Noname manuscript No.
(will be inserted by the editor)

Symbolic Trajectory Evaluation for Word-Level Verification: Theory and Implementation

Supratik Chakraborty · Zurab Khasidashvili · Carl-Johan H. Seger · Rajkumar Gajavelly · Tanmay Haldankar · Dinesh Chhatani · Rakesh Mistry

Abstract Symbolic trajectory evaluation (STE) is a model checking technique that has been successfully used to verify many industrial designs. Existing implementations of STE reason at the level of bits, allowing signals in a circuit to take values from a lattice comprised of three elements: 0, 1, and X. This limits the amount of abstraction that can be achieved, and presents limitations to scaling STE to even larger designs. The main contribution of this paper is to show how much more abstract lattices can be derived automatically from register-transfer level (RTL) descriptions, and how a model checker for the general theory of STE instantiated with such abstract lattices can be implemented in practice. We discuss several implementation issues, including how word-level circuits can be symbolically simulated using a new encoding for words that allows representing X values of sub-words succinctly. This gives us the first practical word-level STE engine, called STEWord. Experiments on a set of designs similar to those used in industry show that STEWord scales better than bit-level STE, as well as word-level bounded model checking.

Keywords symbolic trajectory evaluation · word-level verification · SMT solving · X-based abstraction · hardware verification · RTL verification · invalid-bit encoding · symbolic simulation

S. Chakraborty
Department of Computer Science and Engineering, IIT Bombay, India
Tel: +91-22-25767721
Fax: +91-22-25720022
E-mail: supratik@cse.iitb.ac.in

Z. Khasidashvili
Intel IDC, Haifa, Israel
E-mail: zurab.khasidashvili@intel.com

C.-J. H. Seger
was at Intel, Portland, Oregon, USA when working on this project
E-mail: cjhseger@gmail.com

R. Gajavelly · T. Haldankar · D. Chhatani · R. Mistry
were at Department of Computer Science and Engineering, IIT Bombay, India when working on this project.

1 Introduction

Symbolic Trajectory Evaluation (STE) is a model checking technique that grew out of multi-valued logic simulation on the one hand, and symbolic simulation on the other hand [5]. Among various formal verification techniques in use today, STE comes closest to functional simulation and is among the most successful formal verification techniques used in the industry. In STE, specifications take the form of symbolic trajectory formulas that mix Boolean expressions and the temporal next-time operator. The Boolean expressions provide a convenient means of describing different operating conditions in a circuit in a compact form. By allowing only the most elementary of temporal operators, the class of properties that can be expressed is fairly restricted as compared to other temporal logics (see [14] for a nice survey). Nonetheless, experience has shown that many important aspects of synchronous digital systems at various levels of abstraction can be captured using this restricted logic. For example, it is quite adequate for expressing many of the subtleties of system operation, including clocking schemas, pipelining control, as well as complex data computations [24, 19, 18].

In return for the restricted expressiveness of STE specifications, the STE model checking algorithm provides significant computational efficiency. As a result, STE can be applied to much larger designs than any other model checking technique. For example, STE is routinely used in the industry today to carry out complete formal input-output verification of designs with several hundred thousand latches [19, 18]. Unfortunately, this still falls short of providing an automated technique for formally verifying modern system-on-chip designs, and there is clearly a need to scale up the capacity of STE even further.

The first approach that was pursued in this direction was structural decomposition. In this approach, the user must break down a verification task into smaller sub-tasks, each involving a distinct STE run. After this, a deductive system can be used to reason about the collections of STE runs and verify that they together imply the desired property of the overall design [17]. In theory, structural decomposition allows verification of arbitrarily complex designs. However, in practice, the difficulty and tedium of breaking down a property into small enough sub-properties that can be verified with an STE engine limits the usefulness of this approach significantly. In addition, managing the structural decomposition in the face of rapidly changing register-transfer level (RTL) descriptions limits the applicability of structural decomposition even further.

A different approach to increase the scale of designs that can be verified is to use aggressive abstraction beyond what is provided automatically by current STE implementations. If we ensure that our abstract model satisfies the requirements of the general theory of STE, then a property that is verified on the abstract model holds on the original model as well. Although the general theory of STE allows a very general circuit model [23], all STE implementations so far have used a four valued circuit model. Thus, every bit-level signal is allowed to take one of four values: 0, 1, X or $\top$, where X represents "either 0 or 1" and $\top$ represents "neither 0 nor 1". This limits the amount of abstraction that can be achieved. The main contribution of this paper is to show how much more abstract lattices can be derived automatically from RTL descriptions, and how the general theory of STE can be instantiated with such a lattice to give a practical word-level STE engine that provides significant gains in capacity and efficiency on a set of benchmarks.

Operationally, word-level STE bears some similarities with word-level bounded model checking (BMC). However, there are important differences, the most significant one being the use of X-based abstractions on entire words or on multi-bit slices of words in word-level STE. This allows a wide range of abstraction possibilities, including a combination of user-specified and automatic abstractions – often a necessity for complex verification tasks. Our preliminary experimental results indicate that by carefully using X-based abstractions in word-level STE, it is indeed possible to strike a good balance between accuracy (by cautious propagation of X's) and performance (by liberal propagation of X).

The remainder of the paper is organized as follows. We begin with an overview of the general theory of STE, and of existing STE implementations in Section 2. This leads to the question of whether we can build a practical STE engine that reasons about multi-bit words directly without bit-blasting [12] them. In Section 3, we discuss how words in an RTL design can be split into slices, or *atoms*, such that (most) RTL operators either read or write an entire atom or no part of it. Atoms form the basis of abstracting groups of bits in our work. In Section 4, we elaborate on the lattice of values that this atomization and resulting abstraction generates. Section 5 presents a new way of encoding values of atoms in this lattice. We also discuss how to symbolically simulate RTL operators and compute least upper bounds using this encoding. Section 6 presents an instantiation of the general theory of STE using the above lattice, and discusses an implementation. In Section 7, we discuss various implementation issues that contribute to building a practical word-level STE engine. Experimental results on a set of RTL benchmarks are presented in Section 8, and we conclude in Section 9.

2 An overview of STE

In this section, we briefly review the general theory of STE [23], and discuss existing STE implementations. A digital design M consists of primary inputs, primary outputs, state-holding elements and computational blocks (also called gates or operators) driving internal signals or state-holding elements. Let Sig denote the collection of primary inputs, primary outputs, state elements and internal signals of M. For convenience of exposition, we refer to all elements of Sig as "signals" in the following discussion. In general, signals can be multi-bit wide, and are assumed to take values from a bounded lattice $(\mathcal{D}, \sqsubseteq_{\mathcal{D}}, \sqcup_{\mathcal{D}}, \sqcap_{\mathcal{D}}, \top_{\mathcal{D}}, \bot_{\mathcal{D}})$, where $\mathcal{D}$ denotes the set of possible values, $\sqsubseteq_{\mathcal{D}}$ denotes a partial ordering of the "information content" of values, the operators $\sqcup_{\mathcal{D}}$ and $\sqcap_{\mathcal{D}}$ denote least upper bound and greatest lower bound respectively, $\top_{\mathcal{D}}$ represents an unachievable over-constrained value, and $\bot_{\mathcal{D}}$ represents an unconstrained value with no information content. As an example, if all signals are 1-bit wide, the relevant lattice has four elements $\{0, 1, X, \top\}$, where 0 and 1 denote the usual binary values of bit-level signals, X (representing $\bot$) denotes "either 0 or 1", and $\top$ denotes an unachievable over constrained value or "neither 0 nor 1". The ordering of information in the values 0, 1, X, and $\top$ is as shown in the Hasse diagram in Fig. 1, where a value lower in the order has "less information" than one higher up in the order.

A state of the design M is a mapping $s : \mathsf{Sig} \to \mathcal{D}$. Let $\mathfrak{S}$ denote the set of all states of the design. Clearly $\mathfrak{S}$ forms a lattice – one that is isomorphic to the

product of lattices of values of signals in Sig. In the subsequent discussion, we use $(\mathfrak{S}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ to denote this lattice of states of M.

To model the behavior of M over time, we define a *sequence* of states as a mapping $\sigma : \aleph \to \mathfrak{S}$, where $\aleph$ denotes the set of natural numbers. Given two sequences σ_1 and σ_2, we abuse notation and say that $\sigma_1 \sqsubseteq \sigma_2$ iff for every $t \in \aleph$, $\sigma_1(t) \sqsubseteq \sigma_2(t)$. Let $\mathsf{Tr}_M : \mathfrak{S} \to \mathfrak{S}$ define the transition function of design M. Thus, given a state s of M at time t, the next state of the design at time $t+1$ is given by $\mathsf{Tr}_M(s)$. A *trajectory* for M is a sequence σ such that for all $t \in \aleph$, we have $\mathsf{Tr}_M(\sigma(t)) \sqsubseteq \sigma(t+1)$. Note that not all sequences are trajectories for M.

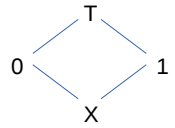

Fig. 1 Ternary lattice

The *symbolic trajectory evaluation logic* of Seger and Bryant [23] provides a simple yet useful language to specify properties of trajectories. Specifically, let $\mathcal{P}$ denote a set of *simple* predicates over $\mathfrak{S}$ (set of states), and let $\mathcal{V}$ denote a set of symbolic Boolean variables. A predicate p is *simple* if there is a unique state $s \in \mathfrak{S}$ such that (i) $p(s)$ is true, and (ii) for every state s' such that $p(s')$ is true, $s \sqsubseteq s'$. The unique state s referred to above is also called the *defining value* of p, and is denoted $\mathsf{def}(p)$. A *symbolic trajectory formula* can now be defined as a formula generated by the following grammar:

$$\varphi \quad ::= \quad p, \text{ where } p \in \mathcal{P} \quad | \quad \varphi \wedge \varphi \quad | \, G \to \varphi \quad | \quad \mathbf{N}\varphi,$$

where G is a Boolean formula over the variables in $\mathcal{V}$, and $\mathbf{N}$ denotes the next-time operator. Formulas like G in the grammar above are also called *guards* in STE parlance. The *depth* of a formula φ is one greater than the number of nested next-time operators in φ. For example, the depth of the formula $\mathbf{N}p \wedge \mathbf{N}(\mathbf{N}p)$ is 3, if p is a simple predicate.

An assignment $\phi : \mathcal{V} \to \{0, 1\}$ assigns a truth value (0 for false and 1 for true) to every variable in $\mathcal{V}$. We say that ϕ *satisfies* a Boolean formula G, denoted $\phi \models G$, if substituting every variable $v \in \mathcal{V}$ that appears in G with $\phi(v)$ causes G to evaluate to true. A sequence of states σ is said to *satisfy* a symbolic trajectory formula ψ with respect to an assignment ϕ, denoted $\sigma \models \psi(\phi)$, if the following hold, where σ^i refers to the i^{th} state in the sequence σ.

