protocol's original specification consisted of several high-level textual descriptions whose structure paralleled that of the EV7 multiprocessing hardware. The original specification classified protocol transaction scenarios according to the states of the hardware blocks involved in the transaction. As a result, in addition to encapsulating the protocol design, the for-

mal specification reflected aspects of the hardware structure and architecture and contained local safety properties in the form of assertions. These features distinguish the EV7's TLA+ specification from specifications consisting of property lists.

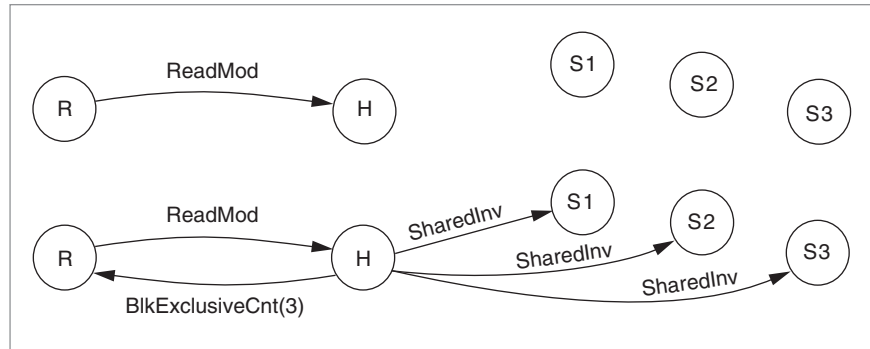
Writing the formal specification required translating informal specifications into a formal TLA+ description at roughly the same abstraction level. The end result was a specification consisting of approximately 2,000 lines of TLA+ code. This process required rigorous thought about the protocol, the implementation, and how they relate. The architects found the process very beneficial.

The directory-based EV7 cache-coherence protocol implements shared memory. Each processor owns the portion of the address space that is physically attached to it, and is called the *home node* for these addresses. The memory controller at the home node tracks outstanding protocol transactions for these addresses and serializes requests for a particular address.

The protocol specification is parameterized by the number of processors and memory addresses being considered. The three specification state variables follow the design's structure. These variables represent

- the cache controller state (**Cbox**),
- the memory controller state (**Zbox** or **DIFT**), and
- messages in flight between processors (**Net**).

Each action in the EV7 protocol specification pertains to one phase of a protocol scenario, such as the example scenario in Figure 3. Figure 4 shows a disjunct in the TLA+ action **ZboxRecvLPResp** describing the protocol phase in Figure 3. In this phase, the requestor node (**R**) sends the home node (**H**) a request for exclusive ownership of an address (**ReadMod**). The home node looks up the states of the cache and the victim buffer (encapsulated in the **probe** variable in line 15), the directory (line 16), and the memory controller (line 14). (A few omitted lines state that the next **Zbox** entry to be processed for this address contains the **ReadMod** command.) The protocol engine then sends invalidate



**Figure 3. EV7 cache-coherence protocol example scenario.**

```

1 MODULE EV7 Protocol Snippet
2 ZboxRecvLPResp(addr) Δ
3   LET ReleaseDIFTEntry(addr) Δ
4     DIFT' = [DIFT EXCEPT! [addr] = [reqQ → Tail(@.reqQ),
5       state → "None",
6       vicseen → "None"]]
7   SetDir (addr, state, sharers) Δ
8     Dir ← [Dir except ![addr] =
9       [owners → sharers, state → state]]
10  SetDirExclusiveIfRemote(pid) Δ
11  IF pid = HomeNode(addr)
12  THEN SetDir (addr, "Stale", StaleOwners)
13  ELSE SetDir (addr, "Exclusive", {pid})
14  IN ∧ dentry.cmd = "ReadMod"
15  ∧ probe = "Mem"
16  ∧ Dir [addr].state = "Shared"
17  ∧ let invaldests Δ Dir [addr].owners \ {dentry.reqPID}
18  IN Send(BlkExclusiveCnt(dentry.reqPID,
19    CohCnt(invaldests))
20    ∪ SharedInvalSingles(invaldests))
21  ∧ SetDirExclusiveIfRemote(dentry.reqPID)
22  ∧ ReleaseDIFTEntry(addr)

```

**Figure 4. Excerpt from the TLA+ specification for the home Zbox phase of the protocol scenario in Figure 3.**

messages (**SharedInv**, line 20) to the nodes holding copies of that cache line (sharers, or **S**). The home node also sends a **BlkExclusiveCnt** (line 18) message to the requestor indicating how many invalidate acknowledgments to wait for before modifying the cache line (line 19).

