Abstraction as a Practical Debugging Tool

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

***Abstract*—We present a procedure for automatically constructing an abstract state model of a hardware design using the definition of the state transition function for the design and a description of the set of observations to be preserved in the abstraction. The procedure iteratively constructs the abstract model by refining a mapping from the states of the original design to the states of the abstraction. The resulting abstraction is guaranteed to be a conservative approximation of the design. We discuss our implementation of the procedure and different design trade-offs involved in making it effective.**

I. Introduction

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 [1], 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.

Abstract interpretation has been a central component in formal verification research since its introduction by Cousot and Cousot in 1977 [2]. It forms the basis of many scalable model checking techniques, for instance counterexample-guided refinements, predicate abstraction, etc. Indeed, our algorithm appears on the surface to be close to well-known techniques for generating predicate abstraction, for example those proposed by Namjoshi and Kurshan [3]. 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, in our context, it is often not desirable that $\mathcal{A}$ be a *complete abstraction* of $\mathcal{D}$ with respect to $\mathcal{O}$. 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. Our procedure is carefully designed with consideration for the trade-offs necessary for its practical utility.

The remainder of the paper is organized as follows. In Section II we introduce preliminary notations and definitions. In Section III, we first introduce a basic abstraction generation algorithm and then successively refine it into one that is usable in practice. In Section IV we discuss how to instantiate the algorithm with RTL operations. We conclude in Section V.

II. Preliminaries

$\mathbb{N}$ denotes the set of natural numbers $\{0, 1, \ldots\}$, and $\{i, \ldots, j\}$ denotes the set of natural numbers between i and j (inclusive). $\mathbb{B}$ denotes the set of Booleans $\{true, false\}$. For set X, $|X|$ denotes the cardinality of X. We use $f(x_1, x_2, \ldots) \triangleq e$ to declare a function named f, with parameters $x_1, x_2, \ldots$, as the expression e with free variables $x_1, x_2, \ldots$. The name λ denotes an anonymous function. For functions f and g, we use $f.g$ to denote their composition: that is, $(f.g)(x) \triangleq f(g(x))$. For a surjective function $f : D \rightarrow R$, the function $\hat{f} : R \rightarrow D$ satisfies for all $r \in R$, $f(\hat{f}(r)) = r$. We define a preorder $\sqsubseteq$ on functions by $(f \sqsubseteq g) \triangleq (\forall x, y : (g(x) = g(y)) \Rightarrow (f(x) = f(y)))$. Informally, $f \sqsubseteq g$ implies

that g preserves more information from its domain than f. We also define the strict order $(f \sqsubset g) \triangleq (f \sqsubseteq g) \wedge (g \not\sqsubseteq f)$.

For any set X, X^* is the set of finite sequences of elements of X. We use $\langle x_1, x_2, x_3, \ldots, x_n \rangle$ to denote the sequence x of length $|x| = n$. The empty sequence is denoted by $\langle\rangle$; for nonempty sequence x, x_+ denotes the sequence $\langle x_2, x_3, \ldots, x_n \rangle$; for sequences $x$and y, $x||y$ denotes the sequence $\langle x_1, x_2, \ldots, x_{|x}, y_1, y_2, \ldots, y_{|y|} \rangle$.

Sequential hardware designs semantically reduce to state models which can be represented by a tuple $\langle S, I, P \rangle$, where S is the set of states, I is the set of inputs, and $P : S \times I \rightarrow S$ is the next-state function that takes the current state and current input and returns the next state.

