Formal verification made easy

by T. Schlipf
T. Buechner
R. Fritz
M. Helms
J. Koehl

Formal verification (FV) is considered by many to be complicated and to require considerable mathematical knowledge for successful application. We have developed a methodology in which we have added formal verification to the verification process without requiring any knowledge of formal verification languages. We use only finite-state machine notation, which is familiar and intuitive to designers. Another problem associated with formal verification is state-space explosion. If that occurs, no result is returned; our method switches to random simulation after one hour without results, and no effort is lost. We have compared FV against random simulation with respect to development time, and our results indicate that FV is at least as fast as random simulation. FV is superior in terms of verification quality, however, because it is exhaustive.

Introduction

Since the Intel floating-point divide bug was published, interest in formal verification has soared. For the interested novice, looking at the huge number of theories and languages (see [1] for an overview), the questions are these: Which approach to choose, which language to learn, and how much mathematics will be required? We think that this is one of the main difficulties impeding the introduction of formal verification. The other big problem has been that formal verification might fail because of model size problems. We therefore had the following goals:

1. No mathematical knowledge at all should be required to use a formal verification tool, nor should it be necessary to learn specific languages. We wanted to use a notation designers already know, which they would find intuitive, and which they use for design purposes as well.
2. Formal verification suffers from the well-known state-space explosion problem. Therefore, FV should be integrated into the overall verification process flow so that, if a model size problem arises, we can easily switch back to random simulation. The effort required to do this should take less than an hour.

Why formal verification?

What are the forces that drive formal verification? In this section, we discuss the most important ones. The first is silicon technology. Over the last few years, circuit densities have increased dramatically, which has allowed designers to implement highly queued systems that offer performance advantages over nonqueued systems. **Table 1** compares two memory bus adapter (MBA) chips from two consecutive CMOS-based mainframe generations manufactured by IBM. Compared with nonqueued systems, queued systems offer higher performance, require many more circuits, and have much larger state spaces; they are therefore much more difficult to verify.

The second force is trends in simulation, which lead naturally to formal verification (see **Table 2**). If we look at the way simulation was done over three successive generations of CMOS-based mainframe systems, we observe the following trends:

1. Constant decrease in model size. In CMOS 1 the most important unit of test was the system model, whereas in

©**Copyright** 1997 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

0018-8646/97/$5.00 © 1997 IBM

Table 1 Comparison of queued and nonqueued systems.

	Bandwidth (MB/s)	*Concurrent operations*	*Transistors*
CMOS 2 MBA	200	3	600,000
CMOS 3 MBA	2000	30	3,700,000

Table 2 Simulation trends of system generations.

System generation	*Main unit of verification*	*Main test approach*
CMOS 1	System model	Deterministic tests
CMOS 2	Chip model	Deterministic tests (initially) Random tests (mid-program)
CMOS 3	Functional unit model	Random tests
CMOS 4	?	?

Table 3 Basic temporal logic operators and their meaning.

Operator	*Interpretation*
AG	For all paths, at every point in time.
AF	For all paths, at some point in time.
AX	For all paths, at the next point in time.
AU	AU has two operands $A[q \cup r]$. It means that for all paths, q is true until r is true.
EG	For some path, at every point in time.
EF	For some path, at some point in time.
EX	For some path, at the next point in time.
EU	EU has two operands $E[q \cup r]$. It means that for some path, q is true until r is true.

CMOS 3 the most important unit of test was one functional unit of a chip.

2. Change from deterministic tests to random tests.

In general, random tests have more error-detection capability than deterministic tests, but unfortunately the debug time for an error found by random simulation is greater than that for deterministic tests. Another problem in random simulation is that there is no answer to the question "If we run random simulation again, will we improve the test coverage?" If we look again at Table 2, the next logical steps in the evolution are the following:

- Verification: Since the control logic is the major source of design errors, verification models containing only the control logic are the most important models. We designate such a unit of test a *module*. (A functional unit consists of both dataflow and control logic.)
- Moving from random tests to exhaustive tests.

Models which contain only control logic are small enough to be formally verified; since FV provides exhaustive tests, FV is the ideal candidate for this next step. If successfully applicable, FV outperforms random and deterministic tests in all areas:

1. Since a complete state space is explored, test coverage is better than that of random tests.
2. Error analysis is easier and faster with an FV tool than with simulation using deterministic handwritten tests, because the FV tool searches for a short path to an error.
3. Clearly, there is no way to show that the properties specified by a designer comprise all of the properties required for a circuit to operate correctly. However, in contrast to random simulation, it is known that all of the specified properties hold.