#### Property verification

Because we don't focus on property verification of the protocol, we discuss it only briefly and refer readers to earlier articles.<sup>1,2</sup> Prior to the effort this article describes, we used the explicit state TLA+ model checker (TLC)<sup>2</sup> to check the specification for consistency and prove properties on small multiprocessor configurations. The number of addresses, processors, and requests per processor define a configuration's size. For

**Table 1. Results of running TLA+ model checker (TLC) version 1.66 on a 625-MHz Alpha workstation.**

| Requests<br>per process | No. of processes |                         |                                     |                      |                          |                                     |
|-------------------------|------------------|-------------------------|-------------------------------------|----------------------|--------------------------|-------------------------------------|
|                         | Three            |                         |                                     | Four                 |                          |                                     |
|                         | Runtime          | Size (no.<br>of states) | Diameter<br>(no. of<br>transitions) | Runtime              | Size (no.<br>of states)  | Diameter<br>(no. of<br>transitions) |
| 1                       | 44 seconds       | 3,000                   | 15                                  | 21 minutes           | 53,000                   | 21                                  |
| 2                       | 33 minutes       | 79,000                  | 25                                  | More than<br>8 hours | More than<br>1.2 million | Unknown                             |
| 3                       | 5 hours          | 700,000                 | 38                                  | More than<br>8 hours | More than<br>2.8 million | Unknown                             |

\* Diameter = Minimum number of transitions required to reach the farthest state.

larger configurations, we proved safety properties manually on the specification.

Property verification of the EV7 specification suffers from the state-space explosion problem—that is, the size of the state space increases sharply with configuration size. Although this problem is orders of magnitude less severe at this level than for an implementation-level model, it still represents the computational bottleneck of our approach, as Table 1 demonstrates.

Despite this bottleneck, a TLA+ specification provides significant benefits beyond acting as a high-level monitor. First, TLC can handle many interesting configurations, such as the late-stage design bug we describe later. Improvements to model-checking technology, such as satisfiability-based methods, will make it possible to handle larger configurations. Second, compositional methods make it possible to model-check much larger configurations. Although difficult to apply at the implementation level for complex designs, compositional techniques are applicable at the TLA+ level. Finally, where TLC and compositional methods still fall short, theorem-proving techniques and limited formal analysis, such as consistency checking or approximate and partial model checking, can still be applied to the specification.

### Constructing the refinement map

The refinement map is the key link in our hybrid verification method. It allows the model checker to be used where it is most suitable—the specification level—while obtaining the implementation’s behaviors by simulating RTL code, which most closely models the implementation.

#### Formalizing refinement

We use *state transition systems* as the

semantics for specifications and hardware implementations. A state transition system is a tuple  $(V, S, s_0, \delta)$ , where

- $V$  is the set of **state variables**;
- $S$  is the set of states in which each state is a value assigned to each variable in  $V$ ;
- $s_0 \in S$  is the initial state; and
- $\delta$  is the transition function from  $S \times \mathbf{Actions}$  to  $S$ , where  $\mathbf{Actions}$  is the set of actions the system can perform.

The transition relation for the specification is obtained from the TLA+ description. Each TLA+ action directly corresponds to an action in the specification’s state transition system ( $\mathbf{Spec}$ ). We similarly obtain the actions for the implementation state transition system ( $\mathbf{Impl}$ ) from the RTL code. If  $\delta(s, a) = s'$ , the transition system can perform action  $a$  in state  $s$  to change the state to  $s'$ . We denote such a transition by  $s \xrightarrow{a} s'$ . Given state  $s$  and action  $a$ , at most one state  $s'$  can exist such that  $s \xrightarrow{a} s'$ . The system nondeterministically picks among possible actions  $a$  at state  $s$ . A state transition system  $\mathbf{run}$  is a finite sequence,  $\rho = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$ , for some  $n \geq 0$  such that  $s_i \xrightarrow{a_i} s_{i+1}$  for all  $0 \leq i < n$ .

We call an implementation state  $s^{\mathbf{Impl}}$  quiescent if no protocol phase is in progress at  $s^{\mathbf{Impl}}$ ; that is, all started phases have been completed. We assume that at any quiescent state  $s^{\mathbf{Impl}}$ , a function  $f_{\text{qui}}(s^{\mathbf{Impl}})$  gives the corresponding specification state. Function  $f_{\text{qui}}$  describes the hardware encoding of the specification state variables.  $\mathbf{Impl}$  refines  $\mathbf{Spec}$  if for every run  $\rho^{\mathbf{Impl}} = s_0^{\mathbf{Impl}} \xrightarrow{a_1} s_1^{\mathbf{Impl}} \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n^{\mathbf{Impl}}$  of  $\mathbf{Impl}$  that ends in a quiescent state, there is a run,  $\rho^{\mathbf{Spec}} = s_0^{\mathbf{Spec}} \xrightarrow{a_1} s_1^{\mathbf{Spec}} \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n^{\mathbf{Spec}}$ , of  $\mathbf{Spec}$  such that

