Protocol Verification as a Hardware Design Aid*

David L. Dill Andreas J. Drexler Alan J. Hu C. Han Yang
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

The role of automatic formal protocol verification in hardware design is considered. Principles are identified that maximize the benefits of protocol verification while minimizing the labor and computation required. A new protocol description language and verifier (both called Murφ) are described, along with experiences in applying them to two industrial protocols that were developed as part of hardware designs.

1 Introduction

Most complex digital designs must be regarded as *concurrent systems:* individual modules run in parallel and must coordinate by explicit synchronization and communication. Complexity will continue to increase, portending a shift in total design effort from, for instance, faster arithmetic circuits, to mechanisms for coordination. Those mechanisms usually involve *protocols:* rules that, if followed by each party in a coordinated action, assure a desired outcome.

Unfortunately, protocol design is a subtle art. Even when a designer exercises the utmost care, there is a strong possibility that he or she will fail to anticipate some possible interaction among rules, resulting in errors and deadlocks. Even worse, the nondeterminism resulting from differing internal states or delays means that resulting errors are often not reliably repeatable, making testing and debugging extremely difficult.

Consequently, the protocol design problem seems an obvious target for computer assistance. However, while protocol simulation is an effective way to catch "obvious" errors, many protocol problems arise when many unusual conditions arise at once; catching these problems reliably would require unrealistic amounts of simulation.

Because of these long-standing problems, formal verification of protocols has stimulated a great deal of interest. In particular, automatic methods ("perturbation analysis") that explicitly enumerate all the system states reached under a particular protocol have been used for many years [16, 1, 8]. Generally, these methods have been applied to communication and network protocols.

We believe that protocol verification is now a digital CAD problem; protocol verifiers should be in every digital designer's toolbox. There have already been some initial steps in this area: AT&T's COSPAN protocol verifier has been used for hardware designs [12], and McMillan's SMV program was recently applied to a cache coherence protocol for a shared-memory multiprocessor [10].

*This research was supported by the National Science Foundation (grant number MIP-8858807), the Defense Advanced Research Projects Agency (contract number N00039-91-C-0138) and by gifts from the Powell Foundation and Mitsubishi Electronics. Most of this work was done using equipment generously donated by Sun Microsystems.

We report here on our experiences using a protocol description language and verifier (both of our own design) called Murφ on some industrial digital design problems. We discuss factors that increase the chance for successful use of a protocol verifier, overview the particular description language and verifier we used, and describe the results of our efforts.

Protocol Verification

Protocols are needed in hardware to achieve coordinated action in the presence of complicating factors such as communication channels that impose long and possibly varying delays or deliver messages unreliably or out-of-order, bounded buffers or other scarce resources that may cause deadlocks, and components that have unpredictable delays and other nondeterministic behavior. Of special interest at this time are protocols for communication over networks or inter-processor switches and protocols for maintaining cache coherence in shared-memory multiprocessors.

Formal verification of a protocol proceeds by describing the protocol in some language and then comparing the behavior of this description with a specification of the desired behavior. A verifier generates states from the description, comparing them with the specification as it goes. If the verifier detects an inconsistency, this fact is reported, along with an example sequence of states that illustrates how the problem can occur, to aid in diagnosis. The description of the protocol can be in many forms: a program-like notation, a collection of finite-state machines, or a Petri net, for example. The simplest specifications are for fixed properties, such as absence of deadlock, or *invariants,* which are properties that should be true of individual states. More sophisticated verification systems can handle specifications in the form of temporal logic formulas or automata.

The usual assumption about the role of formal verification is to provide a guarantee of design correctness. Although this is a worthy goal, it is very difficult to achieve. We take a more "economic" view: the main potential of formal verification is to reduce the cost and time of product development.

One of the most important ways to make verification of large systems possible is *down-scaling* — pretending that they are small systems. Most of the bugs in a protocol to coordinate thousands of processes can be demonstrated using two or three processes. In this case, down-scaling would be formally verifying the protocol with two or three processes. In some sense, this is the opposite end of the spectrum from simulation: instead of testing a small fraction of the possibilities for a large model of the system, we check all of the possibilities for a small instance of the system. Neither method is guaranteed to catch all of the problems, but down-scaling will almost certainly catch some problems that simulation will not (and vice-versa — we are not advocating the elimination of simulation).

2 Description language

The Murφ description was designed to be the simplest possible usable language that supports nondeterministic, scalable descriptions. Murφ meets these particular goals (especially simplicity) better than existing hardware and protocol description languages. [2, 3, 11, 8, 9, 14, 13].

Murφ describes a system using a set of iterated guarded commands, like Chandy and Misra's Unity language (which inspired it) [4].

