

Eliminating Verification Using Automated Formal Interface Contracts

Rishiyur S. Nikhil

Bluespec, Inc.
200 West Street, Waltham, MA 02451
[Email: nikhil@bluespec.com](mailto:nikhil@bluespec.com)

Abstract

In order to build large complex systems that are robust (correct), we need a scalable and composable discipline. Each module's interface should specify a contract between the implementor and the client—if the client drives the module in certain legal ways, then the implementor guarantees that it will behave in certain well-specified ways. This is an old idea in software engineering, and is now finding its way into hardware design through assertions (e.g., PSL or SVA). However, such an interface contract methodology will truly eliminate the “verification bottleneck” only if the contracts are both automatically derived from implementations and automatically enforced at clients. This is the very essence of the phrase “correct by construction”. In this paper we describe how Bluespec SystemVerilog's *Rules* and *Rule-based Interfaces* implement this idea.

1 Introduction

Due to Moore's Law, today we build entire systems on a chip (SoCs), with tens of millions of gates. The chip development cost is enormous, thereby demanding larger markets to recoup costs, but of course large markets are fewer in number. The cost of chip design failure is not merely the expense of operations and materials for a respin, but the potentially huge opportunity cost in missing or shrinking an already-small market window.

A central factor in chip development costs is the so-called “verification bottleneck”. Today, it is estimated that roughly 70% of development resources go into verification. This is obviously the keystone that we must attack in order to solve the complexity problem.

One approach is to reuse existing or purchased “verified” IP blocks (Intellectual Property) to avoid designing a system from scratch. Unfortunately, interfacing to such IP is itself proving to be a complex and costly problem.

Most of the difficulties in verification and IP reuse can be attributed to the lack of any formal methodology in module composition. One cannot build a large edifice on creaky joints—the whole structure will be fragile and can collapse under the slightest unanticipated stress.

We need a systematic, powerful, formal way of specifying *interfaces* of modules, i.e., the contract between implementor and client. It is immediately obvious that interface semantics must include behavior (signaling and scheduling protocols) and not just structure (types and connectivity). Further, interface behavior is intimately tied to the behaviors of a module and its client, and so interface semantics cannot be considered in isolation—they must integrate organically with module behavioral semantics.

This idea, that a module's interface must specify a *contract* between the implementor and the client, is an old idea in software engineering [5], and is today finding its way into hardware design in the form of *assertions* attached to interfaces [6, 11] in languages such as PSL [8] and SVA, or SystemVerilog Assertions [10].

However, this idea will truly eliminate the verification bottleneck only if it is used *constructively*, i.e., when the contract is derived automatically for an implementation and used while creating the client code so that, by construction, it cannot violate the contract. In this way, it does not run into any of the limitations in the manual insertion of interface assertions: correctness of the assertions, completeness of the assertions, simulation speed, and the need to repeatedly check them at each of possibly hundreds of instances of a module.

Of course, one cannot expect automatically to infer the *complete* interface contract for a module—that is an open-ended problem because it encompasses application semantics, not merely design-language semantics, and is intractable in general. However, the *Rules* and *Rule-based Interfaces* advocated in this paper capture a tractable and highly useful subset of interface semantics that can be used by a synthesis tool in the automated, constructive way just described. First, they provide the designer with a framework and vocabulary for thinking about and describing interfaces that are at a much higher level than the raw *port lists* found in RTL (and even in SystemC). Second, they have a formal semantics that the synthesis tool can capture and represent explicitly. Third, these formal semantics are used by the tool when generating code for client modules to guarantee correct use. Since it captures a subset of interface semantics, it of course does not eliminate the verification problem nor the value of some types of assertions on the interface, but collectively, these properties eliminate the verification *bottleneck* by eliminating most of the common errors found in composing modules in hardware designs (illegal port use, race conditions, inconsistent states, wrong signaling, etc.). This frees the designer to focus on the more important, higher-level verification issues. The ideas advocated in this paper have been implemented and validated in Bluespec SystemVerilog [3].