For control logic designs, FV approaches the ideal as a verification tool, which in general allows us to find all errors quickly and debug them.

Background

Symbolic model checking is a fully automated technique which verifies that a set of properties specified with temporal formulas will hold for a given circuit. The complete state space of the circuit is exhaustively traversed. Temporal formulas are described using temporal logic operators. The eight basic temporal logic operators talk about *computation paths*, which are possible in a *computation tree*. A computation tree is an (infinite) tree of all of the possible execution sequences of a system. The root of the tree is the initial state of the system.

A temporal logic operator consists of two parts: the *path quantor* and the *temporal operator*. There are two path quantors: A, which means *on all paths*, and E, which means *for some path*. There are four basic temporal operators: G, which means *always*; F, which means *eventually*; X, which means *next*; and U, which means *until*. See **Table 3** for a short description of the basic temporal operators.

For a detailed treatment of temporal logic, see References [2–7]. There are two very important classes of temporal formulas: *Safety formulas* specify properties that must hold in the complete state space of the circuit, and *liveness formulas* describe the absence of deadlock in the design.

Integrating FV in a design verification methodology

Figure 1 shows the design and verification process flow in which FV could be used. The process flow follows the

well-known V-diagram. In the implementation branch of the V we start with system design and terminate with the module design; in the verification branch, we follow the branch upward, starting from module test and ending with system test. Note that in this process flow the essential new step in system verification is the module test.

Module test is divided into dataflow test (typically done by simulation) and control logic test. Testing the control logic requires a complete understanding of the behavior of the design; this activity should therefore be performed by the designers. Since the control logic is normally the most complex logic, formal verification should be used. Introducing such a new step in the overall process flow requires acceptance by the workers involved. We therefore had the following goals:

1. *Ease the introduction.*
   To ease the introduction of FV for the designers, little knowledge about FV should be required. Ideally no new languages should have to be learned.
2. *Reuse protocol checkers in the next level of verification.*
   We found it very useful to enhance our test models with protocol checkers. These protocol checkers should also be usable for module test, or, if they are developed during module test, they should be integrable in the next level of verification. This approach reduces the amount of work devoted exclusively to module test.
3. *Minimize the risk.*
   Currently, formal verification can fail because of model size problems, rendering useless the whole model-generation effort. To avoid this, we have developed a method which is nearly identical for simulation and formal verification. If formal verification fails, we perform random simulation; the extra effort required typically takes less than an hour.

Experience has shown that the dataflow of a chip can be quickly debugged, and that the control logic is the dominating source of design errors. As mentioned above, this logic should therefore be subject to formal verification. The components of a typical model to test a piece of control logic are shown in **Figure 2**. The model consists of the design which should be tested, and an *environment*. The environment consists of *protocol generators*, which implement the protocols as required by the design, and *protocol checkers*, which implement the rules the design must follow. The following points are important:

1. Our experience shows that all components in the environment of the model can be represented as FSMs. Therefore this notation, which is familiar and intuitive, was chosen as the input language.
2. The protocol checkers are used instead of complex equations in the computation tree logic (CTL) language [2, 7]. We found only simple CTL equations to be intuitively comprehensible; nontrivial CTL equations are hard to understand and prone to error.

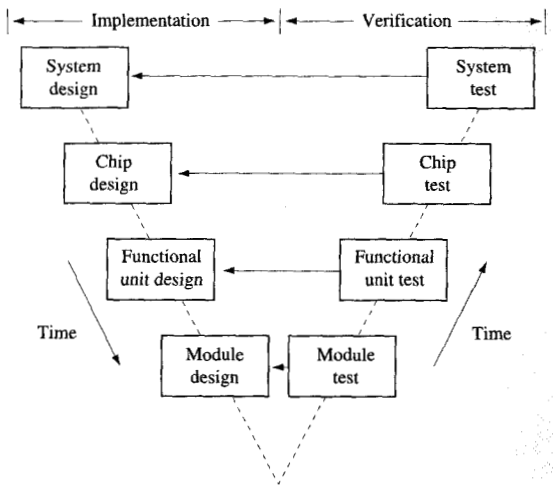

Figure 1

Integrating module test in the design and verification process flow.