## Related work

Our approach for constructing refinement maps uses some of the same concepts as Park and Dill's *aggregation of distributed transactions* method.<sup>1</sup> In their method, when a commit point is encountered, the refinement map computes the lower-level state that would be reached if the system was run to quiescence with no additional transactions (the **completion point**), and computes the corresponding high-level state using a projection similar to  $f_{qui}$  (described in the main article). Making the completion of a transaction part of the refinement map in this way amounts to stopping simulation at each step and simulating the part of the hardware that is processing the transaction to quiescence. This is not practical for complex multiprocessor systems. Because we're not performing a static check, we can avoid this difficulty by attributing hardware state transitions to individual transactions and waiting until a transaction is complete to perform the corresponding abstract state update.

Like us, Shimizu and Dill use the same formal description for collecting coverage information and deriving simulation inputs.<sup>2</sup> Their formal description is a list of interface properties describing a bus protocol. Our work differs in several ways:

- The EV7 cache-coherence protocol specification is at a more abstract level than their cycle-accurate, RTL interface specifications.

- In Shimizu and Dill,<sup>2</sup> properties are localized in time—for instance, properties can't express constraints on bus protocol transactions.
- The EV7 specification reflects the multiprocessing engine's internal architecture and is better suited for measuring coverage and directing simulation.

The EV7 protocol specification can also serve as a detailed functional coverage model similar to those Lachish et al. present.<sup>3</sup> Our specification addresses a key issue—determining whether coverage gaps are true deficiencies in validation—because it's executable, and verification tools that can run on it are available.<sup>3</sup> Lachish et al. discuss techniques that can be used to limit the number of coverage targets for TLC.<sup>3</sup>

## References

1. S. Park and D.L. Dill, "Verification of Cache-Coherence Protocols by Aggregation of Distributed Transactions," *Theory Computing Systems*, vol. 31, 1998, pp. 355-376.
2. K. Shimizu and D.L. Dill, "Deriving a Simulation Input Generator and a Coverage Metric from a Formal Specification," *Proc. 39th Design Automation Conf. (DAC 02)*, ACM Press, 2002, pp. 801-806.
3. O. Lachish et al., "Hole Analysis for Functional Coverage Data," *Proc. 39th Design Automation Conf. (DAC 02)*, ACM Press, 2002, pp. 807-812.

$$f_{qui}(s_n^{Impl}) = s_n^{Spec} \quad (1)$$

Burch and Dill describe this same refinement condition in their processor verification work,<sup>6</sup> and Park and Dill,<sup>7</sup> among others, describe a similar condition in their work on protocol verification (see the "Related work" sidebar).

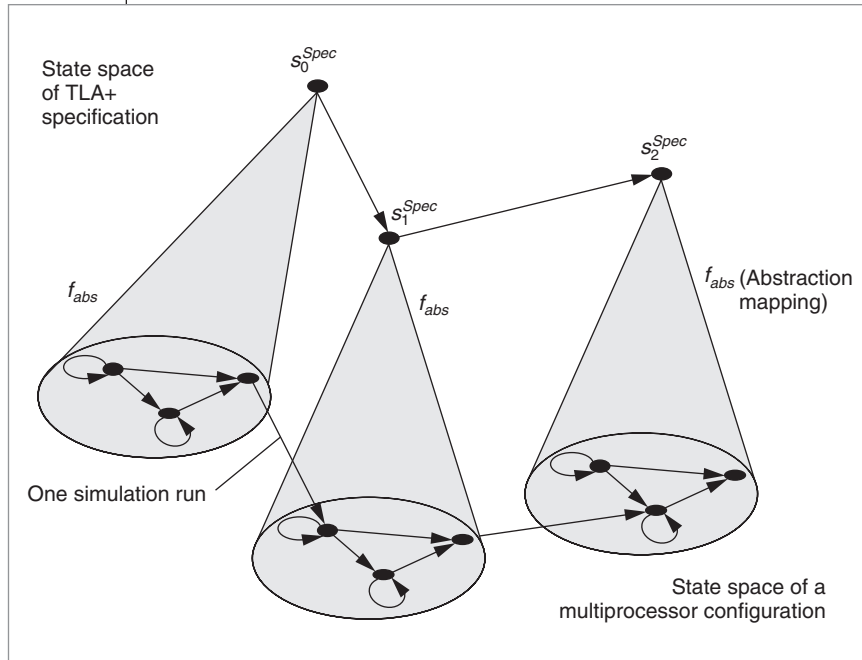
Quiescent states can be rare during multiprocessor runs. Condition 1 is better suited as a refinement criterion for static, exhaustive methods performing this check for all implementation runs  $\rho^{Spec}$ . For refinement checking during simulation, condition 1 doesn't give enough visibility into the design. By the time the multiprocessor reaches a quiescent state, it might have overwritten errors, or the verification engineer might be unable to determine the causes of the errors because they're too far in the past. We instead perform a more stringent, detailed check that's sufficient condition for refinement condition 1. We require that a refinement

map, such as that in Figure 5, translate each implementation state transition,  $s_i^{Impl} \rightarrow s_{i-1}^{Impl}$ , executed during a simulation to a specification state transition,  $s_i^{Spec} \rightarrow s_{i-1}^{Spec}$ .