2.1 Murφ Language

A Murφ description consists of constant and type declarations, variable declarations, procedure declarations, rule definitions, a description of the start state, and a collection of *invariants*. An invariant is a Boolean expression that references the variables.

Rules are written

Rule
Boolean–expression
⇒
stmSeq

Each rule is a guarded command [6], consisting of a *condition* and an *action*. The condition is a Boolean expression consisting of constants, declared variables, and operators. The action is a sequence of statements,

A *state* is a function that gives values to all of the variables. An *execution* of the system is a finite or infinite sequence of states $s_0, s_1, \ldots$, where s_0 is determined by the description of the start state that is part of the description. If s_i is any state in the sequence, s_{i+1} can be obtained by applying some rule whose condition is true in s_i and whose action transforms s_i to s_{i+1}. In general, s_i can satisfy several conditions, so there is more than one execution (nondeterminism). A simulator for Murφ might choose the rule randomly; a verifier must somehow cover all the possibilities. In either case, the invariants are applied whenever a state is explored; if any invariant is violated, an error is reported.

Murφ is well-suited for an asynchronous, interleaving model of concurrency, where atomic steps of individual processes are assumed to happen in sequence, and one process can perform any number of steps between the steps of the other. When two steps are truly concurrent, there will be executions that allow them to happen in either order in the interleaving model. In Murφ, concurrent composition is very easy: to model two processes in parallel, just form a new description using the union of their rules.

Given the importance of down-scaling in verification, we have put some effort into making it possible to change the scale of a Murφ description by changing a single parameter. A Murφ description of a protocol that coordinates n processes can be written with a declared constant (e.g. `NumProcesses`). Then a subrange `Processes : 0..NumProcesses-1` can be declared, and the states of the processes stored in an array indexed by `Processes`. Finally, a collection of nearly identical rules can be defined using the ruleset construct:

Ruleset *formal* : *range* **Do**
ruleSet
Endruleset;

A ruleset can be defined that allows the rules for a process to be instantiated for every process number in the type `Processes`.

A description written in this style can be scaled by changing only the constant declarations.

Quantifiers in expressions also promote scalability:

Forall a: *addressType* **Do** $v[a] = w[a]$ **Endforall**

is a Boolean expression which is true if all $v[a]$ equal $w[a]$ over some given address range.

Statements have sequential semantics, i.e. assignments take place in the environment that has been modified by all previous assignments. The usual conditional statements **if–then–elsif–else** and **switch** (case) are part of Murφ. There is a restricted **for** statement that must have compile-time constant loop bounds. Murφ procedures are essentially "macros" with parameter type-checking. These constructs will probably be generalized in the future, but they were sufficient for examples described here.

2.2 Specifications

Murφ has several features for detecting design errors. First, it can detect *deadlocks*, which are defined to be states with no successors other than themselves. Second, an **Error** statement can appear in the body of a rule (almost always imbedded in a conditional). This feature is especially useful when some branches of an **If** or **Switch** statements are not intended to be reachable. The **Error** statement prints a user-supplied error message and causes an error trace to be printed. There is also an **Assert** statement, which is an abbreviation for a conditional error statement. Finally, the user can define *invariants* in a separate part of the Murφ description. An invariant is a Boolean expression that is desired to be true in every state. When an invariant is violated, an error message and error trace a generated.

These specification facilities are limited, because they do not allow one to directly express properties of sequential behavior. Another important limitation is the lack of general facilities for dealing with liveness or fairness properties. For example, we cannot detect *livelocks*, a deadlock in part of the system is masked by activity in another part of the system. However, we have been able to verify important properties of real examples using only deadlock, error, and invariant checking. The specification facilities of the system will be expanded in the future.

2.3 Murφ Compiler and Verifier

The Murφ compiler takes a Murφ source description and generates a `C++` program, which is compiled together with code for a verifier which check for invariant violations, error statements, assertion violations, and deadlock.

The verifier attempts to enumerate all the states of the system, checking for error conditions as it proceeds. Because space is at a premium in verification, states are represented compactly by encoding all scalar types in the minimum possible number of bits, then concatenating the fields without regard to byte and word alignment constraints. This slows down access to fields somewhat, but is justified by the massive space savings that result. A hash table with double hashing that stores reached states is used to decide efficiently if a newly-reached state is *old* (has been reached already) or *new* (has not been reached already). New states are stored in a queue of *active states* (states that still need to be explored). Depending on the organization of this queue, the verifier does a breadth–first search or a depth–first search. Every state in the hash table has a pointer to a predecessor state that can be used to generate an error trace if a problem is detected. Breadth–first search is used by default, because it causes the error-traces to be as short as possible.