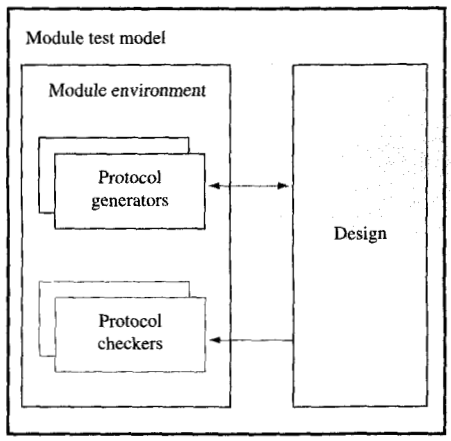

Figure 2

Components of a model for a module test.

The concept of protocol checkers implemented as finite-state machines (FSMs) allows us to automatically generate safety formulas for the formal verification tool. In

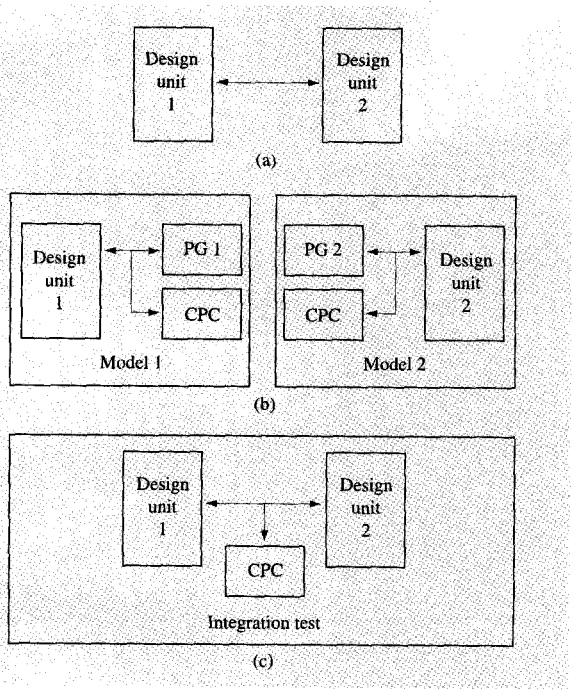

Figure 3

The role of common protocol checkers in the verification process: (a) Hardware structure; (b) two FV models derived from (a); (c) the CPC in the integration test. Protocol generator = PG; common protocol checker = CPC.

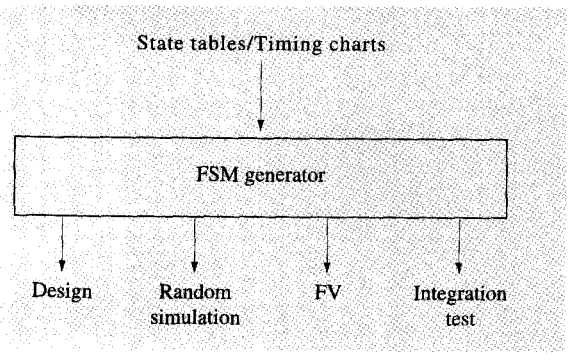

Figure 4

Target languages of the FSM generator.

addition, the protocol checkers can be used in the integration test, improving the overall test process.

Since there are many modules in a complex chip, there are many module test models as well. Therefore, how do we ensure that the module environment description is correct? The technique to achieve this is shown in **Figure 3**, using two design units which communicate via a protocol. To verify these two units, two module test models are generated [Figure 3(a)]. The first model tests design unit 1, and protocol generator 1 is required to implement the protocol in place of design unit 2. The same scheme holds for the second model, which tests design unit 2 and requires protocol generator 2.

In order to check the protocol, a protocol checker is required; since it can be used in both models, it is called a *common protocol checker*. Note that this common protocol checker can also be used in the integration test [Figure 3(c)], further increasing our confidence in the environment description of the module test models.

```
*  Current     Input          Output      Next
*  state       values         values      state
*
     Sx    ,   010        ,   101     ,   Sy
     Sy    ,   101        ,   010     ,   Sy
     Sy    ,   OTHERWISE  ,   010     ,   Sx
```

(a)

```
@    S1    ,   010        ,   101     ,   S1
@    S1    ,   010        ,   101     ,   S2
@    S1    ,   OTHERWISE  ,   010     ,   S3
@    S1    ,   OTHERWISE  ,   010     ,   S4
```

(b)

Figure 5

State table formats for different FSMs: (a) Deterministic machines; (b) nondeterministic machines.

