

Integrating Formal Verification into Computer Organization and Architecture Courses

Lubomir Ivanov

Department of Computer Science

Iona College

New Rochelle, NY 10801

livanov@iona.edu

ABSTRACT

The high complexity of modern hardware and software systems necessitates the use of formal methods for checking the satisfaction of desired properties and the absence of design flaws. Numerous methods have been developed, and some, such as model checking and the ω -automata approach, have found wide acceptance in the computer industry and have led to the development of powerful verification tools. However, the popularity of these methods has not been firmly established in the Computer Science and Engineering curriculum. This paper presents an approach to integrating current verification research results into a typical, small-college Computer Organization and Architecture course.

OVERVIEW OF FORMAL VERIFICATION

The ever-increasing complexity of hardware and software systems and the protocols which govern their behavior has significantly increased the potential for design errors, and, at the same time, limited the usefulness of the classical simulation and testing methods for uncovering design faults. A powerful alternative, which has gained significant popularity, is formal verification. It aims at establishing the correctness of a design by proving facts about a mathematical model of the system under consideration. The proofs are general enough to guarantee that the property being considered is satisfied in all possible cases. An excellent overview of the field of formal verification can be found in [1]. Several different formal verification approaches have emerged:

- In *Automatic Theorem Proving* the model of the system and its specifications are expressed in a given logic. The verification question is specified as a theorem in the logic, and then a formal proof of this theorem is constructed to show that the implementation meets the specifications. The advantage of this method is its power and flexibility. The disadvantage is that the method is not easy to automate. To construct a proof, the designer needs to be very proficient in the specific logic. Moreover, the proofs are often very complex and require a significant level of mathematical skills on behalf of the designer. An example of a well-known theorem prover is HOL [2].
- *Model Checking* expresses system properties as temporal logic formulas, but instead of proving the validity of the logic formula for all possible models, it studies the truth or falsity of a formula with respect to only one specific finite model. This significantly simplifies the verification task, and allows the verification process to be automated. Model checking as a formal verification technique, was invented by Clarke and Emerson [3]. An improved version of the Emerson-Clarke model checking algorithm, referred to as *Symbolic Model Checking*, was introduced by K.McMillan [1]. A popular tool, based on symbolic

model checking, is SMV developed at Carnegie-Mellon University. The tool has been used extensively for verifying commercial level designs including synchronous & asynchronous sequential systems, distributed cache protocols, etc.

- *Equivalence Checking* is a way of verifying that two descriptions of a design are equivalent. A well-known tool, based on equivalence checking with Ordered Binary Decision Diagrams is Verity, developed at IBM [4]. Verity can be used to compare descriptions at transistor-, gate-, and register-transfer language levels.
- Another powerful approach to formal verification is based on the theory of ω -automata [5]. Given a model of the system in terms of an automaton, M, and a property to be verified given as an automaton, P, verifying the satisfaction of the property involves checking for the language containment $\text{Lang}(M) \subset \text{Lang}(P)$ or, equivalently, $\text{Lang}(M) \cap \text{Lang}(P) = \text{Lang}(M)$. Since checking for language containment directly may be very computationally expensive due to the size of the involved automata, a rich methodology has been developed, which allows the decomposition of the models and reduction of the complexity of the language containment test. R.Kurshan's team at Lucent Technologies has implemented their ω -automata methodology into an industrial-level software called FormalCheck. It not only provides accurate testing of various system properties, but counter-examples as well to indicate how exactly the property under verification fails.

Model Checking and several other methods are plagued by the “state explosion” problem – the fact that explicitly representing the system model requires an exponential number of states in the number of system variables. To avoid “state explosion” and other related issues, a number of new verification methods have emerged, which, while relatively less powerful and general, guarantee a significantly improved efficiency. Among these, several methods have been based on using partial orders to describe the dependence or independence of sets of events occurring in a hardware system [6, 7, 8, 9, 10]. The main appeal of using partial orders in modeling and verifying system behavior is in avoiding the study of all possible interleavings of events occurring during a run of the system. In addition, partial order models are usually very clear and intuitive, and the algorithms can be fully automated. In some cases, partial order methods have been shown to reduce the complexity of verifying properties of asynchronous circuits from exponential to polynomial.