Note that transitions  $s_{i-1}^{Impl}$  and  $s_i^{Impl}$  might not be quiescent states. If  $s_i^{Impl}$  is a quiescent state, then  $s_i^{Spec} = f_{qui}(s_i^{Impl})$  is required. If the TLA+ specification doesn't allow any specification-level transition,  $s_{i-1}^{Impl} \rightarrow s_i^{Impl}$ , then the refinement check has failed. If the check succeeds, the refinement map naturally provides a specification-level run that satisfies condition 1. If the check fails at any point, the model checker detects the discrepancy as soon as it occurs. This early warning is valuable, but the refinement map must now be defined for nonquiescent states as well.

## Challenges

The EV7 specification is at a higher abstraction level than the hardware implementation in two regards:



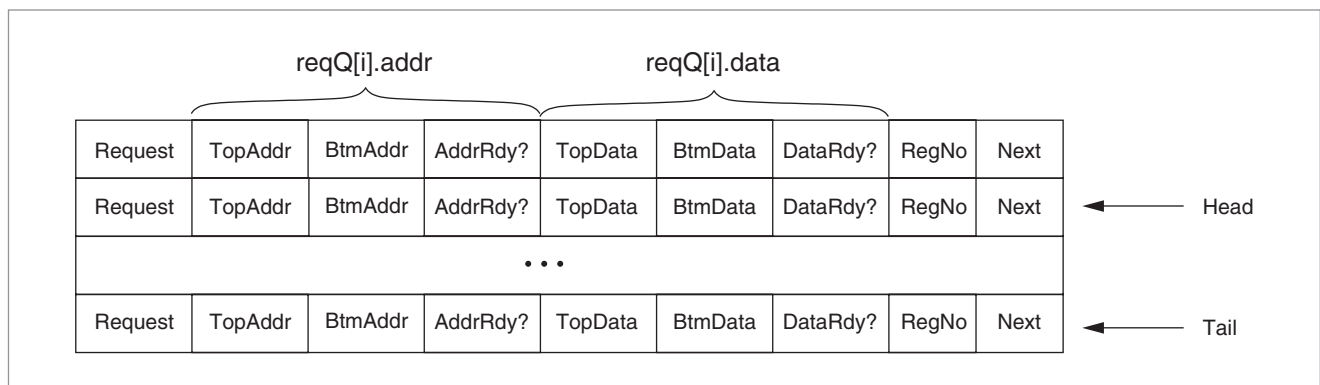
**Figure 5. Map from the state space of a multiprocessor system to the state space of the corresponding TLA+ specification. TLC checks whether transitions  $s_0^{Spec} \rightarrow s_1^{Spec}$  and  $s_1^{Spec} \rightarrow s_2^{Spec}$  are legal.**

- *System state representation.* The value of a specification state variable, such as the cache state, is actually a composite of information stored in a group of implementation state variables. The set of implementation variables that factor into a specification variable can change over time and be distributed across the hardware.
- *Concurrency and atomicity levels.* An execution at the specification level consists of a sequence of atomic protocol transactions, each described by a TLA+ action. At this level, protocol-state variable values appear instantly available. Each action atomically

updates a group of specification variables simultaneously. In the implementation, accessing and updating the variables require sequences of requests and responses spread over time. Moreover, implementation-state updates associated with different protocol transactions occur concurrently.

Suppose we implement the load-and-store unit shown in Figure 2 as an array of entries, as Figure 6 depicts. To see the challenge stemming from what we describe under “system state representation,” observe, for example, that in the hardware implementation, a request’s `.addr` field is a composite of three hardware state variables: the top and bottom halves of the address value (`TopAddr`, `BtmAddr`), and a ready flag (`AddrRdy?`). Suppose the hardware updates `TopAddr`, `BtmAddr`, and `AddrRdy?` at different times. The refinement map composes a value in the range  $\text{Addr} \cup \{\text{NotAddr}\}$  from the `TopAddr`, `BtmAddr`, and `AddrRdy?` values.

