# Integrating Formal Verification Methods with A Conventional Project Design Flow

Ásgeir Th. Eiríksson
Silicon Graphics Inc.,
Mountain View, CA
asgeir@sgi.com

**Abstract**. We present a formal verification methodology that we have used on a computer system design project. The methodology integrates a temporal logic model checker with a conventional project design flow. The methodology has been used successfully to verify the protocols within a distributed shared memory machine.

We consider the following to be the four main benefits to using the model checker. First, it ensures that there exists an accurate high-level machine readable system specification. Second, it allows high-level system verification early in the design phase. Third, it facilitates equivalence and refinement checking between the high-level specification, and the RTL implementation. Finally, and most importantly it uncovered many protocol specification and RTL implementation problems.

## 1. Introduction

The paper presents a formal verification methodology that we have used on a real-life computer system design project.

The methodology integrates the **smv** temporal logic model checker with a conventional project design flow. The methodology has been used to verify the two main protocols within a directory based distributed shared memory machine [Len92, Tan95]. The two protocols are the cache coherence, and the programming Input/Output (PIO) protocols. The cache coherence protocol is the set of rules that ensure that at any time, different processor and I/O caches contain coherent values for the same memory location, and that the order of writes to different locations, as seen from the programmer, are in accordance with the memory consistency model. Memory mapped I/O devices are controlled by streams of write, and read instructions to predefined locations of memory. These streams are referred to as PIO streams, and the protocol governing their operation as the PIO protocol.

A system model is formally verified by showing with mathematical techniques that it conforms to the specified properties. The properties we might wish to verify for a protocol specification are, for example, the absence of deadlock, and that a processor request always receives the expected response. Formal verification amounts to exhaustively, for all possible cases, verifying that a particular model satisfies the specified properties. In contrast to formal verification: conventional simulation methodology can be viewed as verifying that a system model conforms to the diagnostic test suite, i.e the specification of the design in this cases consists of the diagnostic tests. The inherent problem with the simulation approach is writing diagnostic tests that sufficiently cover all the “interesting” corner cases in the operation of the system.

We chose **smv** [McM93] to formally verify the protocol specifications. There are several reasons for this choice. First, **smv** has been successfully used to verify the specifications of other cache coherence protocols [Cla93,Lon93,McM91]. Another reason is that source code is available for the tool, in case any problems are encountered. Finally, we chose **smv** because it can be integrated with a conventional project design flow; an important consideration from an industry perspective.

We did evaluate several other approaches vis-a-vis **smv**: the **voss** [Bry91,Seg93] finite state machine trajectory analysis tool, the **cospan** [Kur94] finite automaton ω-language containment tool, and the HOL [Gor88] higher-order logic proof assistant. Except for **cospan**, we think that the strength of these tools is not a good match for the type of systems we are interested in analyzing

The **cospan** tool is built on powerful theory that uses a property specific refinement capability to counter computational complexity, and the state explosion problem. We do plan to compare the utility of **cospan** and **smv** for the type of properties we are interested in analyzing.

The **voss** tool was not chosen because it is designed to verify implementations of interacting state machines, and as such it lacks the proper behavioral and temporal abstraction capabilities that are necessary to verify protocol specifications. The tool does not accept nondeterministic state machines which are essential when verifying abstract models. Also the tool restricts temporal specifications to the always-in-next-state AX operator, but does not have the CTL eventually operators that are necessary to verify nondeterministic models.

Finally, the HOL approach was not considered because it is manually intensive, and has mostly been successful in reasoning about data paths, whereas the protocol specification is control logic dominated.

