Getting Formal Verification into Design Flow

Arvind[1], Nirav Dave[1], and Michael Katelman[2]

[1] Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology, Cambridge, MA 02139, USA
{arvind,ndave}@csail.mit.edu
[2] Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
katelman@uiuc.edu

Abstract. The ultimate goal of formal methods is to provide assurances about the quality, performance, security, etc. of systems. While formal tools have advanced greatly over the past two decades, widespread proliferation has not yet occurred, and the full impact of formal methods is still to be realized. This paper presents some ideas on how to catalyze the growth of formal techniques in day-to-day engineering practice. We draw on our experience as hardware engineers that want to use, and have tried to use, formal methods in our own designs. The points we make probably have been made before. However we illustrate each one with concrete designs. Our examples support three major themes: (1) correctness depends highly on the application and even a collection of formal methods cannot handle the whole problem; (2) high-level design languages can facilitate the interaction between design and formal methods; and (3) formal method tools should be presented as integrated debugging aids as opposed to one requiring mastering a foreign language or esoteric concepts.

1 Introduction

Over the past few decades, formal techniques have made impressive progress. For example, serious theorems (*e.g.,* relative consistency of AC with ZF) have been mechanically verified [1], and huge improvements in the range and capacity of decision procedures (*e.g.,* linear arithmetic, uninterpreted functions, bit-vectors, SAT solving) have been made [2–6]. In the hardware industry, a number of commercial tools to support formal verification have sprung up [7–12]. These improvements in tools and techniques have gotten the attention of both the research community and industry. Formal methods are used within the research community in numerous case studies ranging from security [13] and wireless protocols [14] to processors [15] and cache-coherence protocols [16] to buffer overflows in software [17]. In the industrial setting, ever-increasing system complexity in both hardware and software has fostered interest in formal methods (*e.g.,* [18–22]). Indeed, in the hardware industry, formal methods are even widely employed for certain types of verification tasks. Sadly, despite all of these advances and a receptive climate, formal methods have had no discernable effect on the day-to-day design flow. Both hardware and software designers still reach verification closure primarily through *ad hoc* testing and low-level debugging.

Designers are paid to make efficient systems within some constraints. The constraints may be on performance (*e.g.,* a video codec must process 30 frames per second at 720p), power (*e.g.,* a cell phone may not dissipate more that 3W), cost (*e.g.,* the chip must not cost more than $5) or some other metric like compatibility (*e.g.,* a DSP that must run all existing applications). In addition there are always limited resources and time-to-market pressures. In such an environment, many designers believe (with some justification) that extensive testing is sufficient to reach the corresponding confidence for the economic or social consequences of failure. It is difficult to find cases where a product is shipped only after it passes a full "formal verification".

The current design flows are strongly biased towards *post-design verification*. Of course design engineers are encouraged to perform unit testing but the primary task of verification rests with a verification team that is often two to three times the size of the design team. Good verification teams prepare elaborate test plans and then employ a horde of engineers to actually write and perform the tests. By way of analogy, this is how the automobile industry used to be in the United States until the early 1980's. It employed more and more inspectors to try to avoid shipping defective automobiles. This proved ineffective in developing higher quality automobiles. Then the Japanese industry started focusing on *zero-defect* components which resulted in drastically improved quality.

The aim of this paper is to provide, via a rich set of examples, insights into the problems that designers face and how formal methods may alleviate some of these problems. We are convinced that designers want their designs to be correct, and would use methods and tools that isolate tricky problems quickly. However, a designer is unlikely to employ a tool that is too hard to use or too slow, or provides information that the designer is at best peripherally interested in. Designers are rarely interested in tools that require daunting specifications in some totally unfamiliar form, or which may not be available at the time of design. If the verification of a WiFi protocol block requires, say, axiomatization of the 802.11a standard in PVS, then the specification task itself would overwhelm the design task. Additionally, large formal specifications are at least as hard to debug as large designs and their engineering utility is questionable even when correct.

We make three points in this paper with the goal of stirring a discussion about how to address the gap between day-to-day engineering practice and the unrealized potential of formal methods:

1. *Correctness depends on the application. Different applications require vastly different formal techniques.* Most designs will benefit from both the application of formal methods and testing via executable specifications (see Section 2).
2. *Formal tools must be tied directly to high-level design languages.* There is a great deal of high-level information that needs to be communicated to formal tools. Much of it is also needed for the design itself. The only practical way to get this information to the tools is by extracting it directly from the design itself; designers are unlikely to restate knowledge in a different notation solely for the sake of verification (see Section 3).
3. *Most formal methods and tools have a post-design bias. Instead they should be presented as debugging aids during the design process.* In the most successful

cases, designers are unaware that they are using a formal method. A good design method is much more likely to be used by the designers if it is enforced by the tools (see Section 4).

In this paper we focus only on hardware design, though our observations may apply equally to software. We also do not address the problems of defective tools (*e.g.,* the compiler itself produces incorrect code, ambiguities in the design language). In the hardware industry there is a tendency to merge the testing of the tool and of the design. We feel strongly that these activities should be kept separate. The designer must have a high degree of confidence that the tools are bug-free or the verification task is truly monumental. Also, we do not focus on manufacturing bugs (*e.g.,* stuck-at-zero faults) or the errors introduced by physical design tools. We also ignore the issue of lack of education in formal methods on part of design engineers; we live in an environment (*i.e.,* MIT) where this is not an issue. We focus only on the technical challenges involved in making formal methods fit within an effective design-debug loop. *Our goal is to help the designer produce designs which, if implemented in a totally automatic way using bug-free tools, would have completely satisfied the specification.*

2 What Needs to be Verified: Examples from Hardware

Over the last decade we have designed a variety of complex digital systems. Our verification methodology has been based primarily on testing and occasionally on handwritten proofs for very difficult parts of the design. In our design explorations we've found a number of places where formal verification could have been highly useful. However, in no instance have we found that formal verification would have replaced testing. This is because testing provides a more-than-adequate guarantee for some aspects of the verification task and setting up a testbench is usually significantly easier than a formal verification tool. Nevertheless, there are situations where a designer cannot get sufficient confidence in the correctness of the design even with extraordinary amount of testing.