FSM generator

To describe finite-state machines, designers use an FSM generator (see **Figure 4**) which requires as input either a state table or a timing chart. This FSM generator produces either a macro (for design usage) for an IBM internal hardware description language or a function (for simulation usage) in a C-like behavioral description language. With an additional option, we are able to generate the environment description in the language required by the formal verification tool.

Since timing charts are converted internally to state tables, only state tables are considered next. **Figure 5** shows the format of a state table. There are four columns,

which describe the current state, the input vector value, the output vector value, and the next state. States are coded symbolically, and input and output values can be represented as binary or symbolic values. In addition, the OTHERWISE statement is supported in the input column. In addition to Mealy and Moore machines [8], two different classes of machines can be represented: deterministic and nondeterministic. Nondeterministic machines require an @ sign to mark state transitions that should be done nondeterministically (e.g., the machine transitions from S1 and the input vector 010 to the states S1 and S2). In simulation, one transition of the set of possible transitions is selected randomly, while in FV all possible transitions are verified exhaustively.

To the states given in the state table, the generator adds an additional state, the *error state*. The state machine enters this state if during execution an input vector is found which is not defined in the state table for the current state (e.g., if in state Sx an input vector 110 occurs). Both simulation and formal verification check to ensure that no state machine ever enters the error state.

Figure 6 shows an example of a request unit which generates a request accompanied by a request vector, and eventually receives a grant. The request vector points to eight different targets (T0, .., T7), which should be selected nondeterministically. The decision to generate a request should also be done nondeterministically. The FSM may nondeterministically transition from the initial state S0 to any of the request states (R_T0, .., R_T7), or may remain in the initial state. In each of the request states, the FSM generates the request signal and a corresponding Req_Vec value (which is symbolically coded T0, .., T7). As long as no grant is received, the FSM remains in the request state; if a grant is received, the FSM returns to the initial state S0.

Since this tool is used for design and simulation, the designers are already familiar with it. There is no need to learn a new language in order to generate an environment description for formal verification.

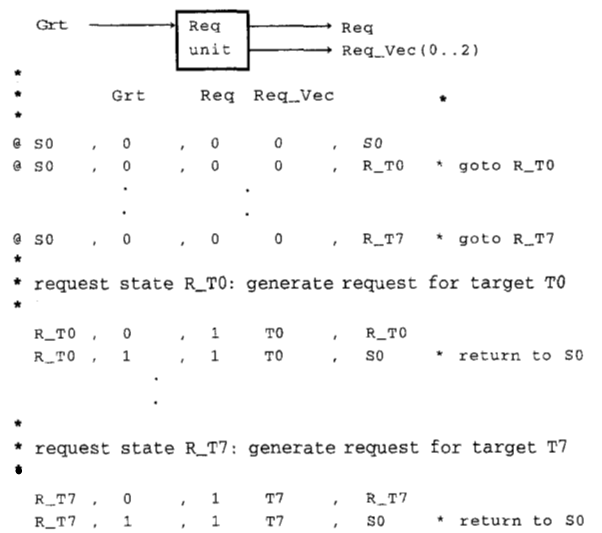

Figure 6

State table for a request unit.

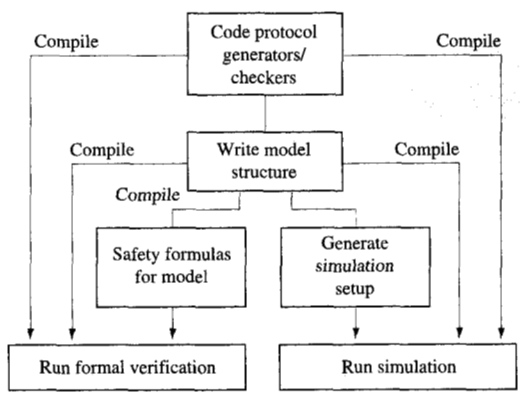

Figure 7

Switching between FV and simulation.

Switching between FV and random simulation

If different target languages can be generated from a single input description, it is obvious that switching between simulation and FV is trivial. **Figure 7** shows the process of switching between FV and random simulation. Obviously, the biggest task is the description of the protocol generators/checkers. Since the models are small, all other activities require little effort.