Organization of this paper

Section 2 is an analysis of current methodology based on today's RTL (Verilog, SystemVerilog and VHDL, and even SystemC), illustrating lack of composability and the way it contributes to the verification bottleneck. Section 3 discusses some partial solutions, including assertions with PSL or SVA. Section 4 discusses how Rules and Rule-based Interfaces take direct aim at the problem. Finally, we summarize and conclude in Section 5.

2 Why current methodology based on RTL leads to the verification bottleneck

Terminology: in the following discussion, we will frequently use the term “IP block” (Intellectual Property block) to refer to a particular module, and the word “client” to refer to the environment (the surrounding module) that uses the IP block.

Module interfaces in RTL are merely port lists. When modules are interconnected, tools merely perform some rather trivial structural checks, e.g., that all ports are connected, that ports are connected to signals of the correct data type/width, etc. VHDL and SystemVerilog [10] have stronger notions of types and type checking, and have facilities to separate interface definitions from module definitions, which alleviate some of the tedium of writing port lists, but in the final analysis, the only semantic checks performed by tools are to ensure correct connection of ports to signals.

The central problem is that interfaces have no semantics of *behavior*, other than the low-level semantics of individual signals. For every module, the designer starts from scratch to roll his own specification for the behavior across the interface. For example, when designing a FIFO block, the designer may decide:

- The logical “enqueue” operation will involve three ports: an input data bus, an output ENQ_READY signal to prevent enqueueing into a full FIFO, and an input ENQ_ENABLE signal which the client drives when it is actually enqueueing some data.
- Similarly, the logical “dequeue” operation will involve three ports: an output data bus, an output DEQ_READY signal to prevent dequeuing from an empty FIFO, and an input DEQ_ENABLE signal which the client drives when it is actually dequeuing some data.
- The logical “clear” operation to empty out the FIFO has a single input CLR_ENABLE signal.

In addition to deciding the ports of the module, the designer will work on the *contract* for correct behavior by the client, involving *Individual Operation Constraints* such as:

1. ENQ_ENABLE should never be asserted if ENQ_READY is false (except see constraints (I) and (II) below).
2. DEQ_ENABLE should never be asserted if DEQ_READY is false.
3. ENQ_ENABLE should be asserted simultaneously with data being valid on the input data bus.

The contract will also have *Multiple Operation Constraints* such as:

- I. ENQ_ENABLE can be asserted even if ENQ_READY is false, if DEQ_ENABLE is asserted simultaneously (because, even though the FIFO is full, the newly enqueued datum can be placed in the slot being vacated by the dequeued datum).
- II. If CLR_ENABLE is asserted, then ENQ_ENABLE and DEQ_ENABLE are ignored (they can be asserted even though the corresponding READY signals are false).

There is actually an interesting implementation choice that affects whether constraint (I) exists. Here, the FIFO designer has chosen to interpret ENQ_READY to mean “not full”, and has left it to the FIFO client to choose to override ENQ_READY in situations where DEQ_ENABLE is also asserted. Another FIFO implementor could make a different choice: to interpret ENQ_READY as “ok to enqueue” and to encode this override *within* the FIFO, i.e., the FIFO itself will assert ENQ_READY whenever DEQ_ENABLE is asserted and the FIFO is full. Thus constraint (I) is eliminated, simplifying the client’s obligation. However, in such an implementation, there is a combinational path inside the module from DEQ_ENABLE to ENQ_READY, and so the client should not have a combinational path outside the module from ENQ_READY to DEQ_ENABLE (this constraint is structural, not behavioral, so many tools will check this)

As a further example of the subtleties involved, consider this. Another FIFO (call it the “bypass FIFO”) could have *exactly the same port list*, but alternate semantics and hence alternate contract constraints:

- III. DEQ_ENABLE can be asserted even if DEQ_READY is false, if ENQ_ENABLE is asserted simultaneously (because, even though the FIFO is full, the dequeued datum will be taken directly from the newly enqueued datum (i.e., the newly enqueued datum is “bypassed” through the FIFO to the dequeue client).

This induces a structural constraint: there is a combinational path inside the module from the “enqueue” input data bus to the “dequeue” output data bus, and so the client should not have a combinational path from the dequeue output data bus to the enqueue input data bus. And, again, the designer could have made an implementation choice to encode (III) within the FIFO, i.e., the FIFO itself will assert DEQ_READY whenever ENQ_ENABLE is asserted and the FIFO is empty. This would induce another structural constraint: there is a combinational path inside the module from ENQ_ENABLE to DEQ_READY, and so the client should not have a combinational path outside the module from DEQ_READY to ENQ_ENABLE.

