

Experience with Applying Formal Methods to Protocol Specification and System Architecture

Mani Azimi, Ching-Tsun Chou*, Akhilesh Kumar, Victor W. Lee,
Phanindra K. Mannava and Seungjoon Park
Intel Corporation, SC12-608, 3600 Juliette Lane, Santa Clara, CA 95054

September 2002

Abstract. In the last three years or so we at Enterprise Platforms Group at Intel Corporation have been applying formal methods to various problems that arose during the process of defining platform architectures for Intel's processor families. In this paper we give an overview of some of the problems we have worked on, the results we have obtained, and the lessons we have learned. The last topic is addressed mainly from the perspective of platform architects.

1. Problems and Results

Modern computer systems are highly complex distributed systems with many interacting components. Architecturally they are often organized like a computer network into multiple *layers*: physical layer, link layer, protocol layer, etc. Most of the problems to which we applied formal methods are the formal verification (FV) of intricate protocols in the protocol and link layers. In addition, we also found several novel uses of binary decision diagrams (BDDs) [3] that are worth mentioning.

1.1. DIRECTORY-BASED CACHE COHERENCE PROTOCOLS

A significant portion of our work centers around the formal modeling and verification of directory-based cache coherence protocols in Scalability Port (SP), which is a family of scalable distributed shared memory architectures based on high-speed point-to-point interconnect technologies and whose first incarnation was implemented in Intel's 870 chipset [2]. Directory-based cache coherence protocols are complex distributed algorithms that must operate correctly in the face of asynchrony, unpredictable message delays, and multiple paths between agents. Due to the astronomical number of possible executions, neither unaided human reasoning nor traditional simulation-based validation can be relied upon to flush out all bugs. Only exhaustive state space exploration offered by FV techniques can produce a high degree of confidence in the correctness of a protocol specification.

* To whom correspondences should be addressed: ching-tsun.chou@intel.com



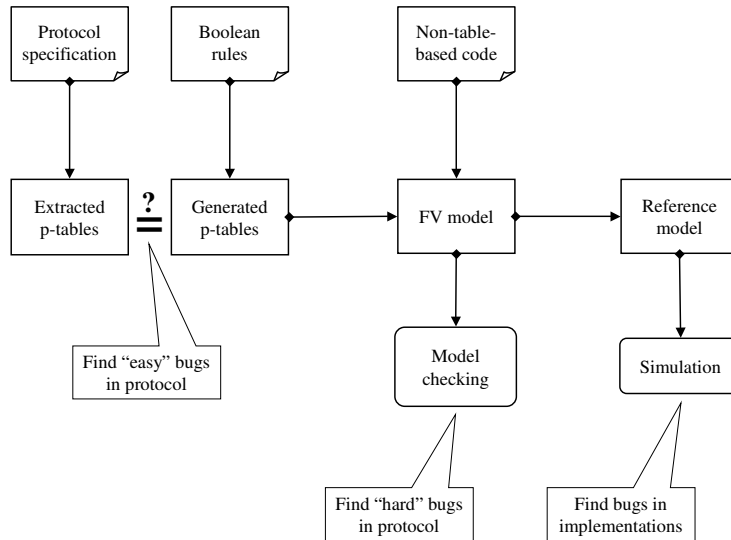


Figure 1. Verification flow of SP cache coherence protocols

Our verification flow is shown in Fig.1. The core logic of the cache coherence protocol is specified by a set of *protocol tables*, or *p-tables* for short. There are two reasons for adopting a tabular representation. First, tables are more structured and precise than texts or pictures and are relatively easy to read by both humans and machines. Second, “easy” errors in tables can be caught by “light-weight FV” based on *Boolean rules*, which will be explained in Sec.1.3.1. After the p-tables are checked by Boolean rules, they are translated by a simple compiler into code fragments that are included into an FV model. The translation guarantees that the p-tables and the FV model assume the same granularity of atomic actions. The FV model is then augmented with assertions expressing desired properties and exhaustively analyzed by symbolic model checking [8]. Finally, the FV model is translated by another, rather nontrivial, compiler into an executable reference model in C for use in simulation. For example, the reference model can be used as a checker of protocol rules that flags an error whenever a simulated microarchitectural or RTL model takes a step that is not allowed by the protocol specification.