Since the model structure contains all FSMs, it is possible to identify all error states in the model. These error states are used in safety formulas. An example is shown in **Figure 8**, where the first formula states that for all possible execution paths (AG = always globally) in every cycle, the condition holds that the state of the FSM REQ_U0 is not the error state.

Safety formulas are an important class of temporal formulas. Since they can be generated automatically from the model structure, all components required by the formal verification tool are available with no FV

```
rule No_Error {
  formula { AG ( REQ_U0.State  != Error_State ) }
                    :
  formula { AG ( REQ_U5.State  != Error_State ) }
  formula { AG ( SLT_U0.State  != Error_State ) }
                    :
  formula { AG ( SLT_U7.State  != Error_State ) }
}
```

Figure 8

Safety formulas.

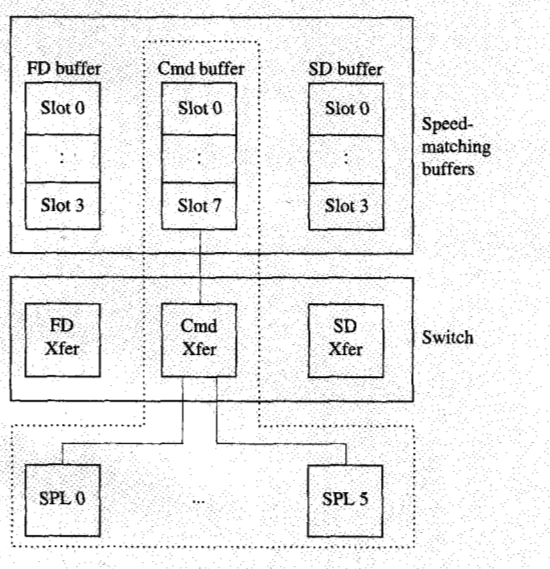

Figure 9

A design example [11]. Fetch data = FD; Store data = SD.

knowledge. Users can therefore use the formal verification tool immediately and get accustomed to it.

RuleBase: A formal verification tool

RuleBase [5, 9, 10] is an industry-oriented formal verification tool developed by the IBM Haifa Research Laboratory. Based on years of experience in practical formal verification of industrial hardware designs, RuleBase offers access to this advanced technology to every designer, not just FV experts. RuleBase uses an enhanced version of SMV [2] as its verification engine, employing the CTL model-checking verification method [7]. SMV is an efficient and robust symbolic model checker developed by Ken McMillan at Carnegie Mellon University.

The primary features of RuleBase are the following:

- Several hardware description languages are supported, including VHDL, Verilog**, and DSL (an IBM internal hardware description language).
- RuleBase is integrated in a variety of design environments and is easily integrable into others.
- Sugar, the RuleBase specification language, provides a way for hardware designers who are not CTL experts to read and write specifications easily.
- Various methods address model size problems:
  1. New, efficient model-checking algorithms.
  2. Enhancements of the original SMV algorithms.
  3. Automatic design reductions which leave only parts that influence formula correctness.
  4. New dynamic binary decision diagram (BDD) ordering.
- Debugging aids support result analysis and process analysis:
  1. Counterexamples are presented as simulation-like timing diagrams.
  2. Counterexamples can be translated into simulation control programs.
  3. Formulas which are trivially correct are detected.
- A graphical user interface allows convenient control over the formal verification process. The user interface facilitates user intervention in the process, while allowing a fully automated verification when the user chooses not to intervene.

RuleBase has been used at various locations within IBM. The list of hardware units successfully verified with RuleBase includes bus bridges, cache controllers, bus interface units, and more. Additionally, RuleBase has been used to formally verify hardware at the architectural level, specifically verification of cache coherence protocols.

An example

Figure 9 shows a functional unit of the memory bus adapter [11]. The switch connects six SPL units to a speed-matching buffer which contains three buffer types: commands, store data, and fetch data. Correspondingly, there are independent processes to transfer commands, store data, and fetch data in the switch. In the following we focus on the arbitration part of the command transfer process. The command buffer has room for eight different commands, which are associated with buffer slots. The availability of a slot is signaled with an `Slt_Avl_x` signal. An SPL requests a command transfer with a `Req` signal and provides a `Req_Vec`, which indicates the command slot it wishes to use. At the end of arbitration two things happen:

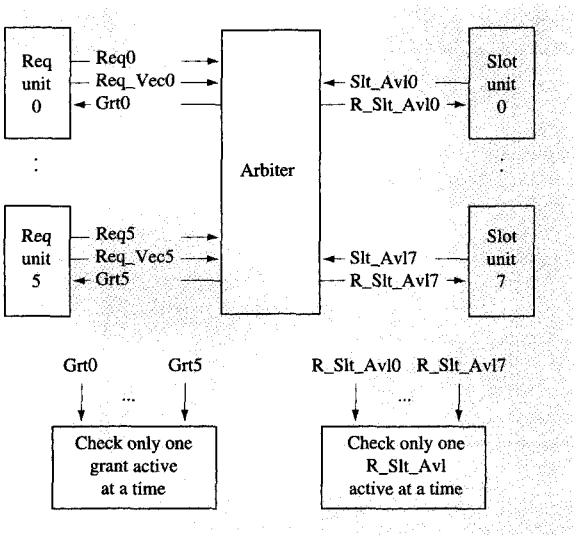

Figure 10

Module test model of the command arbiter.

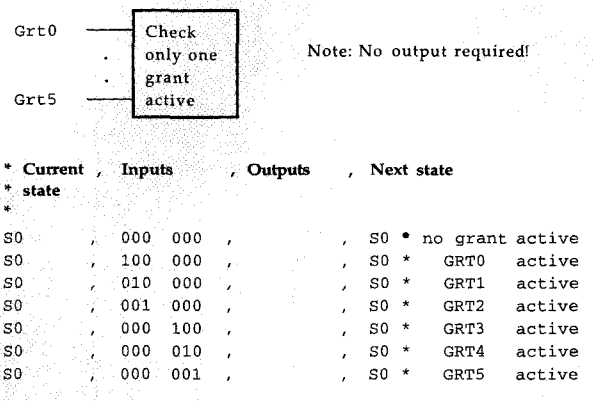

Figure 11

One-grant-active protocol checker.

1. The unit which won the arbitration receives a grant.
2. The `Slt_Avl` bit for the target selected is reset via the `R_Slt_Avl` signal.

If we wish to verify the command transfer process, we need a model of the dotted area in Figure 9. We can abstract most of the behavior of the command buffer and the SPL, and build a model which contains only the relevant behavior for the unit to test, as shown in **Figure 10** for the unit of the command transfer arbiter. Every SPL is replaced by a request unit, which nondeterministically generates requests and `Req_Vec` values and eventually receives a grant from the arbiter. The different slots of the command buffer are represented by eight slot units, which provide the `Slt_Avl` signals. In addition there are two protocol checkers, which check whether more than one grant signal or more than one `R_Slt_Avl` signal is active at a time. (Note that in reality there are more than two protocol checkers.) The principle of the request generator is shown in Figure 6. Note that every request unit will fall into the error state if it receives a grant without having issued a request.

Figure 11 shows a state table which checks that only one grant (or no grant) is active at a time. In this table only the legal conditions are shown; all other input conditions cause the FSM to transit to the error state. Using the built-in error states of the FSMs, it is easy to generate the temporal formulas which specify that no FSM should ever enter the error state. The corresponding formulas for our model are shown in Figure 8. With the

```
rule liveness_properties {
  formula { EF ( FSM_i.State=Sx      ) }
  formula { EF ( FSM_i.State=Sy      ) }
            .
            .
  formula { EF ( FSM_k.State=Sz      ) }
  formula { AG AF ( FSM_i.State=Idle ) }
            .
            .
  formula {AG ( Req ⟶ AF Grt ) }
}
```

Figure 12

Some liveness properties.

environment description and the safety formulas, a formal verification run could be started. After some time the designer becomes comfortable with the tool, gradually adding liveness formulas to the set of safety formulas which were automatically generated. In **Figure 12** an example is given. The first formula states that for some execution path in the future (`EF` = eventually future) the state of `FSM_i` will be some state named `Sx`. Obviously some equations (in addition to those for the safety formulas) are generic and could be generated automatically at the outset. Other formulas, however, are model-specific, such as the last formula, which states that for every request there must eventually be a grant.

Results

The primary concern we had was this: "Do we increase the total verification time by adding the module test to our process flow?" To evaluate this question, we compared the time required for a traditional functional unit simulation with that required for FV. The complete switch was verified with random simulation; part of the switch (the command transfer process) was formally verified. The command transfer process represents approximately 30% of the switch in terms of logic circuits and complexity. Since we did not use FV for the complete switch, we had to extrapolate from the command transfer process. The different cases are shown in **Table 4**. The first column shows the switch simulation, the formal verification of the command transfer process, and the extrapolation of FV to the complete switch. The second column contains the number of lines of code needed to describe the environment for simulation and for the different FV models. The last column shows the time spent (in person-months). There is an order of magnitude difference in lines of code, and the extrapolated FV value indicates that FV is similar to or better than random simulation in terms of time spent. Clearly it is superior in terms of test coverage. We did formal verification after the random functional unit simulation. No errors were found, and no error was found in the actual hardware.

