Debugging Bluespec Designs via Equivalence Checking

Nirav Dave*, Michael Katelman†
* Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139, U.S.A.
ndave@csail.mit.edu
† Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801 USA
katelman@uiuc.edu

We have argued previously [1] that the best way utilize formal methodologies for hardware design and verification is to focus on high-level debugging tools that let the designer think of the verification problem in the same way that he considers the design problem. One key problem with this idea is that while almost all design specifications are nondeterministic to allow designers more freedom in generating high-performance and efficient design, design languages in which designers think of the problem almost all represent the design as a deterministic state machine.

Our work focuses on hardware designed using Bluespec, a language of guarded atomic actions (or *rules*) where the rules can fire nondeterministically. Any rule whose predicate guard is true can be executed. The Bluespec compiler generates efficient synchronous hardware by automatically selecting (or with some manual assistance) the rule execution strategy [2][3].

In Bluespec one can represent a high-level specification of a design using a small number of large rules. For implementation purposes, however, designers may split specification-level rules into collections of smaller rules. For example a specification-level rule A may be split into a set of rules $A_1, \ldots, A_n$. Logically, the designer wants his system to behave as if all of the A_i rules fire one after the other, but to achieve higher parallelism, and better performance, this may not be desirable for the final design. The correctness of an implementation is established by showing that (a) $A_1, \ldots, A_n$ behaves as A, and (b) that it is possible to simulate the refined system, where $A_1, \ldots, A_n$ may fire in any order, with the one where the A_i's only fire in sequence.

We have developed a tool which poses this simulation question to help designers verify that their refinements are correct. To use the tool, the user must provide (a) the sequence of rules $A_1, \ldots, A_n$ resulting from a split, and (b) a Bluespec predicate which signifies when there are no partial transactions of the original A transaction currently in-flight. Both pieces of information are natural for designers to understand deeply enough that this is not a significant imposition.

With this information we automatically generate a series of questions which interactively prove that all finite sequences in the refined system have a corresponding, equivalent sequence when we constrain the firing of the A_i's to always occur in sequence. We exploit the predicate to drastically prune the search space by let us reason about "flushed" starting states.

This tool can be used in two modes. One, the tool can be used to establish bisimulation equivalence between (a) the partitioned system and (b) the restricted system where the "atomic" rule sequences are always executed without interruption. The second mode of operation is a sort of *dual* to the equivalence checking of the first; the tool applies heuristics to quickly show that the two designs are *not* bisimilar. Both modes are similar, but the second can be applied as a quick check without having to wait for the full bisimulation check to successfully complete. In either case, the imposition on the user is the same: the user must provide (a) a set of sequences of rules that constitute transactions, and (b) a predicate characterizing when transactions are in-flight. From this information we automatically generate a candidate bisimulation relation where states that have partially completed transactions get "flushed" by applying sequences of rules that are proper suffixes of the provided transactions.

Internally, our tool uses symbolic simulation to generate logical formulas representing the diagrams which must commute for bisimulation. Our current symbolic simulator reads an intermediate format output by the Bluespec compiler and generates constraints for the bit-vector SMT solver STP [4]. We are able to symbolically execute bounded rule sequences having an arbitrary or fixed schedule and operating from a symbolic, or constrained symbolic, initial state. Our tool is currently able to handle medium-sized designs, including a relatively realistic 6-stage pipelined MIPS microprocessor refined from a 4-stage design. Currently, proving correctness of the refined design and finding a bug in a incorrect version takes on the order of a few hours and 30 minutes, respectively. We believe that with modest improvements to the compiler these times can be reduced to the point that they are well within the designer's delay tolerance.

References

[1] Arvind, Nirav Dave, and Michael Katelman. Getting formal verification into design flow. In *Lecture Notes in Computer Science, FM 2008: Formal Methods Volume 5014/2008 pp.12-32*, 2008.

[2] Nirav Dave, Arvind, and Michael Pellauer. Scheduling as Rule Composition. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Nice, France, 2007.

[3] Thomas Esposito, Mieszko Lis, Ravi Nanavati, Joseph Stoy, and Jacob Schwartz. System and method for scheduling TRS rules. United States Patent US 133051-0001, February 2005.

[4] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, pages 519–531, 2007.