The rest of the paper is organized as follows. We first describe how the model checker is integrated into a con-

**33rd Design Automation Conference ®**

DAC 96 - 06/96 Las Vegas, NV, USA

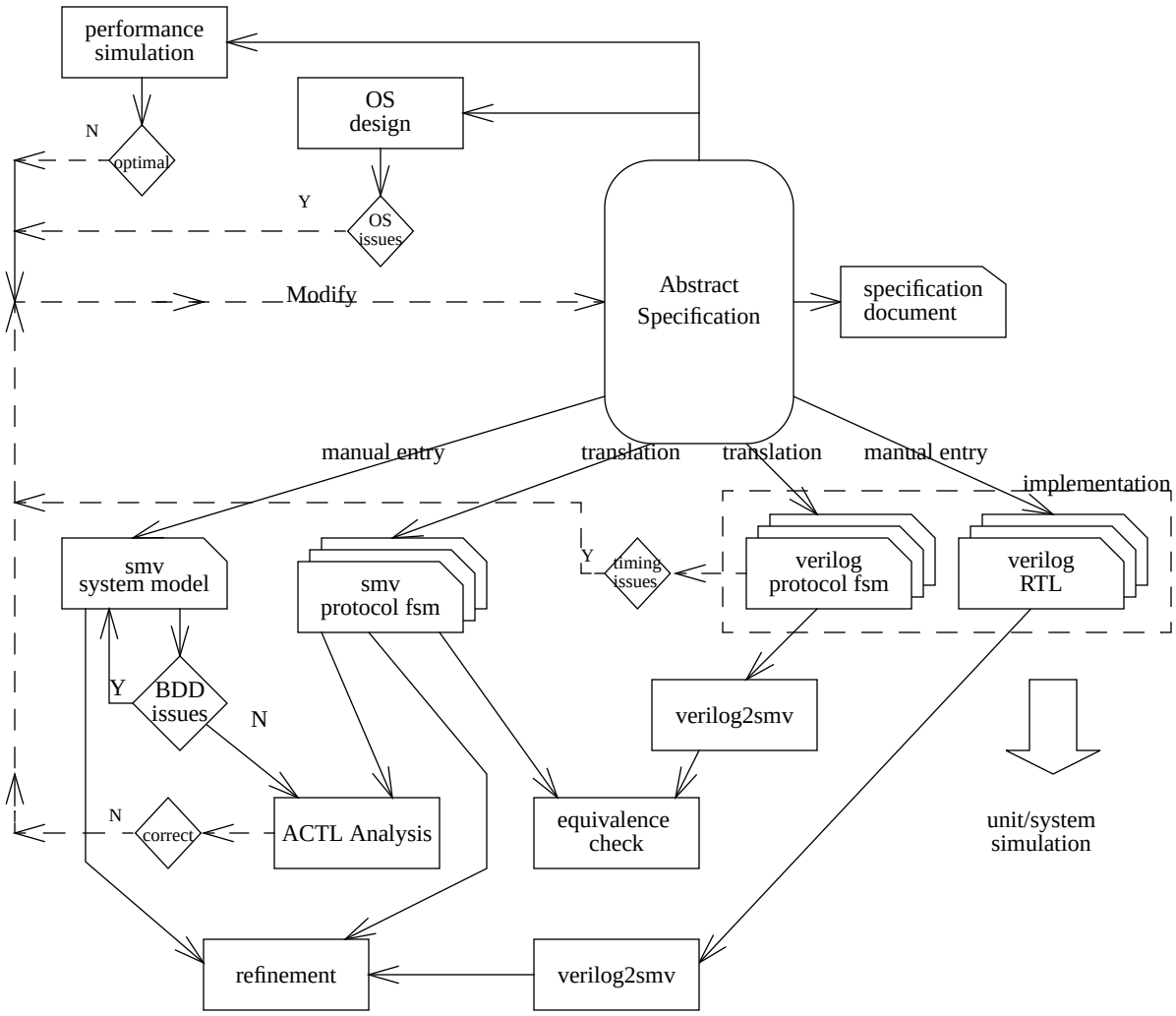


**Figure 1: project design workflow**

ventional project design flow, then describe the model checker run-time environment, and finally discuss open issues and possible future enhancements.

## 2. Design Methodology

We employ top-down methods to maximize the benefits of formal verification/analysis. An overview of the project design workflow is shown in Figure 1.The input to the analysis is an abstract design specification.

The **smv** model is derived from the design specification. Temporal abstraction is used to minimize the granularity of the time scale, and functional dependency analysis is used to eliminate as many state variables as possible. For each state variable we analyze if can be eliminated by re-writing it as a function of other state variables. Finally, we selectively refine the **smv** model, using the RTL implementation.

