Assume-Guarantee Validation for STE Properties within an SVA Environment

Zurab Khasidashvili, Gavriel Gavrielov
Intel Israel (74) Ltd.
Haifa 31015, Israel
{zurabk,ggavriel}@iil.intel.com

Tom Melham
Oxford University Computing Laboratory
Oxford, OX1 3QD, England
melham@comlab.ox.ac.uk

***Abstract*—Symbolic Trajectory Evaluation is an industrial-strength verification method, based on symbolic simulation and abstraction, that has been highly successful in data path verification, especially microprocessor execution units. These correctness results are typically obtained under certain assumptions about how the verified hardware block's inputs are driven, as well as assumptions about the values of these inputs. For correct overall operation, the hardware environment within which the verified block resides is expected to satisfy these assumptions.**

We describe a translation of these proof assumptions into System Verilog Assertions. These are then used as checkers in dynamic validation of the hardware environment within which blocks verified by Symbolic Trajectory Evaluation operate. The result is a pragmatic assume-guarantee method that increases the quality and confidence in verification results, requires little or no modification to the Symbolic Trajectory Evaluation proofs, and leverages pre-existing dynamic validation infrastructure.

I. Introduction

Symbolic Trajectory Evaluation (STE) is a model checking method based on symbolic simulation over a lattice of abstract state sets [1]. STE's combination of abstraction and algorithmic efficiency is especially suited to verification of datapaths and memories, and has been demonstrated on many hard industrial verification problems [2], [3], [4]. A notable success is the verification, using Intel's Forte system [5], of the entire execution cluster of the Intel Core 2 Duo and Core i7 microprocessors [6], [7]

As with most property verification, STE correctness results are usually conditional on various assumptions about the hardware environment within which the verified block is operating. Assumptions are made about how the inputs to the verified block are driven. And it is often assumed that the values presented on these inputs comply with certain constraints, sometimes quite complex ones. Validating these assumptions is part of the well-known *assume-guarantee* paradigm for compositional reasoning [8]. We separately verify a component under assumptions and show that the environment guarantees the assumptions hold.

In this paper, we describe an approach to assume-guarantee validation in which components are verified with STE and the environmental assumptions are translated into System Verilog Assertions (SVA) and checked with dynamic validation. The methodology has been applied on a large scale to STE verifications of a micro-controller unit, and of the microoperations in the execution cluster of a recently-designed Intel microprocessor.

This work encompasses methodology, theory, implementation, and experimental evaluation. Unrestricted STE is capable of expressing quite complex and possibly *implicit* environmental assumptions. Our methodological contribution is to describe a restricted, standardized form for encoding STE properties that enables us to translate their environmental assumptions—whether explicit or implicit—into equivalent, and efficiently checkable, SVA properties. Our theoretical contribution is a simple solution to capturing the semantics of certain uses of STE symbolic indexing variables in the SVA language, where there is no similar concept, and in arguing the soundness of the translation. We also describe an implementation that employs the reflection features of the Forte system's functional scripting language [9] to allow both STE verification and translation from a single source. Finally, we present experimental results from extensive use of the framework on the STE verification environment of a recently taped-out Intel microprocessor.

II. SVA and STE Background

In this section, we give just enough background on SVA and STE to enable the reader to follow the subsequent text and to appreciate the problem our translation addresses. A good tutorial introduction to SVA [10] can be found in [11]. A detailed introduction to STE is given in [1], and a full account of Intel's Forte environment appears in [5].

A. Basic SVA Notation

System Verilog Assertions provide a rich specification language for expressing assumptions and conditions on the values that appear on circuit nodes over time. The most basic form of assertions are expressions built up with operators over bits (circuit nodes) and bit-vectors (vectors of circuit nodes). These include negation (!), conjunction (&&), disjunction (||), implication (<=), equality (==), and bit-vector relations and functions such as comparison and addition. If E is an expression and k a non-negative integer, then '$\$\mathrm{past}(E, k)$' means the value E had k clock ticks ago. The expression '$\$\mathrm{stable}(E)$' says that the current value of E is the same as its value at the previous clock tick.

An SVA *sequence* is an expression that describes a series of events over time. A delay operator is used to specify relative timing of events in the sequence. For example the sequence 'E_1 ##k E_2' means that the expression E_1 holds and then, integer k clock ticks later, expression E_2 holds. Writing '##k E' just means that the expression E holds k clock ticks from now. A repetition may be employed to say that an expression holds over some period of time. Writing '$E\,[*k]$' means that E holds for the next k clock ticks (including now).

Sequences can be combined using conjunction ('and') and disjunction ('or'). They can be inverted, to yield a *property*, with 'not'.

B. STE Verification and the Translation Problem

Verification properties in STE are called *trajectory assertions* and have the form $A \Rightarrow C$, where A and C are formulas of a simple linear-time temporal logic. The intuition is that the *antecedent* formula A describes some initial conditions of the circuit inputs and states, and the *consequent* C specifies the values expected on circuit nodes as a response.

Atomic propositions in A and C take the form '$P \rightarrow \mathrm{n}$ is 0' or '$P \rightarrow \mathrm{n}$ is 1', where 'n' is the name of a circuit node and the *guard* P is a formula of propositional logic. The guard determines when the proposition is asserted: if P is true, then the node n must have the value 0 (or 1 respectively); if P is false, then n can have any value—including, for abstraction efficiency, the don't care value X. Antecedents and consequents are essentially just conjunctions (using ' and ') of these atomic propositions, possibly modified by the next-time temporal operator N.