In this section we illustrate the verification task via a number of examples taken from our own personal experiences and highlight where formal methods could have had significant impact. These examples should also make it obvious to the reader that *proper verification involves domain specific knowledge.*

2.1 Simple Deterministic Designs: *IP Lookup*

The Longest Prefix Match (LPM) function is used in Internet Protocol (IP) packet routers to determine the output port to which an input packet should be forwarded. It is a requirement that the router maintain the ordering of packets between the same source and destination. For cost and power reasons, the memory size must be kept small. This rules out a flat table implementation which even for IPv4 would need 2^{32} elements. Designs often use a tree-structured table which allows one to exploit the similarities in table entries with common prefixes. The lookup procedure essentially reads the table repeatedly using different parts of the IP address. If a result is found, an output is produced, otherwise

another read is performed in the part of the table holding the relevant subtree. For most schemes, an IPv4 lookup requires between 1 and 4 memory reads.

One efficient implementation [23] of this lookup functionality is based on a circular pipeline shown in Figure 1. It includes a FIFO of partially served requests, a pipelined memory, and a completion buffer that ensures that outputs are sent out in the correct order.

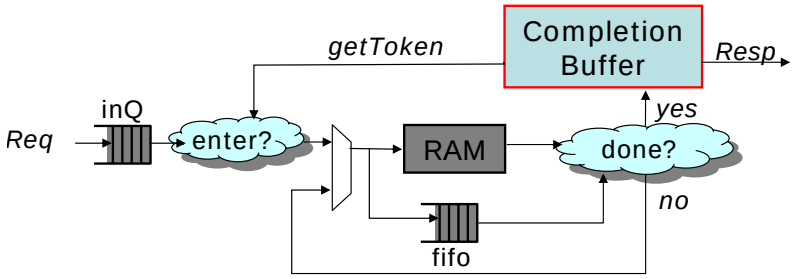

Fig. 1. Circular Pipeline Design

Writing an operational description of IP Lookup is easy. The functional correctness is deterministic and can be implemented in any sequential language as a single lookup in a flat IP table. Indeed, with a simple executable description and random stimulus generation, we can achieve a high degree of confidence via testing that the design does not produce wrong answers. However, checking other requirements is a different story. Do packets come out in order? Does each packet produce a result packet? One may need to check this if there is a danger of dropped packets because the design cannot keep up with the specified input rate. Is there a dead-cycle, that is, can a new packet enter the system in the cycle when an old packet leaves? All these questions are very important for the designer, and to set up tests to check all these properties is not always easy and often not satisfactory. If one could formally state these properties and easily verify them, most designers would take the time to do so.

2.2 Dealing with noise: *802.11a*

802.11a is an IEEE standard for wireless communication [24]. The protocol translates raw bits from the Media Access Control (MAC) into Orthogonal Frequency Division Multiplexing (OFDM) symbols comprised of 64 32-bit fixed-width complex numbers. The protocol is designed to operate at different data rates; at higher rates it consumes more input to produce each symbol. Regardless of the rate, to be acceptable an implementation must be able to generate an OFDM symbol every 4 μs.

The 802.11a specification only specifies how data is to be transmitted. This specification is given operationally as a sequence of stream processing functions (see Figure 2). Each of these functions can be viewed in a way that is similar to the IP lookup function and consequently it is easy to translate the 802.11a specs into an executable sequential program. In fact the block structure in the reference is often directly visible in the implementation. Verification can be performed by comparing the test results from the design against the executable specs. For

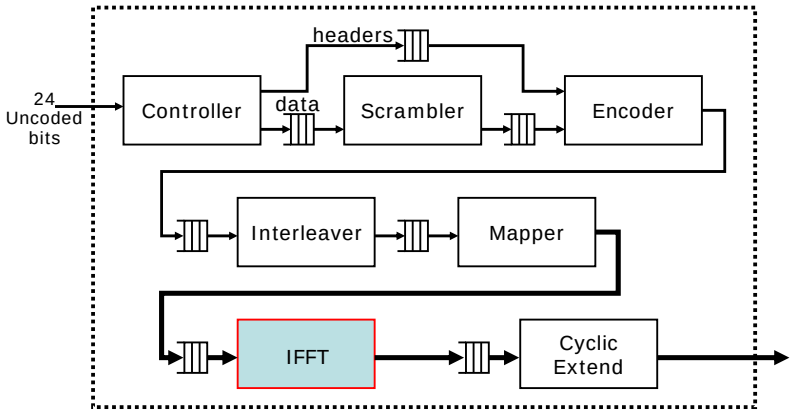

Fig. 2. 802.11a Transmitter Pipeline

debugging purposes it is not uncommon to instrument the standard executable code to capture the bitstream after each functional block and compare it against the internal bitstreams.

On the receiving side, the input is tightly coupled with the digital-to-analog conversion at the transmitter, the noise properties of the transmission medium, analog-to-digital conversion at the receiver and the phase shift between sender and receiver. We can partition this problem into two subproblems: given a transmitted stream, "can the receiver synchronize its phase to match the transmitter's phase?", and "given a noisy transmitted packet can the receiver successfully reconstruct the original packet?". Since we know the noise models the standard is supposed to correct, we can introduce the correctable noise effects on a transmitted packet (including possible phase shifts) relatively easily.

Currently, to reach sufficient confidence that an 802.11a design is transmitting and receiving data reliably, we take into account two more facts. First, the 802.11a codec was designed to reduce the effect of corner cases, which determined the worst-case behavior. Second, since we can always drop data, not getting all the *exact* behavior in corner cases only reduces the space of noise which can be corrected, slightly degrading performance. Additionally, unlike the IP lookup example, all of the design complexity lies in the data transformations, not the control logic. These points make it so that once we have a system where a few packets are sent and received correctly, our confidence that the design is "good enough" to ship becomes very high. Of course more directed testing can be performed to gain more confidence.

What simple directed testing does not cover, however, is the correctness of fractional-level arithmetic, which is used pervasively in 802.11a. Designers need confidence that their numerical logic works (of course, such an arithmetic library is useful for other designs as well). This task is well suited to formal verification.