We proceed in this fashion in order to try to increase the computational efficiency of model checking, and to side-step/avoid the state explosion problem.

It is important to minimize the granularity of the time scale in order to minimize the steps required to explore the reachable state space, and also to minimize the number of iterations during fixed point calculations.

The model checking performed by **smv** is Binary Decision Diagram (BDD) [Bry86] based. The minimization of the number of state variables tends to decrease the size of the BDD data structures within **smv**. In our experience this is always successful for the BDD representing the set of reachable states, but this decrease in the size of the reachable state space BDD is sometimes offset by an increase in the size of the intermediate partitioned transition relation BDDs [Bur91]. The cases where this re-writing of BDD variables is successful, is when a BDD variable can be eliminated by re-writing it as a deterministic, or nondeterministic, function of BDD variables that are close together in the BDD variable order.

Refinement checking is effective at side-stepping

the state explosion problem because different parts of the RTL can be refined separately. Refinement checking is also compuatationally efficient, because it does not involve fixed point calculations; the checking can be performed while deriving the set of reachable states.

A high-level protocol specification consists of a collection of multiple input, multiple output state machine tables, that determine the response to incoming messages in terms of state changes, outputs, and outgoing messages. The tables serve as input to a performance simulation, formal verification with **smv**, verilog RTL state machine generation, and to a text processor specification document generator.

During the early phases of a project, protocol design alternatives are evaluated using a performance simulator.

After performance evaluation trade-off simulations are completed, a protocol typically goes through numerous revisions. The three primary driving forces behind the changes are the following: Operating System (OS) requirements, RTL synthesis timing issues, and protocol problems, uncovered with formal analysis.

There is one accurate machine readable specification. Having one source for the protocol specification has several important benefits. The first is that the different tools are always working with the same version of the protocol. Another is that it is possible to verify any design changes, e.g. due to RTL timing considerations, with the formal verification tool. Finally, once formal verification finds a problem in the protocol, and **smv** verifies the proposed fix, the revised version of the protocol is immediately available to the RTL simulation tools.

### 2.1 smv system model

The model checking performed by smv is BDD based. The structure of the protocol system model is therefore very important in making property verification tractable [McM93,Eir95].

Two conflicting factors have to be taken into account when developing the system model.

First, it is imperative to have all the components of a protocol present in the system model. It is not sufficient, for example, to verify only the processor part of a cache coherence protocol. The system model also has to contain the I/O section of the protocol, i.e. DMA reads and writes. We uncovered several problems in the cache coherence protocol in the area of processor and I/O interaction. If the system model wouldn't have represented all of the cache coherence protocol, these problems would not have been found until system simulation of the processor, memory and I/O sub-systems, at the tail end of the project.

The other factor that has to be taken into account when developing the system model, is that it is essential not to introduce unnecessary detail in the model. First, so as not to add unnecessary complexity to the verification, and second so that the protocol is verified under the most general possible conditions in which it is intended to operate.

It is our experience that if careful attention is paid to the development of the system model, then **smv** is tractable up to 150-200 state variables. For the cache coherence protocol this translates into a model with 3 processors (or 2 processors and 1 I/O, or 1 processor and 2 I/Os), each with 1 cache line, with a 1 bit data value, and 1 directory entry. For the PIO protocol this translates into 2 processors, and 2 I/O modules. Each processor in this case has up to 4 outstanding requests, 3 writes, and 1 read.

We have access to machines with up to 2G bytes of real memory. For larger models than above the problem is not memory size, but rather that **smv** becomes CPU bound. Once an **smv** run-time image exceeds 1-1.2G bytes of real memory, the run-time become prohibitively large.

### 2.2 smv protocol finite state machines

The smv version of the protocol specification tables is created by translating the tables into **smv** case statements. The **smv** case statement is priority encoded, so the first case that is valid for a particular input condition, is evaluated.

The RTL case statements are either synthesized using these same sequential semantics, or using parallel semantics. The latter case relies on the property that the different case are mutually exclusive, i.e. there is at most one case valid for any input combination; the cases are one-hot encoded.

### 2.3 ACTL Analysis

The goal is to verify the following properties of the high-level specifications: there are no deadlocks, the different types of requests, always receive the correct response, there is never unsolicited response, and the safety invariants are never violated.

