4

Formal Modeling and Validation Applied to a Commercial Coherent Bus: A Case Study

Ganesh Gopalakrishnan, Rajnish Ghughal, Ravi Hosabettu, Abdelillah Mokkedem and Ratan Nalumasu
Department of Computer Science, University of Utah
Merrill Engineering Bldg, Salt Lake City, UT 84112, USA
{ganesh, ghughal, hosabett, mokkedem, ratan}@cs.utah.edu

Abstract

The degree to which formal verification methods are adopted in practice depends on concrete demonstrations of their applicability on real-world examples. In this paper, we present our efforts in this regard involving a commercial high-speed split-transaction bus called the Runway. Modern busses such as the Runway deal with so many inter-twined and complex issues that successful application of formal method requires separation of concerns, and the use of the most appropriate tool for each concern. We report our experiments towards this end through the use of the PVS theorem-prover to formally analyze the high-level functional behavior of the bus and the HDL-based model-checker VIS to verify the pipelined arbitration protocol of the bus. The high degree of effort found necessary, as well as the specific abstraction mechanisms found useful in obtaining these formal models are discussed in detail.

1 INTRODUCTION

With the growing importance of multiple CPU systems such as core-based designs and multiprocessors, interfacing processors using either standard or custom-made busses is a problem of growing importance. Modern busses are used for performing both coherent and non-coherent data transfers, and input/output (I/O). They involve multiple levels of detail including coherency processing, split-transaction processing, bus arbitration, flow control, and absolute-timing. Descriptions of these busses (for example, the HP Runway bus [BCS96], or the PCI bus [pci]) are extremely complex, lengthy, often contain inconsistencies [CSZ97], and are very difficult to deal with on a day-to-day basis for designers of complex systems. We have observed in the context of an in-house academic shared memory multiprocessor (SMP) design project called the Avalanche [CKK96] that designers are inundated by complexity of bus specifications and commit subtle but crucial mistakes on a regular basis.

These mistakes are later uncovered very laboriously in the process of time-consuming simulation runs. The ease with which subtle errors can be introduced is very worrisome. In this paper, we report on our experiences in using Formal Methods to describe a commercial bus called the Runway, in the context of an actual multiprocessor design project called the Avalanche[CKK96].

As pointed out by Corella, [CSZ97], one of the greatest advantages of formal modeling is the clarity of thinking it promotes. In addition, formal analysis supported by theorem-proving and model-checking tools allows one to examine various scenarios at a high level, formally prove putative theorems about them, and thereby enhance one's understanding. This paper provides a case study of our work along those lines in the context of the HP Runway bus. Our experience shows that as much effort is involved in creating formal models for various aspects of the bus operation as in proving properties of the models. The creation of such formal models is discussed in detail in this paper, in addition to verification experiments. Our use of a commercial bus as a design example will also, hopefully, redress the imbalance in the classes of examples published in the current literature which includes a preponderance of arithmetic circuits and pipelined processors, but not so much on bus-based systems where a great deal of commercial emphasis is being placed [UC97].

This paper is organized as follows. We survey related work in this section. In Section 2, we describe the Runway bus at an intuitive level. In Section 3, we describe the different versions of a high-level model of the Runway in the language of the theorem prover PVS [OSR92], culminating in a version that is intuitive, simple, and yet captures much of the functional behavior of the Runway. Formal analysis using PVS is also reported in this section. In Section 4, we describe the use of the model-checker VIS [Bra96] to verify the distributed pipelined arbitration protocol of the Runway. Our conclusions appear in Section 5.