- $\sigma^0\widetilde{\sigma} \models p(\phi)$ iff $\mathsf{def}(p) \sqsubseteq \sigma^0$.
- $\sigma \models (\psi_1 \wedge \psi_2)(\phi)$ iff $\sigma \models \psi_1(\phi)$ and $\sigma \models \psi_2(\phi)$
- $\sigma \models (G \to \psi)(\phi)$ iff $\phi \not\models G$, or $\sigma \models \psi(\phi)$
- $\sigma^0\widetilde{\sigma} \models (\mathbf{N}\psi)(\phi)$ iff $\widetilde{\sigma} \models \psi(\phi)$

The *defining sequence* of a symbolic trajectory formula ψ with respect to an assignment ϕ, denoted $[\psi]^\phi$, can now be inductively defined as follows. In the following, $t \in \aleph$ denotes a time instant, and $\mathsf{b} \in \mathsf{Sig}$ denotes a signal in the design M.

- $[p]^\phi(t)(\mathsf{b}) \triangleq \mathsf{def}(p)(\mathsf{b})$ if $t = 0$, and is $\bot$ otherwise.
- $[\psi_1 \wedge \psi_2]^\phi(t)(\mathsf{b}) \triangleq [\psi_1]^\phi(t)(\mathsf{b}) \sqcup [\psi_2]^\phi(t)(\mathsf{b})$
- $[G \to \psi]^\phi(t)(\mathsf{b}) \triangleq [\psi]^\phi(t)(\mathsf{b})$ if $\phi \models G$, and is $\bot$ otherwise.
- $[N\psi]^\phi(t)(\mathsf{b}) \triangleq [\psi]^\phi(t-1)(\mathsf{b})$ if $t \neq 0$, and is $\bot$ otherwise.

By a key result of the general theory of STE [23], for every assignment ϕ and for every symbolic trajectory formula ψ, the defining sequence $[\psi]^\phi$ is the uniquely defined weakest (least with respect to $\sqsubseteq$) sequence satisfying ψ, assuming variables in $\mathcal{V}$ are assigned values as given by ϕ. Note that this may not be a trajectory for the design M.

The *defining trajectory* of ψ with respect to M and assignment ϕ, denoted $[\![\psi]\!]^{\phi}_{M}$, is similarly the weakest (least with respect to $\sqsubseteq$) trajectory of M satisfying ψ, given that variables in $\mathcal{V}$ are assigned values as specified by ϕ. The defining trajectory is defined as follows, where $t \in \aleph$ denotes time and b denotes a signal in M.

- $[\![\psi]\!]^{\phi}_{M}(0)(\mathsf{b}) \triangleq [\psi]^{\phi}(0)(\mathsf{b})$
- $[\![\psi]\!]^{\phi}_{M}(t+1)(\mathsf{b}) \triangleq [\psi]^{\phi}(t+1)(\mathsf{b}) \sqcup \mathsf{Tr}_{M}([\![\psi]\!]^{\phi}_{M}(t))(\mathsf{b})$ for every $t \in \aleph$.

Symbolic trajectory evaluation concerns checking whether a design satisfies a *symbolic trajectory assertion*, for all valuations of the symbolic variables in $\mathcal{V}$. For purposes of this paper, we focus on *iteration-free* symbolic trajectory assertions, as defined in [23]. Specifically, such an assertion is inductively defined as follows: if A (antecedent) and C (consequent) are symbolic trajectory formulas of the same depth, $[A \Rightarrow C]$ is a symbolic trajectory assertion; in addition, if F is a symbolic trajectory assertion, then so is $[A \Rightarrow C]\,;F$.

A design M is said to satisfy $[A \Rightarrow C]$ if every trajectory of M that satisfies A also satisfies C. The central theorem of the theory of symbolic trajectory evaluation [23] asserts that checking the above satisfaction is equivalent to checking if for every assignment ϕ of symbolic variables in $\mathcal{V}$, we have $[C]^{\phi} \sqsubseteq [\![A]\!]^{\phi}_{M}$. In case the above ordering doesn't hold for all assignments ϕ, symbolic trajectory evaluation also determines the set of assignments (represented as a Boolean expression over $\mathcal{V}$) under which $[C]^{\phi} \sqsubseteq [\![A]\!]^{\phi}_{M}$ holds.

Existing implementations of STE reason at the level of bits, i.e. each signal in the design is assumed to be 1-bit wide. The corresponding lattice of values has three non-$\top$ elements, 0 1 and X, as shown in Fig. 1. In order to symbolically reason about ternary values, STE tools usually use dual-rail encoding, in which every bit-level signal v is encoded using two binary variables v_0 and v_1. Intuitively, v_i indicates whether v can take the value i, for i in $\{0,1\}$. Thus, 0, 1 and X are encoded by the valuations $(0,1)$, $(1,0)$ and $(1,1)$, respectively, of (v_1, v_0). By convention, $(v_1, v_0) = (0,0)$ denotes $\top$.

The semantics of logical operators like AND and NOT can be easily extended to work over the $\{0, 1, X, \top\}$ lattice. Specifically, let u_0, u_1, v_0, v_1 be binary values such that (u_1, u_0) and (v_1, v_0) encode ternary values using dual-rail encoding. It is easy to see that $(u_1, u_0)\,\mathsf{AND}\,(v_1, v_0) = (u_1 \wedge v_1, u_0 \vee v_0)$ and $\mathsf{NOT}(u_1, u_0) = (u_0, u_1)$, where $\vee$ and $\wedge$ denote Boolean conjunction and disjunction respectively.

It should be noted, however, that the extension of semantics of logical operators is not unique. In fact the only requirement it must satisfy is monotonicity, i.e., given an n-argument Boolean operator $\circ$ and $a_i \sqsubseteq_{\mathcal{D}} b_i$ for $1 \leq i \leq n$, we must have $\circ(a_1, ..., a_n) \sqsubseteq_{\mathcal{D}} \circ(b_1, ..., b_n)$. The least pessimistic extension $\bullet$ of $\circ$ is defined as:

$$\forall \mathbf{a}_1, \ldots \mathbf{a}_n \in \mathcal{D},\ \bullet(\mathbf{a}_1, \ldots \mathbf{a}_n) = \sqcap_{\mathcal{D}}\{\circ(a_1, \ldots a_n) \mid a_i \in \{0,1\} \ \wedge\ \mathbf{a_i} \sqsubseteq_{\mathcal{D}} a_i\}$$

Although this provides an extension that is as accurate as possible, while still satisfying the monotonicity requirements, in practice computing this extension for an operator taking many inputs can be prohibitively expensive. As a pragmatic solution, the extension of complex operators are derived by first expressing the operator in terms of AND and NOT and then using the extensions of these primitive operators. However, it should be noted that for some situations, this simplistic approach is insufficient. For example, consider an if-then-else operator, denoted as $\mathsf{ite}(c, t, e)$. If we use the least pessimistic extension, the output will be 1 if

both the t and e inputs are 1 and c is X. However, if we use the expression NOT(NOT(c AND t) AND NOT(NOT(c) AND e)), it is easy to see that the result will be X for this assignment.

Using dual-rail encoding, the least upper bound of (u_1, u_0) and (v_1, v_0) is given by $(u_1 \wedge v_1, u_0 \wedge v_0)$. To check whether $(u_1, u_0) \sqsubseteq_{\mathcal{D}} (v_1, v_0)$, it suffices to check if $v_0 \rightarrow u_0$ and $v_1 \rightarrow u_1$, where $\rightarrow$ denotes logical implication. State-of-the-art bit-level STE tools, viz. Forte [24], use dual-rail encoding to effectively reason about ternary-valued signals via their binary encoding. This makes it possible to solve bit-level STE problems by leveraging the significant advances made in Boolean reasoning engines over the past three decades, viz. Binary Decision Diagram (BDD) packages [6,25], And-Inverter-Graph (AIG) packages [3] and propositional satisfiability (SAT) solvers [13].

3 Atomizing words

While dual-rail encoding provides an elegant way to harness the power of Boolean reasoning engines to reason about ternary-valued signals, it has an undesired consequence as well. Specifically, if there are n binary signals in a circuit, post dual-rail encoding, we must deal with $2n$ binary variables in the encoded circuit. This can pose serious scalability issues when verifying designs with wide datapaths, large memories, etc. Attempts to scale STE to large designs must therefore raise the level of abstraction beyond that of individual bits. Raising the level of abstraction in STE has other associated benefits as well. In recent years, there has been significant progress in the development of Satisfiability Modulo Theories (SMT) solvers that incorporate highly sophisticated word-level reasoning engines. Examples of such solvers include Z3 [10], CVC4 [2], Boolector [4], Beaver [15], MathSAT [9], Yices [11] and the like. Therefore, an STE engine that reasons about words can harness the power of word-level reasoning in modern SMT solvers. For circuits with wide datapaths (words), this can lead to significant computational efficiency vis-a-vis bit-level STE engines, when checking symbolic trajectory assertions.

In order to use word-level STE, we must first define a lattice of values for words, and then instantiate the general theory of STE [23] with this lattice. For an m-bit word, two extreme choices for the lattice of values are as follows.

1. A "cautious" lattice that keeps track of the ternary value of each bit in the word independently. Elements of this lattice are m-dimensional vectors of values from $\{0, 1, X, \top\}$ (see Fig. 1), where the i^{th} component of the vector gives the value of the i^{th} bit in the word. Since $\top$ represents an unachievable value for a signal, a vector having $\top$ in at least one component represents an unachievable value for the word. Therefore, all vectors with one or more components set to $\top$ can be collapsed to a single unachievable value for the entire word (top element of the lattice of values).
2. A "liberal" lattice that keeps track of the values of individual bits only if these values are in $\{0, 1\}$. If any bit has the value X, the value of the entire word is liberally treated as unknown (bottom element of the lattice of values). For example, in a 2-bit word, this lattice doesn't make a distinction between the first bit being X and the second bit being X – in both cases, the word is considered to have an unknown value. As in the previous case, if any bit in the

word has the value $\top$, the entire word is considered to have an unachievable value (top element of the lattice of values).

Unfortunately, both the above lattices present practical difficulties in implementation. In the case of the cautious lattice, there are $3^m + 1$ values in the lattice for an m-bit word. Symbolically representing values from such a large lattice and reasoning about them is likely to incur overheads similar to that incurred in bit-level STE. Thus, while the cautious lattice promises to maintain a high degree of precision when reasoning about words, the scalability of an STE engine using this lattice will be poor. The liberal lattice, on the other hand, has only $2^m + 2$ values for an m-bit word. This is almost the same as the count of values the word can take if we allow only $\{0, 1\}$ values for the individual bits. An STE engine using this lattice is therefore likely to scale much better. However, the precision of the analysis will be poor, since any bit being unknown in a wide word renders the value of the entire word unknown. For example, consider a 128-bit word w, whose least significant bit has the value X, while all other bits have the value 0. In the liberal lattice, we must represent the value of w using the bottom element (i.e., unknown value). Extracting the most significant 64 bits of w therefore gives us an unknown value using the liberal lattice, although the extracted bits are really all 0s. Therefore, the lattice of values of words must be carefully chosen to strike a fine balance between precision and scalability. We propose one such approach below in which we split a word into appropriate sub-words, and use a liberal lattice for each sub-word, while keeping track of the values of different sub-words independently, as in a cautious lattice.