If for performance reasons the protocol is implemented with one-hot encoding, we also verify that there is always at most 1 row activated in a table. This is accomplished with the following specification: $AG(\Sigma row_i \leq 1)$, where $row_i$ is one of the input conditions in the table (the AG means that the property should hold true in every state within the reachable state set). It is common that the large tables, the largest table has >600 rows, have problems in this area.

It is also important to check the converse condition,

i.e. that each row is activated for some state of the protocol. This property is verified with an EF($row_i$) specification (the EF means that there should exist a sequence of states, from the initial state, to a state where the property is true). The most common cause of a problem in this area is due to problems in the **smv** system model, i.e. it doesn't generate requests in all cases where it should. If not detected this in turn might mask protocol problems.

The following four types of safety properties are verified: expected state machine input conditions, protocol message invariants, protocol state invariants, and a special case of deadlock.

The specification of a protocol only contains the valid input conditions to each of the different state machines. A state machine input error function is derived from each state machine specification; the error function returns false for the valid input conditions, but true otherwise. The first type of AG specification verifies that an invalid input condition never occurs in the set of reached states.

A protocol message invariant, for example, is the property of the cache coherence protocol that a particular processor can only have at most one outstanding request, targeting a particular cache line.

A protocol state invariant, for example, is the property that if a particular processor has an exclusive cached copy of a cache line, then no other processor, or I/O can have a cached copy of the same cache line.

The version of **smv** that we use verifies the safety properties while the state space is being explored. It is therefore most efficient to verify the safety properties first, and then the deadlock and correctness properties.

The absence of deadlock is verified using a two pronged approach.

The first approach uses a specification of the form AG EF$cond_i$, where $cond_i$ is a possible value for an **smv** state variable (the AG in front of the EF means every state should be considered the initial state). This is not an ACTL property (see McM93 for example), and therefore doesn't hold for refined state variables, but is still very helpful in finding missing transitions in protocol state variables.

Another approach to find deadlock, is verifying that if there is a requests outstanding, then a protocol should always have some messages in flight. The transformation of deadlock detection to a safety properties in this fashion is important because safety properties are checked during the reachable state space exploration, and the check is therefore more computationally efficient then an AG EF specification that entails a fixed point calculation.

Finally, the correctness properties are verified using a specification of the form AG(rq → A(rq-status U resp)), or AG(rq->AF resp), where rq is a protocol request, rq-status is the state maintained by the initiator during the transaction, and resp is the expected response.The first specification above specifies that whenever rq is asserted, the rq-status always is true, until resp becomes true. The latter specification specifies that whenever rq is true, resp always eventually becomes true. An example is a read-shared request by a processor in a cache coherence protocol. In this case rq=read-shared, rq-status is that a request buffer is allocated for this request, and resp is the expected read-shared response, e.g. a shared cache line.

### 2.4 equivalence checking

We use **smv** to ensure that the verilog version of a protocol table is equivalent to the **smv** version of the same table.

The boolean equivalence of the two tables is verified using the methodology shown in Figure 2. Let $f_{rtl,i}$, i=1..N, where N is the number of the next-state and output functions, be the translated verilog version of the next-state, and output functions [McM95a,Bai95]. The inputs to this model are represented with I. The corresponding **smv** model is $f_{smv,i}$, with inputs I.

Boolean equivalence is verified by having I as free variables, and verifying that the specification AG($f_{rtl,i}$=$f_{smv,i}$), is satisfied for each i=1..N.

In practice we've observed three sources of problems in this step. First, when the verilog RTL is not up to date with the current version of the protocol specification. Second, because of errors in the scripts that translate the protocol specification tables to verilog. Finally, some cases, where a designer has manually updated the verilog version of the tables, to supposedly reflect a change in the specification, but hasn't done this correctly.

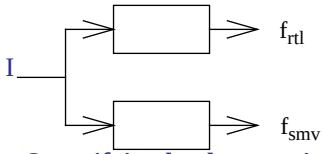

**Figure 2: verifying boolean equivalence**

Equivalence checking is computationally efficient. The largest protocol specification table, has 640 rows, with 74 next-state and output functions. The equivalence check of all the verilog and **smv** files, using one **smv** executable, requires <10 min, and uses 300M bytes of memory.