Related Work

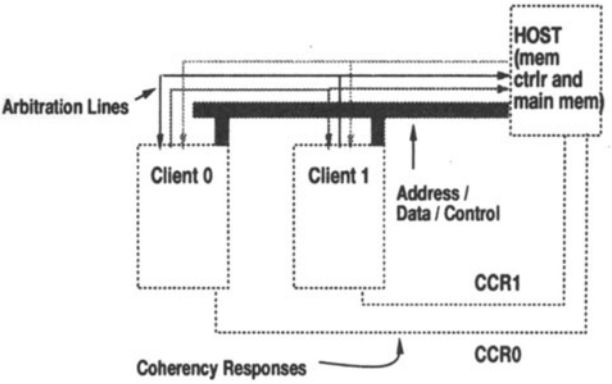
Many past works deal with modeling and verifying cache coherence protocols supported by various busses [CGH+93, GKMK91, McM93]. In [HW94], the identification of deadlocks in the HP Summit Bus Converter through symbolic model-checking was presented. [Hoo93] presents the hand-proof of correctness of an academic distributed real-time arbitration protocol example. In [CSZ97], a formal proof of absence of deadlocks for any acyclic network of PCI busses has been given. The main differences between our work and these works can be summarized as follows: (i) our work is a case study involving a modern commercial bus, and as such is a more challenging example in many ways; (ii) we discuss the creation of a formal high-level model for the bus in PVS *emphasizing the abstraction mechanisms* that have proved to be valuable in obtaining a tractable model and provide mechanical verification of this model; (iii) we apply a modern symbolic model-checking tool, namely VIS, to verify an actual bus arbitration protocol.

2 AN INTUITIVE DESCRIPTION OF THE RUNWAY BUS

Figure 1 shows the simplified view of a two CPU system using the Runway bus. The actual Runway bus supports many modes of behavior including coherent and non-coherent reads and writes, and I/O operations. It obeys a complex cycle-based protocol without aborts or retries. For the purpose of this paper, we provide a highly simplified view of its operation. When a CPU (*client*) attempts to read or write an invalid cache line or write a read-only cache line, it suffers an internal cache miss. In the former case, a Read Shared Private (RSP) transaction is broadcast on the runway (shown as thick line in Figure 1) for that address. In the later case, a Read Private (RP) transaction is broadcast for the address. Depending on who has *ownership* (the currently valid version) of data for that address, another client or the main memory (HOST) responds with the data. When the HOST returns data, it also tells the client the ownership status it can assume of the data, through a signal called the *Client-op*. This request/response paradigm is usually called the *split transaction* mechanism. The following algorithm is used to determine who will supply the most recent version of the data. In response to each bus transaction (including transactions generated by itself), each client generates a *cache coherency response* (CCR). If one of the CCRs is *copyout*, that client promises to supply the data; in this case, HOST does not generate a bus transaction. It is required that the client that made the *copyout* promise later generate a *cache to cache write* (C2CW) transaction directed towards the requesting client with the data. If none of the CCRs is *copyout*, HOST supplies the data—either in the *shared* or *private* mode depending on whether one of the clients has (or doesn't have) a shared copy (determined by looking into the CCR words).

Before every potential bus user (client or HOST) attempts a bus operation, it must become the bus master. Bus mastership at cycle N+2 for operation K is acquired by initiating the arbitration in cycle N, as shown in Figure 1, by driving the request through dedicated arbitration lines also shown in the figure. During cycle N+1, every potential bus user evaluates others' drives and, in conjunction with round-robin pointers for arbitration priorities (not shown), determines who wins bus-mastership for cycle N+2. Those who do not win bus mastership keep-off the bus. As shown in the figure, bus arbitration proceeds in a pipelined manner concurrently with transaction processing.

In order to generate the CCR coherency responses, clients *snoop* the bus and keep the results of snooping around in dedicated FIFOs. The relative sizes of these queues determines how much behind the clients can get in coherency processing, relative to HOST's use of the CCR values. Unfortunately, due to pipelined communication, the HOST only has an outdated view of how full the client snoop queues are. Therefore, HOST starts throttling the client actions by asserting a flow-control signal (part of Control in the figure) whenever its most pessimistic prediction of future events foretell one of the client queues



		Cycle N	Cycle N+1	Cycle N+2	Cycle N+3	Cycle N+4
OP K	Client 0	Drive	Read Read	DriveBus	Pointer Upd	.....
	Client 1	Drive	Read Read	KeepOff	Pointer Upd	
	Host	Drive	Read Read	KeepOff	Pointer Upd	
OP K+1	Client 0	.....	Drive	Read Read	KeepOff	Pointer Upd
	Client 1		Drive	Read Read	DriveBus	Pointer Upd
	Host		Drive	Read Read	KeepOff	Pointer Upd

Figure 1 Simplified View of the Runway Bus

overflowing. In addition to this throttling signal, each client can retain bus mastership for one more cycle by asserting a locally generated *long trans* signal.