In [11, 12, 13] we introduced a new partial-order formal verification method for proving timing properties of complex systems. The method is based on the inductively defined notion of series-parallel posets. In [14, 15] we demonstrated the modeling and verification capabilities of our methodology by verifying the properties of a handshaking communication protocol, the popular PCI local bus protocol, and the MESI snoopy cache coherence protocol used in many modern parallel computer systems as well as in the popular Intel Pentium processor series. Current work involves using the developed methodology and software in the verification of microcode of the Transputer microprocessor, and the behavior of Dataflow parallel computer systems.

INTEGRATING FORMAL VERIFICATION INTO THE UNDERGRADUATE CURRICULUM

The computer industry has already shown considerable interest in formal verification. A number of first generation tools developed at Lucent Technologies and Chrysalis (now part of Avant!), as well as proprietary tools developed at IBM and Motorola are now in use. Companies such as Cisco, Intel, IBM, Motorola and many others have included formal verification as part of their design flow. While the demand is not yet in the hundreds, companies are facing staffing challenges, especially when looking to hire "formal methods literate" engineers in the work place. The research effort, however, has had limited impact in the classroom. The issues of design correctness and system verification are rarely discussed, and considered almost impractical. Few courses familiarize their students with the available design- and verification tools and methodologies. Moreover, whenever the issues of formal correctness are discussed they are presented in isolation, without the essential emphasis on the continuity of the design flow process, of which formal verification is an essential component. Our goal is to integrate research results from the field of formal verification of hardware and software systems into several key courses of the undergraduate Computer Science curriculum. This will increase the students' awareness of the necessity and complexity of the verification task, and provide them with essential practical skills in using CAD and verification software packages for designing, testing, and debugging large-scale systems.

Introducing students to formal verification can be a gradual and gentle process. The very first Computer Science course - Introduction to Programming - offers excellent opportunities for familiarizing the students with the issues of (formal) program correctness as an essential and integral part of the overall software design flow. The level of mathematical sophistication need not be very high. Formal reasoning about program correctness can be introduced through pre- and post-conditions, loop invariants, etc. A number of textbooks have already adopted this approach [16, 17, 18].

The more advanced and powerful formal verification techniques usually require a higher degree of mathematical sophistication on behalf of the students, but that can be gradually developed based on the knowledge acquired in a typical 2nd/3rd semester Discrete Mathematics course. Many courses such as Data Structures and Algorithms, Digital Design, Computer Organization and Architecture, Operating Systems, and Compilers, require Discrete Math as a prerequisite, and, therefore, are a natural place to incorporate formal verification ideas with the regular curricular material. From personal experience, this can be done naturally, efficiently, and with a minimal loss of time and almost no tradeoff in course material. The next section presents a concrete example of how such integration can be accomplished in a typical undergraduate Computer Organization and Architecture course, and upcoming papers will address the issue with regards to Data Structures and Algorithms, and Operating Systems courses.

A CONCRETE EXAMPLE: INTEGRATING FORMAL VERIFICATION INTO AN UNDERGRADUATE COMPUTER ORGANIZATION COURSE

As a concrete example, let us consider integrating some aspects of formal verification into a traditional, undergraduate Computer Organization and Architecture course at a small college. The usual course outline of such a course includes:

- Review of number systems and basic circuits
- Organization of a general CPU
- The fetch-decode-execute instruction cycle, macro-, and micro-instructions, etc.
- ALU Design
- Microprogrammed- & Hardwired Control Unit Design
- Basic notions of pipelining and superscaling
- RISC vs. CISC architectures
- The memory hierarchy
- Cache memory design
- Main memory organization
- A bit on Virtual Memory and hardware support for it
- The Input/Output subsystem
- Parallel Ports. Handshaking Protocols.
- Serial Communication and Protocols
- The system bus. Bus protocols.

The very first lecture offers an opportunity to introduce formal verification as an integral part of the design flow for a new CPU. The need for formal verification may be illustrated with examples of design flaws in real microprocessors (like the Pentium bugs), and other examples, in which formal verification has made a difference. The discussion at this point need not be very detailed. It is important that the students gain an appreciation for the need and the power of formal methods.

The discussion of microprogramming presents the next excellent opportunity to consider formal verification and its importance. To root the discussion in reality, the example of the "halting bug" of the Pentium processors may be cited (the execution of a particular sequence of instructions may halt the machine unexpectedly). At this point, a suitable formal verification formalism must be introduced - model checking, ω -automata, etc. The formalism should be powerful enough to handle the issues of micro-instruction sequencing as well as the following topics of cache design and system bus protocols. The availability of an automatic software tool based on the selected formalism will help reinforce the practical aspect of formal verification and provide hands-on experience for the students. In our case, we chose to introduce students to formal verification with series-parallel posets.