3 Experience on larger examples

We have used Murφ on two hardware designs that are "real" in the sense that they were intended to become commercial products: a directory-based cache coherence protocol for a multiprocessor, and a synchronous link-level communication protocol. In both cases, we began verifying early in the design phase, basing our Murφ descriptions on

an informal design specification. In both cases, we found significant errors and omissions and spent a great deal of time modifying and enhancing the designs to meet our correctness conditions.

Verification goes through several stages: deciding how to model the problem (especially, what details to omit); writing the description; using verification to find description errors; and only then discovering genuine design errors.

When the first serious design error is discovered, the system design needs to be modified. But, even if the modification avoids introducing more bugs, more bugs in the original design are uncovered. The verification process then enters a tight loop redesign-reverify loop much like the more traditional edit-compile-debug that programmers experience.

3.1 Cache coherence protocol

Directory-based cache coherence is a way of implementing a shared-memory abstraction on top of a message-passing network, by recording in a central directory which processors have cached readable or writable copies of a memory location. Maintaining cache coherence can be somewhat complicated. For example, if a processor p wants a writable copy of a location which is cached read-only by processors $\{q_i\}$, a request for a writable copy is first sent from p to the directory. The directory then sends a writable copy to p (which can then proceed) and an invalidation message to every q_i. Each q_i invalidates its copy and sends and acknowledge back to the directory, which is waiting for all the invalidations to arrive before processing any more transactions on that location.

Although this single transaction sounds simple enough, the problem becomes more complicated when one considers scenarios in which several different transactions on the same location have been initiated at the same time, especially when messages are not guaranteed to arrive in the same order they were sent. A protocol verifier methodically explores all of these possibilities.

Since Murφ has no built-in support for message communication, the network was modeled as an array with a counter of the number of messages it contained. Out-of-order message reception was modeled using a rule set that had the position of the message in the array as a parameter. The has the effect of nondeterministically choosing a message to process, regardless of the order of message transmission.

The description has separate scaling parameters (constant declarations) for: number of main memories and directories, number of caches and processors, number of addresses, number of legal memory values, size of directory entry (number of cached entries that can be kept track of), and capacity of the message network.

The specification of the protocol is not complete. Instead, we have specified a set of properties that seem to be obvious necessary conditions for correct operation. Our specification made use of in-line error statements, invariants, and deadlock checking. The in-line error statements were used for several purposes, including reporting on common description errors, such as overflowing the network array or the directory. However, the most important error statements were those that we inserted methodically on every unused branch of an if or case, to flag presumed impossible occurrences. These error statements were especially useful for detecting "unspecified receptions" of messages.

Other properties were specified using three invariants. The first checked for conditions that were empirically likely to be violated by description errors. For example, if the directory state for a particular memory address is "INV" (indicating that there are no cached copies), the directory list of cached copies should be empty.

The other two invariants check for cache consistency properties. One of these basically asserts that there are never two modifiable cached copies of the same address, although the condition is made much more complicated by various exceptions for transient states. For example, in this particular protocol there may legally be two modifiable copies if one is already being written back to main memory. Most of these conditions were determined experimentally by starting with simple invariant, running the verifier, and inspecting the results to see whether the violation is because the invariant is too strong or because of a genuine error.

The final invariant asserts that if a cache entry is read-only, its value is the same as the corresponding value in main memory. This, too, is tempered by various exceptions for transient conditions. For example, the protocol allows a modifiable copy to be converted to a read-only copy by writing back the modified value to main memory and changing the cache entry state. While the writeback message is in transit, the value of the (now) read-only cache entry may be different from the (not-yet-written-back) memory value.

Surprisingly, almost all of the errors found were found with a description consisting of one main memory/directory, two processors/caches, and one memory location with one possible value (zero bits of data). Verifying at this scale required examining on the order of two thousand states. Scaling up to three processors, two values, and two main memories revealed only trivial errors, such as the use of the constant 0 for a value instead of the proper variable. In this case, hundreds of thousands of states were examined. The state explosion problem was only an issue in verifying scaled-up versions of the system, where, in fact, no additional problems were discovered.

Many of the in-line error statements were triggered, every invariant was violated, and several deadlocks were detected. There were many errors in the modeling, particularly in the handling of the network (e.g. failing to remove a message after it had been handled). Another more significant common error was a message arriving at a processor in an unexpected state, detected by an error statement in the default case of a switch statement. In many cases, this represented a legitimate possibility that could be handled by augmenting the design. In other cases, deeper changes were required. Many other errors were manifested in illegal global states, such as two processors having writable copies of a location. Only one memory value was required because most problems that would lead to inconsistencies showed up earlier as illegal states.

3.2 Link-level protocol

We also applied Murφ to the problem of verifying a link-level communication protocol. The protocol is basically a complicated version of the well-known alternating bit protocol, in that it uses one-bit sequence numbers to catch lost and duplicated messages. One of the complications is that the protocol has the capability of transmitting a group of several packets as a single unit.