Such “contract” constraints are typically specified informally in text and waveform diagrams in the data sheet for the FIFO. They are often incomplete or ambiguously written. Sometimes they are plain wrong, i.e., the text spec does not correspond to the actual implementation. In at least one such FIFO data sheet from a major IP vendor, we found these constraints buried in paragraphs spread over several pages.

For each FIFO implementation, if two (or more) processes in the client code need to enqueue into the FIFO, we will also have *Shared Operation Constraints* such as:

- A) Only one process can enqueue, in any particular clock, i.e., there must be some arbitration and selection (muxing) so that only one process can drive the FIFO data input bus and ENQ_ENABLE, together, in any particular clock. (Of course, this is not a verification obligation on the FIFO but on the client, for correct use.)

As seen above, even for a simple, straightforward and familiar block like a FIFO, the behavioral contract can be complex and subtle. It can get quite unmanageable for larger blocks or subsystems with more complex and unfamiliar behavior.

The above activity of the designer is reminiscent of the early days of computer programming where, for each subroutine, the programmer carefully designed a protocol—which arguments were carried in which registers, which arguments were carried in memory, how they were laid out in memory, etc. Another programmer who used the subroutine had to carefully understand this protocol (assuming it was properly documented), and carefully craft his code accordingly. It was worse: the subroutine designer's decisions affected the subroutine user's decisions. For example, choice of register usage inside the subroutine, in turn, affected how registers were used in the client. In other words, if the subroutine were replaced with a functionally equivalent subroutine with different argument and register conventions, the subroutine user may have had to adjust his own code accordingly. Thus, this (lack of) methodology simply did not compose, and simply did not scale. It had a major impact on debugging (verification). It was also very *fragile*, limiting reuse. Today, with high-level languages and compilers, all these issues are removed from the programmer's concern, are never the source of bugs, and never need verification. The compiler systematically produces code that is correct by construction.

The hardware designer using RTL faces the same kind of problem today. For each use of every IP block, he must understand the port protocols assumed by the module designer (assuming they are fully documented). Since each module's port behavior is designed from scratch by the module designer, the IP user has to cope with all the different styles and conventions of different IP designers, i.e., understanding each IP block's port behavior is a completely fresh task. Constraints like (I) above affect the control circuitry in the client module, so that substituting a piece of IP by another which has the same port list and functionality but a slightly different contract also requires changes in the client.

This results in a major lack of scalability in Verification. The designer of module B that uses an instance A1 of IP module A must not only think about the functionality of B (which is the focus of his attention), but he must also think about ways in which B might possibly violate the port behavior contracts of A1. He must devise verification tests for B to ensure that it never drives A in violation of those contracts. This must be repeated for *every* instance of A, e.g., if another module C also uses an instance A2 of module A, then C's verification tests must also include tests that ensure that A2 is not driven in violation of its contracts. The tests for B that try to force A1 into its corner cases are likely to be quite different from the tests for C that try to force A2 into its corner cases. If there are a hundred FIFOs in his design, then these verification checks must be devised and performed separately on each of those hundred instances.

Needless to say, very few engineers have the patience to employ such diligence during block and subsystem tests, and so they often punt the problem to the final “system level” verification. Unfortunately, once we are at system level, it becomes even harder to push some deeply embedded IP blocks into their corner cases.

The complexity of this verification obligation compounds itself as we build up from sub-blocks to blocks to subsystems and systems, i.e., this (lack of) methodology does not *compose* well. Thus, verifying an entire system today is a nightmare, and it is impossible to cover all the corner cases.

To summarize, verification and IP reuse is hard, and increasingly

intractable for large systems, because of the lack of a good formal semantic model for interface behavior, resulting in:

- No reusable methodology:
 - IP designers start from scratch and roll their own interface protocols for each module they design.
 - IP users start from scratch in understanding each IP's peculiar interface protocols.
 - No standard way to document behavior, resulting in missing, incorrect, incomplete, ambiguous, confusing or hard-to-locate documentation.
- A complex verification obligation that is repeated at *every instance* of every module.