The guards in a trajectory assertion are formulas of propositional logic over some Boolean variables. For each assignment of truth-values to the variables, the assertion collapses into a property checkable by three-valued simulation, with the don't care value X on all circuit nodes not forced to 0 or 1 by the antecedent (or the circuit). Given a trajectory assertion, STE *simultaneously* computes all these three-valued simulations, and checks the results against the consequent C.

This arrangement gives STE a native capability for partitioned Boolean abstraction that, by the method known as *symbolic indexing* [12], can be very effective on large but 'semantically regular' datapaths. A full discussion of symbolic indexing and its automation can be found in [13] and [14].

This machinery can encode sophisticated abstraction schemes with complex implicit environmental assumptions. In the work of this paper, however, our methodology imposes constraints on how STE properties are written in order to make their assumptions easily translated into SVA. In the simplest case, we deal with STE sub-formulas of the form

$$x \rightarrow \mathrm{n}\ \textsf{is}\ 1\ \ \textsf{and}\ \ \neg x \rightarrow \mathrm{n}\ \textsf{is}\ 0$$

where x is a Boolean variable that appears nowhere else in the formula and n is a circuit input node. This establishes a one-to-one correspondence between x and n—the circuit input n is 'driven by' the symbolic Boolean variable x. We abbreviate this by writing 'n is x', and more generally allow one to write 'n is E' for any propositional formula E. If a unique, unconstrained Boolean variable is associated with each circuit input in this way, then STE verification is essentially just symbolic simulation.

Environmental assumptions can be added to this scheme by using the *parametric encoding* [15] of Boolean constraints. For example, if the antecedent of a two-input device is

$$\mathrm{a}\ \textsf{is}\ x\ \ \textsf{and}\ \ \mathrm{b}\ \textsf{is}\ y$$

we can make the assumption that at least one input is high by using the parametric technique to verify this property under the assumption $x \vee y$. Using this method, any formula that constrains the variables in an STE antecedent can be taken as an environmental assumption.

In this simple example there is a one-to-one correspondence between circuit nodes and variables, so it is easy to re-express the assumption as an SVA expression over circuit nodes. We just substitute nodes for corresponding variables, giving the SVA expression a || b. The propositional disjunction, $\vee$, becomes SVA disjunction, ||.

We wish, however, to support more subtle use of STE than this—including limited forms of symbolic indexing for abstraction efficiency. We commonly find, for example, STE antecedents with guarded sub-formulas equivalent to

$$P \rightarrow \mathrm{a}\ \textsf{is}\ x\ \ \textsf{and}\ \ Q \rightarrow \mathrm{b}\ \textsf{is}\ x \qquad (1)$$

Now there isn't a direct correspondence between variables and nodes. Sometimes the variable x is associated with node a, and sometimes with node b—and, if P and Q overlap, sometimes with both. There is no obvious node name to replace occurrences in environmental assumptions of the variable x (or, indeed, in the guards of *other* sub-formulas). Moreover, this antecedent makes an implicit assumption about circuit nodes, namely that the value on a is the same as the value on b when P and Q both hold.

A general method to untangle completely unrestricted STE antecedents and environmental assumptions into equivalent SVA properties would be very complex. Our work therefore places methodological restrictions on the form in which STE antecedents are written. Part of our contribution—explained in the sections that follow—is to propose a form that scales to the large verifications done at Intel, is natural for engineers to write, and allows symbolic indexing—while still admitting of a fairly intuitive translation into SVA.

C. The 5-tuple Representation of Antecedents

The Forte implementation of STE is embedded in a functional programming language, *reFLect* [9], similar to ML. STE antecedents are represented by lists of *5-tuples* of the form

$$(\mathit{guard}, \mathit{node}, \mathit{value}, \mathit{start}, \mathit{end})$$

where guard and value are formulas of propositional logic (represented as BDDs), node is a node name (a string), and start and end are non-negative integers. The meaning is that if guard holds, then node has value during the time period from simulation cycle $start$ up to but excluding simulation cycle end. So, for example, the 5-tuple list $[(P, \mathrm{a}, x, 0, 1), (Q, \mathrm{b}, x, 0, 1)]$ is the machine representation of the STE antecedent (1) above. Our methodological restrictions and translation are defined over these 5-tuple lists, which can represent any STE antecedent formula.

III. STE Proof Environment

Our target application is verification of micro-operation (μop) execution by, for example, the execution (EXE) cluster of a contemporary X86 microprocessor. We will call the region of the full chip under verification the *STE unit*.

There are typically several thousand different μops, executed in several sub-units of the STE unit and at several ports. To organize and share STE proof code, μops are divided into groups. The μops in a group are normally executed in the same sub-unit and on the same port. Their STE antecedents are therefore very similar. There can be several μop groups for the same port.