Invariant checking may now be formulated as follows. Given a state modem $M \triangleq \rangle S, I, P \rangle$ and a surjective predicate $\mathit{good} : S \rightarrow \mathbb{B}$, show that good is invariantly true for all states reachable from some initial state s of S. This is denoted by $M, s \models \mathit{good}$, defined as follows.

$$\mathit{run}(s, k, P) \triangleq \begin{cases} s & \mathit{if}\ k = \langle\rangle \\ \mathit{run}(P(s, k_1), k_+, P) & \text{otherwise} \end{cases}$$

$$(s \rightsquigarrow_M t) \triangleq (\exists k \in I^* : t = \mathit{run}(s, k, P))$$

$$(M, s \models \mathit{good}) \triangleq (\forall t \in S : (s \rightsquigarrow_M t) \Rightarrow \mathit{good}(t))$$

We leave the definition of good implicitly assumed for model M for the remainder of the paper. The pair of functions $\Phi \triangleq \langle \Phi_S, \Phi_I \rangle$ is termed an *abstraction witness* if there exists a model $M_\Phi \triangleq \langle S', I', P' \rangle$, such that $\Phi_S : S \rightarrow S'$, $\Phi_I : S \times I \rightarrow I'$, $\mathit{good} \sqsubseteq \Phi_S$ and $\forall s, i : \Phi_S(P(s, i)) = P'(\Phi_S(s), \Phi_I(s, i))$. A model M' is an abstraction of M if it is M_Φ for some abstraction witness Φ. If the input mapping Φ_I is independent of the first state parameter, (i.e., $\forall s, t, i : \Phi_I(s, i) = \Phi_I(t, i)$), then Φ is termed a *complete abstraction*. The key property about the abstraction witness Φ is that invariant check can be transferred.

Theorem 1: For an abstraction witness Φ, if $M_\Phi, \Phi_S(s) \models (\mathit{good}.\hat{\Phi}(s))$ then $M, s \models \mathit{good}$.

Theorem 2: For an abstraction witness Φ, $M_\Phi, \Phi_S(s) \models (\mathit{good}.\hat{\Phi}(s))$ if and only if $M, s \models \mathit{good}$.

Before discussing how to generate effective abstractions, we first point out two trivial ones. First, M is a complete abstraction of itself through the abstraction witness $\Phi(s) \triangleq s$ and $\Phi_I(s, i) \triangleq i$. Second, the model $\langle \mathbb{B}, \mathbb{B}, P'(s, i) \triangleq i \rangle$ is an abstraction through the witness $\Phi_S(s) \triangleq \mathit{good}(s)$ and $\Phi_I(s, i) \triangleq \mathit{good}(P(s, i))$. These two abstractions form the opposite ends of the a spectrum of abstractions, where on the one end we have M (with low degree of approximation and low degree of state reduction) and on the other end we have M' with very high degree of approximation and very high degree of state reduction. The goal of abstraction generation is to find abstract models between these two extremes, with an effective balance of approximation and state reduction. As the definition of M' hints, given any $f : S \rightarrow S'$ with $\mathit{good} \sqsubseteq f$, it is possible to build an abstraction $M_\Phi \triangleq \langle S', S', P'(s, i) \triangleq i \rangle$, with $\Phi_S(s) \triangleq f(s)$ and $\Phi_I(s, i) \triangleq f(P(s, i))$. In the generation of abstractions, we focus on the iterative generation of an abstraction witness Φ which achieves the desired level of state reduction while introducing the least amount of approximation. Once the Φ is determined, we modify the definition of P to update the necessary components of state that affect Φ.

III. Generic Abstraction Generation Algorithm

Our procedure is iterative. In each iteration, we take an abstraction mapping and "propagate" 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 $P(s, i) \triangleq \langle i, s_0, s_1, s_2 \rangle$. 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 $(i > 0)$.

Notice that the state mapping is refined at each step in the sequence, i.e., for all j, $f_j \sqsubseteq f_{j+1}$. In the final step the mapping converges and we set our abstraction witness Φ to be $\Phi_S(s) \triangleq f_2(s)$ and $\Phi_I(s, i) \triangleq (i > 0)$. Note that in this case Φ is a complete abstraction. However, it is not desirable in general to construct a complete abstraction, since it might necessitate preserving too much state information from the original system. In some cases, additional inputs or nondeterminism is useful to hide state information that is irrelevant to the invariant. Furthermore, the determinism in the generation of Φ in this example is also sometimes undesirable in practice, since we would like to generate a range of possible abstractions, each defining a different search space for failures in the invariant check with varying degrees of state reduction and approximation to the original model.