A higher-level formal semantics would, instead, allow each IP block to be verified thoroughly on its own, and then guarantee that it cannot be misused in any context. The emphasis shifts to a thorough verification of a block in isolation, instead of the verification of its numerous instantiations in different contexts.

[*Note:* This analysis is orthogonal to the idea of using “behavioral synthesis” to automate the process of going from scientific algorithms to hardware. Such tools are typically restricted to individual blocks and do not address the composition issues in building complex systems out of individual blocks. The blocks generated by behavioral synthesis tools have the same issues of interfacing and reuse. Indeed, the solution using Rule-based semantics recommended in this paper could be a suitable target for the hardware generated by behavioral synthesis tools.]

3 Partial solutions in related work

There are various partial solutions available in some tools and in the industry.

SystemVerilog [10] allows the designer to define *tasks* in an interface. Such a task can represent an entire interface operation (such as an *enqueue* or *dequeue* into a FIFO), and can encapsulate all the specific port-signaling protocols. Client modules, instead of directly signaling ports, invoke these tasks instead. This allows the designer to define the behavior of transactions on a module in one place, in the interface itself. However, these are just layered on top of port lists, and explained in terms of inlining the tasks wherever they are called, i.e., they do not introduce any fundamental new formal model of behavior on the interface. They are also weak in that if the ports are *shared*, i.e., the task is to be called from multiple concurrent processes, then they do not provide any straightforward way to deal with the necessary multiplexing and arbitration.

The SPIRIT Consortium [9] is attempting to define standards for IP reuse by using XML-based descriptors of IP blocks. But, once again, these are just layered on top of standard RTL and RTL's port lists. It does not introduce any fundamentally improved formal model of behavior.

SystemC uses the concept of *methods* from C++ to describe interfaces as transactions rather than signals. However, *synthesizable* SystemC still relies on RTL-like signals for communicating between modules. But even for modeling purposes, where we may not be interested in synthesizability,

C++'s methods just give an open-ended mechanism, not a semantics or methodology. It is hard to see how any tool can usefully exploit this higher level of abstraction to do any static verification.

The most promising partial solution is the use of *assertions* using PSL [8], SystemVerilog Assertions (SVA) [10], or OVL [1] at interfaces to express correctness conditions on the port protocols [6, 11]. The idea is to attach immediate and temporal assertions to interfaces, using one of the above high-level, declarative, logic-based languages to describe formal correctness properties of the interactions at the interface. Immediate assertions are correctness properties that must be true always, i.e., on every clock. Temporal assertions are correctness properties relating events that may span time (e.g., that an ACK signal must always follow a REQ signal within a certain maximum number of clock cycles).

For our FIFO example of Section 2, all the constraints (1)-(3), (I)-(III) and (A) would have to be expressed in the chosen assertion language.

However, assertions are still only a partial solution, for the following reasons:

- The onus remains on the IP block designer or verification engineer to create necessary and sufficient assertions to correctly and completely characterize the port protocol contracts. Achieving this level of precision is very hard for all but the simplest of behaviors. The designer often finds it hard to translate his intent into a watertight formal logic statement. Constraints (1)-(3), (I)-(III) and (A), when expressed in PSA/SVA/OVL, may be comparable in size to the FIFO implementation itself!
- In the current state of the art, assertions are checked by simulation. This affects overall simulation speed and, in addition, one still has the difficult problem of designing testbenches that will drive internal IP blocks into their corner cases, in order to ensure complete coverage, i.e., have all the assertions been exercised sufficiently to catch all the corner cases?
- In the future, assertions will increasingly be checked statically, using theorem proving, so that coverage will be complete and simulation speed is not affected. However, because assertions are a very powerful language, progress towards this goal is likely to be slow and will take many years. Even assuming that these technologies were available, the (manual) work to add assertions will always be time-consuming and error-prone, and likely to be incomplete.

The question is: is there a more limited style and scope of assertions that is adequate for the job and known to be more tractable than general-purpose assertions? The Rule-based semantics advocated below are an answer.

In summary: while there are many partial steps towards alleviating the verification bottleneck, the lack of a suitable high-level, compositional, formal behavioral semantics is hindering true progress.