For each UOP group, the STE antecedent and any accompanying environmental constraints are divided into the following components:

a) Timed assumptions: This is an STE 5-tuple list that describes how inputs to the the STE unit are driven (over time) when a μop is executed. Timed assumptions must be satisfied by the post-reboot behaviour of the full chip—their validity as STE assumptions is expected to be guaranteed by the hardware that drives the inputs of the STE unit.

b) Timed restrictions: This is an STE 5-tuple list used to ignore uninteresting full chip (and STE unit) behaviour over time—behaviour we do not wish to verify in STE. For example, if μop execution is interrupted by a reset or if certain input validity signals are not asserted when μop execution starts, then we don't want STE to check correctness of the output. Timed restrictions often set STE unit inputs to constants. For example, we might make an input F to ignore the STE unit's behaviour for an incompletely implemented feature that is invoked when that input is T. Such restrictions are either temporary and will be removed at a later stage of the design, or they will be used for case-splitting all possible legal behaviours of the STE unit.

c) Global assumptions: These are conditions on Boolean variables that, in the context of the STE antecedent, characterize relationships between values on the STE unit's input that should be satisfied by the post-reboot behaviour of the full chip whenever a valid μop is executed. Some global assumptions may apply to only part of a μop group. Some global assumptions characterize expected microcode behaviour. As with timed assumptions, the validity of global assumptions is expected to be guaranteed by the hardware that drives the inputs of the STE unit.

d) Global restrictions: These are constraints on Boolean variables that focus the proof on some interesting scenario of the STE unit. This is mainly useful for case splitting. For example, one might want to restrict the verification to a particular group of μops. The condition that the executing μop belongs to that group will be a global restriction on Boolean variables representing the opcode of the μops. As with timed restrictions, global restrictions may be used to ignore specific behaviours of the STE unit temporarily.

The distinction between *assumptions* and *restrictions* is pragmatic—assumptions are expected to be met by the operating environment (i.e. are the subject of our assume-guarantee reasoning) and restrictions are not. The distinction between *timed* and *global* is syntactic—timed constraints come from 5-tuples, and global constraints come from expressions over Boolean variables.

In STE verification, the timed restrictions and assumptions are taken together form the antecedent. The conjunction of the global restrictions and assumptions constitutes a Boolean constraint on STE variables that is taken as an assumption using the parametric technique [15]. Thus the distinction between assumptions and restrictions is irrelevant for the STE proofs. On the other hand, the STE consequent is irrelevant for the assume-guarantee method supported by our translation.

IV. STE to SVA Translation

From the description above, it follows that the restrictions can be violated by legal, full chip simulation traces. But for the parts of traces that do violate the restrictions, the STE unit's output behaviour is (by definition) irrelevant, and STE passes vacuously. In our assume-guarantee method, therefore, the generated SVA must ignore the parts of full chip simulation traces where any restriction is violated.

This is done by using the SVA property generated from the restrictions (timed and global) as a *trigger* for all the *checker* properties generated from the assumptions (timed and global). Every System Verilog Assertion we generate has the following implicational form:

$$\text{not}\; trigger \;\text{or}\; checker$$

In our methodology, the trigger is the same for every generated SVA for a group of μops. One or more checker properties are built from each tuple in the timed assumptions and from each global assumption. We observe that the translation preserves abstraction, in the sense that whenever the STE proof environment makes no assumption about the value on a circuit node (i.e. that node gets value X in the STE simulation), the generated SVA makes no constraint on the value of that node.

A. Preprocessing

Let A be the conjunction of all global assumptions and restrictions. As a preprocessing step, we replace every tuple (g, n, v, s, e) in the timed restrictions and timed assumptions with (T, s, v, s, e) if $A \Rightarrow g$ is a tautology, and omit the tuple if $A \wedge g$ is unsatisfiable. This is sound because the global restrictions are part of the SVA trigger, and any violation of the global assumptions will be caught by the assertions obtained by translating them into SVA.

Similarly, let R be the conjunction of all global restrictions. As a preprocessing step, we inspect each global assumption g. If $R \Rightarrow g$ is a tautology, we do not generate SVA for g; if

$R \wedge g$ is unsatisfiable, we replace g with F. Such an assertion would fail every time the simulation check is triggered, so the STE-to-SVA translation immediately reports such global assumptions.

B. Translation of Ground Tuples

A *ground tuple* is a 5-tuple (T, n, v, s, e) in which the guard is true and v, the value on node n, is a Boolean constant, T or F. If v is T, then the tuple is translated to the SVA sequence $\#\#s\, n\, [*e{-}s]$: after a delay of s, node n is high for $e{-}s$ time units. Similarly, if v is F, the SVA sequence is $\#\#s\, !n\, [*e{-}s]$.

C. Handling STE Boolean Variables

The guard of any non-ground tuple in the timed restrictions and timed assumptions will be a non-constant expression over one or more Boolean variables. The value component may be T, F, or an expression over some Boolean variables. Any of these variables may occur in the propositional constraints that constitute the global assumptions and restrictions.

As suggested in section II, our translation to SVA works by finding, for each Boolean variable used in the STE verification, a group of circuit nodes that can serve as proxies, or representatives, for that variable in the corresponding SVA expressions. Global assumptions and restrictions are re-expressed as SVA constraints on those circuit nodes. Likewise, explicit or implicit relationships among values in the circuit imposed using variables in STE are re-expressed as SVA properties over circuit nodes.

To make this work, we impose some methodological constraints on the way in which STE antecedents are written. We say that a variable x *immediately depends* on y if there is a tuple with x in its value and y in its guard. We introduce a dependency relation between variables as the transitive closure of immediate dependency. Using this concept, we impose the following restrictions on the usage of variables in antecedents. Within each group of μops, the following hold:

- Each Boolean variable, whether in the antecedent or in the global assumptions and restrictions, 'drives' at least one input node of the STE unit. That is, each variable is the entire value expression of at least one timed assumption or timed restriction tuple.
- the dependency relation between variables is a strict partial order. Variable dependency defines a DAG.