The method presented here has been successfully applied to the design process of the MBA chip for the next S/390* generation. **Table 5** shows the number of detected design errors in a functional unit of the chip found by different verification techniques. FV detected 24% of the errors very early in the design flow. In this example simulation started very early, so the majority of errors were still found by simulation. However, there were errors that would also have been found by FV. But far more important is that the majority of the errors found by FV would not have been detected by simulation. **Table 6** shows that only 36% of the errors found by FV could have been detected by simulation without much effort; 24% were rather complex cases that would have been detected eventually, and 40% of the failed FV cases would probably have slipped through simulation as well.

Table 4 Comparison of FV with random simulation.

Case	*Number of lines of code*	*Time spent* (person-months)
Simulation of complete switch	2586	6
FV of command transfer process		1.25
Model 1	30	
Model 2	34	
Model 3	142	
Model 4	229	
Model 5	49	
Model 6	84	
FV of switch (estimated)		4

Table 5 Design errors detected with verification technique.

Verification technique	*Errors found* (%)
Functional unit simulation	41
Formal verification	24
Visual code inspection	20
Chip simulation	15

Table 6 Classification of errors found with FV.

Error class	*FV errors* (%)
Not detectable by simulation	40
Possibly detectable by simulation	24
Easily detectable by simulation	36

Summary

We have developed an approach to integrate formal verification in our design and verification process which has the following advantages:

1. It relies on the familiar and intuitive notation of finite-state machines; therefore, no learning curve is required.
2. It allows us to switch back to simulation within an hour if FV fails because of model size problems.
3. The protocol checkers developed for FV can be used in the chip and system integration tests, which increases confidence in the protocol checkers. In addition, they help to find and isolate errors during integration testing.

Because of the driving forces we have described, we expect that model-checking tools will be accepted by the industry in the very near future. We plan to use FV on a large scale in our next project. Some designers today are enthusiastic when a system simulation model runs successfully for millions of cycles. However, with the large state spaces required for heavily queued systems, this may be not much more important than some millions of water molecules in an ocean. How does one convince oneself and others to introduce FV? The experience of very extensive debugging is certainly quite convincing, because heavily queued systems also generate more errors than their predecessors.

Outlook

From a practical point of view, the following areas deserve more investigation:

- We use message sequence charts [12] as a documentation aid to describe the behavior between components of a module. It would be very useful to compile these charts directly into CTL formulas.
- Most of the FV models which had a state-space explosion problem were characterized by very large structural symmetries. These symmetries induce an equivalence relation between states, which permits a dramatic reduction of state space [13, 14]. Ongoing research in this area seems to be very promising.
- Most designers find it easier to specify rules for the behavior of the dataflow than rules for the control logic. A major goal would be to generate automatically an abstracted model of the dataflow; such a model should be behaviorally equivalent to the real dataflow from a control logic point of view. This would allow the control logic to be verified at the same time as the dataflow.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Cadence Design Systems, Inc.