The main aspect of the functional complexity of busses such as these is their coherency processing algorithm. Split-transaction busses such as the Runway attempt to guarantee the formal memory model known as *coherence*[ABJ+92]—per-location sequential consistency—in an efficient manner*. They thus permit the clients to be very aggressive in handling the notion of ownership. For example, when one client c requests ownership for address a (by effectively missing on a write to a) and is awaiting ownership, it may go ahead and start modifying a in a private copy, fully knowing that its ownership is on the way,

*Additional ordering constraints on the operations issued towards the Runway bus by the CPU gives stronger orderings such as sequential consistency; in our model, we do not consider the CPU imposed orderings.

and that it can incorporate the local changes made when the data actually arrives. When another client $c1$ does likewise, the first client c has to notice the fact, and effectively freeze further updates. When data arrives at c, it merges the local changes caused by it, and immediately gives up ownership as well as the merged data to $c1$.

All references to client identities are maintained through a master-ID for each client, and a transaction-ID for each transaction outstanding. Transaction IDs are finite in number, and can be recycled whenever a transaction is completed. Transaction IDs are a means to save bus bandwidth. For each split transaction that is unfinished, its transaction ID represents the address involved in the transaction, albeit using considerably fewer bits.

Though it is not necessary for all the above details to be fully understood, it must be apparent that modern busses do much more than simply help exchange data. They are, in effect, high-performance coherent data exchanges for multiple CPUs, memory systems and I/O systems. As such they often serve as the "repository for all the unresolved decisions" and invite an aspect of complexity from every device they have to deal with. It must therefore come as no surprise that actual high-end design projects spend enormous amounts of *expert designer time* trying to understand and deal with busses (as, for instance, we have observed in many projects; see also [HP95]). The formal models to be presented in the following sections examine some of the formal modeling and analysis techniques that can mitigate some of this complexity.

3 FORMAL MODELING AND ANALYSIS USING PVS

Creating a formal model for the Runway that encompasses all the details presented in the previous section can be an impossibly hard task. Our initial decision was to back-off from the complexity and create a model that emphasizes the main aspects of the bus, namely the model of the client's participation in transactions. This, in turn, requires capturing the *functional* as well as *timing* details of transaction processing including snooping and buffering. Our first attempt was to build such a model. It ended up occupying several thousands of lines of PVS written over a few man-months of effort. No sooner was this model written than was it apparent that it would be virtually impossible to be dealt with in any formal fashion. Specific things "wrong" with this model were the following:

- The description was mostly in terms of *ad hoc* axioms (*i.e.*, not definitional axioms involving function declarations). The axiomatic style was more or less forced upon us by the high degree of non-determinism inside the client- and host units. It is well known that such collections of large numbers of *ad hoc* axioms are prone to be inconsistent [COR+95].
- Scores of explicit queues were employed in the model. To remain cycle-accurate, we had to model more situations than usually dealt with queues.

For example, we had to model each queue being written, read, and simultaneously read and written. This caused the case analysis to explode.

- Explicit queue models caused a large number of type-checking conditions to be generated. Many of these were automatically provable, but some required tedious human-guidance of PVS through the proof.

The following sections examine various model-simplifications, and associated results.

Simplifying finite-state protocols

Using a theorem-prover in situations where there is considerable "protocol complexity" is rather tedious. Similar situations are much more amenable to being dealt with using model-checkers. For example, in the Runway specification, there are many cases where the cycle-based state machines become unduly complex due to the large number of cases of word-sizes being dealt with. Some of the model simplifications we accomplished at this stage of the specification were by assuming that the Runway could perform most of its activities "in a single cycle". One of the *greatest challenges of applying formal methods* to real-world problems would be to discover ways to systematically deal with such model simplifications. More specifically, one should be able to first proceed with the simplifications to prove overall correctness properties, and later verify that employing the more complex situation would not have caused any different outcomes. The latter step may be accomplished using a model-checker, a cycle constraint checker [GC96], or even a theorem-prover.

