Abstraction as a Practical Debugging Tool

Sandip Ray
University of Texas at Austin
sandip@cs.utexas.edu
http://www.cs.utexas.edu/users/sandip

With the increasing size and complexity of hardware designs, verification of practical hardware systems is becoming increasingly challenging. Formal analysis techniques based on state exploration are limited by the well-known *state explosion* problem. In many cases, the size of a hardware design can even limit the effectiveness of simulation since a significant number of cycles may be required to reach a given boundary condition, even under user-directed input biasing. A common method of attacking this state explosion is to simply reduce the number of state elements in the design of interest by redefining the design with smaller registers, for example by reducing a 64-bit datapath to a 2-bit datapath. The resulting smaller model is then analyzed (through formal or simulation-based techniques) with the goal to discover errors in the original design. However, a key problem with this approach is that it is often difficult to correlate the result of the analysis of the simpler model with the original design.

One solution to this problem is to ensure, through a refinement mapping, that the design can be abstracted to small-state model that provides a conservative approximation. There has been work on formally verifying such correlation between a design and its abstraction, often with a theorem prover. However, in practice it can be a cumbersome process to come up with a sufficient abstraction and subsequent proof of correctness. Additionally, both the abstraction and proof may require substantial edition in order to maintain the correspondence with an evolving hardware design.

We present a procedure to ameliorate the above issue by automatically constructing abstract models of hardware designs modeled at the RTL level. Given an input design $\mathcal{D}$ and a desired observation $\mathcal{O}$ on $\mathcal{D}$, the procedure generates an abstraction $\mathcal{A}$ of $\mathcal{D}$ that is guaranteed to be a conservative approximation of $\mathcal{D}$ with respect to $\mathcal{O}$. In addition, the procedure is parameterized by the amount of approximation desired: by varying this parameter, it is possible to generate a *range* of models, each of which is guaranteed to be an abstraction of $\mathcal{D}$ but which vary on the degree of abstraction and the number of possible states.

Our algorithm for generation of abstraction involves an iterative procedure that "propagates" the necessary observations along the logic of the underlying design from the current state to the previous state. We demonstrate the process below using a simple pipeline example. Assume that the pipeline has three integer-valued latches s_0, s_1, and s_2, that linearly transmits the value of its input i. Schematically, we write the state transition function of the pipeline as $P(s,i) \triangleq [i, s_0, s_1, s_2]$. Suppose the observation of interest is the property $\mathit{good}(s) \triangleq (s_2 > 0)$. The iterative process then begins with the abstraction mapping the state s to the singleton set of abstraction predicates $f_0(s) \triangleq \{\mathit{good}(s)\}$. In each iteration we refine the abstraction by propagating the requirements on a state to the previous state in the transition. In the example, this produces the sequence following sequence of refinements.

$$f_0(s) \triangleq \{(s_2 > 0)\}$$
$$f_1(s) \triangleq \{(s_2 > 0), (s_1 > 0)\}$$
$$f_2(s) \triangleq \{(s_2 > 0), (s_1 > 0), (s_0 > 0)\}$$

The resulting model has state elements corresponding to the members in f_2 and the input mapped to $(i > 0)$.

The approach above apparently resembles a cone-of-influence algorithm. The difference with cone-of-influence arises from the need to generate abstractions that account for interpretation of the different operator semantics. In particular, the procedure handles the semantics of arithmetic functions, bit vectors primitives, and arrays, as well as user-defined modules and functions built on top of such primitives.

Abstract interpretation has been a key component in formal verification research. It forms the basis of many scalable model checking techniques, for instance counterexample-guided refinements, predicate abstraction, etc. A key difference between these approaches and our procedure is our emphasis to support scalability, both in formal verification and in reduction of simulation cycles. The orthogonal requirements of simulation and formal state exploration introduce design trade-offs in the algorithm for constructing the abstraction. For instance, although the model constructed for the example above is a *complete* abstraction, completeness is often not desirable in practice. One reason is that a complete abstraction often retain a prohibitive amount of state information from the original design. Furthermore, determinism in abstraction generation is also undesirable since it is critical to generate a range of abstractions, each defining a different search space with varying degrees of state reduction and approximation to the original design.

We have implemented the procedure above as a tool inside the ACL2 theorem prover, and applied it to abstract several RTL designs. The procedure is parameterized over the set of RTL operators. One key application of the tool is in abstracting pipelined transaction queues, which are ubiquitous in implementations of communication protocols, and are known to be notoriously difficult to abstract manually. Practical experiments show that the procedure is scalable for abstracting large-scale RTL modules.