4 Rules and Rule-based Interfaces address the correctness problem automatically and scalably

Rule-based systems have a long history in the study of semantics, parallelism, and concurrency in Computer Science. There is a rich body of literature on *Term Rewriting Systems* [2, 12] which are the basis of Rules. Rule-based systems have been central to the study of complex systems in Artificial Intelligence [13] and the study of complex parallel programming [4]. All this work is focused on correctness, which is the central objective of verification.

The power of Rules in reasoning about correctness arises from its property of *atomicity* [7]. A rule can express complex, dynamically determined state transitions based on complex dynamic conditions. Nevertheless, atomicity allows the designer to reason about correctness one rule at a time. For each rule, assuming that the system starts in a consistent state, we can check if the rule's state transition leaves the system in a consistent state—atomicity allows us to not have to worry about any other concurrent activity. When the correctness of individual rules is established, the composition of these rules is automatically correct. This makes it scalable to large, complex systems.

The concept of atomicity was the fundamental breakthrough in managing complex concurrency in the software field (operating systems, databases, transaction-processing systems and distributed systems). Rules and Rule-based Interfaces bring the same power to an HDL (Hardware Design/Description Language). It is this formal semantic model that allows us to specify concurrent behavior simply, precisely and formally which, in turn, enables a *scalable* methodology to build complex hardware systems correctly, by construction (i.e., eliminating significant aspects of the verification task).

Referring to the FIFO example of Section 2, instead of port lists, we define interfaces using *rule-based interface methods*, as shown below, using some small extensions to the notation of SystemVerilog:

```
interface FIFOBuf#(type x_t);
    method Action      enq    (x_t  x);
    method ActionValue#(x_t) deq  ();
    method Action      clear ();
endinterface
```

[Note: the *(type x_t)* construct is just SystemVerilog's notation for type parameterization (also known as polymorphism, or genericity), i.e., *x_t* is a type variable representing the type of items stored in the FIFO.]

The *enq* method encapsulates all the ports described in Section 2: the input data bus (whose width depends on the particular data type to which the generic type *x_t* is instantiated), the output ENQ_READY signal, and the input ENQ_ENABLE signal. Similarly, the *deq* method encapsulates all the ports described in Section 2: the output data bus, the output DEQ_READY signal and the input DEQ_ENABLE signal.

In general, method arguments become module input data bus ports. Method results (such as that returned by *deq*), become output data bus ports. A method can have multiple output data bus

ports because return-types can be structs (records) with multiple fields, and vectors. All methods have an output READY signal. All Action and ActionValue methods (like those shown) have input ENABLE signals. Action and ActionValue methods are sequential, i.e., they can cause a state change inside the module. A third kind of method, which we call *value* methods (not shown in this example), are purely combinational—their results are combinational functions of their arguments and internal module state. The compiler optimizes away READY and ENABLE signals of a method if it proves that they are always asserted.

A client module that uses the FIFO contains Rules that operate the *enq* and *deq* methods, as in the example below.

```

module mkClient (...);
  ... instantiate fifo ...

  rule upstream (... cond1 ...);
    ... other actions ...
    fifo.enq (expr1);
  endrule

  rule downstream (... cond2 ...);
    x <- fifo.deq ();
    ... other actions ...
  endrule
endmodule

```

Each rule has an *explicit condition*, depicted above as the expressions *cond1* and *cond2*. These are pure combinational boolean expressions. Each rule also contains one or more *actions* that can be executed atomically only if the rule condition is true. For example, the *upstream* rule contains an action that enqueues the value of expression *expr1* into the FIFO, and the *downstream* rule contains an action that dequeues an item *x* from the FIFO.

The conditions of all methods operated by a rule are incorporated into the overall condition of the rule. For example, the ENQ_READY signal is “AND”ed with *cond1* to determine the overall condition of the *upstream* rule. The DEQ_READY signal is “AND”ed with *cond2* to determine the overall condition of the *downstream* rule.

A rule can only fire (execute) if all its conditions permit. When it fires, all its actions, including all the actions in all the methods that it operates, are executed simultaneously as one composite atomic action. Thus, the *upstream* rule can only fire if ENQ_READY is true, and then the enqueueing action becomes part of the overall atomic action of the rule. When the rule fires, the enqueued data is driven and ENQ_ENABLE is asserted.