There are two types of properties that we have found useful. The first is the standard *safety* properties: whenever the state of a cache is not Invalid, then its data content is up-to-date; no two caches can be in Modified or Exclusive state at the same time; and so on. But, while they are useful for checking that certain “bad things” can never

happen, safety properties don't guarantee that any "good things" will ever happen. So we also assert a set of *weak liveness* properties of the form: $\text{AG EF } (\text{cs} = \text{CS})$, which states that, from every reachable state, there exists an execution placing the "control state" cs of a protocol structure (e.g., the MESI state of a cache) in any one of its possible values CS (e.g., any of the MESI states). By systematically asserting weak liveness properties for all protocol structures and all their control states, we guard against deadlocks, missing cases in protocol tables, and other unexpected protocol scenarios.

Since the complexity of symbolic model checking is very sensitive to the number of state bits in a model, we use the technique of *refinement mappings* [9] to reduce that number. There are two main types of refinement mappings that we have found useful. First, during its lifetime, a transaction in a cache coherence protocol travels through multiple protocol structures, each of which has a "transaction type" field to track the type of the transaction currently occupying it. Most of these fields are redundant in the sense that their values can be derived from that of the one in the originating structure and hence can be eliminated by refinement mappings. Second, we introduce an auxiliary variable to track the most up-to-date data value of any address and express each state variable in the "data path" of the model in terms of the auxiliary variable and when the state variable is "valid". Note, however, that this type of refinement mappings would be less effective if the cache coherence protocol allows multiple "up-to-date" values at the same time for an address. Fortunately, SP is not like that.

Results: It is now routine for us to make changes to the protocol tables, update the Boolean rules to reflect the changes, check the agreement between tables and rules to flush out "easy" bugs, generate an FV model from the tables, apply model checking to the FV model to detect "hard" bugs, trace the bugs back to and fix them in the tables, and re-iterate the cycle. After the FV model is formally verified, a new executable reference model is generated at the click of a button. Generally speaking, complete (in the sense of covering all transaction types, of which there are more than 20) models with 1 home node and 2 caching nodes can be model-checked without any special effort. Our intuition is that, due to the nature of the protocol, this configuration covers "essentially all" scenarios. We are currently in the process of formalizing this intuition.

Not surprisingly, most bugs we found were introduced by major protocol changes (e.g., new transaction types, changes to conflict resolution mechanisms, etc). However, even relatively minor modifications could sometimes have consequences unforeseen by people who knew the protocol very well. All these confirm our earlier statement that, as far

as directory-based cache coherence protocols are concerned, unaided human reasoning is not to be trusted.

1.2. VARIATIONS OF SLIDING WINDOW PROTOCOLS

For electrical and physical design reasons, more and more high-speed interconnects in future computer systems will shift from shared-bus to point-to-point technologies. This is true both for processor interconnects (e.g., the simultaneous bidirectional signaling technology used in 870 [6]) and for I/O subsystems (e.g., PCI ExpressTM [1], which is an upcoming replacement of PCI bus). In the link layer of a point-to-point system, a variety of sliding window protocols (SWPs) perform error recovery, flow control, and other functions. But, due to various design constraints, SWPs in real systems often deviate from “textbook” SWPs in subtle ways. For example, to reduce pin-count overhead in 870, link-level error recovery is implemented using a SWP in which acknowledgements are signaled by tokens that are counted, rather than by sequence numbers. Our experience shows that such seemingly innocuous modifications can in fact be extremely tricky and hence call for formal verification.

Results: We have formally verified the link-level error recovery and flow control protocols in both 870 and PCI Express and have found very subtle problems in some of these protocols. Our work led to both changes in RTL design (for 870) and clarifications of specifications (for PCI Express).

1.3. NOVEL APPLICATIONS OF BINARY DECISION DIAGRAMS

Although BDDs are the real workhorse underlying such widely used FV technologies as symbolic model checking [8] and symbolic trajectory evaluation [11], the users of FV tools seldom manipulate BDDs directly. Yet we have found that BDDs can be used to attack some problems that are not traditionally considered in the FV literature. In addition to BDDs, another prerequisite for the work described below is a good scripting language (e.g., FL in the FORTE environment [7]) that supports easy manipulation of BDDs.