In order to define the abstraction propagation process, we need some functions with specified properties that serve as parameters for the procedure we outline. We assume that the state transition function P is defined by a composition of operators taken from a set of primitive operations Ops. The restriction to unary operations here is merely for presentation reasons, but does not lose generality. Different design languages define different sets of primitive operations. In case of RTL designs of sequential hardware, the primitive operations include bit-string manipulation, and arithmetic and logical combinators.

Given the set Ops, the function P is syntactically a finite sequence of operations in Ops^*. The semantics $\overline{P}$ of a sequence of P will be defined recursively as follows.

$$\overline{P} \triangleq \begin{cases} (\lambda(x) \triangleq x) & \textit{if } P = \langle\rangle \\ P_1.\overline{P_+} & \text{otherwise} \end{cases}$$

We assume the definition of a set of surjective abstraction functions Abs which define the supported reduction of state information. We also assume the associative and commutative operators *meet* ($\sqcap$) and *join* ($\sqcup$) on Abs, where for all α, β, $(\alpha \sqcap \beta) \sqsubseteq \alpha \sqsubseteq (\alpha \sqcup \beta)$. Propagation of abstraction is then formalized by a function $\mathit{prop} : \mathit{Ops} \times \mathit{Abs} \times \mathbb{N} \rightarrow \mathit{Abs}$, which transfers an existing abstraction from the output of a primitive operation to its input. The extra $\mathbb{N}$ parameter for prop is provided to allow the selection from among a set of possible propagations. The extension of prop to a sequence of operations is given by the function $\mathit{SP} : \mathit{Ops}^* \times \mathit{Abs} \rightarrow \mathit{Abs}$ as follows, where $\mathit{random}()$ is a random natural number generator.

$$\mathit{SP}(P, \alpha) \triangleq \begin{cases} \alpha & \textit{if } P = \langle\rangle \\ \mathit{SP}(P_+, \mathit{prop}(P, \alpha, \mathit{random}())) & \text{otherwise} \end{cases}$$

What should be the requirement on SP? We formalize the requirement by the notion of *transfer* as follows. A function $G(f, \alpha, n)$ is a *transfer* if for all f, $\alpha \in \mathit{Abs}$, and n, we have $(\alpha.\overline{f}) \sqsubseteq \overline{G(f, \alpha, n)}$. Informally, a function is a transfer if it returns an abstraction function at the input of f that preserves the information in α at the output of f.

Theorem 3: If $\mathit{prop}(op, \alpha, n)$ is a transfer then $\mathit{SP}(P, \alpha, n)$ is a transfer.

The theorem ensures that the abstraction returned by SP for the input of P preserves the goal abstraction at the output of P. We assume in the sequel that prop is a transfer. In practice this requirement may be too strong and we will discuss the points at which we break this requirement.

A. Extending with Context

The definition of SP above did not take into account the current context at the inputs of an operation in determining how to propagate the abstraction. For instance, assume that Ops contains the equality operation "=",[1] and in some iteration, SP encounters the operation $(x = y)$ such that the current goal at the output of the operation is the identity mapping. Given no additional information other than that prop is a transfer, we must return an identity mapping for x and y. But if we know that y can only take the value 42 then we can propagate the goal to the mapping that preserves only te value 42 but coerces other values of x to 0. Thus, having information of current abstractions and possible values of the inputs of an operator can be useful in computing effective abstractions. In our implementation of SP, we compute these contexts using an abstraction "evaluator" of operators, which iteratively computes the abstraction at the output of an operation given an abstraction of its inputs. We extend the definitions of prop and SP with an additional parameter for the current mapping or context, leading to the following modified definition of SP.

$$\mathit{SP}(P, \alpha, c) \triangleq \begin{cases} \alpha & \textit{if } P = \langle\rangle \\ \mathit{SP}(P_+, \mathit{prop}(P_+, g, \mathit{BC}(g, P_+, c)) & \text{otherwise} \end{cases}$$

where

$$g \doteq \mathit{prop}(P_1, \alpha, \mathit{random}(), \mathit{FC}(P_+, c))$$

Here the function FC (forward context) computes the evaluation of the current context c, and the function BC (backward context) refines the current state context based on the eventual propagation of goal g through P_1. In the implementation, BC only modifies the current state context when a goal is propagated to a variable or signal whose value is shared among different operations; thus, the foal propagated back from the operation augments the context for propagating a different goal for a different operation. Note that even though the extension of SP with context is key to the effectiveness of abstraction generation, it is essentially heuristic.