The idea of splitting words into sub-words for the purpose of simplifying analysis is not new (see e.g. [16]). An aggressive approach to splitting (e.g., bit-blasting) can lead to proliferation of narrow sub-words, making our technique vulnerable to the same scalability problems that arise with dual-rail encoding. Therefore, we adopt a more controlled approach to splitting. Specifically, we wish to split words in such a way that we can speak of an entire sub-word having an unknown value without having to worry about which individual bits in the sub-word have the value X. Towards this end, we partition every word in an RTL design into sub-words, which we henceforth call *atoms*, such that every RTL statement (except a few discussed later) that reads or updates a word either does so for all bits in an atom, or for no bit in an atom. In other words, no RTL statement (except the few discussed at the end of this section) reads or updates an atom partially.

Formalizing atomization: Let w be a word of width m in an RTL design M. Let 0 denote the least significant and rightmost bit position, and $m - 1$ denote the most significant and leftmost bit position of w. For integer constants p, q such that $0 \leq p \leq q \leq m - 1$, we say that the sub-word of w from bit position p to q is a *slice* of w, and denote it by $w[q : p]$. Let $\mathsf{SliceOp}(w, q, p)$ be a generic RTL operator that either reads or writes the slice $w[q : p]$. Concrete instances of $\mathsf{SliceOp}$ are commonly used in RTL designs, e.g. in the SystemVerilog [1] statement `c[4:1] = a[10:7] + b[5:2]`. We say that $\mathsf{SliceOp}(w, q, p)$ *induces an atomization* of w, as shown in Table 1, where Atoms_w denotes the set of atoms into which w is partitioned.

Note that different RTL statements may refer to different slices of the same word w. Consequently, atomizations of w induced by different operators in these

Condition	Atoms_w
$q < m-1$ and $p > 0$	$\{w[m-1:q+1], w[q:p], w[p-1:0]\}$
$q < m-1$ and $p = 0$	$\{w[m-1:q+1], w[q:0]\}$
$q = m-1$ and $p > 0$	$\{w[m-1:p], w[p-1:0]\}$
$q = m-1$ and $p = 0$	$\{w[m-1:0]\} = \{w\}$

Table 1 Computing atoms induced by $\mathsf{SliceOp}(w, q, p)$

RTL statements may not coincide. We consolidate different atomizations of the same word by computing their *coarsest refinement*. Given two sets of atoms, $\mathsf{Atoms}_w^{(1)}$ and $\mathsf{Atoms}_w^{(2)}$, of an m-bit word w, their *coarsest refinement*, denoted $\mathsf{Atoms}_w^{(1)} \Downarrow \mathsf{Atoms}_w^{(2)}$, is a set of atoms of w with the following property: for every $m_1, m_2 \in \{0, \ldots m-1\}$, the 1-bit slices $w[m_1 : m_1]$ and $w[m_2 : m_2]$ belong to the same atom in $\mathsf{Atoms}_w^{(1)} \Downarrow \mathsf{Atoms}_w^{(2)}$ iff they belong to the same atom in both $\mathsf{Atoms}_w^{(1)}$ and $\mathsf{Atoms}_w^{(2)}$. It is easy to verify that the coarsest refinement is uniquely defined, and is a commutative and associative operator. This suggests a simple algorithm for determining the coarsest refinement of all atomizations of a word w induced by operators in various statements in an RTL design. For every word $w[m-1:0]$ in the RTL design, we maintain a working set, $\mathsf{WSetAtoms}_w$, of atoms. Initially, $\mathsf{WSetAtoms}_w$ is initialized to $\{w[m-1:0]\}$. For every operator $\mathsf{SliceOp}$ that refers to a slice of w in an RTL statement, we compute Atoms_w using Table 1, and determine the coarsest refinement of Atoms_w and $\mathsf{WSetAtoms}_w$. The working set $\mathsf{WSetAtoms}_w$ is then updated to the coarsest refinement thus computed, and the above process repeated for every RTL statement in the design.

We illustrate the result of applying the above atomization algorithm on a simple example below. Fig. 2(a) shows a SystemVerilog code fragment, and Fig. 2(b) shows the resulting coarsest refinement of atomization of words, where the solid vertical bars represent the boundaries of atoms. Note that every SystemVerilog statement in Fig. 2(a) either reads or writes all bits in an atom, or no bit in an atom.

Since we wish to reason at the granularity of atoms, we must interpret word-level reads and writes in terms of the corresponding atom-level reads and writes. This can be done either by modifying the RTL, or by taking appropriate care when symbolically simulating the RTL. For simplicity of presentation, we show in Fig. 2(c) how the code fragment in Fig. 2(b) would appear if we were to use only the atoms identified in Fig. 2(b). Note that no statement in the modified RTL updates or reads a slice of an atom. However, a statement may be required to read a slice of the result obtained by applying an RTL operator to atoms (see, for example, Fig. 2(c) where we read a slice of the result obtained by adding concatenated atoms). In our implementation, we do not modify the original RTL. Instead, we symbolically simulate the original RTL, but take care to generate the expressions for various atoms as if they were obtained from simulating the modified RTL.

Once the boundaries of all atoms are determined, we choose to disregard values of atoms in which some bits are set to X, and the others are set to 0 or 1. This choice is justified since all bits in an atom are read or written together. Thus, either all bits in an atom are considered to have values in $\{0, 1\}$, or all of them are considered

to have the value X. This implies that values of an m-bit atom can be encoded using $m+1$ bits, instead of using $2m$ bits as in dual-rail encoding. Specifically, we can associate an extra "invalid" bit with every m-bit atom. Whenever the "invalid" bit is set, all bits in the atom are assumed to have the value X. Otherwise, all bits are assumed to have values in $\{0, 1\}$. We show later in Sections 5.2.1 and 5.2.2 how the value and invalid bit of an atom can be recursively computed from the values and invalid bits of the atoms on which it depends.

Memories and arrays in an RTL design are usually indexed by variables instead of by constants. This makes it difficult to atomize memories and arrays statically, and we do not atomize them. Similarly, if a design has a logical shift operation, where the amount of shift is specified by a variable, it is difficult to statically identify sub-words that are not split by the shift operation. We ignore all such RTL operations during atomization, and instead use special techniques to reason about them. Section 5.2.2 discusses this in further detail.

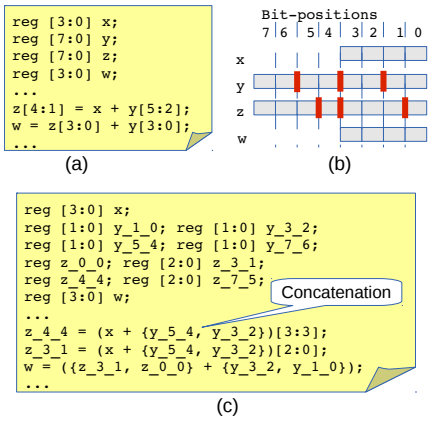

Fig. 2 Illustrating atomization

4 Lattice of atom values

Recall that our primary motivation for atomizing words was to have a liberal lattice for atoms. In other words, if any bit in an atom has the value X, we want to represent the entire atom as having an unknown value, without keeping track of which which bits in the atom had the value X. Therefore, we let an m-bit atom a take values from the set $\{\overbrace{0\cdots00}^{\text{m bits}}, \ldots \overbrace{1\cdots11}^{\text{m bits}}, \mathbf{X}\}$, where $\mathbf{X}$ is a single abstract value that denotes an assignment of X to at least one bit of a. Note the conspicuous absence of values like $0X1\cdots0$ in the above set. Fig. 3(a) shows the lattice of values for a 3-bit atom, ordered by information content. The $\top$ element is added to complete the lattice, and represents an unachievable over-constrained value. Fig. 3(b) shows the lattice of values of the same atom if we allow each bit to take values in $\{0, 1, X\}$. Clearly, the lattice in Fig. 3(a) is shallower and sparser than that in Fig. 3(b).

Consider an m-bit word w that has been partitioned into non-overlapping atoms of widths $m_1, \ldots m_r$, where $\sum_{j=1}^{r} m_j = m$. The lattice of values of w is given by the product of r lattices, each corresponding to the values of an atom of w. For convenience of representation, we simplify the product lattice by collapsing all values that have at least one atom set to $\top$ (and therefore represent unachievable over-constrained values), to a single $\top$ element. It can be verified that the height of the product lattice (after the above simplification) is given by $r+1$, the total number of elements in it is given by $\prod_{j=1}^{r}\left(2^{m_j}+1\right)+1$ and the number of elements at level i from the bottom is given by $\binom{r}{i}\prod_{j=1}^{i} 2^{m_j}$, where

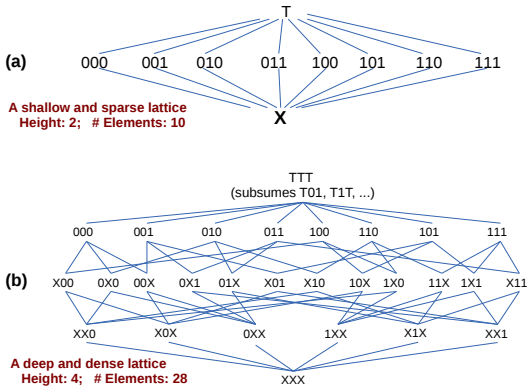

Fig. 3 Atom-level and bit-level lattices

$0 < i \leq r$. It is not hard to see from these expressions that atomization using few wide atoms (i.e., small values of r and large values of m_j) gives shallow and sparse lattices compared to atomization using many narrow atoms (i.e., large values of r and small values of m_j). The special case of a bit-blasted lattice (see Fig. 3(b)) is obtained when $r = m$ and $m_j = 1$ for every $j \in \{1, \ldots m\}$.

Using a sparse lattice is advantageous in symbolic reasoning since we need to encode a small set of values. Using a shallow lattice helps in converging fast when computing least upper bounds – an operation that is crucially needed when performing symbolic trajectory evaluation. However, making the lattice of values sparse and shallow comes at the cost of losing precision of reasoning. By atomizing words based on their actual usage in an RTL design, and by abstracting values of atoms wherein some bits are set to X and the others are set to 0 or 1, we strike a balance between depth and density of the lattice of values, which in turn impacts scalability, on one hand, and precision of reasoning on the other.

5 Symbolic simulation with invalid-bit encoding

As mentioned earlier, an m-bit atom can be encoded with $m+1$ bits by associating an "invalid bit" with the atom. For notational convenience, we use $\mathsf{val}(a)$ to denote the value of the m bits constituting atom a, and $\mathsf{inv}(a)$ to denote the value of its invalid bit. Thus, an m-bit atom a is encoded as a pair $(\mathsf{val}(a), \mathsf{inv}(a))$, where $\mathsf{val}(a)$ is a bit-vector of width m, and $\mathsf{inv}(a)$ is of $\mathsf{Boolean}$ type. Given $(\mathsf{val}(a), \mathsf{inv}(a))$, the value of a is given by $\mathsf{ite}(\mathsf{inv}(a), \mathbf{X}, \mathsf{val}(a))$, where "$\mathsf{ite}$" denotes the usual "if-then-else" operator. For convenience of exposition, we call this encoding "invalid-bit encoding". Note that invalid-bit encoding differs from dual-rail encoding even when $m = 1$. Specifically, if a 1-bit atom a has the value X, we can use either $(0, \mathsf{true})$ or $(1, \mathsf{true})$ for $(\mathsf{val}(a), \mathsf{inv}(a))$ in invalid-bit encoding. In contrast, there is a single value, namely $(a_0, a_1) = (1, 1)$, that encodes the value X of a in dual-rail encoding. We will see in Sections 5.2.2 and 7.3 how this degree of freedom in invalid-bit encoding of X can be exploited to simplify the symbolic simulation

of word-level operations on invalid-bit-encoded operands, and also to simplify the computation of least upper bounds.