To see the challenge stemming from what we describe under “concurrency and atomicity levels,” suppose the circuitry for discharging an entry at the head of the queue first resets the entry’s `Next` field, then notifies the register whose value has been copied to memory, and finally updates the array’s head pointer. Line 30 in the specification (Figure 2) expresses this. Suppose also that the removal of the top entry from the queue occurs after the write request is dispatched to the memory control unit but before the actual memory location is updated. According to the specification, the updates in lines 29 and 30 occur simultaneously. The



**Figure 6. Sketch of the hardware implementing the protocol in Figure 2.**

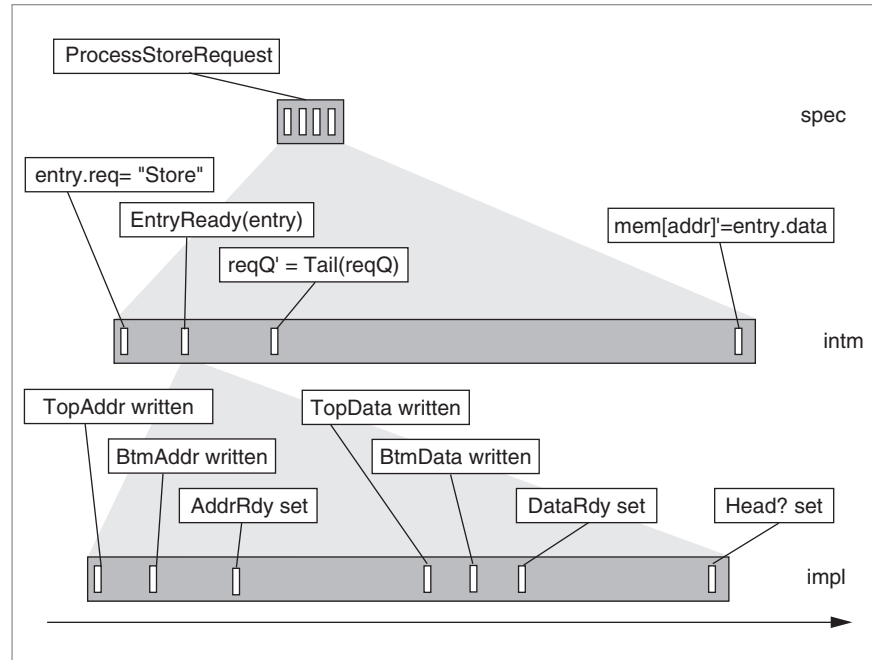
refinement map must translate all the implementation state updates corresponding to an action to one atomic state update at the specification level.

### Two-phase approach

We divide refinement map construction into two steps using an intermediate abstraction level (**Intm**) and expressing the map as the composition of two maps. The first,  $f_{Impl \rightarrow Intm}$ , maps transitions in the implementation to transitions at the intermediate level; the second,  $f_{Intm \rightarrow Spec}$ , maps transitions at the intermediate level to specification-level transitions.

The hypothetical intermediate representation **Intm** is only for better exposition of the mapping process. A state transition model for it is never built. The TLA+ specification guides the definition of **Intm** and  $f_{Intm \rightarrow Spec}$ . **Intm** has the same state variables as the specification but allows only atomic updates to them. However, whereas for every action **Spec** updates a group of state variables simultaneously, **Impl** updates each state variable at a different time. In the load-and-store example, the intermediate model **Intm** requires that, for instance, both the update to **mem[entry.addr]** in line 29 and the removal of the top entry in the request queue (line 30) be atomic. However, **Intm** lets the two state updates associated with the **ProcessStoreRequest** occur at different times, as Figure 7 shows, whereas **Spec** requires these updates to be simultaneous.

Function  $f_{Impl \rightarrow Intm}$  tracks the hardware signal transitions for variables that factor into each intermediate state variable. It aggregates these updates into one atomic update of the state variable (Figure 7). In the **ProcessStoreRequest** example,  $f_{Impl \rightarrow Intm}$  aggregates the updates to **TopAddr**, **BtmAddr**, and **AddrRdy?** to an atomic update of the **.addr** field. The atomic updates of the **.addr** and **.data** fields would then satisfy **EntryReady(entry)**. Function  $f_{Intm \rightarrow Spec}$  aggregates updates to collections of intermediate state variables into one atomic update corresponding to a TLA+ protocol action. In the **ProcessStoreRequest** example,  $f_{Intm \rightarrow Spec}$  aggregates the occurrences of the three preconditions (lines 24 and 25) and the updates to **mem** and **reqQ** into a single atomic state update at the load/store specification level. Each TLA+ action defines