B. Iterative Refinement

Given SP, we now define an iterative procedure AbsGen to compute abstractions by iteratively refining an abstraction mapping. For pedagogical reasons, we first define a simpler function $\mathit{AbsGenComplete}$. To do so, we assume that a function α which maps a state-input pair $\langle s, i\rangle$ can be decomposed into the definition $\alpha(\langle s, i\rangle) \triangleq \langle \alpha_S(s), \alpha_I(i)\rangle$. Then $\mathit{AbsGenComplete}$ is given by the following definition.

$$\mathit{indep}(\alpha) \triangleq (\forall s, t, i : \alpha(s, i) = \alpha(t, i))$$

$$\mathit{AGC}(\alpha) \triangleq \begin{cases} \langle \alpha', \beta_I\rangle & \text{if } (\alpha' \sqsubseteq \alpha) \wedge \mathit{indep}(\beta_I) \\ \mathit{AGC}(\alpha') & \text{otherwise} \end{cases}$$

where

$$\beta \doteq \mathit{SP}(P, \alpha, \alpha)$$

and

$$\alpha' \doteq \beta_s \sqcup \alpha$$

$$\mathit{AbsGenComplete}() \triangleq \mathit{AGC}(\mathit{good})$$

If $\mathit{AbsGenComplete}$ terminates, then it returns a complete abstraction of the original model. We can ensure termination by guaranteeing that every increasing chain $\alpha_1 \sqsubset \alpha_2, \sqsubseteq \ldots$ is finite. But as we mentioned before, completeness might not be desirable in many cases. So we instead consider an alternative function AbsGen that heuristically generates an abstraction with a desired amount of state reduction and approximation. It takes an additional natural number d which gives the minimum number of SP iterations required to build the abstraction, a natural number r which is the minimum factor of state reduction desired for any abstraction, and $v \in \mathit{Abs}$ which is an

[1]Technically, since "=" is binary it cannot be in Ops. We turn it into a unary operation by providing it with a tuple and returning the equality of the first and second items of the tuple.

"upper bound" abstraction used to exclude certain state information from the generated abstraction. The function *AbsGen* returns an abstraction witness and is defined as follows.

$$\mathit{AG}(\alpha, d, r, v) \triangleq \begin{cases} \langle \alpha', \beta_I \rangle & \text{if } (\alpha' \sqsubseteq \alpha) \vee ((\mathit{RF}(\alpha') \geq r) \wedge d \leq 0) \\ \mathit{AG}(\alpha') & \text{otherwise} \end{cases}$$

where

$$\beta \doteq \mathit{SP}(P, \alpha, \alpha)$$

and

$$\alpha' \doteq (\beta_s \sqcap v) \sqcup \alpha$$

The function *RF* (reduction factor) takes an abstraction function α and (heuristically) determines the approximate number of bits needed to encode the domain of α, divided by the number of bits needed to encode the range of α; this ratio defines the amount of state reduction achieved. The parameter d is used to froce a certain minimum number of *SP* iterations and serves the purpose of placing a lower-bound on the amount of allowable abstraction.

IV. Instantiating Abstraction Generation for RTL

The procedure presented above is generic, and can be instantiated with any finite set of primitive operations *Ops* and abstraction functions *Abs*, provided that the necessary derivation functions $\sqcap$, $\sqcup$, *prop*, *RF*, etc. are appropriately defined. We now turn to instantiating the procedure for the abstraction of RTL definitions. For this purpose, we take *Ops* to be the set of standard RTL operations on bit-vectors, including addition, subtraction, comparison, bitwise-conjunction, bitwise-disjunction, concatenation, extraction, generic if-then-else expresssions, etc. For the purpose of this paper, a bit vector is a member of the set $\mathit{BV} \triangleq (\{0,1\})^*$. We assume implicit conversions from $\mathbb{B}$ and $\mathbb{N}$ to *BV* (and the vice versa), where true is converted to $\langle 1 \rangle$ and false to $\langle 0 \rangle$.