The condition of a method or a rule is necessary, but not sufficient, for a rule to fire. In particular, since rules can share resources (such as the FIFO above), simultaneous firing might not be possible while maintaining atomicity, i.e., if simultaneous firing would lead to inconsistent states. The compiler emits *scheduling* logic to ensure that simultaneous firing is only possible if it maintains atomicity.

When compiling a FIFO implementation, the compiler performs a systematic analysis that infers whether the *enq* and *deq* methods can be operated simultaneously safely, and under what conditions. Note, different FIFO designs may or may not permit such simultaneous operation.

This interface information is recorded by the compiler with the FIFO implementation. Then, when compiling *mkClient*, the compiler uses this information to introduce suitable control logic in *mkClient* to guarantee that the *upstream* and the *downstream* rules can fire simultaneously only when conditions permit them to do so safely.

Suppose *mkClient* had two processes that needed to enqueue into the FIFO, i.e., suppose it had another rule, like this:

```

rule upstream_also (... cond_also ...);
  ... other actions ...
  fifo.enq (expr_also);
endrule

```

Because of rule atomicity, the compiler automatically generates all the logic to ensure that either rule *upstream* or rule *upstream_also*, but not both, drive the FIFO's input data bus and ENQ_ENABLE ports in any particular clock. If only one of the conditions *cond1* or *cond_also* is true, then the corresponding rule gets to enqueue. If both are simultaneously true, one of the rules gets priority (the programmer can control which rule gets priority).

The above discussion shows how interface method semantics fit organically into the semantics of the rules that invoke them. Interface method conditions are integrated into rule conditions. Interface method actions become part of rule actions, with the same semantics of atomicity. It is for these reasons that we say that interface methods are *Rule-based*, i.e., methods are simply parts of rules, and a rule can be viewed as a *composition* of the methods it operates.

In particular, the compiler automatically *derives and enforces* all the contract requirements (1)-(3), (I)-(III) and (A) described in Section 2. The compiler, *by construction*, ensures that the design meets all the contract requirements wherever the FIFO is instantiated, as in *mkClient*. It is therefore impossible for *mkClient* to drive the FIFO into an inconsistent state. Similarly, in addition to *mkClient*, the design may contain hundreds of other instances of the FIFO module, but in each case the compiler ensures that the FIFO's contract is met.

Further, these assurances are transitive, so that *mkClient* can truly be viewed as a black box. Its interface contract, in turn, is automatically derived by the compiler. If *mkClient* is itself instantiated repeatedly, the compiler ensures that its contract is met in each case. Thus, we are guaranteed that none of them can drive their FIFO instances into inconsistent states, no matter what their environments. In other words, the interface contracts for *mkClient*, in turn, ensure that the contracts for the FIFOs are met. All these assurances are ensured *statically* by the compiler, before simulation.

Thus, these properties of Rules and Rule-based interfaces dramatically simplify the Verification problem. A whole host of

functional errors that are very common in RTL designs—signaling errors, signal sampling errors, race conditions, and so on—almost all of which can be traced to a failure of atomicity, are eliminated, by construction, when the behavior is expressed in terms of Rules and Rule-based Interfaces. In summary, the benefits are:

- Rule-based interface methods are a higher-level of abstraction than raw ports.
- Action, ActionValue and value methods provide a simple and common vocabulary and model for module interactions and signaling. Instead of the tedium of rolling one's own signaling protocol, timing diagrams and assertions for each module, the implementer just uses one of these off-the-shelf mechanisms. Similarly, the user of an IP block, instead of struggling to interpret ad-hoc interface descriptions, can instantly understand its signaling protocol if its interface is expressed in these terms. The user is also relieved of the tedium of designing the control logic for individual and shared interfaces.
- Because of Rules and Rule-based Interfaces, the compiler automatically infers the complete signaling and scheduling contract exported at the interface of an IP block, and the compiler automatically ensures that this contract is met at every instance of the IP block, when it generates the surrounding contextual code. Because these assurances apply at every level of the module hierarchy, the entire system is a more robust edifice built on a strong and stable foundation.