The idea is to ensure that for every Boolean variable in STE there is a circuit node that can be used as an SVA expression in the translation that denotes the value of that variable. In fact the conditions are a little more complex than the above, because in STE Boolean variables can both drive nodes and be used in expressions conditionally, depending on guards and the global restrictions and assumptions. The general rule is that whenever a variable is used, it must also be 'defined', in the sense of driving a specific circuit node.

Both these conditions are methodological—STE works without them, but then it is not amenable to the assume-guarantee reasoning we're trying to support. One way to deal with variables that do not drive circuit nodes would be to perform a case splitting on them. But this is not practical because each case split can double the amount of generated SVA. The acyclicity restriction ensures there is at least one circuit node that can serve as a representative for each variable.

D. Finding Circuit Node Representatives for Variables

The fundamental element of our algorithm is translation of propositional expressions over Boolean variables into equivalent SVA expressions over circuit nodes. The mapping of Boolean and bit-vector operations—negation, conjunction, binary addition, and so on—to corresponding SVA operations is straightforward. Variable translation is handled as follows.

Let x be any variable that occurs, on its own, as the value expression of any tuple in the timed restrictions or timed assumptions. There may be several such tuples. For any x, we partition the set of all such tuples into subsets $\mathcal{T}_x^g$, where g is a distinguished *maximal guard* among all the guards of tuples in the subset. More precisely, g is logically implied by each of the guards of the tuples in $\mathcal{T}_x^g$. (If there is more than one such maximal guard, we make an arbitrary choice). Moreover, we suppose this partitioning is such as to ensure that if we have any two subsets $\mathcal{T}_x^{g_1}$ and $\mathcal{T}_x^{g_2}$, then $g_1 \wedge g_2$ is unsatisfiable.

The idea is that when g holds, the node components of the tuples in $\mathcal{T}_x^g$ are all candidate representatives for the variable x in our translation. We choose one such candidate as follows. Let $s(x, g)$ be the earliest start time of the tuples with a maximal guard in $\mathcal{T}_x^g$, and let $n(x, g)$ be the node component of the tuple in $\mathcal{T}_x^g$ with this earliest start time. (In case of ties, we make an arbitrary choice.) Let f (for *future*) be any integer greater than or equal to the end time of every tuple in $\mathcal{T}_x^g$ for all relevant guards g. We define a standard SVA 'name' for the value of variable x when g holds as follows:

$$\mathit{node}(x, g, f) = \$\mathrm{past}(n(x, g), f - s(x, g))$$

Suppose, for example, that

$$\mathcal{T}_x^P = \{(P, \mathrm{a}, x, 2, 5), (Q, \mathrm{b}, x, 1, 4)\}$$

where $Q \Rightarrow P$. Then $\mathit{node}(x, P, f)$ is the SVA expression '$\$\mathrm{past}(\mathrm{a}, f{-}2)$', for any $f \geq 4$. The function of f is to relativize time points with respect to a reference point of time somewhere beyond the end of the relevant segment of the STE simulation run for these tuples. We discuss this further in section IV-H.

Given a Boolean expression P, we compute a family of SVA translation instances as follows. Let $\mathcal{V}_P = \{x_1, \ldots, x_k\}$ be the set of all variables in P, together with all the variables they depend on, according to our dependency relation. For each variable x_i, where $1 \leq i \leq k$, we choose a maximal guard g_i that appears in one of the tuples in which x_i is the value expression. Let f be any integer greater than or equal to the end time of every tuple in every $\mathcal{T}_x^g$, for $x \in \mathcal{V}_P$ and all relevant guards g. We define a mapping from variables to SVA expressions:

$$\theta_f = \{x_i \mapsto \mathit{node}(x_i, g_i, f) \mid 0 \leq i \leq k\}$$

Thus θ_f is determined by choosing a relevant maximal guard for each variable that P depends on. For any Boolean formula Q, we write $Q\theta_f$ to denote the translation of Q into an SVA expression, with Boolean variables mapped to SVA subexpressions using θ_f and with the obvious replacement of Boolean operations by SVA operations.

For a given Boolean expression P and a given θ_f defined relative to P, we define a translation instance of P to an SVA expression as follows:

$$exp(P, \theta_f) = ((g_1\theta_f \;\&\&\; \cdots \;\&\&\; g_k\theta_f) <= (P\theta_f))$$

where $g_1, \ldots g_k$ are the guards chosen for the variables $x_1, \ldots, x_k$ in defining θ_f. The methodological restriction that variable dependency forms a DAG ensures that the domain of θ_f covers all the variables that occur in $g_1, \ldots, g_k$. We then define an SVA *sequence* for P,

$$seq(P, \theta_f) = \#\#f\; exp(P, \theta_f),$$

by applying a top-level delay operation '$\#\#f$'.

1) Conflicting Mappings: There may be many translations of a given propositional formula P into an SVA sequence, one for each choice of guards for the variables that P depends on—i.e. one for each mapping θ_f we can construct. Our overall translation needs to cover all relevant cases, but some irrelevant cases may be eliminated as follows.

