Design and Implementation of an Abstract Interpreter for VHDL

Charles Hymans

STIX, École Polytechnique, 91128 Palaiseau, France
charles.hymans@polytechnique.fr

Abstract. We describe the design by abstract interpretation of a static analysis for the popular hardware language VHDL. From a VHDL description, the analysis computes a superset of the states reachable during any simulation run. This information is useful in the validation of safety properties of hardware components. The construction of the analysis is based on the formal definition of a semantics for VHDL. Soundness with respect to this semantics is shown. Various techniques allow a compromise between the desired accuracy and the cost of the final algorithm. We present a few examples and detail the essential implementation choices.

1 Introduction

We present the design of a static analysis for VHDL. It computes a superset of the states that may be encountered during any simulation run of a description. Following the methodology of abstract interpretation [2], we first define the semantics of a subset of VHDL. A sound static analysis is then obtained from this formalization by abstraction. We make our construction generic in the underlying symbolic domain used to represent the possible values that signals may take. That way, it is possible to plug in various back-ends so as to attain the best compromise between precision and efficiency. This work extends [5]. Arrays, variables, for-loops and until clause in wait statements were previously not considered. A finer abstraction of the state-space, which keeps track of the history of computation, is proposed. All implementation details are new.

Motivating example. We consider a component which performs the multiplication of an input matrix by a constant matrix. The input matrix is fed one coefficient at a time through a wire `DI` on rising edges of the clock `CLK`. New coefficients are signaled by setting a flag `DSI` high and need not be given in consecutive cycles. Similarly, the result is produced on `DO` while the flag `DSO` is set. We write a test-bench made up of the input generator of Fig. 1 and the checker of Fig. 2. The generator stimulates the design to do the multiplication of a unique matrix `INPUT`. It does this an unbounded number of times and waits arbitrarily long between each coefficient. The checker simply asserts the values read on `DO` when `DSO` is high are the correct results of the multiplication. Our prototype implementation is able to determine, without any human intervention, that the

D. Geist and E. Tronci (Eds.): CHARME 2003, LNCS 2860, pp. 263–269, 2003.

```
initial INPUT := (1,1,0,1);
process
  for I in 0 to 3 loop
    wait on CLK until CLK;
    DSI <= FALSE;
    while random loop
      wait on CLK until CLK;
    end loop;
    DSI <= TRUE; DI <= INPUT(I);
  end loop;
end process;
```

Fig. 1. Input driver

```
initial RESULT := (-4,17,-9,10);

process
  for J in 0 to 3 loop
    wait on CLK until CLK;
    while (not DSO) loop
      wait on CLK until CLK;
    end loop;
    assert DO = RESULT(J);
  end loop;
end process;
```

Fig. 2. Output Checker

$$
\begin{array}{rcll}
M & ::= & P_1 \,\texttt{|}\, \ldots \,\texttt{|}\, P_n & \text{(Parallel composition)} \\
P & ::= & C\,\texttt{;}\,P \mid \epsilon & \text{(Sequence)} \\
C & ::= & \texttt{v := } e & \text{(Variable assignment)} \\
 & \mid & \texttt{s <= } e & \text{(Signal assignment)} \\
 & \mid & \texttt{a(}e_1\texttt{) := } e_2 & \text{(Array assignment)} \\
 & \mid & \texttt{wait on } W \texttt{ until } b \texttt{ for } t & \text{(Suspension)} \\
 & \mid & \texttt{while } b \texttt{ do } P \texttt{ end} & \text{(Iteration)} \\
 & \mid & \texttt{if } b \texttt{ then } P \texttt{ end} & \text{(Selection)} \\
e, b & ::= & i \mid \texttt{true} \mid \texttt{false} \mid \texttt{random} \mid \texttt{v} \mid \texttt{s} \mid \texttt{a(}e\texttt{)} & \\
 & \mid & \texttt{not } b \mid b_1 \texttt{ and } b_2 \mid b_1 \texttt{ or } b_2 \mid b_1 \texttt{ = } b_2 \mid e_1 \texttt{ < } e_2 & \\
 & \mid & e_1 \texttt{ + } e_2 \mid e_1 \texttt{ * } e_1 &
\end{array}
$$

where `v` is a variable, `s` a signal and `a` an array identifier; W is a possibly empty set of signals; t is a strictly positive integer or ∞; and i is an integer.

Fig. 3. Syntax

assertion `DO = RESULT(J)` is never broken. Note that this is not practicable by conventional simulation.