This approach enables the designer to rapidly put together large systems with complex concurrency. It changes the focus in the overall flow: during system-level verification, it eliminates the need to drive all internal blocks to their respective corner cases. Instead, the focus is on the larger and more important questions of system-level architecture and features.

Experience and Validation

The ideas presented in this paper have been implemented in the Bluespec SystemVerilog system and compiler [3], and have been in use for several years. Bluespec, its customers and university partners have implemented well over a hundred designs with this methodology, for both ASIC and FPGA, with sizes from few thousand gates to 15M-20M gates. There is now substantial empirical evidence supporting the claims of dramatically lower verification effort, higher designer productivity, and IP reuse. There is also substantial evidence showing no performance penalty, in area or speed, compared to coding in RTL directly, for expressing the design using Rules and Rule-based interfaces.

5 Summary and Conclusion

Interface semantics are very weak (practically nonexistent) when working with raw module port lists in RTL (and synthesizable SystemC). The ensuing weakness of compositional semantics is at the root of the “verification bottleneck”—as we build larger and more complex systems, they become increasingly creaky because of the weak “joints” in the system.

With RTL semantics, even with a focused verification effort at the IP block level, you’ve only succeeded in establishing its correctness under proper use. But, this proper use assumption

cannot be made upon future instantiation—an IP block’s proper behavior is dependant on external logic as well as internal logic. Because so many errors can be introduced when blocks are instantiated, the benefit of focusing verification at the IP block level gets blunted – and, instead shifts primarily to the system level, where verification teams must consciously, but indirectly, steer the internals of a design into all its potential corner cases.

Strong formal interface semantics are thus a necessity to eliminate the bottleneck. Assertions are a good step, but have their limitations—the work to add assertions is time-consuming and error-prone, and likely to be incomplete; checking assertions adds simulation overhead and can be difficult to ensure coverage; and formal verification of assertions in all their generality is intractable.

Rule and Rule-based Interfaces provide a powerful, high-level semantic model for behavior and interaction. They provide a high-level vocabulary and simple semantic model for designers in creating and understanding module interfaces. Compilers can automatically derive contracts for exported module interfaces, and can automatically use this information to ensure correct usage at module instances. All this has no performance penalty compared to directly writing RTL. As a result, a major class of errors and bugs in large system design is eliminated, by construction.

6 References

- [1] *Open Verification Library (OVL) 1.0*, Accellera, <http://www.accellera.org/activities/ovl/>, 2005.
- [2] *Term Rewriting and All That*, F. Baader and T. Nipkow, Cambridge Univ. Press, 1998, 300pp.
- [3] *Bluespec System Verilog Reference Guide*, www.bluespec.com, 2004-2006
- [4] *Parallel Program Design: a Foundation*, K. Mani Chandy and J. Misra, Addison Wesley, 1998, 516pp.
- [5] *Design by Contract*, Wikipedia, http://en.wikipedia.org/wiki/Design_by_contract. (See also Eiffel Software, <http://www.eiffel.com>)
- [6] *Assertion-Based Design, 2nd Ed.*, H. D. Foster, A. C. Krolnik and D. J. Lacey, Springer, 2004, 414 pp.
- [7] *Atomic Transactions: In Concurrent and Distributed Systems*, N. A. Lynch, M. Merritt, W. E. Weihl and A. Fekete, Morgan Kaufman Series in Data Management Systems, 1993, 476 pp.
- [8] *PSL—Property Specification Language*, IEEE Std 1850, <http://standards.ieee.org>, September 2005
- [9] *The SPIRIT Consortium* for industry level cooperation in developing standards for IP description, <http://www.spiritconsortium.org>
- [10] *SystemVerilog—Unified Hardware Design, Specification, and Verification Language*, IEEE Std 1800-2005, <http://standards.ieee.org>, November, 2005
- [11] *Assertion-Based Verification*, Synopsys, Inc., http://www.synopsys.com/products/simulation/assertion_based_wp.pdf, March 2003, 14 pp.
- [12] *Term Rewriting Systems*, Terese, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 2003, 884 pp.
- [13] *Artificial Intelligence, Third Edition*, P. H. Winston, Addison Wesley, 1992, 691pp.