To illustrate the notion of a series-parallel poset, consider the following circuit:

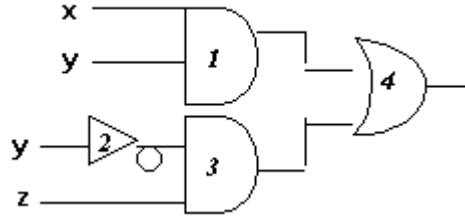


Fig.1 A Simple Combinational Circuit

Let us denote the event “gate i produces a valid output” by e_i . Then the behavior of the system in Fig.1 due to a change in the inputs is given by the following expression:

$$\mathbf{B} = (e_1 \otimes (e_2 \bullet e_3)) \bullet e_4$$

The expression describes succinctly the relationship among events occurring in the system. The output of gate #1 does not depend on the computation of any other gate, but the output of gate #3 depends on that of gate #2. The output of gate #4 depends both gate #1 and gate #3. In general, when two (partially ordered sets of) events are independent of each other, the expressions describing their internal dependencies are combined by the shuffle operation (\otimes). If a (set of) event(s) precedes the occurrence of another (set of) event(s), the expressions of the two are combined by concatenation (\bullet). Besides behaviors, one can express system properties, which we are attempting to verify in the same series-parallel poset format. For example:

Property 1: "The output of gate #1 becomes valid before that of gate #2"

$$\mathbf{P}_1 = e_1 \bullet e_2$$

Property 2: "The output of gate #4 becomes valid before that of gate #2"

$$\mathbf{P}_2 = e_4 \bullet e_2$$

To show the correctness or incorrectness of a specific property, two predicates are used - Sometimes Satisfied (SS), and Always Satisfied (AS). For example *Property 1* is sometimes satisfied within the behavior \mathbf{B} (i.e. $SS(\mathbf{B}, \mathbf{P}_1) = \text{TRUE}$) because the events e_1 and e_2 are independent and can, therefore, occur in any order. However, it is not true that the property is always satisfied, i.e. $AS(\mathbf{B}, \mathbf{P}_1) = \text{FALSE}$. Similarly $SS(\mathbf{B}, \mathbf{P}_2) = \text{FALSE}$, and $AS(\mathbf{B}, \mathbf{P}_2) = \text{FALSE}$. The above verification predicates (and a few more) have been implemented in a software package, which automates the verification process and provides the user with a convenient GUI interface. With the help of the software, it is easy to illustrate (at least partially) the microcode verification for real-world microprocessors.

Further on in the course, the topic of cache coherence is usually discussed. A typical example is the Modified/Exclusive/Shared/Invalid (MESI) snoop cache coherence protocol presented in most Computer Architecture textbooks [19]. The MESI protocol is usually described either as a state diagram indicating state transitions for the different cache controllers in a distributed processor environment, or in the context of the events occurring as a result of read/write actions taken by the various CPUs in the system, which affect the state of the cache controllers. Each of the cache controllers repeatedly engages in one of the several transactions, independently of the other cache controllers in the system. For the sake of illustration we present the series-parallel poset expression modeling the behavior of a system of n independent distributed cache controllers. The behavior of a single cache controller is modeled by the following series-parallel poset expression:

$B_{cache_controller_i} = (I + B_{read_hit_i} + B_{read_miss_i} + B_{write_hit_i} + B_{write_miss_i})^*$, where:

$$B_{read_hit_i} = cr_i$$

$$B_{read_miss_i} = send_snoop_rq_i \bullet (\otimes_{j=1..n, j \neq i} rcv_snoop_rq_j) \bullet ((\otimes_{k \in SH} send_sh_k) \bullet rcv_sh_i \bullet mr_i \bullet S + send_sh_E \bullet (S_E \otimes (rcv_sh_i \bullet mr_i) \bullet S_i) + send_stop_M \bullet send_cl_M \bullet (S_M \otimes (rcv_stop_i \bullet rcv_cl_i) \bullet S_i)) + mr_i \bullet E_i$$

$$B_{write_hit_i} = ((bcast_I_i \bullet (\otimes_{k \in SH} (rcv_bcast_I_k \bullet I_k))) + 1) \bullet cw_i \bullet (M_i + 1)$$

$$B_{write_miss_i} = send_rwitm_i \bullet (\otimes_{j=1..n, j \neq i} rcv_rwitm_j) \bullet ((I_E + \otimes_{k \in SH} I_k) \bullet mr \bullet cw_i \bullet M_i + send_stop_M \bullet mw_M \bullet (I_M \otimes (rcv_stop_i \bullet mr_i \bullet cw_i) \bullet M_i))$$