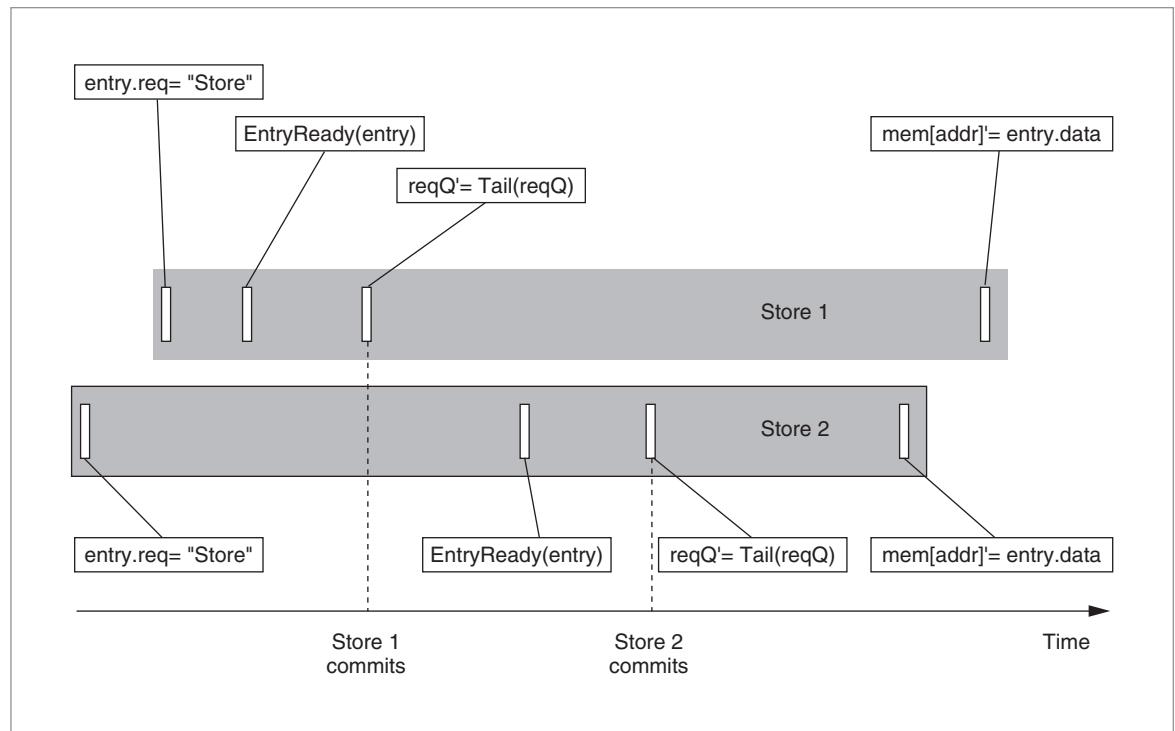
**Figure 7. Functions  $f_{Impl \rightarrow Intm}$  and  $f_{Intm \rightarrow Spec}$  map a collection of variable updates to one atomic transaction.**

a group of state variables whose updates must be aggregated by  $f_{Intm \rightarrow Spec}$  to an atomic update at the specification level. Here, the refinement check that must be performed is whether this update is consistent with the specification.

Our approach to constructing  $f_{Impl \rightarrow Intm}$  and  $f_{Intm \rightarrow Spec}$  is similar to Park and Dill's *aggregation of distributed transactions* method.<sup>7</sup> We also associate a **commit point** in the lower-level description with each atomic transaction at the higher level. Transactions in progress in the lower-level system trigger a change in the higher-level state only at commit points. Figure 8 shows commit points for two transactions.

To gather all the information required to perform the high-level state update at the commit point, we wait until the transaction reaches its completion point. For example, when a **ProcessStoreRequest** transaction is committed (that is, the corresponding entry is removed from the head of the linked list), the associated memory location may not be updated yet. We wait for the update to determine the value of **mem[addr]** at the new specification state. We then reflect the transaction's effect on the abstract state.

We believe that our two-phase mapping approach is well-suited to complex, concurrent implementations whose specifications can be written as transaction-based protocols in the style of the EV7 protocol. In such designs, a large team of designers implements hardware



**Figure 8. Commit points of two `ProcessStoreEntry` transactions.**

blocks, with each designer constructing the portion of  $f_{impl \rightarrow intm}$  relating to his or her block. Architects responsible for the overall design can then construct  $f_{intm \rightarrow spec}$ . Our recent work on the refinement checking of concurrent software designs provides further evidence of the applicability of our approach.<sup>8</sup>

#### Implementation of the map

We implemented the refinement map as a C++ module linked with the simulator (Figure 1). The mapping module computes  $f_{abs}$  incrementally to avoid examining all of the implementation state. At each clock phase, after the simulator computes the state transition  $s_{i-1}^{impl} \rightarrow s_i^{impl}$ , the mapping module is invoked and checks if any implementation variable of interest has changed. It then determines whether the corresponding specification state transition,  $s_{i-1}^{spec} \rightarrow s_i^{spec}$ , is legal using TLC if  $s_i^{spec}$  is different from  $s_{i-1}^{spec}$ . TLC is used only to check a single specification-level state transition. The time spent by TLC, which runs as a parallel process, to check and record one such transition is negligible compared with the simulation cost.