Symbolic simulation and computation of least upper bounds (*lub*) are key components of symbolic trajectory evaluation. We discuss below how to compute the *lub* of two invalid-bit encoded values. In order to symbolically simulate an RTL design in which every atom is invalid-bit encoded, we must first determine the semantics of word-level RTL operators with respect to invalid-bit encoding. Towards this end, we present below a generic technique for computing the value component of the invalid-bit encoding of the result of applying a word-level RTL operator. Subsequently, we discuss how the invalid-bit component of the encoding is computed. All terms and formulas used in the following discussion are assumed to be in the theory of bit-vectors with equality.

5.1 Computing least upper bounds

Let $a = (\mathsf{val}(a), \mathsf{inv}(a))$ and $b = (\mathsf{val}(b), \mathsf{inv}(b))$ be invalid-bit encoded elements in the lattice of values for an m-bit atom. Let c be an element of the same lattice, defined as follows.

(i) If $(\neg\mathsf{inv}(a) \wedge \neg\mathsf{inv}(b) \wedge (\mathsf{val}(a) \neq \mathsf{val}(b))$, then $c = \top$.
(ii) Otherwise, $\mathsf{inv}(c) = \mathsf{inv}(a) \wedge \mathsf{inv}(b)$ and $\mathsf{val}(c) = \mathsf{ite}(\mathsf{inv}(a), \mathsf{val}(b), \mathsf{val}(a))$ (or equivalently $\mathsf{val}(c) = \mathsf{ite}(\mathsf{inv}(b), \mathsf{val}(a), \mathsf{val}(b))$).

Lemma 1 *The least upper bound of a and b in the lattice of values for an m-bit atom, is c.*

Proof In case (i), $lub(a, b)$ is clearly $\top$, which is also the value of c. If the condition in case (i) does not hold, we have three sub-cases.

- $\mathsf{inv}(a)$ is true and $\mathsf{inv}(b)$ is false: Since the value of a is $\mathbf{X}$, we must have $lub(a, b) = b$. The invalid-bit encoding of c in this case is $(\mathsf{val}(b), \mathsf{false})$. Clearly, $lub(a, b) = c$.
- $\mathsf{inv}(a)$ is false and $\mathsf{inv}(b)$ is true: Similar to the above sub-case.
- Both $\mathsf{inv}(a)$ and $\mathsf{inv}(b)$ are false and $\mathsf{val}(a) = \mathsf{val}(b)$: In this case, the values of a and b are identical and non-$\mathbf{X}$. Therefore, $lub(a, b) = a$ (or equivalently b). The invalid-bit encoding of c in this case has false for the invalid bit component, and $\mathsf{val}(a)$ (or $\mathsf{val}(b)$) for the value component. Therefore, $lub(a, b) = c$. □

Note the freedom in defining $\mathsf{val}(c)$ in case (ii) above. This freedom comes from the observation that if $\mathsf{inv}(c) = \mathsf{true}$, the actual value of $\mathsf{val}(c)$ is irrelevant. Furthermore, if the condition in case (i) is not satisfied and if both $\mathsf{inv}(a)$ and $\mathsf{inv}(b)$ are false, then $\mathsf{val}(b) = \mathsf{val}(a)$, even if the symbolic representations of $\mathsf{val}(a)$ and $\mathsf{val}(b)$ are different. This allows us to simplify the expression for $\mathsf{val}(c)$ by replacing it with $\mathsf{val}(b)$ or $\mathsf{val}(a)$, depending on which is simpler to represent.

5.2 Symbolically simulating RTL operators

Let op be a word-level RTL operator of arity k, and let *rexpr* be the result of applying op on $v_1, v_2, \ldots v_k$, i.e., $rexpr = \mathsf{op}(v_1, v_2, \ldots v_k)$. For each i in $\{1, \ldots k\}$, suppose the bit-width of operand v_i is m_i, and suppose the bit-width of *rexpr*

is m_{rexpr}. In general, *rexpr* is an expression in the theory of bit-vectors with equality, and the bit-width of *rexpr* is defined in the standard way (see, for example, Chapter 6 of [20]). We assume that each operand is invalid-bit encoded, and we are interested in computing the invalid-bit encoding of a specified slice of the result, say $rexpr[q:p]$, where $0 \leq p \leq q \leq m_{rexpr} - 1$. Let $\langle \mathsf{op} \rangle : \{0,1\}^{m_1} \times \cdots \times \{0,1\}^{m_k} \rightarrow \{0,1\}^{m_{rexpr}}$ denote the RTL semantics of op. For example, if op denotes 32-bit unsigned addition, then $\langle \mathsf{op} \rangle$ is the function that takes two 32-bit operands and returns their 32-bit unsigned sum. In the following discussion, for notational convenience, we abuse notation and use $\mathsf{inv}(expr)$ and $\mathsf{val}(expr)$ to denote the invalid bit and value of a bit-vector expression *expr*.

5.2.1 Computing the value component

The value component of the invalid-bit encoding of $rexpr[q:p]$, i.e. $\mathsf{val}(rexpr[q:p])$, can be computed if we know $\langle \mathsf{op} \rangle$ and $\mathsf{val}(v_i)$, for every $i \in \{1, \ldots k\}$. Significantly, we do not need $\mathsf{inv}(v_i)$ for any $i \in \{1, \ldots k\}$ to compute $\mathsf{val}(rexpr[q:p])$. The following lemma formalizes this, assuming $\langle \mathsf{op} \rangle(\mathsf{val}(v_1), \mathsf{val}(v_2), \ldots \mathsf{val}(v_k)))[q:p]$ is well-defined. Note that this precludes some corner cases for specific operators, viz. division by zero. For purposes of our discussion, we assume that these cases are handled by separate checks, and do not consider them below.

Lemma 2 *Let* $vexpr = \big(\langle \mathsf{op} \rangle(\mathsf{val}(v_1), \mathsf{val}(v_2), \ldots \mathsf{val}(v_k))\big)[q:p]$. *Then vexpr correctly computes* $\mathsf{val}(rexpr[q:p])$, *where* $rexpr = \mathsf{op}(v_1, v_2, \ldots v_k)$.

Proof By definition of invalid-bit encoding, if $\mathsf{inv}(rexpr[q:p])$ is true, the value of $\mathsf{val}(rexpr[q:p])$ does not matter. Hence, we focus on the case where $\mathsf{inv}(rexpr[q:p])$ is false. By definition, in this case, $rexpr[q:p]$ has a value in $\{0,1\}^{q-p+1}$. If the invalid bits of all operands v_i are false, then $\big(\langle \mathsf{op} \rangle(\mathsf{val}(v_1), \mathsf{val}(v_2), \ldots \mathsf{val}(v_k))\big)[q:p]$ clearly computes the value of $\mathsf{val}(rexpr[q:p])$. Otherwise, suppose $\mathsf{inv}(v_i) = \mathsf{true}$ for some $i \in \{1, \ldots k\}$. By the definition of invalid-bit encoding, v_i can have any value in $\{0,1\}^{m_i}$. However, since $\mathsf{inv}(rexpr[q:p])$ is false, it must be the case that $rexpr[q:p]$ evaluates to the same value in $\{0,1\}^{q-p+1}$, regardless of what value v_i takes in $\{0,1\}^{m_i}$. Therefore, we can set v_i to any value in $\{0,1\}^{m_i}$, and specifically to $\mathsf{val}(v_i)$, in order to obtain the correct value of $\mathsf{val}(rexpr[q:p])$. By repeating this argument for all v_i such that $\mathsf{inv}(v_i)$ is true, we see that $\big(\langle \mathsf{op} \rangle(\mathsf{val}(v_1), \mathsf{val}(v_2), \ldots \mathsf{val}(v_k))\big)[q:p]$ correctly computes $\mathsf{val}(rexpr[q:p])$. □

Example 1 Consider four invalid-bit encoded atoms, $a = (00, \mathsf{false})$, $b = (00, \mathsf{true})$, $c = (01, \mathsf{false})$ and $d = (01, \mathsf{true})$, where each atom is 2 bits wide. We wish to compute the invalid-bit encoding of $a +_2 c$, $b +_2 c$, $a +_2 d$ and $b +_2 d$, where $+_2$ denotes the two-bit arithmetic sum of its operands. Note that in only one of the four cases (i.e. $a +_2 c$) is the invalid bit of the sum false. However, by Lemma 2, the value 01 (computed as $00 +_2 01$) correctly gives the value component of the sum in each of the four cases. Indeed, when the invalid bit of the sum is true, the value component is irrelevant, and 01 is as good as any other value.

Lemma 2 tells us that when computing $\mathsf{val}(rexpr[q:p])$, we can effectively assume that invalid-bit encoding is not used. This significantly simplifies symbolic simulation with invalid-bit encoding. Note that this simplification would not have been possible if we did not have the freedom to ignore $\mathsf{val}(rexpr[q:p])$ when $\mathsf{inv}(rexpr[q:p])$ is true.

5.2.2 Computing the invalid bit component

We now turn to computing $\mathsf{inv}(rexpr[q:p])$, where $rexpr = \mathsf{op}(v_1, v_2, \ldots v_k)$. In general, this depends on $\mathsf{inv}(v_i)$ for all $i \in \{1, \ldots k\}$, and on $\mathsf{val}(v_j)$ for all j such that $\mathsf{inv}(v_j) = \mathsf{false}$. We first discuss how to obtain the invalid bit precisely. Unfortunately, the resulting formula involves quantifiers, and eliminating the quantifiers or reasoning about them may not always be computationally efficient in practice. Hence, we also present sound and simple approximations of $\mathsf{inv}(\mathsf{op}(v_1, \ldots v_k)[q:p])$ for several commonly occurring RTL operators (i.e. op).

Every argument v_i in $\mathsf{op}(v_1, v_2, \ldots v_k)$ can be thought of as representing the value of an m_i-bit wide atom, and hence is an element in the lattice defined by $\{\mathbf{0}, \mathbf{1}, \ldots\ \mathbf{2^{m_i}} - \mathbf{1}, \mathbf{X}, \top\}$. Let $\sqcap_m$ and $\sqsubseteq_m$ denote the greatest lower bound and partial ordering, respectively, in the lattice of values for an m-bit atom. The least pessimistic value of $rexpr[q:p]$ is then given by $\sqcap_{q-p+1}\{\langle\mathsf{op}\rangle(u_1, u_2, \ldots u_k)[q:p] \mid u_i \in \{0,1\}^{m_i} \wedge v_i \sqsubseteq_{m_i} u_i\}$. If we use invalid-bit encoding, the formula for $\mathsf{inv}(rexpr[q:p])$ must evaluate to true whenever the least pessimistic value of $rexpr[q:p]$, as given above, is $\mathbf{X}$.