It is not uncommon in such designs that one transforms a block from an obviously correct implementation into a higher-performing one. For example, one may transform a large combinational circuit into a folded pipeline to reduce area. These transformations will always need to be correct in the functional sense but may not result in equivalent FSMs. It is possible to describe these transformations in such a way that the functional behavior is abstracted away,

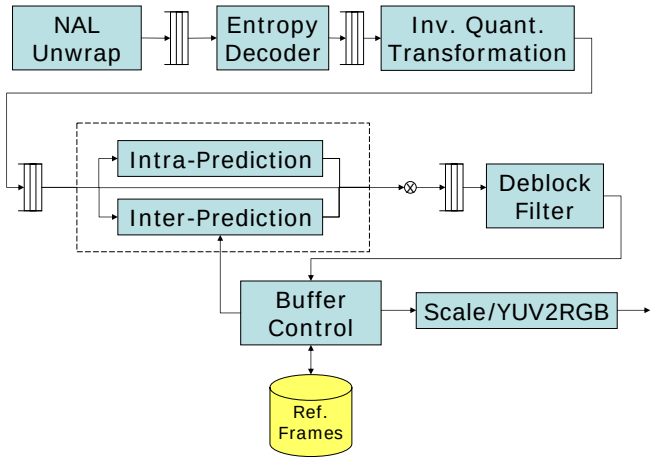

Fig. 3. H.264 Decode Block Level Diagram

i.e., passed in as a parameter. It would be incredibly useful if transformations of this sort were verified formally so that we could do architectural exploration without adding to the testing burden.

2.3 Specification of a Lossy System: *The H.264 Video CODEC*

The H.264 Advanced Video Codec is an ITU standard for encoding and decoding video with a target coding efficiency twice that of H.263 and with comparable quality to H.262 (MPEG2) [25]. H.264 enables PAL (720×576) resolution video to be transmitted at 1Mbit/sec. Like other video coding standards, H.264 specifies only how to reconstruct a video from an encoded bitstream, not how a video is encoded. The goal of the encoder designer is to produce as compressed a bitstream as possible without degrading the user-perceivable quality. Sometimes the encoder also has the constraint of how much computation can be performed because the encoding may have to be done in real time or on a handheld device such as cell phone or camcorder.

Since the encoding is lossy, what does it even mean to correctly encode a video? We could compare the original video with the results of encoding and then decoding the same video, but how does one classify user-perceivable differences? A number of heuristics which approximate user-perceivable differences exist, but these are too crude for verification purposes. Consequently there is no hard or fast rule that the design *must* ensure. (Fortunately, a few errors here and there are unlikely to be catastrophic in this application).

The decoder can be described as a relatively complex dataflow graph as shown in Figure 3. Unlike the 802.11a transmitter which had a significant but still manageable description, the decoding reference for H.264 is 80 thousand lines of C and an English specification that runs into hundreds of pages! Neither of these descriptions is complete: the English is ambiguous in many places and the C code represents only a deterministic representation of the codec. While many rich and complex transformations can be applied on the C code, some arbitrary choices require significant high-level knowledge to find, effectively ruling them out without additional knowledge. Thus, complete understanding of the codec

requires significant use of both of these specifications in tandem. We believe that H.264 is complex enough that there is virtually no hope of ever directly generating a complete formal specification, especially one that could be ready during the design process.

Given these complications, designers are limited to testing for all but the simplest sub-blocks. For the decoder, we test by taking a good mix of interesting videos, encoding them, and then checking that the output of the reference C code and design match. As with the 802.11 transmitter, by instrumenting the reference code one can also generate internal bitstreams at the output of functional blocks and compare them against the design.

The main use of formal methods in this example, like 802.11, would be restricted to testing arithmetic libraries, correctness of transformations for performance, etc. One could also formally verify some tricky parts of dataflow in the pipeline. For example, one could prove that the inter-prediction block does not read the reference frame before it has been properly constructed. However, in case of such an error the bitstream is unlikely to match with the C reference code and the error would be caught by running a few sample videos.

2.4 Nondeterminism: *Cache Coherence*

Verifying the correctness of a cache coherence protocol for shared memory systems presents a unique challenge. On one hand, there is no ambiguity about a correct answer; each Load is supposed to return a value from a set of possible Store values. On the other hand, both the protocol and its operating environment exhibit nondeterminism, *i.e.,* time dependent behavior. This is also an area where absolute correctness is expected. After all, software systems rely on the perfect behavior of Load-Store instructions on any machine!

Protocols are often described abstractly in tabular form where each line represents a valid transition in the protocols abstract state space. These transitions (or *rules*) are allowed to be applied in any order. Even though many rules can be applied at a time, their behavior is understandable as if the rules were applied one at a time in some order. This representation captures the nondeterminism of both the protocol and its environment accurately. These tabular descriptions are often independent of the number of processors or the size of caches.

Cache coherence protocols present two challenges in verification: how to incorporate nondeterminism in the verification framework and how to state formally the correctness criterion so that it is of use in the verification process. An operational model of a protocol, *i.e.,* an interpreter that applies protocol rules in some deterministic order is of limited use because it explores a very small subset of the nondeterministic state space. Even for cleverly designed tests it is hard to convince oneself that the protocol functions correctly in all cases and enters neither deadlock nor livelock in any circumstance. The verification problem is further exacerbated by the fact that modern protocols tend to be very complex for performance reasons and are understood by very few implementers.

Model checking has proven quite useful in identifying obscure bugs in cache coherence protocols. Indeed, for these reasons cache coherence protocols are well-trodden ground in the formal methods literature (*e.g.,* see [26]). This literature also shows the limitations of model checking if one is looking for an absolute

guarantee of correctness for this problem. First, the designer/verifier builds an abstraction of the real design, because the tools cannot handle the real design. Second, the designer looks for a set of invariants, such as "a dirty copy of an address cannot exist in more than one cache", whose proof will guarantee the correctness of the whole protocol. Third, to keep the state space from exploding, model checking is applied to a small machine configuration (*e.g.,* three caches and two unique addresses). The designer has to convince himself that if there is a bug in the protocol it will show up in this small system.

We think all these steps are fraught with problems; there is always a chance of omitting an important detail in the abstraction process. Without a formal proof it is difficult to convince oneself that a set of invariants is sufficient to verify the whole protocol. Finally, one needs to prove, that the correctness of the simple system configuration that was checked implies the correctness of all possible system configurations.