Separating the functional and timing aspects

Even with the above model-simplifications, the model for the Runway behavior in PVS proved to be extremely large and unwieldy. Even leaving the cycle-level details aside, there is enough complexity in the Runway bus in terms of how it handles coherency response generation, how each cache unit updates its line state, how the *Client-ops* are generated, and how the next value of the bus is defined in terms of the current system state. One can also specify bus arbitration abstractly in this model. A purely functional-style model capturing this behavior was written. Two additional simplifications were made to obtain a model of tractable size. The first was to add a bit-control called `lock` per client per line to lock the line once the client requests a data for this line; the subsequent requests for the same line by the same client are frozen till the data comes back. The second simplification was to guard the RSP (Read-Shared-Private) and RP (Read-Private) transactions with the following condition: RSP_i and RP_i subsequent transactions for a line a are frozen if

the client i has a pending C2CW in its *snoop* queue to supply the data of the line a to another client, until the pending C2CW is flushed out of the *snoop* queue. This simplifies the *snoop* queue at each client to one-size buffer and thus we don't have to consider explicit queues to model C2CWs.

The use of a bit-control in the cache lines

In the real runway implementation each cache line of a client can assume several coherency states, some of which may be explicitly captured through tags and others captured through a combination of tags and the protocol state. The terminal coherency states of the cache protocol are: Invalid (the copy is not valid), Private-Clean (the copy is valid, clean, and no other client has a valid copy of the same line), Dirty (the copy is valid, dirty i.e., modified, and no other client has a valid copy of the same line), and Shared (the copy is valid, clean, and no other client has a dirty copy of the same line). There are other transient coherency states such as Invalid-data-pending, Shared-pending-promotable, etc. We simulate these transient states by augmenting the four terminal states with a bit called `lock`, which is `true` if the client is in one of the transient states (i.e., it is waiting for data), `false` otherwise. This bit-control has exactly the same effect as using transient coherency states but we feel that it makes the control in the model clearer and simpler.

Figure 2 presents a higher-level view of our client-model state machine; data and bit-control states are not represented in the figure. The lines with arrows show the transitions of the cache state at the originating client. The text near each line describes the conditions at the other clients that caused the transition, as well as the effect on the other client's state. For example, from the invalid state, a load miss will always cause a RSP transaction. The terminal state for the load miss will be either Private-Clean (if any other client had the cache line invalid or dirty) or Shared (if another client had the cache line shared or private-clean.

The use of guards to deal with delay

The situation leading to delayed CCR responses is best explained using an example. Consider the situation where a client C1 is owning a cache line. Suppose a client C2 issues a bus transaction requesting ownership of the line. C1 then gives up ownership and generates a *copyout* promise (as explained in Section 2). Further, C1 backs its promise to supply the data by generating a C2CW instruction which awaits its turn to win the Runway bus arbitration. In this state, the HOST module still does not have the most recent value for this cache line; it would obtain that from the C2CW transaction only when it goes on the bus. Meanwhile the host also has noticed C1's promise to supply C2's

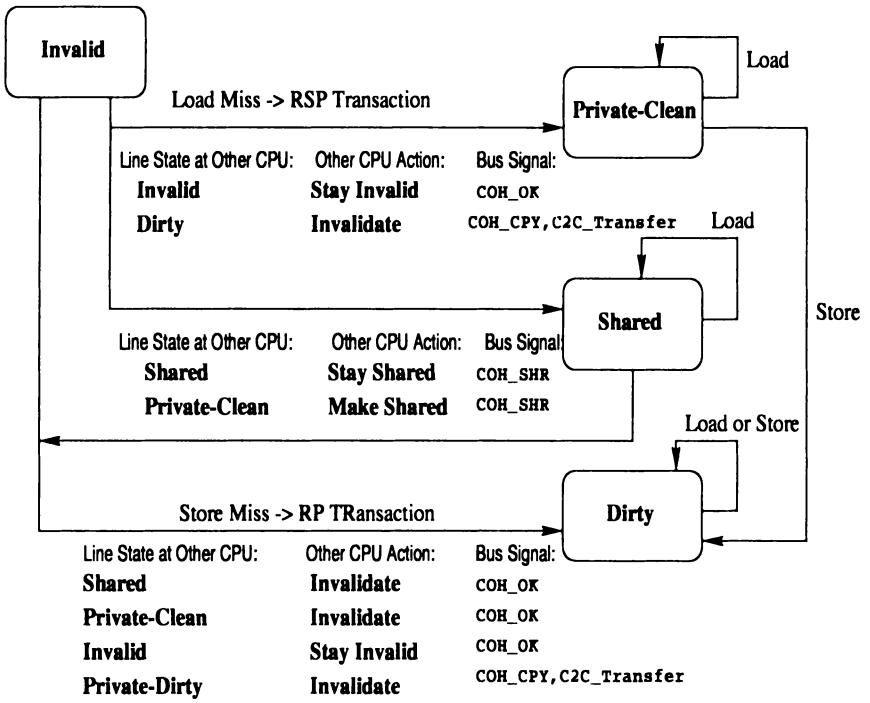


Figure 2 Cache state transitions resulting from the CPU instructions.