We assume that $\langle\mathsf{op}\rangle(u_1, u_2, \ldots u_k)[q:p]$ evaluates to a value in $\{0,1\}^{q-p+1}$ if $u_i \in \{0,1\}^{m_i}$ for all $i \in \{1, \ldots k\}$. This assumption can be easily verified to hold for all RTL operators op in a language like SystemVerilog, with the exception of a few corner cases, viz. division by zero. As mentioned earlier, we assume that such corner cases are handled by separate checks, and do not worry about them in the subsequent discussion. Recalling the structure of the lattice of values for a $(q-p+1)$-bit atom, it is now easy to see that the least pessimistic value of $rexpr[q:p]$ is $\mathbf{X}$ iff $\langle\mathsf{op}\rangle(u_1, u_2, \ldots u_k)[q:p]$ evaluates to two distinct values in $\{0,1\}^{q-p+1}$, for two different tuples $(u_1, u_2, \ldots u_k)$, where $u_i \in \{0,1\}^{m_i}$ and $v_i \sqsubseteq_{m_i} u_i$ for all $i \in \{1, \ldots k\}$. We formalize this idea below to obtain a formula for $\mathsf{inv}(rexpr[q:p])$.

Let $\mu \in \{0,1\}^k$ denote a k-dimensional vector, and let μ_j denote the j^{th} component $(1 \leq j \leq k)$ of μ. For example, if $k = 5$ and $\mu = 10010$, then $\mu_2 = \mu_5 = 1$ and $\mu_1 = \mu_3 = \mu_4 = 0$. For every $\mu \in \{0,1\}^k$, define χ^μ to be the formula $\left(\bigwedge_{i:\mu_i=1} \mathsf{inv}(v_i)\right) \wedge \left(\bigwedge_{j:\mu_j=0} \neg\mathsf{inv}(v_j)\right)$. Furthermore, define $\overline{Y}$ to be the sequence of free variables $(y_1, y_2, \ldots y_k)$, where each y_i is a free variable of bit-width m_i (i.e. bit-width of v_i) that does not occur in any of $\mathsf{val}(v_1), \ldots, \mathsf{val}(v_k)$. We use $\overline{Y}$ and $\overline{Z}$ to denote two such distinct sequences, where y_i and z_i are *distinct* free variables for every i. As a natural extension, we also use $\langle\mathsf{op}\rangle(\overline{Y})$ to denote the bit-vector term obtained by applying the semantics of op on arguments that are the newly introduced free variables y_i; the interpretation of $\langle\mathsf{op}\rangle(\overline{Z})$ is analogous. For example, if $k = 5$, then $\langle\mathsf{op}\rangle(\overline{Y}) = \langle\mathsf{op}\rangle(y_1, y_2, y_3, y_4, y_5)$. Using the above notation, we can now give a formula for $\mathsf{inv}(rexpr[q:p])$.

Lemma 3 *Let* $rexpr = \mathsf{op}(v_1, v_2, \ldots v_k)$. *Then* $\mathsf{inv}(rexpr[q:p])$ *is given by*

$$\bigvee_{\mu \in \{0,1\}^k} \left(\chi^\mu \ \wedge\ \exists\overline{Y}\exists\overline{Z} \left(\begin{array}{c} \bigwedge_{i:\mu_i=0} ((y_i = z_i) \wedge (y_i = \mathsf{val}(v_i))) \\ \wedge \\ \left(\langle\mathsf{op}\rangle(\overline{Y})[q:p] \ \neq\ \langle\mathsf{op}\rangle(\overline{Z})[q:p]\right) \end{array} \right) \right)$$

Proof Consider any tuple $((\mathsf{val}(v_1), \mathsf{inv}(v_1)), \ldots (\mathsf{val}(v_k), \mathsf{inv}(v_k)))$ of invalid-bit encoded values of $(v_1, \ldots v_k)$. Without loss of generality, and using the notation introduced above, let $\mu \in \{0,1\}^k$ be a k-dimensional vector such that $\mathsf{inv}(v_i) = \mathsf{true}$ iff

$\mu_i = 1$. It follows that χ^μ is true, and χ^λ is false for all $\lambda \in \{0,1\}^k \setminus \{\mu\}$. The formula for $\mathsf{inv}(rexpr[q:p])$ therefore simplifies to $\exists \overline{Y} \exists \overline{Z} \left(\begin{array}{c} \bigwedge_{i:\mu_i=0} ((y_i = z_i) \wedge (y_i = \mathsf{val}(v_i))) \\ \wedge \\ \left(\langle \mathsf{op} \rangle(\overline{Y})[q:p] \neq \langle \mathsf{op} \rangle(\overline{Z})[q:p]\right) \end{array} \right)$. Denote this formula by Φ^μ.

From the semantics of first-order logic, Φ^μ evaluates to true iff there exist two tuples of values $\overline{Y^*} = (y_1^*, \ldots y_k^*)$ and $\overline{Z^*} = (z_1^*, \ldots z_k^*)$ such that the following conditions hold: (i) $y_i^*, z_i^* \in \{0,1\}^{m_i}$ for all $i \in \{1, \ldots k\}$, (ii) $y_i^* = z_i^* = \mathsf{val}(v_i)$ for all i such that $\mu_i = 0$, and (iii) $\langle \mathsf{op} \rangle(y_1^*, \ldots y_k^*)[q:p] \neq \langle \mathsf{op} \rangle(z_1^*, \ldots z_k^*)[q:p]$. Given condition (i), condition (iii) is equivalent to requiring that $(\langle \mathsf{op} \rangle(y_1^*, \ldots y_k^*)[q:p]) \sqcap_{q-p+1} (\langle \mathsf{op} \rangle(z_1^*, \ldots z_k^*)[q:p]) = \mathbf{X}$; it also implies that $\overline{Y^*}$ and $\overline{Z^*}$ are necessarily distinct. Since $\mu_i = 1$ iff $\mathsf{inv}(v_i) = \mathsf{true}$, and since $\mathsf{inv}(v_i) = \mathsf{true}$ iff v_i represents the value $\mathbf{X}$ (bottom of the lattice of values of an m_i-bit atom), it follows that given condition (i), condition (ii) is equivalent to asserting that $v_i \sqsubseteq_{m_i} y_i^*$ and $v_i \sqsubseteq_{m_i} z_i^*$ for every $i \in \{1, \ldots k\}$. It is now straightforward to see that Φ^μ evaluates to true iff $\sqcap_{q-p+1}\{\langle \mathsf{op} \rangle(u_1, u_2, \ldots u_k)[q:p] \mid u_i \in \{0,1\}^{m_i} \wedge v_i \sqsubseteq_{m_i} u_i\}$ is $\mathbf{X}$. Since the choice of μ, i.e. which $\mathsf{inv}(v_i)$ is false and which is true, was arbitrary, the formula in Lemma 3 evaluates to true iff the least pessimistic value of $rexpr[q:p]$ is $\mathbf{X}$. Thus, the formula gives the correct value of $\mathsf{inv}(rexpr[q:p])$. □

Unfortunately, eliminating the quantifiers in the formula for $\mathsf{inv}(\mathsf{op}(v_1, v_2, \ldots v_k)[q:p])$ may not always be easy. Reasoning about such quantified formulas in downstream analysis, viz. when finding satisfying assignments using an SMT solver, may also be computationally inefficient in general. Therefore, we approximate $\mathsf{inv}(\mathsf{op}(v_1, v_2, \ldots v_k)[q:p])$ in a sound manner for some RTL operators op. Informally, we allow $\mathsf{inv}(rexpr[q:p])$ to evaluate to true (denoting $rexpr[q:p] = \mathbf{X}$) even in cases where a careful calculation would have shown that $rexpr[q:p]$ is not $\mathbf{X}$. Formally, a value v in the lattice of values for $rexpr[q:p]$ is said to be a *sound approximation* for $rexpr[q:p]$ if v is lower than or equal to the least pessimistic value of $rexpr[q:p]$ in the lattice order $\sqsubseteq_{q-p+1}$. Let $iexpr$ be a formula in the theory of bit-vectors with equality, and $vexpr$ be the bit-vector expression given by Lemma 2. It is easy to see that $(vexpr, iexpr)$ soundly approximates $rexpr[q:p]$ using invalid-bit encoding *if and only if* the following condition, henceforth denoted $\mathsf{SoundCond}$, holds.

$\mathsf{SoundCond}$: *Whenever $iexpr$ evaluates to* false*, the least pessimistic value of $rexpr[q:p]$ is non-*$\mathbf{X}$.

Note that $\mathsf{SoundCond}$ is equivalent to saying that the formula for $\mathsf{inv}(rexpr[q:p])$, as given in Lemma 3, logically implies $iexpr$. A sound approximation of $rexpr[q:p]$ may set $iexpr$ to true even when the least pessimistic value of $rexpr[q:p]$ is non-$\mathbf{X}$. As a degenerate case, if $iexpr$ is identically set to true, we obtain a sound, but hopelessly conservative, approximation of $rexpr[q:p]$. Striking a fine balance between conservativeness and computational efficiency of the approximation is key to building a practically useful symbolic simulator using invalid-bit encoding. Lemma 2 gives an easy way to obtain $vexpr$. Our experience indicates that simple approximations of $\mathsf{inv}(rexpr[q:p])$ can also be carefully chosen when the formula in Lemma 3 is not amenable to easy simplification, to yield a sound approximation of $rexpr[q:p]$. We have derived templates for calculating $\mathsf{inv}(\mathsf{op}(v_1, \ldots v_k)$ either exactly or approximately for all word-level RTL operators op that appear in our benchmarks. We present below a discussion of how this is done for a subset of im-

portant RTL operators; the cases for the other operators is similar. Importantly, we use a recursive formulation for calculating/approximating $\mathsf{inv}(rexpr[q:p])$. This allows us to recursively compute invalid bits of atoms obtained by applying complex sequences of word-level operations to a base set of atoms.

For notational convenience, we say that a formula computes the invalid bit of an RTL operation *exactly* if it is semantically equivalent to the formula given in Lemma 3. If, on the other hand, a formula satisfies SoundCond, but is not equivalent to the formula in Lemma 3, we say that it computes a *sound approximation* of the invalid bit.

Bit-wise logical operations: Let $\neg_m$ and $\wedge_m$ denote bit-wise negation and conjunction operators respectively, for m-bit words. If a, b, c and d are m-bit words such that $c = \neg_m a$ and $d = a \wedge_m b$, the following formulas can be used for computing $\mathsf{inv}(c[q:p])$ and approximating $\mathsf{inv}(d[q:p])$ respectively.

$$\mathsf{inv}(c[q:p]) = \mathsf{inv}(a[q:p]) \tag{1}$$

$$\begin{aligned}\mathsf{inv}(d[q:p]) = {} & \big(\mathsf{inv}(a[q:p]) \vee \mathsf{inv}(b[q:p])\big) \wedge \big(\mathsf{inv}(a[q:p]) \vee (\mathsf{val}(a[q:p]) \neq \mathbf{0})\big) \\ & \wedge \big(\mathsf{inv}(b[q:p]) \vee (\mathsf{val}(b[q:p]) \neq \mathbf{0})\big)\end{aligned} \tag{2}$$

Proposition 1 *Equation (1) computes the invalid bit of* $(\neg_m a)[q:p]$ *exactly, and equation (2) gives a sound approximation of the invalid bit of* $(a \wedge_m b)[q:p]$.