The formula P we are translating can be the guard or value expression of a tuple, a global restriction, or a global assumption. When P is a guard or a value component, we use the conjunction $R \wedge A$ of global restrictions and assumptions to filter out mappings that are ruled out by them ('conflicting mappings'). If, for some selection of guards $g_1, \ldots, g_k$, we find that $g_1 \wedge \ldots \wedge g_k \wedge R \wedge A$ is unsatisfiable, then we do not include the corresponding θ_f among the mappings we use when computing relevant translation instances of P. Similarly, when P is a global assumption, then we use the conjunction of all global restrictions, instead of $R \wedge A$, to eliminate impossible mappings.

We denote the set of all non-conflicting mappings for a Boolean formula P by $maps(P)$, and we define

$$Exp(P, f) = \&\&_{\theta_f \in maps(P)}\, exp(P, \theta_f)$$
$$Seq(P, f) \;= \#\#f\; Exp(P, f)$$

for any sufficiently large value of f.

E. Generating Equality Expressions for Variables

STE antecedents commonly drive several different circuit nodes with the same variable. This encodes an implicit assumption that our translation must capture, namely that the same value appears on all the circuit nodes that are driven by the same variable.

For a pair of tuples (g_1, n_1, x, s_1, e_1) and (g_2, n_2, x, s_2, e_2), we define an *equality constraint* as follows:

$$\#\#f \begin{pmatrix} Exp(g_1 \wedge g_2, f) \\ \quad <= \\ \$\mathrm{past}(n_1, f{-}e_1) == \$\mathrm{past}(n_2, f{-}e_2) \end{pmatrix}$$

Note that here we need to use a reference time f and 'past' expressions, since we cannot write $\#\#e_1\, n_1 == \#\#e_2\, n_2$ in SVA syntax.

Of course we simplify $g_1 \wedge g_2$ when one of g_1 or g_2 implies the other. And when $g_1 \wedge g_2$ is unsatisfiable, we optimize by not generating an equality constraint for this pair.

1) Equality within timed restrictions: We divide the restrictions that have a variable as the value expression into groups, according to the variable. For each group, with defining variable x, we generate equality constraints as follows. Partition the group into maximal subsets $\mathcal{T}_x^{g_i}$, with $1 \leq i \leq k$. For each set $\mathcal{T}_x^{g_i}$, generate an equality constraint between the tuple with guard g_i and every other tuple in the set. Then generate an equality constraint between the tuple in $\mathcal{T}_x^{g_i}$ with guard g_i and the tuple in $\mathcal{T}_x^{g_j}$ with guard g_j, for $1 \leq i < j \leq k$. The SVA sequence conjunction of all these constraints is used as a conjunct in the trigger.

2) Equality within timed assumptions: Similarly, we divide the timed assumptions that have a variable as the value expression into groups by variable. Generate equality constraints that link the node values of the tuples within each group, using a partitioning by maximal guards as for the timed restrictions. Each of these constraints serves as a checker property in the SVA generated from the timed assumptions.

3) Equality relating timed restrictions and timed assumptions: Finally, we generate equality constraints for any variable x that occurs as the value expression in the timed restrictions and in the timed assumptions. The constraints are generated according to the partitionings introduced above. For each subset $\mathcal{T}_x^{g_a}$ of the timed assumptions and each subset $\mathcal{T}_x^{g_r}$ of the timed restrictions, generate an equality constraint between the tuple in $\mathcal{T}_x^{g_a}$ with guard g_a and the tuple in $\mathcal{T}_x^{g_r}$ with guard g_r. The conjunction of all such constraints is used as a conjunct in the trigger.

F. Generating SVA for Global Assumptions and Restrictions

Let $R_1, \ldots, R_n$ be all the global restrictions. For each $1 \leq i \leq n$, we generate the SVA sequence $Seq(R_i, f)$. Note that this sequence covers all possible mappings of variables to nodes that arise from choosing combinations of guards for the variables R_i depends on. We then include the SVA sequence conjunction of all these sequences as a conjunct of the trigger.

Let $A_1, \ldots, A_m$ be all the global assumptions. For each A_i with $1 \leq i \leq m$, and for each θ_f in $maps(A_i)$, we generate a corresponding SVA sequence $seq(A_i, \theta_f)$ as a checker expression in the SVA generated by our translation.

G. SVA Properties for each Antecedent Tuple

For any tuple (g, n, v, s, e), we define its *relevant variables* to be all variables that occur in g or v, plus the variables on on which these depend. For each mapping θ_f covering all the relevant variables of a given tuple, we generate one or more SVA expressions as follows.

If the tuple has the form (g, n, T, s, e), i.e. the value component is constant T, we generate the SVA property

$$\mathrm{not}(Seq(g, f))\ \mathrm{or}\ (\#\#s(n)\, [*e{-}s])$$

Similarly, if the tuple has the form (g, n, F, s, e) we generate

$$\mathrm{not}(\mathit{Seq}(g, f)) \text{ or } (\#\#s(!n)\,[*e{-}s])$$

These simply say, in SVA, that the nodes mentioned in these tuples have the constant values given by the STE antecedent within the stated windows of time.

For every tuple (g, n, x, s, e) with a variable x as its value component, we generate the SVA property

$$\mathrm{not}(\mathit{Seq}(g, f)) \text{ or } (\#\#s{+}1(\$\mathrm{stable}(n))\,[*e{-}s{-}1])$$

This says only that the value on node n is stable over the period during which it is driven by the variable x in STE; linkage with ocurrences of x in other tuples is handled by the equality constraints. We omit the generation of this stability condition if $e - s \leq 1$.