This is one area where model checking coupled with mechanical theorem proving can be very useful in proving the correctness of a protocol. But to be useful in practice, the implementation must be generated automatically from the protocol description that is used in the verification process. We will also report our attempts at mechanical theorem proving of a complex cache coherence protocol in Section 4.

2.5 Simple Specification, Complex Design: *Processors*

Microprocessors encompass many important architectural concepts that occur throughout the wider spectrum of digital logic design. At the same time, "absolute correctness" is what is required of a microprocessor. People are more tolerant of software bugs than a microprocessor which "sort of works". Given the economic importance of microprocessors it is not surprising that microprocessors have pushed verification research and continue to be a rich source of technical problems and proving ground for new verification tools. In addition, the size and complexity of modern microarchitectures has led to a situation where functional verification is a significant contributor to the microprocessor design cycle.

Specifying the correctness of a microprocessor is relatively straightforward: it must respect the semantics of the target instruction set (ISA). Modulo certain complexities (*e.g.,* virtual memory, exceptions) a typical ISA consists of basic arithmetic, memory, and branching instructions that together constitute a simple but extremely expressive programming language. Unlike the previous example of H.264 or 802.11a, this specification can be defined via a simple (one-instruction-at-a-time) processor implementation or some other software ISA interpreter in a straightforward manner. The proof obligation is then that the more complex implementation matches the simpler one. These days microprocessors are often multicore or appear in shared memory systems. Consequently, cache coherence issues discussed in the previous section can be treated as a subset of microprocessor verification problem.

The classic technique for establishing that a particular microarchitecture is a correct implementation of an ISA is to show that its state transitions can be simulated by a much simpler and obviously correct "reference implementation"

of the ISA. Since a real implementation has much larger state space than a reference implementation, one has to provide abstraction functions that map the elements of the real implementation to the elements of the reference implementation. Most abstraction functions are based on *flushing* or *killing*. The flushing abstraction [15, 27–29] returns an ISA state by completing partially executed instructions in the pipeline. In contrast, killing squashes partially executed instructions and returns the system to the ISA state corresponding to the last committed instruction. Processors are really nondeterministic (consider interrupts and shared memory) and when reasoning about nondeterministic behavior, killing has advantages over flushing because there is a unique last instruction, but there can be several possible futures for uncommitted instructions [29].

Intellectually stimulating as these ideas are, their impact on commercial processor development is minimal. The biggest impediment is that none of these ideas are actually applied to a real implementation, *i.e.,* an RTL description from which the gates may be synthesized automatically or semi-automatically. They are applied to an abstraction of the implementation, where the abstraction is motivated as much by what the tool can handle as what needs to be verified.

Nevertheless correct processor design remains one of the most promising areas for formal verification. In addition to test codes (micro-benchmarks) and model checking, mechanical theorem proving would be needed to gain confidence that a processor with caches, TLBs and branch predictor and cache-coherence engine works in all cases.

3 High-Level Design Languages are a Prerequisite for Incorporating Formal Methods into Design

In the previous section we have shown through examples that domain specific knowledge is essential to formulate a verification plan for a design. Furthermore, formal verification requires some sort of specification or high-level architectural information to state properties to be proven. What we argue in this section is that this sort of information is communicated best as part of the design and hence, *directly* through the design language. This isn't just a vehicle to communicate high-level information to various tools, it is also a way to communicate *to the designer* the results of the checking and to help him understand and fix errors. For example, type checking is most meaningful when the typing system allows for rich type structures defined by the user. The ideal situation occurs when the abilities of the formal analysis engine and the intention of the high-level language concept match, so that the underlying discipline can be *enforced*. The best example of this is again typing, where the intent of the system is to better manage which operations go with which data, and type checking algorithms are able to statically enforce this discipline.

This section is organized around a set of high-level concepts which we feel can improve both design and formal analysis. They do not all neatly fit the ideal case described above, but nevertheless elucidate important areas where language and formal methods should fit together to the betterment of the design process. *No existing language embodies all of these concepts, but several incorporate a subset.*

3.1 Static Type Checking

Type systems are one of the most accepted ideas in software engineering and appear in varying degree of sophistication in almost all languages. Algebraic types and record constructors, in particular, allow a designer to design a rich type structure allowing for notions like choice and grouping in user-defined types. For instance, a `Maybe` type groups a data value with a valid bit. This guarantees that the value is accessible only when the bit is true.

The purpose of typing is to avoid operations on improperly structured objects; and when checked it helps designers avoid a large class of mistakes and actually speeds up design. The reason that typing and type checking is successful is because the high-level intent of types is made completely clear *by the language*. Therefore the errors returned by a type check seem natural to the designer and can be addressed quickly. In addition, type information also serves as useful documentation for the designers.

Type correctness is best enforced *statically*, at compile time, and has proved incredibly successful in this role. In fact, static type checking has gotten so pervasive that most engineers do not even consider it to be a formal method; *formal methods are about correctness, type checking is just common sense*. We have seen that other forms of enforcing a type discipline fail to work as well. For example, enforcing type correctness dynamically, as in Scheme or Perl, pinpoints bugs much later in the design process than static type checking. Extra-lingual attempts in Verilog – a language with weak typing – invariably fail, because the typing enforced by the compiler does not match the extra-lingual discipline, and therefore is not checked automatically. Designers cannot be bothered to do hand checking.

3.2 High-Level Parameterization

Often when an engineer starts to design a functional block, a number of similar sub-blocks become apparent. For instance, one can imagine variations of a FIFO with different sizes or different types of data elements. It makes little sense for designers to make each of these variations separately; rather, one should have a design which can be supplied the element type and the size as parameters. Using a modern type system the type of element can be specified polymorphically so that all the attendant FIFO operations accept and return only correct types.

Parameterization allows us to abstract unnecessary details and factor the proof obligation. Parameterization by data type shows that the specifics of the data element are unimportant to verifying the FIFO's behavior. Secondly, parameterization of the FIFO size allows us to make proofs across all FIFO sizes, amortizing verification costs.

In hardware this sort of parameterization is very common and keeping to a small set of stateful building blocks can dramatically help in reducing complexity. It is also likely to result in designs that are highly reusable. However, an important requirement in hardware design is that parameterization should not add extra logic – all the effects of parameterization should disappear when the block is instantiated. This can be accomplished by a compiler via *static elaboration*, a simpler form of *partial evaluation*.

