Semi-formal Verification of Memory Systems by Symbolic Simulation*

Husam Abu-Haimed, Sergey Berezin, and David L. Dill

Stanford University
{husam,berezin,dill}@stanford.edu

Abstract. We propose a debugging method for data-path intensive systems, in particular, memory systems. The approach is based on strengthening invariants by deriving constraints on data in the design using symbolic simulation with constrained inputs. A new heuristic is introduced for finding the appropriate input constraints for the symbolic simulation. We give up soundness in order to gain more automation and efficiency, minimizing or even eliminating the required manual effort. While it is no longer possible to prove the correctness of the design, experimental results demonstrate that the technique is quite effective in finding design errors.

1 Introduction

Most hardware systems of interest today are much larger than what can be reliably tested by conventional methods, and some form of formal verification becomes a necessity. In order for non-expert users to be able to apply formal methods, the tools must be mostly automatic. Some of the most successful approaches to date are model checking [5], theorem proving [11], and validity checking [12]. However, these approaches are often applicable only to relatively small systems, or require significant manual guidance.

In this paper, we are interested in verifying memory systems and similar data-intensive designs. Due to the large sizes of data structures used in memories, we model them as infinite systems. Proving the correctness of such designs usually boils down to proving an invariant. The approach we propose can be used in verifying arbitrary safety properties which can be expressed as invariants.

The standard way to prove invariants for infinite systems is by induction over time. Most of the time, however, the invariant we want to prove is not inductive and has to be strengthened. Often, invariants are strengthened manually in a very tedious iterative process that requires experience and familiarity with the design. This is the most difficult and time consuming part of the verification process.

* This research was supported by GSRC contract DABT63-96-C-0097-P00005, by National Science Foundation CCR-0121403, and by King Fahd University of Petroleum and Minerals, Saudi Arabia. The content of this paper does not necessarily reflect the position or the policy of GSRC, NSF, or the Government, and no official endorsement should be inferred.

D. Geist and E. Tronci (Eds.): CHARME 2003, LNCS 2860, pp. 158–163, 2003.

Many techniques have been proposed in the literature to partially automate the process of strengthening invariants [7,8,6,13,14,3,9,10,2,15,4].

In a previous work [1] we introduced a method for strengthening and proving invariants by the technique called *consistency testing* which uses symbolic simulation. In that method, the user may have to supply the consistency test manually, and the tool then constructs the remaining part of the inductive invariant, proves it, and verifies that the supplied test satisfies certain properties to guarantee soundness.

In this work, we propose a similar method, but without the soundness check and with a simplified induction scheme. The consistency test is replaced by *input constraints* constructed automatically using a special heuristic. This results in a potentially unsound method, but it becomes completely automatic and serves as a very efficient debugging tool. Besides skipping the soundness check, the efficiency is also gained by reducing the number of cycles in symbolic simulation compared to the previous method. We use CVC [12] as a symbolic simulator and a validity checker in our experiments.

Our approach is based on the empirical observation from several examples that most of the invariants in data path intensive systems can be obtained by symbolically simulating the system for a few cycles with specific inputs. The inductive step is then proven only for the states that can be reached by such symbolic simulation, instead of for all reachable states. In order to complete the proof, we need to show that all the reachable state are included in this set of states. However we do not discuss this problem in the paper.

Instead, we give up this soundness check and propose our approach as a debugging tool. We tested the effectiveness of this approach by applying it to several examples of memory systems. In all the examples we considered, it was able to find all design errors in addition to several errors we inserted to test the effectiveness of our approach. This gives us confidence in the effectiveness and reliability of our approach as a debugging technique.

The paper is organized as follows. Sections 2 and 3 formally introduce induction on time and functional equivalence, followed by a detailed description of our verification technique in section 4. An automatic technique for finding input constraints is given in section 5. Section 6 concludes the paper.

2 Induction on Time

We model a hardware design as a *transition system* $T = (S, s_0, N, R, D_{\text{in}}, D_{\text{out}})$, where S is a non-empty (and possibly infinite) set of states, $s_0 \in S$ is the *initial state*, D_{in} and D_{out} are the domains of *inputs* and *outputs*, $N : S \times D_{\text{in}} \to S$ is the *transition function*, and $R : S \times D_{\text{in}} \to D_{\text{out}}$ is the *output function*. We write $N(s, \alpha^\ell)$ to denote the final state of running T on the input sequence α^ℓ of length ℓ starting from the state s:

$$N(s, \alpha^\ell) = \underbrace{N(N(\ldots N}_{\ell}(s, \alpha_0), \alpha_1), \ldots, \alpha_{\ell-1}).$$

It is important to note that a single transition in T can actually represent a complex transaction in the real hardware implementation requiring multiple cycles of execution.