request and is essentially done with handling the C2 originated transaction. Now, if another client C3 generates a request for the same line,

- C1's CCR would indicate that it does not have the ownership *anymore*
- C2's CCR would indicate that it does not have the ownership *yet*
- C3's CCR would indicate that it does not have the ownership (*of course!*)

Noticing these CCRs, the HOST would supply the data. However since the cache to cache write from C2 has not gone out, the HOST would end up supplying the incorrect (old) data to C3.

A solution to this situation adopted in the Runway bus is that C1 *delay* its coherency response for C3's request until after the C2CW corresponding to C2's request had gone out on the runway. However, expressing this "delay trick" in a purely functional style that is devoid of timing is very difficult. Our solution to this problem was to distinguish between *enabled* and *disabled* C2CW transactions. A C2CW transaction is said to be *enabled* if the client that issued the transaction has the current data*. Otherwise the transaction is disabled, and the C2CW cannot be issued onto the runway until the current data is received. Using this notion, a guard is introduced on every HDR

*Recall that, we can use `lock` to determine whether a client has the current data.

```
InvCtrl(runway): boolean = forall (j:Mid)(a:Addr):
    (ctrl_state(cache(runway)(j)(a)) = DIRTY
    => forall (k:Mid): k /= j => ctrl_state(cache(runway)(k)(a)) = INVALID)
       AND
    (ctrl_state(cache(runway)(j)(a)) = PRIVATE_CLEAN
    => forall (k:Mid): k /= j => ctrl_state(cache(runway)(k)(a)) = INVALID )

InvCoh1(runway): boolean = forall (j:Mid)(a:Addr): (
    ctrl_state(cache(runway)(j)(a)) = PRIVATE_CLEAN
    AND NOT(lock(cache(runway)(j)(a)))
    => data(cache(runway)(j)(a))=memory(runway)(a) )

InvCoh2(runway): boolean = forall (j:Mid)(a:Addr): (
    ctrl_state(cache(runway)(j)(a)) = SHARED
    AND NOT(lock(cache(runway)(j)(a)))
    => data(cache(runway)(j)(a))=memory(runway)(a) )

InvCoh3(runway): boolean = forall (j,k:Mid)(a:Addr): (
        ctrl_state(cache(runway)(j)(a)) = SHARED
    AND ctrl_state(cache(runway)(k)(a)) = SHARED
    AND NOT(lock(cache(runway)(j)(a)))
    AND NOT(lock(cache(runway)(k)(a))) )
    => data(cache(runway)(j)(a)) = data(cache(runway)(k)(a))
```

Figure 3 Coherency invariants. **runway** indicates state of the memory, clients, and other buffers.

transaction; this guard indicates that there is no pending *enabled* C2CW for that address. This solution captures the same operational effect of "delaying the CCRs" but is more appropriate to deal with in a functional style. In the previous example, C1's C2CW is an enabled transition. Hence, the host would supply the data corresponding to C3's request only after the C1's C2CW has appeared on the bus.

Results of formal analysis using PVS

We have finished a purely functional specification (written in PVS higher-order language) of the Runway bus along the above lines and proved a number of interesting properties using PVS. A full description of this specification is available on the web [Mok97]. The PVS theory representing our Runway model is parameterized by the number of clients, the range of addresses, and the range of data values. Hence, the properties we prove are general and hold for any number of clients, arbitrary memory and cache size, and data width. It is impossible to get the same result using a pure model-checking technique.