3.3 Modularity

Perhaps the most important high-level abstraction for a designer is modularity. The purpose of modularity is to elucidate high-level functionality through the encapsulation of implementation details. This enhances readability of the code and improves its reuse amongst designers and between projects. Proper modularity also permits the designer to have several different implementations of the same interface. For example, small (*i.e.,* one or two element) FIFOs may be implemented very differently from larger (*i.e.,* several hundred element) FIFOs, but may have the same interface. In addition, modularity may serve as a natural place for formal tools to divide up proof obligations into tractable pieces.

Hardware also presents an opportunity to design a family of modules which may differ only in their concurrency properties. For example, one can imagine several different types of FIFOs for different design situations. For a FIFO to be used in a pipeline, it is necessary that enqueue and dequeue can happen concurrently, and the effect of dequeue should take place before the effect of enqueue. On the other hand for a FIFO to be used in a rate matching buffer, one also needs concurrent enqueue and dequeue but the effect of enqueue to take place before the effect of dequeue. Formal specification of such properties in a FIFOs interface would dramatically benefit the verification process.

Additionally, a system with strong modularity may admit modular refinement, where the designer derives various implementations from an original design serving as a "golden" specification. These derivations come in two types: design-independent transformations which are provable solely based on the language semantics, and design-specific refinements whose correctness depends upon domain-specific knowledge. Both of these changes are clear places where formal methods are useful, especially the provably correct transformations as they give the designer a useful toolbox of easy but powerful design choices he can make. For example, we have shown that using a few pipelining combinators we can effectively take a functional description of FFT expressed as a pure combinational circuit and quickly generate various "folded pipeline" versions [30]. (In a folded pipeline the same "stage logic" is reused across several cycles to implement different pipeline stages). Formally proving that these have the same input-output behavior would mean that showing the correctness of a combinational design (which is straightforward) would imply the correctness of the folded design (which is much more challenging).

Many software languages have strong enough modular interfaces that proper isolation is guaranteed. However, most hardware description languages (*e.g.,* Verilog, VHDL) have modules which serve only as structural abstractions, and do not easily allow one to abstract the behavior across module boundaries. Verilog designers do use modules as abstractions, but because of the complications due to its concrete timings, transferring these abstractions of interfaces to formal tools is quite difficult.

In contrast, modules in Bluespec [31], a language in which we have written all the examples discussed in this paper, require a stronger method-oriented interface which groups related ports into methods. Bluespec semantics is based on Guarded Atomic Actions (*i.e.,* rules), which is the same semantic model that underlies Unity [32]. In Bluespec every method has a notion of being "ready to

be applied", and a compiler enforced microprotocol guarantees that a method cannot be applied unless it is ready. This allows the designer to decouple the timings of interactions from different methods, giving them the ability to play with the intra-cycle ordering. A formal tool can gain information about how a circuit can be used because the method calling protocol is uniform and implied by the language semantics, rather than in comments preceding the module definition.

3.4 Unified Language for Design and Specification

"Golden" reference specifications tend to be given in a different language than the one used in implementation. This is because implementation languages often require more details than what people writing the reference model wish to deal with. Ideally, a good high-level language would not only allow one to write good designs but also, when possible, to write reference specifications as well. This may not be possible if the specification is so abstract as to not even be executable. If it is possible then design becomes simply a refinement of the specification. The designer is saved from the major task of manually translating from the reference specification language into the implementation language.

It is important to note that refinement is only natural when moving between the high-level reference and detailed implementation does not involve significant changes in the concurrency or semantic model. For instance, Verilog has two separate "sublanguages": Behavioral Verilog which is generally used for rough-cut behavioral specifications, and Synthesizable Verilog which is used to the represent implementable designs. While these two languages have the same syntax, the simulation semantics of Verilog does not always match the Verilog synthesis semantics. While there may be a way to refine from the initial high-level specification to a synthesizable implementation, the semantic difference significantly hinders such a transformation. SystemC experiences this same problem as well. Its high-level simulation semantics closely resembles OS-thread concurrency, and therefore is a mismatch for the underlying hardware model that it needs to describe. Esterel [22] represents a significant improvement because its semantic model is consistent with synchronous FSMs. These more closely correspond with the eventual hardware implementation. Similarly, Bluespec offers a better basis for formal methods in this regard as its nondeterministic guarded-atomic-action semantics are consistent from high-level specification to implementation.

A key for languages to be able to operate at both levels is to have a large selection of high-level constructs which have solid refinable reference implementations. This can be addressed partially via good standard libraries in a high-level language where commonly used circuits like arithmetic units, register files, FIFOs and memories can be encapsulated and expressed cleanly. This is trickier than what one might expect because, in most hardware descriptions languages module interfaces are time sensitive.

3.5 Handling Nondeterminism

Closely related to the previous point is a language's ability to express and resolve nondeterminism. An important aspect in verification is the ability to express inherent nondeterminism in the correctness specification. A specification

of a speculative processor should permit an unspecified number of speculative instructions to be executed before the speculation is resolved. The cache coherence protocol described in Section 2.4 has nondeterminism in the memory access stream, and with the 802.11 specification, there is probabilistic nondeterminism having to do with the transmission medium.

With the exception of Bluespec, hardware design languages do not permit one to express nondeterminism. An oft heard remark about nondeterminism is that it significantly complicates reasoning about systems (See, for example, Berry's comment about the need of determinacy in Esterel [33]). This ignores the fact that nondeterminism can be viewed as an axis of flexibility for implementation purposes. Once we have learned to deal with nondeterminism, to quote Dijkstra [34]: "*[It] is no longer frightening. On the contrary! We shall learn to appreciate it, even as a valuable stepping stone in the design of an ultimately fully deterministic mechanism.*"

For example, Bluespec semantics permits nondeterminism in the selection of rules to be executed in a given cycle. The compiler removes this nondeterminism to generate the final hardware in a process called *scheduling*. It has been shown that the compiler can generate efficient hardware automatically, but the user can also provide guidance from the source code if necessary. We can think of a Bluespec design as a nondeterministic specification and the additional information the designer passes to the compiler to choose a good scheduler as implementation details. This flexibility allows designers to explore many different design options easily.