Proof The correctness of equation (1) follows from the definition of bit-wise negation. The soundness of the approximation in equation (2) follows from the observation that if the Boolean expression on the right hand side of the equation evaluates to false, the result of the bit-wise and operation is necessarily non-**X**. Hence, SoundCond holds. □

Example 2 Consider the following 2-bit wide invalid-bit encoded atoms: $a_1 = (00, \mathsf{false})$, $a_0 = (11, \mathsf{true})$, $b_1 = (01, \mathsf{true})$ and $b_0 = (00, \mathsf{false})$. We use a_1a_0 to denote the 4-bit wide word obtained by concatenating a_1 and a_0, with $(a_1a_0)[3:2] = a_1$ and $a_1a_0[1:0] = a_0$. The interpretation of b_1b_0 is similar. Suppose $d = a_1a_0 \ \wedge_4 \ b_1b_0$. Using Lemma 2 and equation (2), the invalid-bit encoding of $d[3:2]$ is obtained as follows.

- $\mathsf{val}\,(d[3:2]) = (0011 \wedge_4 0100)[3:2] = 00$.
- $\mathsf{inv}\,(d[3:2]) = (\mathsf{false} \vee \mathsf{true}) \wedge (\mathsf{false} \vee (00 \neq 00)) \wedge (\mathsf{true} \vee (01 \neq 00)) = \mathsf{false}$

Note that we have used $\mathsf{inv}\,((a_1a_0)[3:2]) = \mathsf{inv}(a_1) = \mathsf{false}$ in the above calculation. Similarly, we have used $\mathsf{inv}\,((b_1b_0)[3:2]) = \mathsf{inv}(b_1) = \mathsf{true}$. To obtain the least pessimistic value of $d[3:2]$, we must compute the bit-wise conjunction of each element of $\{0000, 0001, 0010, 0011\}$ (i.e. values of a_1a_0) with each element of $\{0000, 0100, 1000, 1100\}$ (i.e. values of b_1b_0), and then find the greatest lower bound of the $(\cdot)\,[3:2]$ slices of the results of the conjunctions. It is easy to see that each of the above conjunctions yields 0000, and therefore all the $(\cdot)\,[3:2]$ slices are 00. It follows that their greatest lower bound is also 00. Observe that this matches the invalid-bit encoding of $d[3:2]$ obtained above, i.e. $(00, \mathsf{false})$.
Suppose we now wish to compute the invalid-bit encoding of $d[3:0]$. Following the same reasoning as above, the least pessimistic value of $d[3:0]$ is 0000. However, equation (2) gives the following:

- $\mathsf{val}\,(d[3:0]) = (0011 \wedge_4 0100)[3:0] = 0000$.
- $\mathsf{inv}\,(d[3:0]) = (\mathsf{true} \vee \mathsf{true}) \wedge (\mathsf{true} \vee (0011 \neq 0000)) \wedge (\mathsf{true} \vee (0100 \neq 0000)) = \mathsf{true}$

Note that we have used $\mathsf{inv}\,((a_1a_0)[3:0]) = \mathsf{true}$ since $\mathsf{inv}(a_0) = \mathsf{true}$. Similarly, we have used $\mathsf{inv}\,((b_1b_0)[3:2]) = \mathsf{true}$. Clearly, the invalid-bit encoding of $d[3:0]$, i.e. $(0000, \mathsf{true})$ is a sound approximation of the least pessimistic value.

The invalid bits of other bit-wise logical operators (like or, xor, nor, nand, etc.) can be obtained by first expressing them in terms of $\neg_m$ and $\wedge_m$ and then using the above approximations. Alternatively, they can also be computed directly such that SoundCond holds.

If-then-else statements: Consider an RTL conditional assignment statement "`if (BExpr) then x = Exp1; else x = Exp2;`". Symbolically simulating this statement gives $x = \mathsf{ite}(\mathsf{BExpr}, \mathsf{Exp1}, \mathsf{Exp2})$, where BExpr is a Boolean expression (or bit-vector expression of bit-width 1), and Exp1 and Exp2 are bit-vector expressions of the same bit-width as that of x. The formula for $\mathsf{inv}(x[q:p])$, as given by Lemma 3, can now be simplified as follows.

$$\begin{aligned}\mathsf{inv}(x[q:p]) &= \mathsf{ite}(\mathsf{inv}(\mathsf{BExpr}),\, temp_1,\, temp_2),\ \text{where} \\ temp_1 &= \mathsf{inv}(\mathsf{Exp1}[q:p]) \vee \mathsf{inv}(\mathsf{Exp2}[q:p]) \vee (\mathsf{val}(\mathsf{Exp1}[q:p]) \neq \mathsf{val}(\mathsf{Exp2}[q:p])) \\ temp_2 &= \mathsf{ite}(\mathsf{val}(\mathsf{BExpr}),\, \mathsf{inv}(\mathsf{Exp1}[q:p]), \mathsf{inv}(\mathsf{Exp2}[q:p]))\end{aligned} \tag{3}$$

Proposition 2 *Equation (3) computes the invalid bit of* $\mathsf{ite}(\mathsf{BExpr}, \mathsf{Exp1}, \mathsf{Exp2})[q:p]$ *exactly.*

Proof We consider two cases.

- If $\mathsf{inv}(\mathsf{BExpr}) = \mathsf{false}$, then $\mathsf{inv}(x[q:p])$, as given by Lemma 3, is semantically equivalent to $\mathsf{ite}(\mathsf{val}(\mathsf{BExpr}),\ \mathsf{inv}(\mathsf{Exp1}[q:p]),\ \mathsf{inv}(\mathsf{Exp2}[q:p]))$.
- If $\mathsf{inv}(\mathsf{BExpr}) = \mathsf{true}$, then the value of BExpr can be either true or false. The following four sub-cases arise.
  - If both $\mathsf{inv}(\mathsf{Exp1}[q:p]$ and $\mathsf{inv}(\mathsf{Exp2}[q:p])$ are false and if $\mathsf{val}(\mathsf{Exp1}[q:p]) = \mathsf{val}(\mathsf{Exp2}[q:p])$, i.e. $\mathsf{Exp1}[q:p]$ and $\mathsf{Exp2}[q:p]$ have identical non-$\mathbf{X}$ values, then regardless of the value of BExpr, we have $x = \mathsf{Exp1}$ (or equivalently, Exp2). Hence the formula in Lemma 3 evaluates to false, and the least pessimistic value of $x[q:p]$) is non-$\mathbf{X}$. In this case, equation (3) correctly computes $\mathsf{inv}(x[q:p])$ as false.
  - If $\mathsf{inv}(\mathsf{Exp1})[q:p]$ is true, then by letting BExpr take the value true, it is easy to see that the formula in Lemma 3 evaluates to true, and the least pessimistic value of $x[q:p]$ is the same as that of $\mathsf{Exp1}[q:p]$, i.e. $\mathbf{X}$. Note that equation (3) correctly computes $\mathsf{inv}(x[q:p])$ as true in this case.
  - If $\mathsf{inv}(\mathsf{Exp2})[q:p]$ is true, then by letting BExpr take the value false, the same argument as above shows that equation (3) correctly computes $\mathsf{inv}(x[q:p])$ as true.
  - Suppose $\mathsf{inv}(\mathsf{Exp1})[q:p] = \mathsf{inv}(\mathsf{Exp2})[q:p] = \mathsf{false}$, but $\mathsf{val}(\mathsf{Exp1}[q:p]) \neq \mathsf{val}(\mathsf{Exp2}[q:p])$. The formula in Lemma 3 simplifies in this case to

$$\exists y \exists z\, (\mathsf{ite}(y, \mathsf{val}(\mathsf{Exp1}[q:p]), \mathsf{val}(\mathsf{Exp2}[q:p])) \neq \mathsf{ite}(z, \mathsf{val}(\mathsf{Exp1}[q:p]), \mathsf{val}(\mathsf{Exp2}[q:p]))\,.$$

It is easy to see that this simplifies further to true. In other other words, the least pessimistic value of $x[q:p]$ is $\mathbf{X}$. Note that equation (3) correctly computes $\mathsf{inv}(x[q:p])$ as true in this case. □

Example 3 Consider $a = (001, \mathsf{false})$, $b = (000, \mathsf{false})$ and $c = (0, \mathsf{true})$, where a and b are 3-bits wide, and c is 1-bit wide. Using Lemma 2 and equation (3), the invalid-bit encoding of $d = \mathsf{ite}(c, a, b)[2:1]$ is computed as follows.

- $\mathsf{val}(d) = \mathsf{ite}(0, 001, 000)[2:1] = 00$.
- $\mathsf{inv}(d) = \mathsf{ite}(\mathsf{true}, temp_1, temp_2)$, where $temp_1 = \mathsf{inv}(a[2:1]) \vee \mathsf{inv}(b[2:1]) \vee (00 \neq 00) = \mathsf{false}$, and $temp_2 = \mathsf{ite}(0, \mathsf{false}, \mathsf{false}) = \mathsf{false}$. Therefore, $\mathsf{inv}(d) = \mathsf{false}$.

Note that the least pessimistic value of d is

$$\sqcap_2(\mathsf{ite}(0, 001, 000)[2:1],\ \mathsf{ite}(1, 001, 000)[2:1]) \ = \ \sqcap_2\,(00, 00) \ = \ 00.$$

This matches the invalid-bit encoded value of d computed above, i.e. $(00, \mathsf{false})$.

Word-level addition: Let $+_m$ denote an m-bit addition operator. Thus, if a and b are m-bit operands, $a +_m b$ generates an m-bit sum and a 1-bit $carry$. Let the carry generated after adding the least significant r bits of the operands be denoted $carry_r$. We discuss below how to compute sound approximations of $\mathsf{inv}(sum[q:p])$ and $\mathsf{inv}(carry_r)$, where $0 \leq p \leq q \leq m-1$ and $1 \leq r \leq m$.

It is easy to see that the value of $sum[q:p]$ is completely determined by $a[q:p]$, $b[q:p]$ and $carry_p$. Therefore, we can compute $\mathsf{inv}(sum[q:p])$ as follows.

$$\mathsf{inv}(sum[q:p]) = \mathsf{inv}(a[q:p]) \vee \mathsf{inv}(b[q:p]) \vee \mathsf{inv}(carry_p) \tag{4}$$

Proposition 3 *Equation (4) computes the invalid bit of $sum[q:p]$ exactly*

Proof Since the i^{th} bit of sum is obtained by XOR-ing the i^{th} bit of a, the i^{th} bit of b and the carry out from the $(i-1)^{st}$ bit position, if any, it follows that if any of $a[q:p]$, $b[q:p]$ or $carry_p$ is $\mathbf{X}$, $sum[q:p]$ can have at least two distinct values, and hence its least pessimistic value is $\mathbf{X}$. On the other hand, if all of $\mathsf{inv}(a[q:p])$, $\mathsf{inv}(b[q:p])$ and $\mathsf{inv}(carry_p)$ are false, then $a[q:p]$, $b[q:p]$ and $carry_p$ have non-$\mathbf{X}$ values. Hence, there is no uncertainty in the value of $sum[q:p]$, i.e. $sum[q:p]$ is non-$\mathbf{X}$. Therefore, the formula for $\mathsf{inv}(sum[q:p])$ in equation (4) is equivalent to that in Lemma 3. □

The computation of $\mathsf{inv}(carry_p)$ (or $\mathsf{inv}(carry_r)$ in general) is more interesting, and deserves special attention. We identify three cases below, and argue that $\mathsf{inv}(carry_p)$ is false in each of these cases. In the following, $\mathbf{0}$ denotes the p-bit constant $00\cdots0$.