The definitions of the main invariants we have proved for the correctness of the Runway cache coherence protocol are given in Figure 3. **InvCtrl** asserts a cache coherency-state correctness: if a line is Dirty (or Private-Clean) at a client it must be Invalid at the others. This invariant is necessary for

```
InvC2CW(runway): boolean = (forall (i,j:Mid) (a:Addr):
    ( Allowed_trs(runway)(c2cs(runway)(i)(a)) and j/=i
        => Not(Allowed_trs(runway)(c2cs(runway)(j)(a))) )
        AND
    ( ctrl_state(cache(runway)(i)(a))=PRIVATE_CLEAN
        AND NOT(lock(cache(runway)(i)(a)))
        => NOT(exists (k:Mid): Allowed_trs(runway)(c2cs(runway)(k)(a)))))

InvList(lt: list[Transaction], mem:Memory): recursive boolean =
                lt/=null and Data(car(lt))/=UNDEF
              => Data(car(lt))=mem(Addr(car(lt)))
                 and InvList(cdr(lt),mem)
      MEASURE (length(lt))

InvHDR(runway): boolean = (forall (a:Addr):
    InvList(hdr(runway)(a),memory(runway)))
```

Figure 4 These invariants assert the correctness of `c2cs` and `hdr` buffers.

proving `InvCoh1` and `InvCoh2`. `InvCoh1` (resp. `InvCoh2`) asserts that if a line is Private-Clean (resp. Shared) at a client and the data has been already returned then the data value for this line is the same as the memory value for that address. `InvCoh3` indicates that when two clients share a line `a`, they agree on the data. Note that `InvCoh3` is a logical consequence of `InvCoh2`.

To prove `InvCoh1` and `InvCoh2` we needed two other invariants asserting the correctness of the behavior of the buffers `c2cs` and `hdr` shown in Figure 4. `InvC2CW` asserts that always there exists at most one enabled C2CW transaction per cache-line and that if a client has a non-locked private-clean copy of a line then there is no enabled C2CW transaction for this line on the runway. `InvHDR` asserts that for every enabled HDR transaction on the runway, the data transported by this transaction is the same as the memory data for this line.

Proof summary

Every invariant proof is done by proving that the invariant is true at the initial state and is preserved by every allowed transaction. For example, the proof of `InvCtrl` is done by proving the two following theorems:

```
InvCtrl_Init: THEOREM InvCtrl(Init_Runway)
InvCtrl_thm: THEOREM InvCtrl(runway)
                    and Allowed_trs(runway)(trans)
                    => InvCtrl(eval(trans,runway))
```

where `eval(trans,runway)` defines the transition relation corresponding to the current transaction `trans` on the runway and `Allowed_trs` defines the

```
FindClean_lemma4: LEMMA (ctrl_state(cache(runway)(j)(a)) = PRIVATE_CLEAN AND
        forall (k:Mid): k /= j => ctrl_state(cache(runway)(k)(a)) = INVALID)
    => FindClean(cache(runway),a,N)=j

FindClean_lemma5: LEMMA (forall (n:Mid):
    (FindClean(cache(runway),a,n)=j and j/=0)
    => ctrl_state(cache(runway)(j)(a)) = PRIVATE_CLEAN)

FindDirty_lemma4: LEMMA (ctrl_state(cache(runway)(j)(a)) = DIRTY AND
    forall (k:Mid): k /= j => ctrl_state(cache(runway)(k)(a)) = INVALID)
    => FindDirty(cache(runway),a,N)=j

FindDirty_lemma7: LEMMA ( FindDirty(cache(runway),a,n)=j) and j/=0
    => ctrl_state(cache(runway)(j)(a)) = DIRTY

RSP_lemma7: LEMMA ( ctrl_state(cache(j)(a)) = DIRTY AND j/=i AND
    forall (k:Mid): k /= j => ctrl_state(cache(k)(a)) = INVALID )
    => ctrl_state(Upd_CS_RSP(cache, i, a)(j)(a)) = INVALID
```

Figure 5 Examples of some proved lemmas.

set of all possible (enabled) transactions. The definition of these functions is given in [Mok97].

We should note that PVS proofs of these invariants are often very tedious and sometimes impossible to complete if we try to construct them from scratch. For instance, we have proved over 30 lemmas for completing the proof of these invariants. These lemmas mainly deal with the correctness of the basic definitions from which the main functions of the PVS model are defined. Some examples of these lemmas are listed in Figure 5.

The proof the invariant `InvHDR` of Figure 4 uses induction on the length of the hdr queue. This is the only invariant which requires induction on the queue. All the other invariants are proved by intensive use of case analysis. We should note that in many cases the same proof structure is repeated with different instantiations to prove different subgoals. This causes the proofs to be very long and tedious and we believe that the possibility to define parameterized proof schemes and to use them with different instantiations could be very helpful for these situations.