Nondeterminism also results in a simplification of the verification process. For example we have shown how a cache coherence protocol [35] can be specified naturally in Bluespec. The original protocol, after we made sure rules were selected fairly for execution, served immediately as a working implementation. Assuming that the original protocol was correct, this implementation was guaranteed to be correct. The verification task now only requires us to show that further refinements to the design preserve this correctness.

3.6 Property Specification

Property specification is a mechanism through which relatively simple *assertions* can be made about a design. For example, in C the `assert` macro can be used to halt execution when, at a prespecified execution point, the state of the program is determined to be bad. Assertions on the hardware side are used somewhat differently, as *monitors* of the circuit's dynamic behavior over multiple cycles. When behavior satisfying the assertion is witnessed, the event is recorded and reported to the user. Alternatively, a design can be proven to always satisfy the assertion. For example, a typical assertion is that the state of some register never has more than one bit set to 1 (*i.e.,* it is 1-hot encoded).

SystemVerilog [36], a proper extension to Verilog, adds a number of language features to Verilog, including an assertion language, objects, and a more sophisticated type system. The assertion language is a combination of regular expressions and temporal logic. The designer benefits greatly from having Verilog embedded directly within the language for defining assertions. For example,

assertions are pervasive across module instances so it is possible to define a one-hot register module. This module can be reused repeatedly in the design and even across designs.

Specification languages have to be limited in their expressivity to make the proof obligations for automatic decision procedures tractable. The BAT system [5] is interesting in the sense that it is geared towards handling bit-level implementations and incorporates sophisticated decision procedures to prove non-trivial assertions about them. In its present form BAT lacks many properties of high-level design languages, but it may be an appropriate compilation target for languages like SystemVerilog or Bluespec.

Even though many decision procedures are totally automatic, in practice, using them requires that we limit the complexity of the assertions as well as the design unit over which they are considered. Given this limitation, there may be room for specialized solvers aimed at common classes of local assertions.

3.7 FSM Equivalence and Automatic Retiming

In the EDA industry we have seen wide adoption of equivalence checking technologies. All of the major EDA vendors supply such tools [8–10], which provide some guarantee that low-level transformations on netlists result in functionally equivalent circuitry. These systems allow *retiming* optimizations to be tried without any worries about the correctness of the system has changes. This work has been highly successful for two major reasons. First, the algorithms are effective on real-world designs. Second, the optimizations which are allowed can quickly eke out crucial system performance improvements.

Intel's Integrated Design and Verification (IDV) environment built on the Forte [20] formal verification system is a more cutting-edge example of successful integration of formal methods into the design process. The input to IDV is an executable and synthesizable model expressed in a general-purpose reflected functional language (reFLect). The tool allows the designer to transform the circuit in a way that maintains a sequential refinement relation at every step. The transformation process is used throughout – from high-level algorithmic transformations down to detailed physical placement changes. This allows the tool to catch implementation bugs as soon as they are made. In order to remove specification bugs, the Forte tool also has more sophisticated formal analysis capabilities which require more user intervention. For example, it has been used to verify the correctness of an x86 instruction-length decoder and formally link an x86 floating-point unit with the IEEE specification expressed using real numbers.

3.8 Formalized Testing

Property specification is used for more than just assertions about one-hot registers or constraints on complicated protocols, they are widely used to define *functional coverage goals*. Instead of an assertion firing to indicate a breach of protocol (*e.g.,* a one-hot register has two 1 bits) the satisfaction of a property now indicates greater coverage of the planned test space (*e.g.,* pipeline flush occurred). Coverage-driven testing is becoming very popular with languages such as SystemVerilog [36] that integrate coverage goal specification with the design.

Integration of the languages makes the testing goals clearer (similar syntax is used) and simply more convenient.

Coverage goals given in a language such as SystemVerilog advance verification practice by *formalizing* the testplan and removing ambiguity from the English descriptions. This saves time in developing directed test cases, reduces redundant work, and allows tools to automatically manage the testing effort. The management tools that accompany simulators yield essential information for steering the testing effort. Without such tools, it is not even clear when we have exercised a particular behavior. The view into the logic may only be through waveforms, and going through the generated waveform data from even a simple case can be daunting. Techniques also exist for generating test-stimulus through formal methods, most of which rely on having formalized coverage goals to guide the underlying deduction mechanisms.

4 Issues with Incorporating Formal Methods into Design

We have seen the development of a rich variety of formal tools over the last two decades: temporal logic model checkers (*e.g.,* SMV [2], SPIN [37]); theorem provers (*e.g.,* ACL2 [38], PVS [39], Isabelle/HOL [40]); automated decision and semi-decision procedures (*e.g.,* BAT [5], Z3 [4], UCLID [3], Yices [41], Alloy [42]); and other specification languages and tools (*e.g.,* B [43], IOA [44], TIOA [45]). Most tools today can handle much larger problems, have better libraries, and are much more robust than a decade ago. In spite of these advances these tools have at best seen marginal penetration in the design community. The community of users of these tools has not gone far beyond the tool designers, who tend to be highly inclined mathematically.

The only way to use any of these tools is to learn an entirely new system often involving its own mathematical concepts which are divorced from the concepts used in most designs. The barrier to entry is so high that the effort required is almost never justified by the economic gains. We think that there is an obvious way to fix this problem, though it requires a change in the mindset of the formal methods community:

1. *Tools must be invokable from the design language in a seamless manner. This implies that the tools must be able to take unmodified source as input as well as report results in a manner consistent with the language.*
2. *If possible, tools should be entirely automatic requiring no user intervention to facilitate the proof process. Alternatively, if tools require some user guidance, this information must be provided through the design language.*

In this section we demonstrate how the lack of these two characteristics make current methods extremely difficult to integrate into design flows. We do this by considering the possible verifications of two designs described previously using current tools. In one case we focus on a model-checking-based platform, and in the other we consider theorem proving.

4.1 IP Lookup: *using model checkers in practice*

Consider what the process would be to verify the IP lookup design of Section 2.1 using the SPIN model checker. The design is written in Bluespec, but the SPIN

tool accepts Promela as input. The first thing that needs to be done is to convert the Bluespec into Promela. While this could possibly be done *mechanically*, as it stands today the designer must translate by hand. The same is true for most design languages and this is highly undesirable. A manual translation can easily introduce new behaviors into the design, or remove behaviors that existed before translation. Therefore, the translation itself has to be verified, making this approach a non-starter.