1.3.1. *Rule-based checking of tables*

As mentioned in Sec.1.1, we use “light-weight FV” based on Boolean rules to flush out “easy” bugs in long and complex tables in specification documents. Our methodology is based on the observation that the contents of a specification table are not random, but typically exhibit a great deal of regularities. It is possible to capture those regularities as *Boolean rules* (or, simply, *rules*) that express the expected relationships

among the entries in any row of the table. Our experience shows that, in fact, it is not hard to articulate sufficiently many rules to completely characterize a table in the sense that the set of rows that satisfies the constraints imposed by the rules is exactly the same as the set of rows in the manually constructed table in the specification document. Thus the correctness and completeness of the latter can be checked by comparing it against the former. Any disagreement between the two indicates an error in one or both.

The reader may ask: why not dispense with manual table construction altogether and generate tables directly from rules? The reason is that enumerating a table's rows explicitly and coding rules that implicitly define the rows are two fundamentally different but complementary activities. When one starts to construct a new table, it is far easier to capture one's intention by enumerating rows than by coding rules, because one has no clear idea about the general shape of the table at that point. As the table matures and grows in size, inspection becomes more tedious and less fruitful in finding bugs. Then it is time to switch to the complementary point of view, observe regularities in the table, and code rules to capture them. The table and the rules are checked against each other and eventually converge to the same set of rows. Ideally, the table and the rules should be coded by two different persons, in order to maximize the difference in points of view. Later, when the table undergoes modification (as specifications inevitably do over time), one does not rely on visual inspection to ensure that the internal consistency of the table is maintained. Rather, the consistency conditions have been captured by the rules, which can now be automatically compared against the new table to catch consistency violations.

Results: When rule-based table checking is applied to a set of tables for the first time, dozens of problems are typically found. Most of these problems are trivial, such as misspelled, misplaced, or otherwise incorrect entries. But some of the problems are more serious, such as missing cases and systematic mistakes across multiple tables. After several iterations, the tables and rules quickly converge to agree with each other. From then on the maintenance of tables and rules does not require much work, unless the change is major. In particular, we found that changing the rules to keep up with the tables rarely requires more work than changing the tables themselves.

1.3.2. *Search for minimal deadlock-free wormhole routing schemes*

There is a large body of research literature on deadlock-free wormhole routing schemes (e.g., see [5, 10]). However, when given a specific interconnect topology, general deadlock-free routing techniques found in the literature (e.g., dimension routing [5]) sometimes do not yield a minimal

routing scheme (i.e., one in which every path used by the scheme is a shortest path). Sometimes it may not even be obvious whether a minimal deadlock-free routing scheme exists at all. If the topology is not too large, BDD-based techniques can be used to exhaustively search for such schemes, as described below.

By definition, a minimal routing scheme uses only shortest paths between pairs of nodes in a topology. But there are only a finite number of shortest paths between any pair of nodes. Thus one can introduce sufficiently many Boolean variables, called *selector variables*, to index the set of shortest paths between all pairs of nodes, so that each truth assignment to the selector variables corresponds to a minimal routing scheme. A routing scheme induces a *channel dependency graph*, as follows. The vertices of the graph are the virtual channels available in the topology, which in the simplest case can be identified with directed links in the topology (i.e., only one virtual channel per direction per link). The edges of the graph represent the dependency relation between channels imposed by the paths used by the routing scheme: channel c depends on channel c' whenever the routing scheme uses a path that goes through c and then c' . It is not hard to see that the channel dependency graph can be represented by a BDD, among whose variables are the selector variables mentioned earlier. (More precisely, the BDD represents a family of channel dependency graphs, one per truth assignment to the selector variables, which chooses a minimal routing scheme.) It is shown in [5] that a routing scheme is deadlock-free if and only if the channel dependency graph it induces is acyclic. A graph is acyclic if and only if its transitive closure contains no self-loop. The transitive closure of a graph represented by a BDD can be computed using fixpoint iteration. Conclusions: (1) the problem of whether a topology permits a minimal deadlock-free routing scheme can be reduced to that of whether there is a truth assignment to the selector variables so that the transitive closure of the resulting channel dependency graph has no self-loop; (2) the latter problem is solvable by BDD manipulation (at least when the topology is not too large).