Note: The set SH includes all caches holding the cache line in a *shared* state. The subscripts E and M indicate that the signals are sent by a cache, holding the line in *exclusive* or *modified* state

Signal	Interpretation
cr	Cache Read
$send_snoop_rq$	Send snoop request
rcv_sh	Receive signal that the cache line is shared
mr	Memory read (followed by a cache line fill)
S	Change state to "Shared"
rcv_stop	Received stop signal from another cache controller
rcv_cl	Receive a cache line directly from another cache controller
E	Change state to "Exclusive"
$bcast_I$	Broadcast an "Invalidate" signal to all other cache controllers
cw	Cache write
M	Change state to "Modified"
$send_rwitm$	Issue a "Read with Intent to Modify" signal
Rcv_snoop_rq	Receive snoop request
$send_sh$	Send signal that the cache line is shared
$send_stop$	Send a stop signal to stop the main memory read of another cache controller
$send_cl$	Send a cache line directly to another cache controller
rcv_I	Receive an "Invalidate" signal
rcv_rwitm	Receive "Read with Intent to Modify" signal
mw	Main memory write

The behavior of a system of n cache controllers is given by:

$$\mathbf{B} = \mathbf{B}_{cache_controller_1} \otimes \mathbf{B}_{cache_controller_2} \otimes \dots \otimes \mathbf{B}_{cache_controller_n} = \otimes_{i=1..n} \mathbf{B}_{cache_controller_i}$$

The details of the actual verification appear in [15]. In class, the above model was presented, explained, and a few properties were verified with the help of the verification software.

The last topic of a typical Computer Organization and Architecture course is the discussion of system buses. Invariably, the discussion centers on the example of the popular PCI bus protocol used in most Pentium-based PCs. The *Peripheral Component Interconnect* (PCI) bus protocol is a processor-independent bus protocol, which allows 32-bit or 64-bit high-speed data transfers between devices. PCI was announced by Intel in 1990, and has since become a virtual standard among computer manufacturing companies. A *PCI bus transaction* involves a *master* (initiator) and a *slave* (target) device. There are several types of transactions available such as *configuration reads and writes*, *memory/IO reads and writes*, *interrupt acknowledge*, and *special cycles*. For each type of transaction, the sequence of events occurring during the transaction can be expressed as a series-parallel poset expression. The overall *behavior of the PCI bus* can then be specified as follows:

$$\mathbf{B}_{PCI} = (\mathbf{B}_{IO_Read} + \mathbf{B}_{IO_Write} + \mathbf{B}_{Conf_Read} + \mathbf{B}_{Conf_Write} + \mathbf{B}_{Mem_Read} + \mathbf{B}_{Mem_Read_Line} + \mathbf{B}_{Mem_Read_Multiple} + \mathbf{B}_{Mem_Write} + \mathbf{B}_{Mem_Write_Invalidate} + \mathbf{B}_{Intr_Ack} + \mathbf{B}_{Special_Cycle})^*$$

In the above expression, each \mathbf{B}_i represents a particular type of PCI bus transaction. Using the verification software, it was easy to illustrate the verification of a few PCI properties, and the presence of two bugs in the PCI local bus protocol specification (which have been independently verified in [20]).

The introduction of the above material into the course curriculum of CS311 Computer Organization and Programming at Iona College, NY required very little change in the presentation of the regular material. One of the changes was the elimination of the detailed discussion of control hazard resolution techniques such as Branch Prediction with a History Table. Significant as this topic may be, its removal was outweighed by the benefits of introducing aspects of formal verification into the course. Moreover, the topic of pipelining and hazard resolution is discussed in greater depth in the subsequent CS411 Computer Architecture course.