After translation, the next barrier that we come up against is specifying the property to be verified. Consider the property that there is a one-to-one correspondence between inputs and output of the IP lookup design. Currently, the designer is expected to be able to represent this in LTL. This is a significant challenge as such logics are a completely foreign way of thinking for most designers. Worse, since we are limited by LTL we cannot even represent the possibly unbounded size of the input-output relation. The designer is left wondering whether this is a result of him not knowing how to express the property, LTL being restrictive, or an actual issue in his design.

Assuming that the property actually can be represented in LTL, it is likely that a direct translation will be completely untenable. That is, given the propositional nature of LTL, the only view into the state is through *unary predicates*. Therefore, if a direct translation of the code is done, then this number will be gigantic. For example, given a 32-bit register, each of the 2^{32} states would need to be encoded separately to get all information out of the design. These numbers quickly become far larger than any model checker can reasonably handle. Alternatively, the designer must abstract away certain states and prove that a much smaller representation preserves the property being checked. For example, with IP lookup a reasonable abstraction would use certain key assumptions about the lookup table (*e.g.,* that a lookup chain in the table is of length 1 to 4). Representing such abstraction naturally in the design language is a challenging open problem..

However, even if we assume some abstraction is done correctly, what happens if SPIN evaluates the correctness property and reports a bug? The designer must now translate this failed path back into something useful to reason about. In this case, the ideal way to express this would be the relevant indexes in the IP lookup and, the inputs questions, and the timings of the rules in the system. Extracting this from the given path is tedious for a designer. Any help in making the data more accessible would pay significant dividends.

For formal methods to work effectively in an engineering design flow, it is important to prevent the user from having to jump through hoops. But how would the designer want to be able to use a formal system? First, the designer wouldn't have to manually translate from the design language (in this example Bluespec). Second, apropos Section 3.4, a correctness specification would also be given as part of the design in some natural dialect of Bluespec. In this case it may involve adding virtual state to represent unique request tags which can be verified to occur in sequential order at output. By informing the system of the invariant that should be maintained, (*i.e.,* packet identifiers leave in sequential order) compilation would either prove or disprove this assertion. If wrong, the system would give an initial state and an understandable sequence of rule firings (a single semantic step in Bluespec) witnessing the failure. This would allow the

designer to stay focused on his design in the language to which he is used while incorporating the verification task easily with compilation, a task he must do frequently in design.

4.2 Cache Coherence: *using theorem provers in practice*

In his dissertation, Xiaowei Shen described an adaptive cache coherence protocol called Cachet [46, 47] and proved it correct. This proof was very long and complicated and it was decided that it should be proved mechanically using PVS [16] to guarantee no mistakes were made. The mechanical proof considered a significant subset of the protocol. Despite already having completed a hand-worked proof of the complete protocol, it took an engineer skilled both in design and formal methods six months of effort to complete, a significant expenditure. Clearly, this amount of effort is much too burdensome for general integration in the design flow. However, it is still worthwhile to consider how this would apply for verifying the implementation.

A full proof of our implementation requires effort just to get an implementation into PVS. One possible approach would be to synthesize the code and then feed it into PVS as one large Boolean next-state function. Of course, the user then has to reason at this horribly tedious level of detail. Alternatively, we could try to axiomatize Verilog or Bluespec semantics directly in PVS, but this adds a layer of indirection in the proof which may complicate things. Ideally the user could express booleans which represented the interesting properties to be verified (*e.g.,* no multiple modified version of an address) directly in the design.

Secondly, even with a good representation of the code in PVS, the proof becomes much more complicated based on the fact that any implementation is bound to involve many system details not elucidated by the protocol. For example, while the abstract proof could assume that all coherence messages are received, in an implementation with finite buffering and various routing logic, this too must be verified. This would be quite difficult to formulate, let alone prove in PVS.

5 Conclusion

Formal methods have come a long way, but for reasons we have outlined in this paper, a large gap remains between current formal methodologies and engineering practice. Except in a few instances, formal methods are not tightly integrated into the design-debug loop. We have argued that for widespread adoption formal methods must be invoked through high-level design languages and must present a semantic model that makes sense to the designer. Incorporation of assertions in SystemVerilog is an example of a good start but its effectiveness is limited. The weak semantics of Verilog, inability to express the full spectrum of interesting correctness properties, and the capacity of current tools all take away from usability.

The range of examples in this paper have shown that even for a single application often more than one formal technique is needed to show correctness. It may not be desirable to try to unify too many concepts into one specification

language. For example, should every design language be so powerful that it can express probabilistic correctness? Or, to put this another way, just because someone might develop a system where probabilities are important (*e.g.,* our 802.11 transmitter) should probabilities be in the language? It is a difficult question to answer as it is not clear where one should draw the line. For example, in Bluespec nondeterminism is inherent in the language, but its observability remains latent in the current set of tools. In this case, probabilities are entirely foreign to this model and it is not clear how they will affect the system.

Based on our design experience in Bluespec, we think we can express, programmatically, many assertions that we would like to prove about the design. This may require the introduction of extra state and rules, but semantically it requires no new concepts for the user. If these assertions and associated code are syntactically identifiable in the source then it should be straightforward to eliminate them once the design is deemed to be working correctly. Such a method may provide a continuum between "proof by simulation" and proof by formal means. The real technical challenge is how to place restrictions on this verification code so that the decision or semi-decision procedures have a high chance of success. We plan to pursue this line of research in the future.

Acknowledgments

This paper and much of the examples discussed was done as part of the CSAIL-Nokia collaboration. This work was also supported by NSF grant CCF-0541164. We would like to thank Rishiyur Nikhil, George Harper, Srini Devadas, Daniel Jackson, and Pete Manolios for their generous advice on the penultimate draft of this paper. We are also thankful to Carl Sager for clarifying our understanding of the use of Forte tool inside Intel.