Verification with Murφ caught several fundamental errors in the initial design. Many of these stemmed from a group of packets being disrupted by the retransmission of another single packet or group of packets. Redesigning the protocol to be correct and also meet given performance goals was quite difficult, and required over a month of effort (with countless iterations of the verification– redesign cycle).

Three major properties were specified. The first two were that that messages were not lost or duplicated. These were specified entirely by in-line error checks by exploiting *data independence* [15]: the control of the protocol does not depend on the data being sent.

The description checks for lost and duplicated packets by sending exactly one packet with a "1" value; all other packets have value 0. The time at which to send the "1" packet packet is chosen nondeterministically. If there is a possibility of a packet being lost or duplicated, there is a possibility that that packet will be the "1" packet. So it is sufficient to verify that the "1" packet is not lost or duplicated. This trick works because the verifier considers *all* of the possible rule executions.

The third property was that a group of packets arrived together. This property was checked by choosing to send no more than one group of packets, all of where the data value in every packet of the group was 1. All other packets carried the value 0. The description contains in-line error statements that look for the acceptance of at most one group of packets with the all values set to 1. A group has been disrupted if and only if some of its packets are 1 and some are 0.

We believe that this specification is essentially complete. However, the same approach as we suggested for the cache coherence protocol of comparing the implementation protocol with another, more abstract protocol could be applied. In this case, the more abstract protocol would model communication over a reliable channel.

Verifying the link-level protocol generally required dealing with larger state spaces than the cache-coherence protocol; however, all detected design errors were found examining fewer than 1 million states, which used 11 megabytes of memory.

4 Conclusions

In summary, automatic formal protocol verification can be a valuable design aid if

- it is used by a designer in the earliest design phases;
- it is regarded as a debugging tool, not a guarantee of total correctness;
- the system is modeled at a high level of abstraction; and
- the system description is down-scaled.

The adoption of these principles maximizes the utility of verification given the current state of the art: they gain maximum economic advantage by catching the most expensive design errors as early as possible, and reduce the sizes of the state spaces that need to be explored, making verification computationally feasible.

Our findings on the industrial examples we have tried are:

- There are many bugs to be found in the early design phase. Verification finds them quickly.
- The state explosion problem was not severe (because of adherence to the principles above).
- We were able to catch many errors using relatively weak specification methods, such as invariants and deadlock checking.

Formal verification is also feasible without these assumptions, for example in comparing low-level sequential circuits [5, 7]. Techniques will advance in the future to increase the payoff for a broader range of problems. However, for the near-term future, we believe that the highest payoff can be obtained with these principles.

Acknowledgements

We would like to thank Ken McMillan for his advice and help with the examples.

References

[1] J. Billing and M.C. Wilbur-Ham. Automated protocol verification. In M. Diaz, editor, *Protocol Specification, Testing, and Verification, II*, pages 77–100. Elsevier Science Publishers B.V. (North-Holland), 1982.

[2] Ed Brinksma. A tutorial on lotos. In *Protocol Specification, Testing, and Verification V*, pages 171–194. Elsevier Science Publishers B.V. (North-Holland), 1986.

[3] S. Budkowski and P. Dembinski. An introduction to estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems*, 14(1):3–24, 1987.

[4] K. Mani Chandy and Jayadev Misra. *Parallel Program Design — a foundation*. Addison-Wesley, 1988.

[5] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[6] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[7] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. MIT Press, 1989.

[8] Gerard J. Holzmann. *Automated Protocol Validation in Argos, Assertion Proving and Scatter Searching*, pages 163–188. Computer Science Press, Inc., 1987.

[9] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[10] K. L. McMillan and J.Schwalbe. Formal verification of the gigamax cache-consistency protocol. In *Proceedings of the International Sympoium on Shared Memory Multiprocessing*, pages 242–251. Information Processing Society of Japan, 1991.

[11] F. Orava. Formal Semantics of SDL Specifications. In *Protocol Specification, Testing, and Verification VIII*, 1988.

[12] K. Sabnani S. Aggarwal, R.P. Kurshan. A calculus for protocol specification and validation. In *Protocol Specification, Testing, and Verification, III*, pages 19–34. Elsevier Science Publishers B.V. (North-Holland), 1983.

[13] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.

[14] *IEEE Standard VHDL Language Reference Manual*. March 1988.

[15] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*. Association for Computing Machinery, 1986.

[16] Pitro Zafiropulo, Colin H. West, Harry Rudin, D.D. Cowan, and Daniel Brand. Towards analyzing and synthesizing protocols. *IEEE Transactions on Communications*, COM-28(4), April 1980.