The portion of the hardware involved in the map was described using approximately 20,000 lines of hardware description language (HDL) code. The map itself took 8,000 lines of C++ code, excluding comments.

#### TLC model checker

Using the refinement map, TLC can be used as a monitor during all simulations, providing an iterative process for simultaneously debugging the specification, implementation, and refinement map. When TLC reports that transition  $s_{i-1}^{spec} \rightarrow s_i^{spec}$  violates the specification, there is a bug either in the implementation, the specification, or the refinement map. Inspection of the simulation run and the specification-level run constructed by the map up to that point help determine which one of these is the case.

We also use TLC to store the specification states visited during simulations. In this way, we collect coverage information at the same time that we test protocol conformance. Using the formal specification and the map for coverage data collection makes the process more rigorous.

To keep coverage data manageable, we use a TLC feature called **views**. A view  $v$  is a function from a large state space to a smaller one, expressed in terms of the state variables in a TLA+ specification. We can use a view to designate a subset of specification variables as **coverage variables**. One natural choice for a view is the tuple of variables on which the protocol phases' case splits are based at one processor. These were the same variables (all of which are specification-

state variables) used in the text description for the protocol and for coverage measurement during prior simulations. The late-stage design bug described in the next section was directly related to a gap according to this notion of coverage.

### Generating tests to improve coverage

Possibly the most important benefit of our approach is the partial automation of coverage-guided test-case generation. In later stages of the functional validation process, specification states that have not been exercised during simulation are identified as coverage gaps. These then serve as targets for TLC, which explores the protocol state space and generates a trace to the target if one exists. This process also distinguishes coverage gaps due to insufficient simulation from those corresponding to unreachable states at the specification level. Users can then give gaps in the former category priority when writing tests.

A TLC-generated path to a target state is easy to understand, and is a valuable aid for simulation input generation. Because the abstract specification closely reflects the EV7 design's structure, translating this path into an EV7 simulation run is manageable. Model checkers' capacity limitations pose a hurdle for the approach's practicability. Techniques such as partial order or symmetry reductions help ease these limitations. Furthermore, advances in model-checking technology, such as improved satisfiability checkers, directly translate into improved test generation. Although the limitations can't be denied, the improvement over test generation by reasoning at the implementation level is significant.

To demonstrate our approach's viability, we selected a bug from the EV7 bug database and showed that our technique would have revealed it. We chose a bug that was discovered only after the first hardware prototype was built and tested in an eight-processor configuration, as Figure 9 shows.

Initially, a memory address (**addr0**), whose home node is processor 0, is in a shared state in processors 0, 1, and 2. The home node asks for exclusive access to the address **SharedtoDirty [1]** (in Figure 9) and gets it. Later, the home node evicts this line from its cache (the **Victim** message in Figure 9). In the meantime,

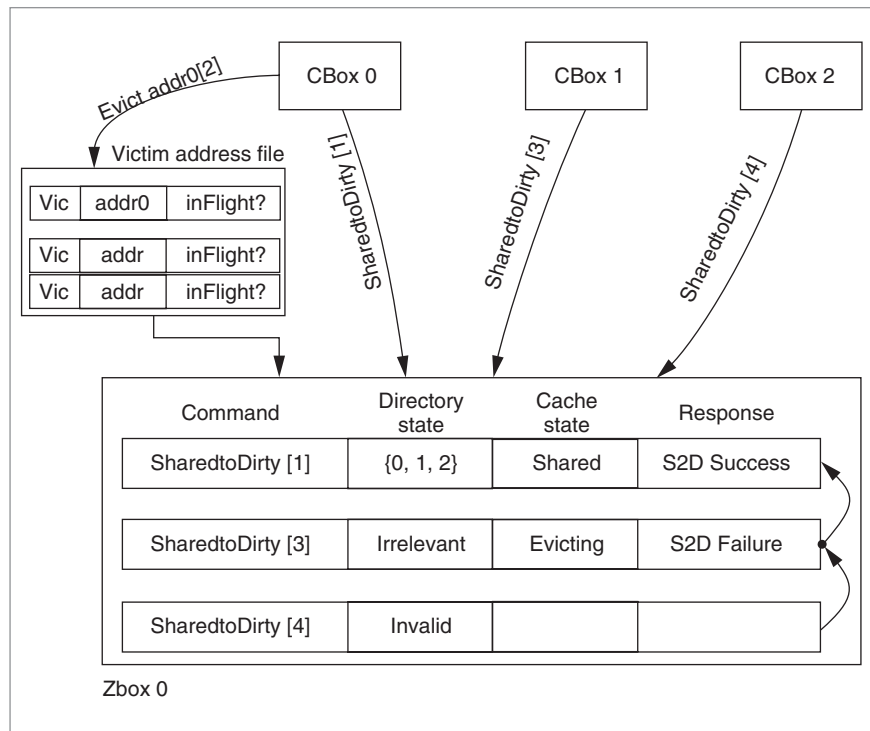


Figure 9. Late-stage design bug.