Our implementation has efficient support for arrays or sequence for bit vectors, but we do not present them here for pedagogical reasons. Instead, we assume that an RTL state is defiend by a sequence of bit vectors, and the set S is the set BV^*. We now discuss the different mapping operations. A *linear mapping* is either the identity mapping $\lambda(x) \triangleq x$, or is of the form $\lambda(x) \triangleq \mathit{ite}((\mathit{Lo} \leq x \leq \mathit{hi}), \mathit{Lo}, f(x))$, where *ite* is the standard if-then-else operation, *Lo* and *Hi* are constant bitvectors, and f is a linear mapping. A *bitvector mapping* is either a linear mapping f or is defined by $\lambda(x) \triangleq (f(\langle x_1, \ldots, x_i \rangle) || g(\langle x_{i+1}, \ldots, x_j \rangle))$ where i and j are natural number constants, "||" is the bitvector concatenation operation, and f and g are bitvector mappings. The set of abstraction functions *Abs* is then restricted to contain functions Φ defined by: $\Phi(\langle x_1, \ldots, x_n \rangle) \triangleq \langle f_1(x_1), \ldots, f_n(x) \rangle$.

The bulk of the complexity is in the definition of *prop*. To understand the issues we consider one example. Consider the instance of *prop* for the $\mathit{ite}(x, y, z)$ operator, and let α be the map at the output. If the function *prop* returns the map α on both y and z and the identity mapping on x then the requirements on *prop* being a transfer are satisfied. But *ite* is a prolific operation and the best abstraction is not always obtained by including the informationfrom all tests in the *ite* expression. So we break the transfer property for *ite* by randomly selecting once for each field (or register) in the next-state function. This is based on a heuristic observation that there is a high correlation between tests within the next-state expression for a given register. The selection is where the additional natural number selection parameter for *prop* is used to select from a range of possible propagations.

Given a library of functions *prop* it is straightforward to produce *SP* as outlined in the preceding section. The strategy is to design a procedure that crawls over the body of P starting from the output, using *prop* to propagate the output mapping at the goal of an operation to its input. We often take into account the context when applying a specific mapping. For instance, consider the expression $(x + y)$ where the goal mapping is $A(z) \triangleq \mathit{ite}((4 \leq z \leq 6), 4, \mathbf{0}(z))$ where $\mathbf{0}(z)$ is the all-zero mapping. It y is alwars the constant 2 then we can use this information to propagate the mapping $B(x) \triangleq \mathit{ite}((2 \leq x \leq 4), 2, \mathbf{0})$ to x.

V. Conclusion

We have presented a procedure for automatically generating abstractions of state machines represented in RTL. The procedure illuminates different design trade-offs involved in the design of a scalable abstraction tool in practice. 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.

Our approach 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: our procedure is designed with the goal of scalability at the sake of verification completeness. Furthermore, the procedure handles the semantics of arithmetic functions, bit vectors primitives, and arrays, as well as modules and functions built on top of such primitives.

Acknowledgements

This material is based upon work supported in part by DARPA and the National Science Foundation under Grant No. CNS-0429591 and by the Semiconductor Research Corporation under Grant No. 08-TJ-149. The author thanks Rob Sumners for helping with the development of an early version of the procedure discussed in the paper.

References

[1] M. Abadi and L. Lamport, "The Existence of Refinement Mappings," *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, May 1991.

[2] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Approximation or Analysis of Fixpoints," in *Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL 1977)*. Los Angeles, CA: ACM Press, 1977, pp. 238–252.

[3] K. S. Namjoshi and R. P. Kurshan, "Syntactic Program Transformations for Automatic Abstraction," in *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV 2000)*, ser. LNCS, E. A. Emerson and A. P. Sistla, Eds., vol. 1855. Springer-Verlag, Jul. 2000, pp. 435–449.