4 MODEL-CHECKING THE RUNWAY ARBITRATION PROTOCOL

The formal model used in PVS model assumes a very simple arbitration scheme: we simply pick any legal next Runway transition that is allowed in the current state. In this section, we discuss how the actual Runway arbitration protocol was modeled and verified.

Arbitration protocols used in modern busses such as the Runway are complicated by several factors. First and foremost, a formal description is not readily available; often, it has to be gleaned by reading English descriptions,

Network statistics	Gates	979
	Primary outputs	4
	Latches	41
Reachability analysis	FSM depth	6
	States	10^9
	MDD Size	30,872
	Time	122 sec
Correctness	Without reductions	132 sec
	With property specific reductions	10 sec

Table 1 Model-checking statistics with VIS running on an UltraSparc-1 with 512 MB memory

timing diagrams, and (if one is lucky to get one) detailed HDL descriptions. Second, these protocols involve not only cater to the basic arbitration mechanism, but also often involve actions taken during the initialization of the system and actions taken for flow-control (throttling of the bus by the bus-master, etc.). In our experience, a modern symbolic model-checking tool such as VIS is quite capable of analyzing these models and providing useful insights, provided one can obtain *reliable* models of these protocols by reading manufacturer's documentations*.

Figure 1 illustrates the details of the arbitration process. As shown, for the k th bus operation desired by a bus client (OP K) at cycle N+2, all bus clients express their intend to drive (or not drive) the bus at cycle N itself. The HOST also expresses its intentions to *throttle* the bus in specific ways in cycle N itself, by generating suitable *client_op* signals. During cycle N+1, each bus client evaluates the truth-value of the drives of all other clients and of the HOST. Based on this, the bus "winner" for cycle N+2 is chosen, and the winner drives the bus at this time. During cycle N+3, the round-robin pointers for determining the arbitration priority are updated by all the clients. All this activity goes on pipelined with the pipelined arbitration process for OP K+1 to be effected during cycle N+3 (most Runway operations take only one cycle; exceptions are not discussed here).

A model for the distributed pipelined arbitration process was created using VIS Verilog [Bra96]. VIS supports Verilog simulation, and a host of verification algorithms including CTL model-checking [McM93], language emptiness checks, etc. VIS proved to be an effective tool to develop specifications

*The complexity of this task is very often under-estimated in the academic world, as we discovered in our work.

incrementally by allowing us to "fake" missing parts through its facility of non-deterministic input wires. This facility was also used (with suitable constraints) to simulate the round-robin pointers that would have caused additional state explosion if faithfully modeled. Specific results obtained using VIS are as follows:

1. Using the symbolic model-checking facility supported by VIS, we have model-checked to prove that the arbitration algorithm ensures at-most one bus driver at any time. During this experiment, property-specific reductions achieved considerable reduction in model-checking time.
2. In another experiment, symbolic model-checking of the Runway arbitration algorithm exposed a possible misunderstanding in a timely fashion. Some members of the Avalanche design team believed that two signals named `Client-op` and `Effective-Client-op` were implied to be "almost the same", by the documentation. We model-checked an invariant `Client-Op = Effective-Client-op` which revealed that this assumption was incorrect, generating a useful error-trace.
3. In a final experiment, (using an idea attributed to Geist et.al. of IBM Israel) we typed in formulae of the form **`AG not(true-fmla)`** and generated an error-trace that acts as a simulation sequence for *establishing* **`true-fmla`**. This facility will be used for deriving test vectors for the Runway HDL models that the Avalanche group is creating.

Statistics pertaining to our model-checking experiments are summarized in Table 1. This table shows that the "complexity" of these protocols in terms of the number of gates, latches, and reachable states is well within the reach of today's symbolic model-checking tools. Also, the property-specific model reduction heuristic used in VIS proved to be of great value in reducing the model-checking time.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a case study (that is still in progress) of a commercial split-transaction bus, which was modeled and formally analyzed using a theorem-prover and also model-checked using a finite-state model-checker. Our preliminary experience tells us that the added insight obtained while writing formal descriptions, and the ability to test one's understanding using challenge queries on the descriptions may be the two most factors in favor of using formal methods in real design projects. It is also clear that the prevalent practice of providing only informal documentations of high-end digital systems has to change to one of using a judicious combination of informal and formal descriptions, if we were to make good use of the power of formal methods to facilitate understanding and to eliminate "guess-work" in designing around pre-existing complex digital systems.