Results: We have applied the methodology outlined above to several topologies that came up during the interconnect architecture exploration process. In one of them, there are 64 pairs of nodes between each of which two alternative shortest paths exist (all other pairs of nodes have unique shortest paths between them), giving rise to 64 selector variables. After several hours of BDD computation, we found that none of the 2^{64} possible minimal routing schemes is deadlock-free. It is hard to imagine how such a result could be obtained without BDD.

Acknowledgements: We would like to thank Jay Jayasimha and Aniruddha Vaidya for valuable help on this problem.

1.3.3. Search for fault-tolerant link initialization sequences

During link initialization the two state machines at the two ends of a link must coordinate their transitions by exchanging *delimiters*. The precise bit patterns denoting these delimiters must be *fault-tolerant* in the sense that a small number of misrecognized bits cannot lead one delimiter to be confused with another delimiter, with the same delimiter but in a wrong time frame, or with the bits that are sent between delimiters. (Since one of the purposes of link initialization is to “train” the receiver circuitry, misrecognition of bits is more likely during initialization than after it.)

This delimiter selection problem has several parameters. Each of the d delimiters is D bits long. The detection process aims to tolerate any e -bit errors in any E -bit windows. The receiver may use a C -bit comparator ($C \geq D$) to detect the delimiters from the received bit stream. (Because the link operates at very high speeds, only simple comparator-based designs can be used.) For example, suppose that we have two 4-bit delimiters “0000” and “1111”. When they are transmitted within alternating bit streams, either “...010100000101 ...” or “...010111110101 ...” is received, where the delimiters are underlined for clarity. Any single-bit errors in 8-bit windows cannot cause a wrong detection when the receiver uses a 4-bit comparator. However, the same two delimiters cannot tolerate 2-bit errors in 8-bit windows using 4-bit comparators, because the first bit stream above may be corrupted to “...010101000001 ...” and lead to the “detection” of the right delimiter in the wrong time frame, or to “...011110000101 ...” and lead to the “detection” of the wrong delimiter, where the corrupted bits are slanted and the mis-detected delimiters underlined.

Results: For historical reasons, we wrote a C program that, given a set of parameters, exhaustively enumerates all possible sets of delimiters and tests each of them against all possible error patterns. We are currently in the process of re-coding so that we can perform the same task symbolically using BDDs, since exhaustive enumeration in C is running out of steam due to increasing parameter values. We include the problem here because its spirit is very similar to the problem in Sec.1.3.2 and our approach to it is clearly influenced by “formal methods thinking”.

2. Lessons Learned

Formal modeling steered us toward more precise and concrete protocol specifications than we would have written without it. Even a very abstract formal model requires one to spell out what exactly one means

by each protocol structure or action. The result is a specification that is somewhat less abstract, but much more unambiguous, than traditional architectural specifications. In particular, our protocol specification is really a high-level implementation of the protocol, where all main protocol structures are defined and their major actions identified. We found that such “high-level details” made it much easier to think through the consequences of our decisions. Indeed, in the early stages of defining SP cache coherence protocols, more bugs were found during formal modeling than by model checking.

Formal modeling also turned out to be an excellent way to help architects articulate their ideas. Sometimes the protocol being designed is not very complicated as an end product, but it can be quite tricky for the architect to capture his ideas unambiguously and assess their viability if he expresses them only in English. Some sort of high-level modeling is needed here to provide the “definiteness” that greatly helps the architect understand the nature of his problem. Presumably both formal and simulation models should satisfy this need equally well. But, for protocol design, a formal model in fact provides feedback faster than a simulation model does, because the exhaustive nature of formal verification removes the burden of generating tests and collecting coverage, which is nontrivial due to the distributed and asynchronous nature of protocols.

Formal verification gave us much higher confidence in the correctness of our protocol specifications than we would have had without it. It is our view that certain protocols, such as directory-based cache coherence protocols and certain variations of sliding window protocols, are simply too complex for unaided human reasoning alone to get correct. This is not to say that one cannot know such a protocol well enough to be able to reason about it productively (e.g., to tell what changes are needed when a new transaction type is added). But one should not have the illusion that unaided reasoning by itself is sufficient to foresee all possible corner cases.