### 2.5 smv/rc refinement analysis

To make the high-level protocol models tractable within **smv**, it is necessary abstract away as much detail as possible. The abstraction is usually achieved by replacing logic blocks with non-deterministic functions. Scheduling or arbitration for example, is typically mod-

elled with a free nondeterministic variable, that can have a value corresponding to each of the possible choices. We use **smv** to selectively refine the abstract model, using the verilog RTL implementation.

Refinement checking formally analyses that a particular implementation preserves the ACTL properties that have been proven for the abstract model. The methodology used is shown in Figure 3.

Assume that there is a function $f_{abs}$ in the model M. This function can be either deterministic or nondeterministic, and either combinational or sequential. Further assume that the function $f_{abs}$ is implemented in the verilog RTL with the function $f_{rtl}$. The function $f_{rtl}$ typically is a function of the variables in M, and also introduces additional state variables.

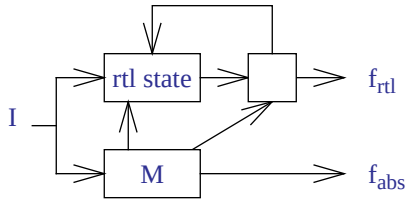


**Figure 3: refinement checking**

The function $f_{rtl}$ is a refinement of $f_{abs}$ if the following ACTL specification is true: AG ($f_{rtl} \subseteq f_{abs}$). To verify this property **smv** verifies that for each state in the reachable state space, the value of $f_{rtl}$ is one of the values of $f_{abs}$, all other state variables having the same value.

The refinement check is either performed by **smv**, or the **rc** tool [McM95b]. The rc tool automatically creates the required AG specification, but we revert to using smv when it is necessary to specify the order of the BDD variables during the refinement checking.

Refinement analysis is time consuming because in order to make it tractable, it requires that the person performing the refinement analysis have a thorough understanding of the design. This can only be accomplished if there is close co-operation between this person and the hardware designer(s). At this point we have only used this approach for a few critical blocks. In each case this has uncovered subtle RTL problems.

Refinement checking performed in this fashion is computationally tractable because only one function is refined at a time.

## 3. Run-Time Environment

A regression run that verifies all the properties of the two protocols, for all the different configurations, contains 250-300 CTL properties. The memory requirement of **smv** for the different configurations is from 30-800M, and the run-time from 10 min - 24 hours, pr. property (150MIPS machine). We have access to 200 machines, with up to 2G bytes of real memory.

To minimize the time required for mini, and full regressions, only 1 property is verified pr. **smv** run, and a batch scheduling utility is used to schedule the different **smv** invocations on the available machines.This setup allows a 2 day turn around on a full regression.

The regression runs are managed with manually created make files. The decision as to which regression to run after a particular design change is also manual. An improvement would be if the decision could somehow be integrated with the formal verification tool being employed.

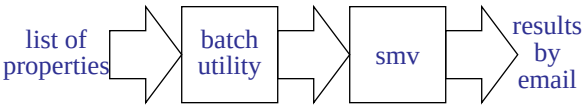


**Figure 4: running smv in batch mode**

## 4. Conclusions

A formal verification methodology, is based on, and therefore ensures that there exist an accurate high-level machine readable system specification. This is never the case in a conventional project design flow. Invariably in that environment, the simulation diagnostics become, over time, the only accurate system specification!

An unexpected advantage to having an accurate machine readable specification, is that it enables the generation of simulation diagnostics directly from the specification. This is used extensively in practice.

If there is access to a large pool of large machines, then model checking with **smv** has sufficient capacity to analyze complete abstract specifications of real-life protocols within the aggressive time schedule of a computer design project.

Refining abstract models using the RTL implementation is feasible in practice. This step is time consuming, and requires a thorough understanding of the design by the person performing the refinement analysis. Each time that we've refined a model using the RTL design we've uncovered subtle design problems.

Our experience with refinement analysis, leads us to conclude that formal verification/analysis can not be an independent activity separated from the design process; as is typically the case in conventional system verification methodology. The design and the analysis has to be integrated as closely as possible to maximize the benefits.