One aspect of writing formal descriptions for high-end digital systems is that one needs to be willing to start using suitable abstractions early on, to avoid getting inundated by the complexity of the problem. Only by gaining sufficient insight into using abstractions in new domains can one develop suitable specification styles. Another aspect of using formal descriptions has been pointed out in [CSZ97]: namely that in the interest of detecting problems early on, hand-proofs are highly recommended initially. After our initial experience with the Runway bus, we now find this observation to be a bit more true. On the other hand, it is very difficult to find a mechanical proof without having a pencil-and-paper proof.

Modern digital systems are complex precisely because they are meant to be fast. Almost every aspect of their design reflects one decision that the designer has taken to optimize some aspect of its performance. One of the greatest challenges of formal verification methods research is how to encourage (and facilitate) designers to document these optimizations in such a manner that the formal methods tool user can later benefit from the added knowledge. Tractable use of formal verification tools must then consist of first "undo"-ing the optimizations to prove something overall, and then re-introducing the optimizations to show that nothing of concern was affected.

REFERENCES

[ABJ+92] Mustaq Ahamad, Rida Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. Technical Report GIT-CC-92/34, 1992. Revised 1993; FTP path: ftp.cc.gatech.edu/pub/tech_reports.

[BCS96] William R. Bryg, Kenneth K. Chan, and Nicholas S.Fiduccia. A high-performance, low-cost multiprocessor bus for workstations and midrange servers. *Hewlett-Packard Journal*, pages 18–24, February 1996.

[Bra96] Robert Brayton. VIS: a verifier for interacting systems. In *Computer Aided Verification*, New Brunswick, New Jersey, July 1996. Tool demo.

[CGH+93] Edmund Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David Long, Ken McMillan, and Linda Ness. Verification of the futurebus+ cache coherence protocol. In D. Agnew, L. Claesen, and R. Camposano, editors, *Proceedings of the 11th International Conference on Computer Hardware Description Languages and their Applications*, 1993.

[CKK96] John B. Carter, Chen-Chi Kuo, and Ravindra Kuramkote. A comparison of software and hardware synchronization mechanisms for distributed shared memory multiprocessors. Technical Report UUCS-96-011, University of Utah, Salt Lake City, UT, USA, September 1996.

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS, June 1995. http://www.csl.sri.com/sri-csl-fm.html.

[CSZ97] Francisco Corella, Robert Shaw, and Cui Zhang. A formal proof of absence of deadlock for any acyclic network of PCI buses. In *Hardware Description Languages and their Applications*, pages 134–156. Chapman Hall, 1997.

[GC96] Pierre Girodias and Eduard Cerny. Interface timing verification using CLP, 1996. Dept of Computer Science, IRO, University of Montreal.

[GKMK91] Stein Gjessing, Stein Krogdahl, and Ellen Munthe-Kaas. A top down approach to the formal specification of SCI cache coherence. In *Computer Aided Verification*, pages 83–91, 1991. LNCS 575.

[Hoo93] Jozef Hooman. Specification and verification of a distributed real-time arbitration protocol. In *Real-Time Systems Symposium*, pages 284–293. IEEE CS Press, Los Alamitos, CA, 1993.

[HP95] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach (Second Edition)*. Morgan Kaufman, 1995. Appendix-E.

[HW94] Cherly Harkness and Elizabeth Wolf. Verifying the summit bus converter protocols with symbolic model checking. *Formal Methods in System Design*, 4(2):83–98, 1994.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993.

[Mok97] Abdelillah Mokkedem. A PVS model of runway, 1997. http://www.cs.utah.edu/~mokkedem/pvs/.

[OSR92] Sam Owre, Natarajan Shankar, and John Rushby. Pvs: A prototype verification system. In *11th International Conference on Automated Deduction (CADE), Saratoga Springs, NY*, pages 748–752, June 1992.

[pci] "PCI Local Bus Specification", Revision 2.1, PCI Special Interest Group, June 1995. Phone: 1-800-433-5177.

[UC97] Cary Ussery and Simon Curry. Verification of large systems in silicon. In *Hardware Description Languages and their Applications*, pages 215–239. Chapman Hall, 1997.