1. If $\big(\mathsf{inv}(a[p-1:0]) \vee \mathsf{inv}(b[p-1:0])\big) = \mathsf{false}$, then both $\mathsf{inv}(a[p-1:0])$ and $\mathsf{inv}(b[p-1:0])$ must be false. Therefore, there is no uncertainty in the values of either $a[p-1:0]$ or $b[p-1:0]$, and $\mathsf{inv}(carry_p) = \mathsf{false}$.
2. If $\big(\neg\mathsf{inv}(a[p-1:0]) \wedge (\mathsf{val}(a[p-1:0]) = \mathbf{0})\big)$, then the least significant p bits of $\mathsf{val}(a)$ are all 0. Regardless of $\mathsf{val}(b)$, it is easy to see that in this case, $\mathsf{val}(carry_p) = 0$ and $\mathsf{inv}(carry_p) = \mathsf{false}$.
3. This is the symmetric counterpart of the case above, i.e., $\big(\neg\mathsf{inv}(b[p-1:0]) \wedge (\mathsf{val}(b[p-1:0]) = \mathbf{0})\big)$.

We now approximate $\mathsf{inv}(carry_p)$ by combining the conditions corresponding to the three cases above. In other words,

$$\begin{aligned}\mathsf{inv}(carry_p) = &\big(\mathsf{inv}(a[p-1:0]) \vee \mathsf{inv}(b[p-1:0])\big) \wedge \\ &\big(\mathsf{inv}(a[p-1:0]) \vee (\mathsf{val}(a[p-1:0]) \neq \mathbf{0})\big) \wedge \\ &\big(\mathsf{inv}(b[p-1:0]) \vee (\mathsf{val}(b[p-1:0]) \neq \mathbf{0})\big) \end{aligned} \tag{5}$$

Proposition 4 *Equation (5) gives a sound approximation of* $\mathsf{inv}(carry_p)$.

Proof It follows from the arguments given above that if the right-hand side of equation (5) evaluates to false, then the value of $carry_p$ is necessarily non-$\mathbf{X}$. Hence, $\mathsf{SoundCond}$ holds. □

Example 4 Consider the following 2-bit wide invalid-bit encoded atoms: $a_0 = (11, \mathsf{true})$, $a_1 = (00, \mathsf{false})$, $b_0 = (00, \mathsf{false})$ and $b_1 = (11, \mathsf{false})$, with the usual interpretations of a_1a_0 and b_1b_0. Let $sum = a_1a_0 +_4 b_1b_0$. Using Lemma 2 and equations (4) and (5), the invalid-bit encoding of $sum[3:2]$ is approximated as follows.

- $\mathsf{val}(sum[3:2]) = (0011 +_4 1100)[3:2] = (1111)[3:2] = 11$.
- $\mathsf{inv}(sum[3:2]) = \mathsf{inv}(a_1) \vee \mathsf{inv}(b_1) \vee \mathsf{inv}(carry_2) = \mathsf{inv}(carry_2)$. To obtain a sound approximation of $\mathsf{inv}(carry_2)$, we use equation (5). Therefore, $\mathsf{inv}(sum[3:2]) = (\mathsf{inv}(a_0) \vee \mathsf{inv}(b_0)) \wedge (\mathsf{inv}(a_0) \vee (\mathsf{val}(a_0) \neq 0)) \wedge (\mathsf{inv}(b_0) \vee (\mathsf{val}(b_0) \neq 0))$. Since $b_0 = (00, \mathsf{false})$, this gives $\mathsf{inv}(sum[3:2]) = \mathsf{false}$.

Note that the least pessimistic value of $(a_1a_0 +_4 b_1b_0)[3:2]$ is

$$\sqcap_2 \begin{pmatrix} (0000 +_4 1100)[3:2], (0001 +_4 1100)[3:2], \\ (0010 +_4 1100)[3:2], (0011 +_4 1100)[3:2] \end{pmatrix} = 11.$$

This is exactly what $(\mathsf{val}(sum[3:2]), \mathsf{inv}(sum[3:2])) = (11, \mathsf{false})$ represents using invalid-bit encoding.

Suppose we now wish to calculate $\mathsf{inv}(carry_3)$. Using equation (5), we obtain this as $(\mathsf{inv}(a[3:0]) \vee \mathsf{inv}(b[3:0])) \wedge (\mathsf{inv}(a[3:0]) \vee (\mathsf{val}(a[3:0]) \neq 0)) \wedge (\mathsf{inv}(b[3:0]) \vee (\mathsf{val}(b[3:0]) \neq 0))$. Since $\mathsf{inv}(a_0) = \mathsf{true}$, we use $\mathsf{inv}(a[3:0]) = \mathsf{true}$. Similarly, since $\mathsf{val}(b_1) = 11$, we have $\mathsf{val}(b[3:0]) \neq 0$. Therefore, $\mathsf{inv}(carry_3) = \mathsf{true}$ by equation (5). However, from the least pessimistic value analysis done above, we know that the least pessimistic value of $carry_3$ is 0. Thus, equation (5) gives a sound approximation of $\mathsf{inv}(carry_3)$ in this case.

Word-level division: Let $\div_m$ denote an m-bit division operator; this is among the most complex word-level RTL operators for which we have derived an approximation of the invalid bit expression. If a and b are m-bit operands, $a \div_m b$ generates an m-bit quotient, say $quot$, and an m-bit remainder, say rem. Furthermore, a is called the dividend and b is called the divisor in $a \div_m b$. We wish to compute $\mathsf{inv}(quot[q:p])$ and $\mathsf{inv}(rem[q:p])$, where $0 \leq p \leq q \leq m-1$. For notational convenience, we represent constant bit vectors by their unsigned integer interpretations. Since division by zero is undefined, we assume that the value $\mathbf{0}$ is excluded from the lattice of values of b for purposes of the following discussion. The case of $a \div_m b$ with $(\mathsf{val}(b), \mathsf{inv}(b)) = (0, \mathsf{false})$ leads to a "divide-by-zero" exception, and is assumed to be handled separately.

The following expressions give approximations for $\mathsf{inv}(quot[q:p])$ and $\mathsf{inv}(rem[q:p])$. In these expressions, we assume that i is a non-negative integer that does not appear in the RTL and is such that $2^i \leq \mathsf{val}(b) < 2^{i+1}$. Note that this assumption precludes the possibility of b taking the value $\mathbf{0}$ from its lattice of values.

$$\begin{aligned}
\mathsf{inv}\left(quot[q:p]\right) &= temp_1 \wedge \left(\mathsf{inv}(b) \vee temp_2\right), \text{ where} \qquad (6)\\
temp_1 &= \mathsf{inv}\left(a[m-1:p]\right) \vee \left(\mathsf{val}\left(a[m-1:p]\right) \neq \mathbf{0}\right) \text{ and}\\
temp_2 &= \mathsf{ite}\left(\mathsf{val}(b) = \mathbf{2^i}, temp_3, (i<p) \vee \mathsf{inv}\left(a[m-1:p]\right)\right), \text{ where}\\
temp_3 &= (p+i \leq m-1) \wedge \mathsf{inv}\left(a[\min(q+i, m-1) : p+i]\right)\\
\mathsf{inv}\left(rem[q:p]\right) &= temp_1 \wedge \left(\mathsf{inv}(b) \vee \ temp_4\right), \text{ where} \qquad (7)\\
temp_4 &= \mathsf{ite}\left(\mathsf{val}(b) = \mathbf{2^i}, (i>p) \wedge \mathsf{inv}\left(a[\min(q, i-1) : p]\right), i \geq p\right)
\end{aligned}$$

Although equations (6) and (7) are formulated assuming $2^i \leq \mathsf{val}(b) < 2^{i+1}$, the assumption itself does not appear in the equations, and must be taken into account separately. We will see later that a word-level STE problem is solved by generating a formula in the theory of bit-vectors with equality, such that every satisfying assignment of the formula gives a counterexample to the verification problem. We incorporate assumptions like $2^i \leq \mathsf{val}(b) < 2^{i+1}$ in the final bit-vector formula representing the verification condition for an STE problem. This ensures that every assignment of i and $\mathsf{val}(b)$ in a counterexample satisfies the assumptions made with respect to them.

Proposition 5 *Equations (6) and (7) give sound approximations of* $quot[q:p]$ *and* $rem[q:p]$*, respectively.*

Proof We first claim that if $temp_1$ in equations (6) and (7) evaluates to false, then both $\mathsf{inv}(quot[q:p])$ and $\mathsf{inv}(rem[q:p])$ are false, regardless of the value of b (assuming b does not take the value $\mathbf{0}$). To see why this is true, note that if $\mathsf{val}(a[m-1:p]) = \mathbf{0}$ and $\mathsf{inv}(a[m-1:p]) = \mathsf{false}$, then $\mathsf{val}(a) < 2^p$. Since dividing an unsigned integer by another unsigned integer can neither yield a quotient nor a remainder that exceeds the dividend, it follows that both $quot$ and rem are $< 2^p$. Therefore, $quot[q:p] = rem[q:p] = \mathbf{0}$, and both $\mathsf{inv}(quot[q:p])$ and $inv(rem[q:p])$ are false.

If $temp_1$ evaluates to true, then for the right-hand side of equations (6) and (7) to evaluate to false, we must have $\mathsf{inv}(b) = \mathsf{false}$. Hence, we focus only on the case when $b \neq \mathbf{X}$. We consider two sub-cases below.

- $\mathsf{val}(b) = \mathbf{2^i}$: In this case, $a \div_m b$ effectively shifts a right by i bit positions, and the least significant i bits of a forms the remainder. Therefore, $\mathsf{val}(quot[q:p])$ is $a[q+i:p+i]$ if $q+i \leq m-1$; it is $a[m-1:p+i]$ padded to the left with $q+i-(m-1)$ 0s if $q+i > m-1 \geq p+i$, and it is $\mathbf{0}$ if $p+i > m-1$. It follows that if $p+i > m-1$, then $\mathsf{val}(quot[q:p]) = \mathbf{0}$ and $\mathsf{inv}(quot[q:p]) = \mathsf{false}$. Otherwise, $\mathsf{inv}(quot[q:p]) = \mathsf{inv}(a[k:p+i])$, where $k = \min(q+i, m-1)$.
  It is also easy to see that $\mathsf{val}(rem[q:p])$ is $a[q:p]$ if $i > q$; it is $a[i-1:p]$ padded with $q-(i-1)$ 0s to the left if $q \geq i > p$, and it is 0 if $i \leq p$. Therefore, if $i \leq p$, we have $\mathsf{inv}(rem[q:p]) = \mathsf{false}$; otherwise, $\mathsf{inv}(rem[q:p]) = \mathsf{inv}(a[k:p])$, where $k = \min(q, i-1)$.