A state s is called *reachable* in a transition system T, if there is an input sequence α^ℓ such that $s = N(s_0, \alpha^\ell)$, where s_0 is the initial state of T. In this paper, we only consider *safety properties*, or *invariants* over the set of reachable states. We say that a transition system T satisfies a safety property $Q(s)$, if $Q(s)$ holds for every reachable state s of T. This can be stated as follows:

$$\forall \ell, \alpha^\ell.\ Q(N(s_0, \alpha^\ell)). \tag{1}$$

The conventional way of proving (1) is by induction on time, when Q is first shown to hold in the initial state s_0, and then the transition function N is shown to preserve Q:

$$Q(s_0), \qquad \forall s, \sigma.\ Q(s) \Rightarrow Q(N(s, \sigma)). \tag{2}$$

In practice, this induction scheme requires finding an inductive the invariant, which is often the hardest and most tedious part of verification process.

3 Functional Equivalence

We prove correctness of systems using the idea of *functional equivalence*. The problem is stated as follows. Given two systems, the concrete system T^c (the system we want to verify) and the abstract system T^a (which defines the required functionality of T^c), prove that T^c is functionally equivalent to T^a. Two systems are said to be functionally equivalent if they produce the same sequence of outputs for the same sequence of inputs. Formally, this is expressed as follows:

$$\forall \ell, \alpha^\ell, \lambda.\ R^c(N^c(s_0^c, \alpha^\ell), \lambda) = R^a(N^a(s_0^a, \alpha^\ell), \lambda). \tag{3}$$

If we define $Q(s)$ to be $\forall \lambda.\ R^a(s^a, \lambda) = R^c(s^c, \lambda)$, (3) becomes $\forall \ell, \alpha^\ell.\, Q(N(s, \alpha^\ell))$, which is the same as formula (1). So, we can use the same induction principle given by (2) to prove the functional equivalence (3) of the two modules.

4 The Verification Method

In this section we introduce our approach through a simple example. We show how the direct use of (2) to prove the correctness of a memory system fails. Then we show how our method can be used to deal with the problem.

Consider a small example of a read-only memory with a single-line cache given in figure 1(a). To verify the correctness of this design, we show that it is functionally equivalent to a simple (uncached) array of data in figure 1(b). Since the memories are read-only, the input to both modules is the address ($D_{\text{in}} = \mathsf{Addr}$), and the output is the data read from that address ($D_{\text{out}} = \mathsf{Data}$).

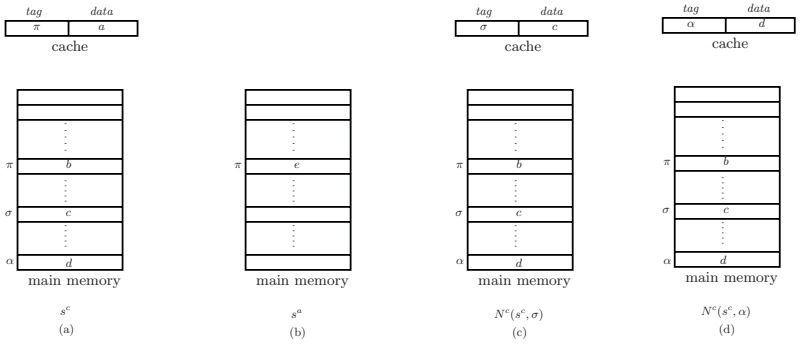

Fig. 1. Memory Example

The transition systems T^c and T^a are defined as follows. The abstract state s^a of T^a is just an array M indexed by Addr and holding the Data elements. The next state function N^a is the identity function, and $R^a(s^a, \lambda) = M[\lambda]$. The concrete state s^c of T^c contains the state of the cache in addition to the same array M. Initially, in s_0^c, some arbitrary address is cached such that the cache is coherent with the main memory M. The next state function $N^c(s^c, \lambda)$ adds the address λ and the data stored under that address $M[\lambda]$ to the cache, yielding the new state. The output function $R^c(s^c, \lambda)$ is similar to N^c, except that it returns the data associated with the address λ.

Unfortunately, proving the functional equivalence of the two memories by simple induction fails. Consider the state in figure 1 (a) and (b) where $a \neq b$ and $a = e$. In this case, s^c and s^a are functionally equivalent and hence the induction hypothesis $Q(s)$ is satisfied. However, transitioning to the next state by reading some address $\sigma \neq \pi$ brings T^c to a new state $s^{c\prime}$, shown in figure 1 (c), where the address π is no longer cached. Therefore, reading π again yields $b \neq a$, which no longer agrees with T^a. The induction fails in this case because it starts out from an incoherent state, which is not reachable. The natural way to strengthen the invariant is to require the state to be *coherent*. In this example it means that the cached value must be the same as in the main memory. So, in general, we can strengthen invariants for such systems by asserting their coherence.