References

1. Paulson, L.C.: The Relative Consistency of the Axiom of Choice Mechanized using Isabelle/ZF. London Mathematical Society Journal of Computation and Mathematics **6** (2003)
2. McMillan, K.L.: Symbolic Model Checking: An Approach to the State Explosion Problem. PhD thesis, Carnegie Mellon University (1992)
3. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In: 14th International Conference on Computer Aided Verification (CAV), Copenhagen, Denmark. (2002)
4. de Moura, L., Bjorner, N.: Z3: An Efficient SMT Solver. In: 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary. (2008)
5. Manolios, P., Srinivasan, S.K., Vroon, D.: BAT: The Bit-Level Analysis Tool. In: Proceedings of the 19th International Conference on Computer Aided Verification (CAV), Berlin, Germany. (2007)
6. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th conference on Design automation (DAC), Las Vegas, NV. (2001)
7. Mentor Graphics Corp.: 0-In® Formal Verification `www.mentor.com/products/fv/abv/0-in_fv/`.

8. Synopsys, Inc.: Formality® Equivalence Checker www.synopsys.com/products/verification/.
9. Mentor Graphics Corp.: FormalPro™ www.mentor.com/products/fv/ev/formalpro/.
10. Cadence Design Systems, Inc.: Cadence® Encounter® Conformal® Equivalence Checker www.cadence.com/products/digital_ic/conformal/index.aspx.
11. Jasper Design Automation, Inc.: JasperGold® Verification System www.jasper-da.com/products_jaspergold.htm.
12. Cadence Design Systems, Inc.: Incisive® Formal Verifier www.cadence.com/products/functional_ver/incisive_formal_verifier/index.aspx.
13. Meadows, C.: The NRL Protocol Analyzer: An Overview. The Journal of Logic Programming **26**(2) (1996) 113–131
14. Kwiatkowska, M.Z., Norman, G., Sproston, J.: Probabilistic Model Checking of the IEEE 802.11 Wireless Local Area Network Protocol. In: Proceedings of the Second Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification, (PAPM-PROBMIV), Copenhagen, Denmark. (2002)
15. Burch, J.R., Dill, D.L.: Automatic Verification of Pipelined Microprocessor Control. In: Proceedings of the 6th International Conference on Computer Aided Verification (CAV), Stanford, CA. (1994)
16. Stoy, J.E., Shen, X., Arvind: Proofs of Correctness of Cache-Coherence Protocols. In: Proceedings of FME'01: Formal Methods for Increasing Software Productivity, London, UK, Springer-Verlag (2001) 47–71
17. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI), San Diego, CA. (2002)
18. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Portland, OR. (2002)
19. Russinoff, D.M.: A Case Study in Fomal Verification of Register-Transfer Logic with ACL2: The Floating Point Adder of the AMD Athlon™ Processor. In: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design (FMCAD), Austin, TX. (2000)
20. Seger, C.J., Jones, R., O'Leary, J., Melham, T., Aagaard, M., Barrett, C., Syme, D.: An Industrially Effective Environment for Formal Hardware Verification. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **24**(9) (Sept. 2005) 1381–1405
21. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ Analyzer. In: Proceedings of the 14th European Symposium on Programming (ESOP), Edinburgh, UK. (2005)
22. Berry, G.: The Foundations of Esterel. (2000) 425–454
23. Arvind, Nikhil, R.S., Rosenband, D.L., Dave, N.: High-level synthesis: an essential ingredient for designing complex ASICs. In: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Jose, CA. (2004)
24. IEEE: IEEE standard 802.11a supplement. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. (1999)
25. International Telecommunication Union: H.264. www.itu.int/rec/T-REC-H.264
26. Joshi, R., Lamport, L., Matthews, J., Tasiran, S., Tuttle, M., Yu, Y.: Checking Cache-Coherence Protocols with TLA+. Form. Methods Syst. Des. **22**(2) (2003)
27. Arvind, Shen, X.: Using Term Rewriting Systems to Design and Verify Processors. IEEE Micro **19**(3) (May 1999) 36–46
28. Krstić, S., Jones, R.B., O'Leary, J.: Mothers of Pipelines. Electron. Notes Theor. Comput. Sci. **174**(8) (2007) 7–22

29. Manolios, P.: Correctness of Pipelined Machines. In: Formal Methods in Computer-Aided Design (FMCAD), Austin, TX. Volume 1954 of LNCS., Springer-Verlag (2000)
30. Dave, N., Pellauer, M., Gerding, S., Arvind: 802.11a Transmitter: A Case Study in Microarchitectural Exploration. In: Proceedings of Formal Methods and Models for Codesign (MEMOCODE), Napa, CA (2006)
31. Bluespec, Inc. Waltham, MA: Bluespec SystemVerilog Ver. 3.8 Reference Guide. (Nov. 2004)
32. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, Reading, Massachusetts (1988)
33. Berry, G.: The Esterel v5 Language Primer Version v5_91. (2000)
34. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall PTR, Upper Saddle River, NJ, USA (1997)
35. Dave, N., Ng, M.C., Arvind: Automatic Synthesis of Cache-Coherence Protocol Processors Using Bluespec. In: Proc. of Formal Methods and Models for Codesign, Verona, Italy (2005)
36. : Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language (IEEE Std. 1800-2007)
37. Holzmann, G.J.: The SPIN MODEL CHECKER: Primer and Reference Manual. Addison-Wesley (2003)
38. Kaufmann, M., Moore, J.S., Manolios, P.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Norwell, MA, USA (2000)
39. Owre, S., Rushby, J.M., Rushby, J.M., Shankar, N.: PVS: A Prototype Verification System. In: Proceedings of the 11th International Conference on Automated Deduction (CADE), Saratoga Springs, NY. (1992)
40. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer-Verlag, London, UK (2002)
41. SRI International: Yices. yices.csl.sri.com/index.shtml
42. Jackson, D.: Alloy: A Lightweight Object Modelling Notation. ACM Trans. Softw. Eng. Methodol. **11**(2) (2002) 256–290
43. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, New York, NY, USA (1996)
44. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC), Vancouver, British Columbia. (1987)
45. Archer, Lynch, M.H.L., Mitra, N., Umeno, S., S.: Specifying and Proving Properties of Timed I/O Automata in the TIOA Toolkit. In: Proceedings. Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE), Napa, CA. (2006)
46. Shen, X.: Design and Verification of Adaptive Cache Coherence Protocols. PhD thesis, MIT, Cambridge, MA (2000)
47. Shen, X., Rudolph, L., Arvind: CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems. In: Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing, IEEE Computer Society (1999)