2 An Operational Semantics for VHDL

To be able to reason about VHDL descriptions, we first formally define their semantics. Formalizations close to ours can be found in [3,4,6]. We suppose an elaboration phase – similar to the one presented in the standard [1] – compiles the description into a program of the kernel language of Fig. 3. Programs manipulate integers, booleans and statically allocated arrays. Note we deliberately ban delayed signal assignments (signal assignments with an after clause). They do not appear in the designs we wish to validate. and add much complexity since, in their presence, the precise layout of the memory used by a program is not known statically.

We express the execution of a program P as a small-step operational semantics. Program statements ${}^{l}C$ are uniquely tagged with labels l that are taken

$$\text{var}\ \frac{{}^{l}\texttt{v := } e \qquad \rho \vdash e \Longrightarrow v}{(l,\rho) \rightharpoonup (next(l), \rho[\texttt{v} \leftarrow v])} \qquad \text{sig}\ \frac{{}^{l}\texttt{s <= } e \qquad \rho \vdash e \Longrightarrow v}{(l,\rho) \rightharpoonup (next(l), \rho[\overline{\texttt{s}} \leftarrow v])}$$

$$\text{suspend}\ \frac{{}^{l}\texttt{wait on } W \texttt{ until } b \texttt{ for } t \qquad c = (next(l), W, b, t)}{(l,\rho) \rightharpoonup (c,\rho)}$$

$$\text{enter}\ \frac{{}^{l}\texttt{while } b \texttt{ do } {}^{l'}C\texttt{;}P \texttt{ end} \qquad \rho \vdash b \Longrightarrow \texttt{true}}{(l,\rho) \rightharpoonup (l',\rho)} \qquad \text{exit}\ \frac{{}^{l}\texttt{while } b \texttt{ do } P \texttt{ end} \qquad \rho \vdash b \Longrightarrow \texttt{false}}{(l,\rho) \rightharpoonup (next(l),\rho)}$$

Fig. 4. Sequential execution

$$\Pi\ \frac{\forall j < i : c_j \notin \mathcal{L} \qquad c_i \in \mathcal{L} \qquad (c_i,\rho) \rightharpoonup (c_i',\rho')}{(c,\rho) \to (\langle c_1,\ldots,c_i',\ldots,c_n\rangle,\rho')}$$

$$\Delta\ \frac{\forall j : c_j = (l_j, W_j, b_j, t_j) \qquad \rho' = update(\rho) \qquad \exists j : wake(W_j, b_j, \rho, \rho') \qquad c_i' = \begin{cases} l_i & \text{if } wake(W_i, b_i, \rho, \rho') \\ c_i & \text{otherwise} \end{cases}}{(c,\rho) \to (c',\rho')}$$

$$\Theta\ \frac{\forall j : c_j = (l_j, W_j, b_j, t_j) \qquad \rho' = update(\rho) \qquad \forall j : \neg wake(W_j, b_j, \rho, \rho') \qquad \begin{array}{c} \exists j : t_j \neq \infty \\ t = \min\{t_i \neq \infty\} \end{array} \qquad c_i' = \begin{cases} l_i & \text{if } t_i = t \\ (l_i, W_i, b_i, t_i - t) & \text{if } t_i \neq \infty \\ c_i & \text{otherwise} \end{cases}}{(c,\rho) \to (c',\rho')}$$

Fig. 5. Simulation algorithm

from a set $\mathcal{L}$. The label of the unique statement which follows ${}^{l}C$ in the control flow graph of the enclosing process is fetched with $next(l)$. The point of execution in a process is determined by the label of the statement that is to be executed next. The control point of a suspended process is augmented with a list of signals W, a condition b and a duration t. The duration is either a strictly positive integer or ∞ to indicate the absence of a timeout. A global environment ρ stores values of variables and signals. We denote by $\overline{x}$ the location where the future value of a signal x lies. We impose the syntactic restriction that no signal is assigned by more than one process. Hence, it is sufficient to remember only one future value for every signal.