Now suppose we simulate the incoherent state s^c for one step with the input constraint $C(s^c, \alpha) \equiv \alpha \neq \pi$. The resulting state $s^{c\prime}$ is shown in figure 1(d). Clearly, state $s^{c\prime}$ is coherent, and the induction (2) for such a state is valid. Formally, (2) is restricted to the set of states Σ' defined as follows:

$$\Sigma' = \{(s^a,\, s^{c\prime}) \mid \exists s^{c\prime\prime}, \alpha.\ s^{c\prime} = N^c(s^{c\prime\prime}, \alpha) \wedge \alpha \neq \pi\}.$$

The induction (2) with Σ' becomes:

$$Q(s_0), \qquad \forall s \in \Sigma', \sigma.\ Q(s) \Rightarrow Q(N(s, \sigma)). \tag{4}$$

Proving (4) does not complete the proof of correctness for the memory system; it simply says that the concrete system behaves according to the specifications when started from any state in Σ'. To complete the proof of correctness,

we need to prove that all reachable states Σ are included in Σ'. That can be done by proving the following induction:

$$\Sigma'(s_0), \qquad \forall s, \sigma.\ \Sigma'(s) \Rightarrow \Sigma'(N(s,\ \sigma)), \tag{5}$$

In general, (5) is undecidable. For some memory systems, however, proving it can be a matter of a simple intuition of the designer. For cases where we fail to prove (5), our approach can still be used as an effective debugging tool. For the cache example, it is easy to show that (5) is valid and that completes the proof of correctness for this example.

The general idea in our approach is to find an *input constraint* $C(s,\ \alpha^k)$ on an input vector α^k that when executed on an arbitrary state s will remove the incoherences in it. For instance, in the example above, the read from $\alpha \neq \pi$ removes the incoherence by causing c to be copied from the main memory to the cache.

5 Finding Input Constraints

Data path intensive systems consist mainly of registers interconnected by buses (or *links*). Each link has a condition or predicate associated with it. When the condition is *true*, the data is transfered along the link. In any system transition many data transfers may happen. These data transfers imply some constraints on the state of the system. In the cache example, the data transfer from the main memory to the cache implies the constraint that the cache and the main memory are always coherent. In general, we can control which data transfers happen in each system transition by constraining the inputs. In the cache example, we constrained the input by $\alpha \neq \pi$.

Our heuristic looks for the right input constraints that will exercises the right links and get the data synchronized. The idea is to look at counterexamples of failed proofs. Suppose we try to prove

$$\forall s, \sigma.[Q(s) \Rightarrow Q(N(s, \sigma))]. \tag{6}$$

If the proof fails, we get back a counterexample C. Intuitively, C defines the data transfers that contributed to the failure of the proof. Based on our assumption, the proof failed due to incoherences between the data involved in these transfers. If we simulate s for one transition and exercise the same links as in C, we are likely to get rid of these incoherences. Let c_i be the condition associated with a link l_i. If l_i is activated in C, its condition c_i becomes true in C. Let Σ' be the set of states where every c_i holds for each link l_i activated in C. That is, the input constraint becomes $C(s,\ \alpha) = \bigwedge_i c_i$. Then we try to prove:

$$\forall s' \in \Sigma', \sigma.[Q(s') \Rightarrow Q(N(s', \sigma))]. \tag{7}$$

By simulating s with the constraint $C(s,\ \alpha)$, it is likely that we will get rid of the incoherences. If (7) is not valid, we get a new counterexample and repeat

the process. If at any point we get a counterexample with the same set of activated links as in any previous counterexample, we report it as a potentially true counterexample. The user can also put a limit on the number of iterations to guarantee termination.

6 Conclusion

In this paper, we presented an automatic technique for finding design errors in memories and data path systems. The method is based on a semi-formal version of invariant checking using symbolic simulation with automatically generated input constraints. We tested the method on various types of memory systems (one and two-level direct-mapped cache, set-associative cache, and a memory system with SDRAM controller), and the method found all the bugs in these designs without any manual effort, which demonstrates its effectiveness. The longest runtime was for the two-level direct-mapped cache, and it took 10 minutes on a machine with a 800MHz Pentium processor.

References

1. Husam Abu-Haimed, Sergey Berezin, and David L. Dill. Strengthening invariants by symbolic consistency testing. In *CAV'03*, volume 2725 of *LNCS*, 2003.
2. Saddek Bensalem, Yassine Lakhnech, and Hassen Saidi. Powerful techniques for the automatic generation of invariants. In *CAV'96*.
3. Nikolaj Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. In *Theoretical Computer Science*, 1997.
4. Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *CAV'94*.
5. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
6. Michael Colon and Tomas E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *CAV'98*.
7. Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In *FMCAD'02*.
8. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV'97*.
9. Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1993.
10. John Rushby. Integrated formal verification: Using model checking with automated abstraction, invariant generation, and theorem proving. In *SPIN'99 workshop*.
11. N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.
12. A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *CAV'02*.
13. Jeffrey X. Su, David L. Dill, and Clark W. Barrett. Automatic generation of invariants in processor verification. In *FMCAD'96*.
14. Jeffrey X. Su, David L. Dill, and Jens U. Skakkebæk. Formally verifying data and control with weak reachability invariants. In *FMCAD'98*.
15. A. Tiwari, H. Rueß, H. Saidi, and N. Shankar. A technique for invariant generation. In *TACAS'01*.