Finally, we observe that formal verification helped to uncover many protocol specification and RTL implementation problems. The designers have acknowledged that some of these problems would never have been found in simulation, and a few due to there subtle symptoms, would not have been found on the test floor.

## 5. Open Issues & Future Enhancements

When a design specification is changed, it is necessary to re-verify the modified design specification. This is a critical time consuming task. It would increase the efficiency of the regression if the formal verification tool could derive which properties need to be re-verified in response to a particular design change.

The state explosion problem prevented us from running large protocol configurations, i.e. >3 processors for the cache coherence protocol, and >2 processors for the PIO protocol. We plan to investigate, if integrating formal verification closely with the design process can increase the capacity of the analysis. We have observed in practice, that it is possible in some cases to make design changes that have no affect on performance, but that increase the capacity of formal analysis.

## 6. Acknowledgments

We are indebted to Ken McMillan of **Cadence Berkeley Labs**, who suggested to us the refinement methodology that we used, provided us with the newest version of **smv**, consulted extensively on the **smv** system model development, refinement analysis, and graciously shared his experience from prior formal verification projects. We would also like to thank the members of the **Leg Hub** team at **Silicon Graphics Inc.**; in particular Jim Laudon, Dan Lenoski, Kianoosh Naghshineh, and Alex Silbey. This formal verification methodology, would not have been successful without their input, and strong support.

## 7. References

[Bai95] Stan Bailes, v2smv, Internal SGI Inc., verilog to smv translator.

[Bry86] R. E. Bryant, “Graph Based Algorithms for Boolean Function Manipulation”, IEEE Trans. on Comp., C-35, pp. 677-681, 1986.

[Bry91] R. E. Bryant, D. L. Beatty, and C. J. Seger, “Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation”, Proc. 28th ACM/IEEE Design Automation Conf., 1991

[Bur91] J.R. Burch, E. M. Clarke, and D. E. Long, “Symbolic Model Checking with Partitioned Transition Relations”, VLSI 91: Proceedings of the IFIP TC 10/WG 10.5 International Conf. on VLSI, Edinburgh, Great Britain, 1991

[Cla93] E. M. Clarke, O. Grumberg, H. Hirashi, S. Jha, D.E. Long, K.L. McMillan, and L. A. Ness, “Verification of the Futurebus+ cache coherence protocol”, Proc. 11th Intl. Symp. on Computer. Hardware Description. Lang. and their Application, 1993

[Eir95] Ásgeir Th. Eiríksson, and Ken L. McMillan, “Using Formal Verification/Analysis Methods on the Critical Path in System Design: A Case Study”, Proc. Computer Aided Verification Conf. ,Liege, Belgium, LNCS 939, Springer Verlag, 1995.

[Gor88] M. J. C. Gordon (ed), “HOL: A Proof-Generating System for Higher-Order Logic”, Kluwer SECS 35, pp. 73-128, 1988.

[Kur94] R. P. Kurshan, “Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach”, Princeton University Press, 1994

[Len92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D Weber, A. Gupta, J. Hennessy, M. Horowitz, M. Lam, “The Stanford Dash Multiprocessor”, IEEE Computer, vol. 25, pp. 63-79, March 1992.

[Lon93] D. E. Long, “Model Checking, Abstraction and Compositional Verification”, Ph.D. Thesis, CMU 1993

[McM91] K. L. McMillan, J. Schwalbe, “Formal Verification of the Encore Gigamax cache consistency protocol.”, Int. Symposium on Shared Memory Multiprocessors, 1991.

[McM93] K. L. McMillan, “Symbolic Model Checking”, Kluwer Academic Publishers, 1993

[McM95a] K. L. McMillan, vl2smv: verilog to smv translator, Cadence Berkeley Labs, 1995.

[McM95b] K. L. McMillan, rc: refinement checker, Cadence Berkeley Labs, 1995.

[Seg93] C. J. Seger, R. E. Bryant, “Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories”, Tech. Report 93-8, Dept. of Computer Science, University of British Columbia, Aug. 1993.

[Tan95] A. S. Tanenbaum, “Distributed Operating Systems”, Prentice-Hall, 1995