For tuples (g, n, v, s, e) where the value component v is not a constant or a variable, we generate the SVA property

$$\mathrm{not}(\mathit{Seq}(g, f)) \text{ or } \#\#f(\$\mathrm{past}(n, f - s) == \mathit{vname})$$

where vname is a fresh SVA identifier defined to be equal to $\mathit{Exp}(v, f)$.

The SVA properties generated using these rules from the tuples in timed restrictions all become conjuncts of the trigger. Each SVA property generated from a timed assumption tuple becomes a checker expression.

H. Summary Overview of the Algorithm

As already mentioned, the top-level SVA properties generated for assume-guarantee validation by our method are all of the form '$\mathrm{not}\ \mathit{trigger}\ \mathrm{or}\ \mathit{checker}$'. There is one global $\mathit{trigger}$ property per μop group, the conjunction of:

- all SVA properties generated from the tuples in the timed restrictions (section IV-G),
- the equality constraints for variables within the timed restrictions (section IV-E1),
- the equality constraints relating timed restrictions and timed assumptions (section IV-E3), and
- SVA sequences for the global restrictions (section IV-F).

For each of the conjuncts in this trigger property, our implementation chooses an appropriate value for the reference time 'f' of our translation. The efficiency of checking the resulting SVA in simulation depends critically on the depth of simulation required by the 'past' operators in our properties. Our implementation therefore includes a complex algorithm that aims to minimise f for each conjunct in the trigger.

In SVA syntax, the trigger property is written as a conjunction of the above constraints—is illustrated by this example:

$$\begin{array}{l} \mathit{property\ trigger};\\ @(`\mathit{SYS\_CLK})\\ \quad((\#\#10(\$\mathrm{stable}(\mathrm{opcode}[2:0]))\,[*1]) \text{ and}\\ \quad\ (\#\#5(!\mathrm{reset})\,[*12]) \text{ and}\\ \quad\ (\#\#7(\mathrm{uop\_valid})\,[*2]) \text{ and}\\ \quad\ \#\#10(\mathrm{uop\_group\_condition}))\\ \mathit{endproperty} \end{array}$$

This aligns the time units within our conjunct sequences with, $\mathit{SYS\_CLK}$, which is the reference clock in the system.

The individual $\mathit{checker}$ properties comprise:

- each SVA property generated from the tuples in the timed assumptions (section IV-G),
- the equality expressions for variables within the timed assumptions, (section IV-E2),
- the SVA sequence for each global assumption (section IV-F).

Again, the reference time f for translation of each SVA property is minimised for simulation efficiency.

For each checker, an SVA assertion is defined that says the checker must hold triggered. Suppose the reference time f is 10. Then a typical example is the following stability property for the tuple $(\mathrm{T}, \mathrm{output}, \mathit{variable}, 4, 9)$:

$$\begin{array}{l} \mathit{output\text{-}stability\text{-}assertion} : \mathit{assert\ property}\\ \mathrm{not}\ \mathit{trigger} \text{ or } \#\#10(\$\mathrm{stable}(\$\mathrm{past}(\mathrm{output}, 5))\,[*4] \end{array}$$

A pragmatic optimization, in the default flow, is that we do not generate SVA corresponding to STE tuples that drive clocks. (In STE verifications, clocks need to be driven explicitly by the antecedent.) If the clocks do not behave correctly, this will be caught by other validation activities.

V. Implementation

Our translation method is implemented within Intel's Forte system [5]. Forte is essentially a programming environment based around the *reFLect* functional programming language [9], and large-scale STE verification efforts are to a great degree a programming (or scripting) activity in nature. STE model checking is invoked through *reFLect* library function calls, and trajectory assertions for verification (lists of 5-tuples) are generated by writing functional programs that compute them. Our translation is also implemented as a *reFLect* functional program that computes SVA texts from lists of 5-tuples together with the Boolean expressions stating global restrictions and assumptions.

A. Usage of Reflection

The STE proof environment contains calls to functional program code that generate many Boolean formulas for each μop group being verified—guards, value expressions, global assumptions, and global restrictions. This code is user-defined and of arbitrary complexity, and therefore unsuitable as the source for our translation into SVA. On the other hand, the resulting formulas are represented in *reFLect*, and passed to the STE model checking engine, as BDDs (or, optionally, a form of AIGs [16]). This makes them too low-level for translation into SVA—once we have evaluated down to a BDD, much useful structure is lost and compact translation is difficult.

Our implementation solves this problem by exploiting the reflection features of *reFLect* to 'intercept' the evaluation of function calls that generate Boolean formulas at a stage suitable for translation into SVA. The *reFLect* language includes a primitive datatype, *term*, whose elements are the abstract syntax trees of *reFLect* programs themselves. Functions can take terms as arguments, analyse their structure, and return terms as results, in any programmable way. We exploit this to replace the BDDs that occur in 5-tuples by terms that represent syntax of the *reFLect* function calls that generate these BDDs, at a level of elaboration suitable for translation to SVA.