References

1. M. Yoeli, *Formal Verification of Hardware Design*, IEEE Computer Society Press, Los Alamos, NM, 1990.
2. K. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, Boston, 1993.
3. J. Burch, E. Clarke, K. McMillan, and D. Dill, "Symbolic Model Checking: 10^{20} States and Beyond," *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, June 1990, pp. 428–439.
4. J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill, "Symbolic Model Checking for Sequential Circuit Verification," *IEEE Trans. Computer Aided Design* **13,** No. 4, 401–424 (April 1994).
5. I. Beer, S. Ben-David, D. Geist, R. Gewirtzman, and M. Yoeli, "Methodology and System for Practical Formal Verification of Reactive Hardware," *Lecture Notes in Computer Science* **818,** 182–193 (1994).
6. D. Geist and I. Beer, "Efficient Model Checking by Automated Ordering of Transition Relation Partitions," *Lecture Notes in Computer Science* **818,** 299–310 (1994).
7. E. Clarke and E. Emerson, "Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic," *Lecture Notes in Computer Science* **31,** 52–71 (1981).
8. John Hopcroft and Jeffrey Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley Publishing Co., Reading, MA, 1979.
9. I. Beer, S. Ben-David, C. Eisner, and A. Landver, "RuleBase: An Industry-Oriented Formal Verification Tool," *Proceedings of the 33rd Design Automation Conference*, Las Vegas, 1996, pp. 655–660.
10. *RuleBase Formal Verification Tool: User's Manual*, IBM Science and Technology, Haifa Research Laboratory, Haifa, Israel 31905.
11. G. Doettling, K. J. Getzlaff, B. Leppla, W. Lipponer, T. Pflueger, T. Schlipf, D. Schmunkamp, and U. Wille, "S/390 Parallel Enterprise Server Generation 3: A Balanced System and Cache Structure," *IBM J. Res. Develop.* **41,** No. 4/5, 405–428 (1997, this issue).
12. K. Turner, Ed., *Using Formal Description Techniques*, John Wiley & Sons, Inc., New York, 1993.
13. E. Clarke, R. Enders, and T. Filkorn, "Exploiting Symmetry in Temporal Logic Model Checking," *Formal Methods in System Design* **9,** 77–104 (1996).
14. C. Ip and D. Dill, "Better Verification Through Symmetry," *Formal Methods in System Design* **9,** 41–75 (1996).

Received December 12, 1996; accepted for publication July 10, 1997

Thomas Schlipf *IBM Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (SCHLIPF at BOEVM3).* Mr. Schlipf studied electrical engineering at the University of Karlsruhe. In 1985, after working for a time at the Robert Bosch Company, he joined the IBM S/390 Development Laboratory in Boeblingen. Since then he has been working on the hardware design of I/O chips and now leads the MBA team. Mr. Schlipf's interests are in the areas of computer architecture and formal verification. He is a member of the IEEE.

Thomas Buechner *IBM Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (BUECHNER at BOEVM4).* Dr. Buechner received his M.S.E.E. degree from the University of Karlsruhe, Germany, in 1988, and the Ph.D. degree in computer science from the University of Stuttgart, Germany, in 1996. In 1993 he joined the IBM development laboratories in Boeblingen, Germany, where he is currently responsible for logic design and verification of complex and high-performance ICs for the IBM S/390 CMOS Parallel Enterprise Servers. His research interests include VLSI testing, formal verification, and computer-aided design methods. From 1989 to 1993 he worked as an R&D engineer at the Custom Processors and Test Department of the Institute for Microelectronics Stuttgart, Germany, a federally funded ASIC design/fabrication/test center. During this time he was responsible for the design of several custom processors, and he did research on cost-effective test methods for semicustom ASICs. Dr. Buechner has been a member of the IEEE since 1989; he served as a technical committee member and session chair of the IEEE International ASIC Conference in 1996 and 1997.

Rolf Fritz *IBM Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (FRITZR at BOEVM3).* Mr. Fritz studied electrical engineering at the Fachhochschule Offenburg. In 1981, after four years with Wandel u. Goldermann developing measurement sets for telecommunications, he joined the IBM S/390 development laboratories in Boeblingen. Starting in a firmware department, he developed code for recovery and error reporting for various S/390 systems. In 1993 he joined the MBA hardware team, where he has been responsible for the design and in part for the verification of the sense and control logic.

Markus Helms *IBM Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (HELMS at BOEVM4).* Mr. Helms studied electrical engineering at the Berufsakademie Stuttgart, with emphasis on telecommunications. His studies were accompanied by practical training at IBM in Sindelfingen. In 1993 he joined the IBM S/390 development laboratories in Boeblingen, where he was responsible for design verification and simulation of I/O adaptor chips. His current job is design and verification of the MBA for the next two CMOS generations.

Juergen Koehl *IBM Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (KOEHL at BOEVM4).* Dr. Koehl is an Advisory Engineer in the VLSI Design Centre. He received his Ph.D. in mathematics from Bonn University in 1987; his Ph.D. thesis was on algebraic cycles on Hilbert modular surfaces. From 1987 to 1989 he was a Research Staff Member at the Institute for Discrete Mathematics in combinatorial optimization for VLSI design. He joined IBM in 1989 and is responsible for physical design methodology. Dr. Koehl is Chairman of the ITG/GI/GMM Fachgruppe "Layout" and a senior member of the IEEE.