Another change necessitated by the introduction of the formal verification material was the shortening of the presentation on Virtual Memory. Only a brief outline of the concept of Virtual Memory was presented, with the bulk of details left for the subsequent Operating Systems class. This not only helps introduce the issues of formal modeling and verification into the Computer Organization syllabus, but also emphasizes the continuity among courses.

One final change was the elimination of the review lecture on analysis and design of general sequential circuits in the beginning of the course. The material was previously presented mostly for completeness of the discussion, and was hardly ever used in the remainder of the class. Therefore, its elimination was not a major issue.

CONCLUSIONS

In this paper we considered the issue of integrating formal verification into the undergraduate Computer Science curriculum, and, more specifically, into the syllabus of a Computer Organization and Architecture course. This can be accomplished gradually, with a minimal loss of time and little change to the breadth and depth of the material normally presented in the course. The advantages, however, are significant. Students will develop an understanding, appreciation, and working knowledge of the overall design process, and gain hands-on experience in using real-world design and verification software. The experience will reinforce the close relationship and interdependence among the various Computer Science disciplines, and the bond between theory and applications. Overall, students will be better prepared for the demands of working in the industry and academia of tomorrow.

REFERENCES

- [1] K. McMillan, "*Symbolic Model Checking*", Kluwer Academic Publishing, 1993
- [2] M.J.C.Gordon and T.F.Melham, "*Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*", Cambridge University Press, 1993
- [3] E.M.Clarke and E.A.Emerson, "*Synthesis of Synchronization Skeletons for Branching Time Temporal Logic*", Logic of Programs Workshop, vol.131 of LNCS, Springer-Verlag, 1981
- [4] A.Kuehlmann, A.Srinivasan, and D.P.LaPotin, "*Verity - A Formal Verification Program for Custom CMOS Circuits*", IBM Journal of Research and Development, v.39 Jan./March 1995
- [5] R.Kurshan, "*Computer Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*", Princeton Series in CS, 1994
- [6] M.Nielsen, G.Plotkin, and G.Winskel, "*Petri Nets, Event Structures, and Domains*", part I, Theoretical Computer Science, 13:85-108, 1981
- [7] V.Pratt, "*Modeling Concurrency with Partial Orders*", International Journal of Parallel Programming, 15, 1, c. Nov.1986
- [8] P. Godefroid, "*Partial Order Methods for the Verification of Concurrent Systems: an Approach to the State Explosion Problem*", Doctoral Dissertation, University of Liege, 1995
- [9] R. Nalumasu, G. Gopalakrishnan, "*A New Partial Order Reduction Algorithm for Concurrent System Verification*", IFIP, 1996
- [10] D. Peled, "*Combining Partial Order Reductions with On-the-Fly Model Checking*", Journal of Formal Methods in Systems Design, 8 (1), 1996
- [11] L.Ivanov, R.Nunna, S.Bloom, "*Modeling and Analysis of Non-Iterated Systems: An Approach Based on Series-Parallel Posets*", Proceedings of ISCAS'99, Orlando, FL, 1999
- [12] L.Ivanov, R.Nunna, "*Formal Verification with Series-Parallel Posets of Globally-Iterated Locally-Non-Iterated Systems*", Proceedings of the MWSCAS'99, Las Cruces, NM, 1999
- [13] L.Ivanov, R.Nunna, "*Formal Verification: A New Partial Order Approach*", Proceedings of ASIC/SOC'99, Washington DC, 1999
- [14] L. Ivanov, R. Nunna, "*Modeling and Verification of an Interconnect Bus Protocol*", Proceedings of IEEE MWSCAS'2000, Lansing, MI, 2000
- [15] L.Ivanov, R.Nunna, "*Modeling and Verification of Cache Coherence Protocols*", Proceedings of IEEE ISCAS'2001, Sydney, Australia, 2001
- [16] R.Kruse, A.Ryba, "*Data Structures and Program Design in C++*", Prentice Hall, 1999
- [17] W.Savitch, "*Problem Solving with C++: The Object of Programming*", AWL1996
- [18] M.Main, W.Savitch, "*Data Structures and Other Objects Using C++*", AWL, 1997
- [19] W. Stallings, "*Computer Organization and Architecture*", 5th Edition, Prentice Hall, 1996
- [20] D. Wang (with E. Clarke), "*Formal Verification of the PCI Local Bus: A Step Towards IP Core Based Systems-On-Chip Design Verification*", Master's Thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, May 1999