Suppose there is a *reFLect* function, xor say, that normally computes a BDD, but which we wish our translation to see. We overload the *reFLect* function identifier 'xor' with an alternative version that takes terms, rather than BDDs, as arguments and produces the *reFLect* syntax tree consisting of an application of the xor function to these terms. This alternative xor builds an exclusive-or *expression* rather than a BDD. We do this for all functions we want to stop evaluation at—i.e. all those functions that we wish to form the vocabulary for the source language of our translation to SVA. The arbitrarily complex STE proof environment built on these primitives will then, through overloading, have two interpretations—one that computes 5-tuples with BDDs, for STE, and one that computes 5-tuples with terms for translation into SVA.

By this method we are able to have a single Forte source for both running STE and generating SVA. Verification engineers need to make very few changes to their existing STE environment to enable this new flow. Moreover we ensure that the Boolean expressions we translate are at a suitable abstraction level for to produce compact, efficiently checkable, and human-readable SVA.

B. Vectorization

Primitive STE antecedents express everything in terms of individual bits, and it is difficult to ensure that the STE proof environment keeps bit-vectors intact through to the level at which translation to SVA begins. As a second step in preprocessing, therefore, we perform *vectorization*—grouping of the nodes of STE tuples that belong to the same bit-vector. This enables much more compact and efficient SVA to be generated, with vector operations instead of bit-level ones. Tuples whose nodes belong to the same vector (we can tell from their names) and whose guards are the same are merged into a vector tuple $(g, \vec{n}, \vec{v}, s, e)$. We do this whenever all the bits of $\vec{v}$ are variables whose names indicate they belong together, or all the bits of $\vec{v}$ are T or F. Constant vectors are translated to hexadecimal numbers in SVA.

VI. Experimental Results

Our assume-guarantee mechanism has been used at Intel on two major examples, a micro-controller unit, and μop execution in the execution cluster of a recently-designed processor. We give some results from the second of these.

SVA properties were generated and checked by dynamic validation for every μop group (except multiplication and division) in the STE proof environment of the execution cluster. A total of 36 μop groups were covered, comprising 1,035 μops in total. The number of μops per group ranged from 1 to 111, with an average of around 29. A total of 3,616 SVA checker properties were generated, of which 3,061 were from global assumptions, 471 were from constant assignment tuples, and 84 were from equality constraints. There were no equality constraints relating timed restrictions and timed assumptions, and no non-trivial stability constraints—these parts of our translation were tested in other exercises.

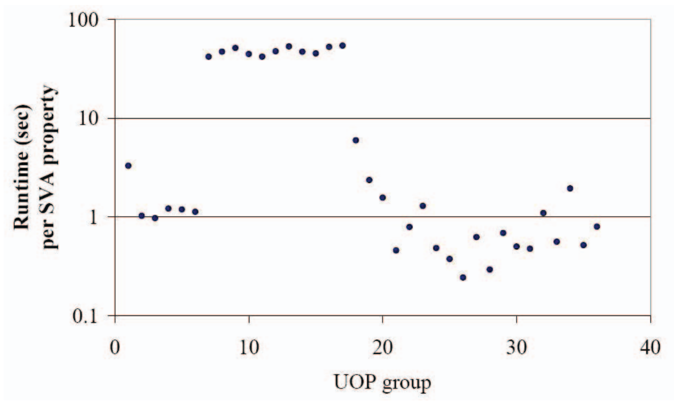

Fig. 1. Runtime per SVA Property Generated for EXE Cluster μop groups.

VCS, a 3-valued simulator from Synopsys, was run on all 3,616 SVA checker properties in one VCS session with 173 cluster level tests. The runtime was approximately 54.5 hours. Running the same tests on the execution cluster without any SVA checks (just computing the waveforms of all nodes) takes around 27 hours. So the overhead of checking the SVA generated by our method was a factor of around 2.

The data in figure 1 show the runtime for our translation (in seconds, on a log scale) per SVA property generated, for each group. The average runtime is 15.4 seconds. The high runtimes for some groups is due to the pre-processing steps of our algorithms, which require intensive BDD computations. This high runtime is compensated for by a reduction in SVA properties generated, and so in VCS simulation time. For example, in one of the groups taking over 500 seconds, the number of timed restrictions was reduced by pre-processing from 138 to 28, and number of timed assumptions from 86,970 to 1,991. For another group, the numbers are 92 to 24 for timed restrictions and 86,790 to 1,969 for timed assumptions. By contrast, for a μop group with runtime 11 seconds, the numbers are 92 to 24 and 4,078 to 2,017.

Without vectorization, it is estimated we would have at least two orders of magnitude more assertions, with very large combinational expressions, and it would not be practical to check all the SVA properties in VCS. More importantly, we found a huge benefit in keeping symbolic the bit-vector that codes for the instruction field of the μops in each group. In the STE proofs, the μop code sometimes needs to be made concrete for STE capacity reasons—essentially these proofs case split over the different kinds of μops in each group. In our translation, however, we code the μop instruction with Boolean variables, and translate the group as a unit. (Groups 21–36 are too complex to handle in this way, and were run with explicit μops.) A user who run all the sessions without this symbolic UOP feature generated 100 times more properties, beyond the capacity of VCS.

A. STE Environment Violations and Bugs Found

1) Unused variables: Our methodology requires every variable in the global assumptions to drive at least one circuit node—our tool reports an error if this is violated. In our experiments, there have been tens of such cases. Most of the time such variables were redundant, and the STE proofs still ran after cleaning up the global assumptions and restrictions to eliminate them. In rare cases, debugging unused variables revealed true environment violations, and after correcting them the STE proofs failed. In such cases, debugging unused variables lead to eliminating false positives in the STE proofs.