An expression e evaluates to a value v in an environment ρ, which we express by the judgment $\rho \vdash e \Longrightarrow v$. The meaning of expressions is defined by structural induction in the classical way. Figure 4 shows the sequential execution of an individual process. Paraphrasing the sig rule: the right-hand side expression is evaluated in the current environment; the resulting value is then scheduled for

the next cycle at location $\overline{x}$; and control is transferred to the next statement. The three rules of Fig. 5 are enough to completely characterize the simulation algorithm of VHDL. Processes are run concurrently as long as possible thanks to the first rule. Once all processes are suspended, the global environment is updated so that signal assignments encountered during the last simulation cycle take effect:

$$update(\rho)(x) = \begin{cases} \rho(\overline{x}) & \text{if } x \text{ is a signal,} \\ \rho(x) & \text{otherwise.} \end{cases}$$

The Δ rule reactivates any process for which the value of some signal in the sensitivity list W was changed during the last cycle, and the condition b is met:

$$wake(W, b, \rho, \rho') = (\exists x \in W : \rho(x) \neq \rho'(x)) \wedge (\rho' \vdash b \Longrightarrow \texttt{true}) \,.$$

Finally, if no process activity can be resumed by Δ then the final rule advances simulation time by the smallest timeout.

3 The Abstract Interpreter

The set $\mathcal{O}$ of all prefixes of execution traces from some initial state s_0 can be constructively expressed as the least fixpoint of the continuous operator :

$$\mathbb{F}(X) = \{s_0\} \cup \{s_0 \dots s_k s_{k+1} \mid \exists s_0 \dots s_k \in X : s_k \to s_{k+1}\} \,.$$

This fixpoint is not effectively computable or even finitely representable. So we adopt the methodology of abstract interpretation [2] to obtain a decidable approximation. We proceed in two steps.

Generic Abstract Domain. We build an abstract domain to encode sets of traces. We collect environments and group them according to the history of computations that led to their creation. Collections of environments are further abstracted thanks to an abstract numerical domain $\mathcal{N}$. Numerical domains provide finite descriptions for sets of tuples of scalar values. We call $\gamma_{\mathcal{N}}$ the concretization function on the numerical domain. The way environments are grouped depends on a function κ which creates a token h from an execution trace. Formally, a collection of abstract environments X represents the traces:

$$\gamma(X) = \{s_0 \dots s_k \mid h = \kappa(s_0 \dots s_k) \wedge (c, \rho) = s_k \wedge R = X(c, h) \wedge \rho \in \gamma_{\mathcal{N}}(R)\} \,.$$

Both the numerical domain $\mathcal{N}$ and the grouping function κ are left as parameters of our construction. Hence, we have two orthogonal means to adjust the precision and efficiency of our analyzer.

$$
\begin{aligned}
[\![^{l}\texttt{v := } e]\!]^\sharp R &= \{\langle next(l), \mathsf{assign}_{\texttt{v}\leftarrow e}(R)\rangle\} \\
[\![^{l}\texttt{s <= } e]\!]^\sharp R &= \{\langle next(l), \mathsf{assign}_{\texttt{\#s}\leftarrow e}(R)\rangle\} \\
[\![^{l}\texttt{a(}e_1\texttt{) := } e_2]\!]^\sharp R &= \{\langle next(l), \mathsf{assign}_{\texttt{a(}e_1\texttt{)}\leftarrow e_2}(R)\rangle\} \\
[\![^{l}\texttt{wait on } W \texttt{ until } b \texttt{ for } t]\!]^\sharp R &= \{\langle c, R\rangle \mid c = (next(l), W, b, t)\rangle\} \\
[\![^{l}\texttt{while } b \texttt{ do } {}^{l'}C\texttt{;} P \texttt{ end}]\!]^\sharp R &= \{\langle l', \mathsf{select}_b(R)\rangle, \langle next(l), \mathsf{select}_{\texttt{not } b}(R)\rangle\} \\
[\![^{l}\texttt{if } b \texttt{ then } {}^{l'}C\texttt{;} P \texttt{ end}]\!]^\sharp R &= \{\langle l', \mathsf{select}_b(R)\rangle, \langle next(l), \mathsf{select}_{\texttt{not } b}(R)\rangle\}
\end{aligned}
$$

where `#s` is a new expression to reference the future value of a signal: $\rho \vdash \texttt{\#s} \Longrightarrow \rho(\overline{\mathtt{s}})$.