Formal verification also made it less risky for us to modify protocol specifications. In an industrial environment, a protocol specification inevitably evolves over time, in response to customer requirements, design constraints, backward compatibility, and a host of other factors. The greatest risk when a protocol is modified is usually not the modification *per se*, but unexpected interactions between the modification and the rest of the protocol. We minimize that risk by applying formal methods to both the “syntax” (via rule-based table checking) and “semantics” (via model checking) of the protocol. Furthermore, by using automatic extraction and translation, we minimize the possibility of disagreements

between the specification document and the protocol tables and models that are actually analyzed by FV tools.

In conclusion, we believe that architectural definition affords a rich and fruitful area for the application of formal methods, for several reasons. First, architectural definition is at the right level of abstraction for formal methods to be applied to system-level verification in a meaningful manner; any system description with more lower-level details would overwhelm the limited capacity of currently available formal techniques and tools. Second, applying formal techniques at the definition stage makes architects aware of the verification issues and challenges and encourages them to select from a plethora of possible schemes one that is most amenable to complete verification. Third, a close relationship between architects and FV experts developed through their interactions also encourages a more meaningful selection and abstraction of issues to be scrutinized by formal methods.

Last but not least, since a typical implementation exercises only a subset of all allowed options in an architectural specification, applying formal methods to the specification enables the exploration of design spaces that are beyond the scope of any particular implementation. This aspect becomes even more important when system components are designed by different companies and hence close collaborations between component design and validation teams are impossible, as is often the case for Intel's architectural specifications. This becomes even more challenging when systems are expected to work across multiple generations of designs based on a single specification. Applying formal methods at the definition stage produces more robust and mature specifications. For instance, the initial version of PCI bus specification had a deadlock issue for peer-to-peer accesses [4], which was not discovered and documented until a later version of the specification and thus required revisions of existing PCI bridges to support the functionality correctly. We hope that by integrating the use of formal methods into the development of architectural specifications, we can minimize the risk of similar occurrences in the future.

References

1. PCI ExpressTM web site: www.pcisig.com/specifications/pci_express
2. F. Briggs, M. Cekleov, K. Creta, M. Khare, S. Kulick, A. Kumar, L.P. Looi, C. Natarajan, S. Radhakrishnan, and L. Rankin, "Intel 870: A Building Block for Cost-Effective, Scalable Servers", *IEEE Micro*, vol. 22, no. 2, pp. 36–47, March/April 2002.
3. R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. on Computers*, vol. C-35, no. 8, pp. 677–691, August 1986.

4. F. Corella, R. Shaw, and C. Zhang, “A formal proof of absence of deadlock for any acyclic network of PCI buses”, pp. 134–156 of *Hardware Description Languages and Their Applications*, Chapman & Hall, 1997.
5. W.J. Dally and C.L. Seitz, “Deadlock-Free Message Routing in Multiprocessor Interconnection Networks”, *IEEE Trans. on Computers*, vol. C-36, no. 5, May 1987.
6. M. Haycock and R. Mooney, “A 2.5GB/s Bidirectional Signaling Technology”, *Hot Interconnects V*, pp. 149–156, August 1997.
7. R.B. Jones, J.W. O’Leary, C.-J.H. Seger, M.D. Aagaard, and T.F. Melham, “Practical Formal Verification in Microprocessor Design”, *IEEE Design and Test of Computers*, vol. 18, no. 4, pp. 16–25, July/August 2001.
8. K.L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.
9. K.L. McMillan, “A Compositional Rule for Hardware Design Refinement”, *Computer Aided Verification (CAV’97)*, O. Grumberg (Ed.), LNCS 1254, pp 24–35, June 1997.
10. L.M. Ni and P.K. McKinley, “A Survey of Wormhole Routing Techniques in Direct Networks”, *IEEE Computer Magazine*, vol. 26, no. 2, pp. 62–76, 1993.
11. C.-J.H. Seger and R.E. Bryant, “Formal verification by Symbolic Evaluation of Partially Ordered Trajectories”, *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–190, March 1995.