processors 1 and 2 ask for exclusive access to the same line (**SharedtoDirty [3]** and **SharedtoDirty [4]**, respectively). The victim message remains pending in the victim buffer because too many other memory requests are keeping the memory controller busy. The home node refuses **SharedtoDirty [3]** because of the eviction in progress. The protocol design implicitly assumes that after this refusal, the victim will make its way to the memory controller and reset the associated in-flight bit. **Zbox0** processes **SharedtoDirty [4]** as if no victim message is in flight. But then **Zbox0** receives the victim message while processing **SharedtoDirty [4]**. Because designers didn't anticipate this scenario, they didn't specify a next state in the protocol engine specification or implementation. This unexpected victim message causes an assertion violation during a model-checking run using TLC on a single-address, three-processor configuration. This particular model-checking run took less than 5 minutes and about 30 Mbytes of memory on a 625-MHz Alpha server. We confirmed with the architects that had they used TLC to perform the protocol-level trace before discovering the bug, they could have easily produced a corresponding run in the implementation.

We repeated the simulation run that exercised this bug using simulation inputs from the bug database, this time

using TLC as a correctness monitor. The simulation was consistent with the buggy version of the specification throughout the run. This proved that the protocol-level error trace was an actual bug in the implementation and wasn't due to the protocol specification being too loose.

**WE ARE APPLYING SIMILAR** techniques to the runtime verification of highly concurrent software designs.<sup>8</sup> Checking refinement at runtime or during simulation is a powerful and scalable verification technique. The key challenge in making this approach practical is devising an easy-to-use, intuitive method for defining a refinement map. This article presents such a method specific to concurrent hardware designs described using transactions. In our ongoing work, we are exploring methods specific to the runtime verification of highly concurrent software designs. ■

### Acknowledgments

We thank Maurice Steinman, Brian Lilly, Luka Bodrozic, Kathy Menzel, Jonathan Nall, Rajeev Joshi, Scott Kreider, and Scott Taylor for helping us understand the EV7 specification and design, as well as the bug described in the article.

### References

1. R. Joshi et al., "Checking Cache-Coherence Protocols with TLA+," *Formal Methods for System Design*, vol. 22, no. 2, 2003, pp. 125-131.
2. Y. Yu, P. Manolios, and L. Lamport, "Model Checking TLA+ Specifications," *Proc. IFIP Working Conf. Correct Hardware Design and Verification Methods (CHARME 99)*, LNCS 1703, Springer-Verlag, 1999, pp. 54-66.
3. L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley, 2002.
4. B. Cohen, S. Venkataramanan, and A. Kumari, *Using PSL/Sugar with Verilog and VHDL: Guide to Property Specification Language for Assertion-Based Verification*, VhdlCohen Publishing, 2003.
5. B. Batson and L. Lamport, "High-Level Specifications: Lessons from Industry," *Proc. Formal Methods for Components and Objects (FMCO 02)*, LNCS 2852, Springer-Verlag, 2002, pp. 242-261.
6. J. Burch and D. Dill, "Automatic Verification of Pipelined Microprocessor Control," *Proc. Int'l Conf. Computer-Aided Verification (CAV 04)*, LNCS 818, Springer-Verlag, 2004, pp. 68-80.
7. S. Park and D.L. Dill, "Verification of Cache-Coherence Protocols by Aggregation of Distributed Transactions," *Theory Computing Systems*, vol. 31, 1998, pp. 355-376.
8. S. Tasiran and S. Qadeer, "Runtime Refinement Checking of Concurrent Data Structures," *Proc. 4th Workshop on Runtime Verification*, Elsevier Electronic Notes in Theoretical Computer Science, 2004.



**Serdar Tasiran** is an assistant professor at Koç University in Istanbul, Turkey. His research interests include tools and techniques for systems design and verification. Tasiran has a PhD in electrical engineering and computer sciences from the University of California, Berkeley. He is a member of the IEEE and the ACM.



**Yuan Yu** is a research scientist at Microsoft Research. His research interests include model checking, formal verification, and static and dynamic program analysis. Yu has a PhD in mathematics from the University of Texas at Austin.

**Brannon Batson** is a processor architect at Intel, where he works on next-generation Itanium processor family (IPF) processors. His research interests include memory system design and multiprocessor cache coherence. Batson has a BS and an MS in computer and electrical engineering from Purdue University.

■ Direct questions and comments about this article to Serdar Tasiran, College of Engineering, Koç University, Rumeli Feneri, Sariyer, Istanbul, Turkey 34450; stasiran@ku.edu.tr.

**For further information on this or any other computing topic, visit our Digital Library at <http://www.computer.org/publications/dlib>.**