2) Assertion failures: When an assertion generated by our translation fails in dynamic validation, there are two possible reasons. First, the assertion may simply be too strong, and eliminating the corresponding assumption (implicit or explicit) from the STE environment simply strengthens the the STE proofs. Second, the assertion may catch a real environment violation—a *potential* false positive in the STE proofs. Running the generated SVA for the 173 cluster level tests mentioned above revealed tens of wrong or redundant assumptions in the STE environment.

3) Bugs found: After the cleanup stage, the 3,616 SVA were run in VCS on 1,100 core level tests. Two bugs were discovered in the design as a result of this activity. These bugs were found in the interaction between the microcode and the execution cluster, rather than in the execution cluster itself. This is to be expected, because the assume-guarantee activity supported by our work starts at a relatively late stage of validation, when the bugs that have already been found using STE have been fixed.

VII. Conclusion

There is, of course, a vast literature on assume-guarantee reasoning in formal verification. In this paper, we have described a pragmatic method for checking the assumptions of STE proofs by dynamic validation in an SVA environment, rather than checking them by proof. The method has reasonable computational overhead, doesn't require users to make substantial modifications to their existing STE proofs, and has shown useful benefits in experimental usage. We have found that our methodological constraints are not burdensome in practice, and that we can handle all the forms of environmental specification that arise in industrial examples.

The task of verifying the quality of STE antecedents has also been addressed at Intel by translation into checkers (in a different language) with less aggressive vectorization [7]. This translation imposes a different structuring methodology on the STE environment that allows more flexible symbolic indexing, but also rules out implicit equality constraints coded by tuples. The assertions generated appear larger and significantly more numerous than with our method. On the other hand, this method can handle more complex STE proof environments.

Recently we have also used the SVA produced by our translation to generate stuck-at-tests for execution cluster outputs. Here, the SVA properties were used as assumptions. This experimental application aims to leverage the effort put into creating an STE proof environment for post-silicon validation.

Acknowledgments

The authors thank Yulik Feldman and Dmitry Korchemny for helpful advice about SVA, and Zenya Belfer for assistance with running the VCS simulator. Shachar Finkelstein was largely responsible for deploying STE2SVA and provided us with benchmark data. Roope Kaivola kindly provided helpful comments on a draft of the paper.

References

[1] C.-J. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, Mar. 1995.

[2] M. Pandey, R. Raimi, R. E. Bryant, and M. S. Abadir, "Formal verification of content addressable memories using symbolic trajectory evaluation," in *Design Automation Conference*. ACM Press, 1997, pp. 167–172.

[3] T. Schubert, "High Level Formal Verification of Next-Generation Microprocessors," in *DAC'03: Proceedings of the 40th conference on design automation*. ACM Press, 2003, pp. 1–6.

[4] R. Kaivola and K. R. Kohatsu, "Proof engineering in the large: formal verification of Pentium® 4 floating-point divider," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 3, pp. 323–334, 2003.

[5] C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme, "An industrially effective environment for formal hardware verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 9, pp. 1381–1405, Sept. 2005.

[6] A. Flaisher, A. Gluska, and E. Singerman, "Case study: Integrating FV and DV in the verification of the Intel® Core™ 2 Duo microprocessor," in *Formal Methods in Computer Aided Design: FMCAD 2007*, J. Baumgartner and M. Sheeran, Eds. IEEE Computer Society, 2007, pp. 192–195.

[7] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodova, C. Taylor, V. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in Intel core i7 processor execution engine validation," in *Computer Aided Verification, CAV 2009: Proceedings*, ser. LNCS, A. Bouajjani and O. Maler, Eds. Springer-Verlag, 2009.

[8] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[9] J. Grundy, T. Melham, and J. O'Leary, "A reflective functional language for hardware design and theorem proving," *Journal of Functional Programming*, vol. 16, no. 2, pp. 157–196, Mar. 2006.

[10] *1800: IEEE Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language*. IEEE Computer Society, 2007.

[11] H. D. Foster and A. C. Krolnik, *Creating Assertion-Based IP*. Springer-Verlag, 2008.

[12] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger, "Formal hardware verification by symbolic ternary trajectory evaluation," in *ACM/IEEE Design Automation Conference*. ACM Press, Jun. 1991, pp. 397–402.

[13] T. F. Melham and R. B. Jones, "Abstraction by symbolic indexing transformations," in *Formal Methods in Computer-Aided Design: 4th International Conference, FMCAD 2002*, ser. LNCS, M. D. Aagaard and J. W. O'Leary, Eds., vol. 2517. Springer-Verlag, 2002, pp. 1–18.

[14] S. Adams, M. Björk, T. Melham, and C.-J. Seger, "Automatic abstraction in symbolic trajectory evaluation," in *Formal Methods in Computer Aided Design: FMCAD 2007*, J. Baumgartner and M. Sheeran, Eds. IEEE Computer Society, 2007, pp. 127–135.

[15] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Formal verification using parametric representations of Boolean constraints," in *ACM/IEEE Design Automation Conference*, Jul. 1998.

[16] L. Hellerman, "A catalog of three-variable Or-Invert and And-Invert logical circuits," *IEEE Transactions on Electronic Computers*, vol. EC-12, Jun 1963.