Fig. 6. Equations for the abstract sequential execution

$$
\Pi^\sharp \frac{\begin{array}{c}\forall j < i : c_j \notin \mathcal{L} \qquad c_i = l_i \qquad {}^{l_i}C \qquad \langle c_i', R'\rangle \in [\![^{l_i}C]\!]^\sharp R \\ c' = \langle c_1, \ldots, c_i', \ldots, c_n\rangle \qquad h' = \mathsf{record}_{(c,c')}(h)\end{array}}{(c, h, R) \rightsquigarrow (c', h', R')}
$$

$$
\Delta^\sharp \frac{\begin{array}{c}\forall j : c_j = (l_j, W_j, b_j, t_j) \qquad (c', R') \in \mathsf{update}(c, R) \qquad c' \neq c \\ h' = \mathsf{record}_{(c,c')}(h)\end{array}}{(c, h, R) \rightsquigarrow (c', h', R')}
$$

$$
\Theta^\sharp \frac{\begin{array}{c}\forall j : c_j = (l_j, W_j, b_j, t_j) \qquad (c, R') \in \mathsf{update}(c, R) \\ \exists j : t_j \neq \infty \qquad t = \min\{t_i \neq \infty\} \\ c_i' = \begin{cases} l_i & \text{if } t_i = t \\ (l_i, W_i, b_i, t_i - t) & \text{if } t_i \neq \infty \\ c_i & \text{otherwise}\end{cases} \qquad h' = \mathsf{record}_{(c,c')}(h)\end{array}}{(c, h, R) \rightsquigarrow (c', h', R')}
$$

Fig. 7. Abstract simulation semantics

Abstract Semantic Transformer. We systematically derive from its concrete counterpart an abstract simulation algorithm (see Fig. 6 and 7). The transition relation $\rightsquigarrow$ mimics in the abstract domain the concrete execution of processes. It is expressed in terms of a few primitives that operate on the numerical domain: assign undertakes assignments, select asserts boolean conditions, singleton builds the representation of a unique environment. Each of these operations must obey a soundness condition. For instance select must be such that:

$$\{\rho \in R \mid \rho \vdash b \Longrightarrow \texttt{true}\} \subseteq \gamma_{\mathcal{N}}(\mathsf{select}_b(R)) \; .$$

Finally, our algorithm consists in computing the least fixpoint of the following monotonic function:

$$\mathbb{F}^\sharp(X)(c', h') = X_0(c', h') \sqcup \bigsqcup\{R' \mid \exists (c, h) : R = X(c, h) \wedge (c, h, R) \rightsquigarrow (c', h', R')\} \; .$$

The static analysis is correct. Indeed, thanks to the properties enforced on the basic numerical operators, one can prove that we have:

$$\mathcal{O} \subseteq \gamma(\mathrm{lfp}\ \mathbb{F}^\sharp)\ .$$

Implementation We implemented the abstract interpreter in OCaml. Executions that went through distinct branches of if-statements are distinguished and for-loops are unrolled. For the back-end, we chose the domain of constants which we encode with balanced binary trees. The major advantage is to improve sharing, which in turn speeds up many operations. All abstract environments computed during the analysis are placed in a hashtable. It is not necessary to keep them all in memory, rather we store only the ones at the entry point of loops. Once the fixpoint has been reached, we can rebuild the missing environments in a single last pass. This dramatically reduces memory consumption. The fixpoint is computed with a standard worklist algorithm. The analysis was able to automatically verify various instances of the introductory example.

4 Conclusion

We have shown the staged design of an abstract interpreter for a subset of VHDL. It is based on a formalization of the simulation algorithm. As such, it has the ability to handle non-synthesizable descriptions. This permits its early integration in the design cycle. With a first implementation, we successfully verified non-trivial properties on a VHDL component. We hope to have demonstrated the adequacy of the approach as an automatic means to validate fairly complex safety properties. We were careful to separate concerns as much as possible so that our analyzer can be easily improved by local modifications. In fact, we can now focus on more efficient numerical domains tailored to prove specific classes of properties. We need no longer concern ourselves with the idiosyncrasies of the VHDL dialect.

Acknowledgments. We are grateful to P. Cousot, R. Cousot, F. Logozzo, X. Rival and E. Upton for help, comments and discussions.

References

1. ANSI/IEEE Std 1076–1987. *IEEE Standard VHDL Language Reference Manual*, 1988.
2. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992. (The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/~cousot`.).
3. K. Goossens. Reasoning about VHDL using operational and observational semantics. In *Correct Hardware Design and Verification Methods*, volume 987 of *Lecture Notes in Computer Science*, 1995.

4. M. Gordon. The semantic challenge of Verilog HDL. In *10th Annual IEEE Symposium on Logic in Computer Science*, pages 136–145, 1995.
5. C. Hymans. Checking safety properties of behavioral VHDL descriptions by abstract interpretation. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 444–460, 2002.
6. V. Rodrigues, D. Borrione, and P. Georgelin. An ACL2 model of VHDL for symbolic simulation and formal verification. In *XIII Symposium on Integrated Circuits and Systems Design*, 2000.