– $\mathbf{2^i < \mathsf{val}(b) < 2^{i+1}}$: In this case, we claim the following propositions are true.
P1: If $i \geq p$, then $\mathsf{inv}(quot[q:p])$ can be approximated by $\mathsf{inv}(a[m-1:p])$.
P2: If $i < p$, then $\mathsf{inv}(rem[q:p]) = \mathsf{false}$
To see why these claims are true, note that $\mathsf{val}(a)$ can be written as $a_1 \cdot 2^p + a_2$, where a_1 and a_2 are the integer representations of $a[m-1:p]$ and $a[p-1:0]$, respectively. Clearly, $0 \leq a_2 < 2^p$. Considering quotients and remainders on division by $\mathsf{val}(b)$, suppose $a_1 = k_1 \cdot \mathsf{val}(b) + r_1$ and $a_2 = k_2 \cdot \mathsf{val}(b) + r_2$, where $0 \leq r_1, r_2 < \mathsf{val}(b)$ and $k_1, k_2 \geq 0$. Suppose further that $2^p \cdot r_1 + r_2 = k_3 \cdot \mathsf{val}(b) + r_3$, where $0 \leq r_3 < \mathsf{val}(b)$ and $k_3 \geq 0$. It is an easy exercise to see that the quotient obtained on dividing $\mathsf{val}(a)$ by $\mathsf{val}(b)$ is $2^p \cdot k_1 + k_2 + k_3$, and the remainder is r_3. Thus, $\mathsf{val}(quot) = 2^p \cdot k_1 + k_2 + k_3$ and $\mathsf{val}(rem) = r_3$. We discuss what happens when $i \geq p$ and $i < p$.
  – If $i \geq p$, then $\mathsf{val}(b) > 2^i \geq 2^p > a_2$. Since $\mathsf{val}(b) > a_2$, we have $k_2 = 0$ and $r_2 = a_2 < 2^p$. It follows that $quot = 2^p \cdot k_1 + k_3$. If $k_3 < 2^p$, then $quot[q:p]$ depends only on k_1, which in turn, depends only on $a[m-1:p]$ and $\mathsf{val}(b)$. Therefore, if $\mathsf{inv}(a[m-1:p])$ is false (i.e. $a[m-1:p]$ is non-$\mathbf{X}$), there is no uncertainty in the value of $quot[q:p]$. Hence $\mathsf{inv}(quot[q:p])$ is false if $(i \geq p) \wedge \neg\mathsf{inv}(a[m-1:p])$ holds.
  We now show that k_3 is indeed strictly less than 2^p. Since $2^p \cdot r_1 + r_2 = k_3 \cdot \mathsf{val}(b) + r_3$, after rearranging terms, we get $k_3 \cdot \mathsf{val}(b) - 2^p \cdot r_1 = r_2 - r_3$. If possible, let $k_3 = 2^p + d$, where $d \geq 0$. Substituting for k_3, we get $2^p \cdot (\mathsf{val}(b) - r_1) + d \cdot \mathsf{val}(b) = r_2 - r_3$. Since $\mathsf{val}(b) > r_1$, the left hand side of the above equation is at least as large as 2^p, while the right hand side is at most r_2, which, in turn, is less than 2^p. This gives a contradiction, and therefore, k_3 must be strictly less than 2^p.
  – If $i < p$, we have $rem = r_3 < \mathsf{val}(b) < 2^{i+1} \leq 2^p$. Therefore, $\mathsf{val}(rem[q:p]) = 0$, and $\mathsf{inv}(rem[q:p])$ is false in this case.

The above arguments show that when the right-hand side of equation (6) is false, then $quot[q:p]$ is indeed non-$\mathbf{X}$. Similarly, when the right-hand side of equation (7) is false, $rem[q:p]$ is non-$\mathbf{X}$. Hence, $\mathsf{SoundCond}$ is satisfied in both cases. □

Example 5 Consider $a_0 = (11, \mathsf{true})$, $a_1 = (00, \mathsf{false})$ and $b = (0101, \mathsf{true})$. Let a_1a_0 denote the 4-bit word formed by concatenating a_1 and a_0, as in Example 4. Clearly, if we use a cautious lattice, the value of a_1a_0 would be $00XX$ and that of b would be $XXXX$ (with the restriction that b cannot take the value 0000). Therefore, a_1a_0 represents an unsigned integer in the interval $[0, 3]$, and b represents an unsigned integer in the interval $[1, 15]$. Let $quot$ and rem denote the quotient and remainder obtained on dividing a_1a_0 by b, i.e. $a_1a_0 \div_4 b$. We assume that b, being the divisor, cannot take the value 0000 in its lattice of values. The invalid-bit encodings of $quot[3:2]$ and $rem[3:2]$ can be obtained using Lemma 2 and equations (6) and (7) as follows. Note that $p = 2$, $q = 3$ and $m = 4$ in this example. In addition, since $\mathsf{val}(b) = 5$, we have $i = 2$.

– $\mathsf{val}(quot[3:2]) = (\text{quotient of } (0011 \div_4 0101))[3:2] = (0000)[3:2] = 00$.
– $\mathsf{inv}(quot[3:2]) = temp_1 \wedge (\mathsf{true} \vee temp_2)$, where $temp_1 = \mathsf{inv}(a_1) \vee (\mathsf{val}(a_1) \neq 0) = \mathsf{false}$. Therefore, $\mathsf{inv}(quot[3:2] = \mathsf{false}$.
– $\mathsf{val}(rem[3:2]) = (\text{remainder of } (0011 \div_4 0101))[3:2] = (0011)[3:2] = 00$.
– $\mathsf{inv}(rem[3:2]) = temp_1 \wedge (\mathsf{true} \vee temp_4) = \mathsf{false}$, since $temp_1 = \mathsf{false}$.

Thus, $quot[3:2] = (00, \mathsf{false})$ and $rem[3:2] = (00, \mathsf{false})$. The least pessimistic value of $quot[3:2]$ is obtained by computing the greatest lower bound of the $(\cdot)[3:2]$

slices of quotients obtained on dividing elements in the set $\{0000, 0001, 0010, 0011\}$ by elements in the set $\{0001, 0010, \ldots 1111\}$. It is easy to see that each of the slices thus obtained is 00; hence their greatest lower bound is 00. Similarly, the least pessimistic value of $rem[3:2]$ can also be seen to be 00. These values coincide with those obtained above, i.e. $(00, \mathsf{false})$ for both $quot[3:2]$ and $rem[3:2]$, using invalid-bit encoding.

In the above example, both $quot[3:2]$ and $rem[3:2]$ ended up having no uncertainty in their values although both the divisor and quotient had uncertain values. This was because $temp_1$ evaluated to false. In the next example, we show that the same thing can happen even when $temp_1$ evaluates to true.

Example 6 Consider $a_0 = (11, \mathsf{true})$, $a_1 = (01, \mathsf{false})$ and $b = (0100, \mathsf{false})$. We wish to compute $quot[3:2]$ and $rem[3:2]$, obtained on dividing a_1a_0 by b, as in the previous example. Note that $p = 2$, $q = 3$, $m = 4$ and $i = 2$ in this example.

Proceeding as in Example 5, we get:

- $\mathsf{val}(quot[3:2]) = (\text{quotient of } (0111 \div_4 0100))[3:2] = (0001)[3:2] = 00$.
- $\mathsf{inv}(quot[3:2]) = temp_1 \wedge (\mathsf{false} \vee temp_2)$, where $temp_1 = \mathsf{inv}(a_1) \vee (\mathsf{val}(a_1) \neq 0) = \mathsf{true}$, and $temp_2 = \mathsf{ite}\big((0100 = 2^2), temp_3, \mathsf{false} \vee \mathsf{inv}(a_1)\big) = temp_3$. In turn, $temp_3 = (2 + 2 \leq 4 - 1) \wedge \mathsf{inv}(a_1[1:1]) = \mathsf{false}$. Therefore, $\mathsf{inv}(quot[3:2] = \mathsf{false}$.
- $\mathsf{val}(rem[3:2]) = (\text{remainder of } (0111 \div_4 0100))[3:2] = (0011)[3:2] = 00$.
- $\mathsf{inv}(rem[3:2]) = temp_1 \wedge (\mathsf{false} \vee temp_4) = temp_4$, since $temp_1 = \mathsf{true}$. Since $temp_4 = \mathsf{ite}\big((0100 = 2^2), (2 > 2) \wedge \mathsf{inv}(a_0[1:1])\big) = \mathsf{false}$, we have $\mathsf{inv}(rem[3:2]) = \mathsf{false}$.

Therefore, $quot = (00, \mathsf{false})$ and $rem = (00, \mathsf{false})$. It can be verified that this matches exactly the least pessimistic values obtained using a cautious lattice.

The next example shows that equations (6) and (7) can give true for $\mathsf{inv}(quot[q:p])$ and $\mathsf{inv}(rem[q:p])$, even when the corresponding least pessimistic values are non-**X**.

Example 7 Consider $a_0 = (00, \mathsf{false})$, $a_1 = (01, \mathsf{true})$, $b_0 = (00, \mathsf{true})$ and $b_1 = (11, \mathsf{false})$. Let $a = a_1a_0$ and $b = b_1b_0$, as before. Clearly, a represents an unsigned integer in $\{0, 4, 8, 12\}$ and b represents an unsigned integer in $\{12, 13, 14, 15\}$. Suppose we wish to compute $a \div_4 b$, and are interested in $\mathsf{inv}(quot[1:1])$ and $\mathsf{inv}(rem[1:1])$. It is easy to see that the set of possible values of the quotient is $\{0, 1\}$ and that for the remainder is $\{0, 4, 8, 12\}$. Therefore, the least pessimistic values for $quot[1:1]$ and $rem[1:1]$ are both 0, i.e. non-**X**.

Let us now compute $\mathsf{inv}(quot[1:1])$ and $\mathsf{inv}(rem[1:1])$ from equations (6) and (7). Since $\mathsf{inv}(b_0) = \mathsf{true}$, we have $\mathsf{inv}(b) = \mathsf{true}$. Similarly, since $\mathsf{inv}(a_1) = \mathsf{true}$, we have $\mathsf{inv}(a[3:1]) = \mathsf{true}$. Furthermore, $p = 1$, $q = 1$, $m = 4$ and $i = 3$ in this case. Using equations (6) and (7), it is now easy to see that $\mathsf{inv}(quot[1:1]) = \mathsf{inv}(rem[1:1] = \mathsf{true}$.

Memory/array reads and updates: Let A be a 1-dimensional array, i be an index expression, and `x` be a variable and `Exp` be an expression of the base type of A. On symbolically simulating the RTL statement "`x = A[i];`", we update the value of x to $\mathsf{read}(\mathsf{A}, \mathsf{i})$, where the read operator is as in the extensional theory of arrays (see [26] for details). Similarly, on symbolically simulating the RTL statement

"`A[i] = Exp`", we update the value of array A to $\mathsf{update}(\mathsf{A}_{\text{orig}}, \mathsf{i}, \mathsf{Exp})$, where A_{orig} is the (array-typed) expression for A prior to simulating the statement, and the update operator is as in the extensional theory of arrays.

Since the expression for a variable or array obtained by symbolic simulation may now have read and update operators, we must find ways to compute sound approximations of the invalid bit for expressions of the form $\mathsf{inv}(\mathsf{read}(\mathsf{A}, \mathsf{i})[q : p])$. Note that since A is an array, the symbolic expression for A is either (i) A_{init}, i.e. the initial value of A at the start of symbolic simulation, or (ii) $\mathsf{update}(\mathsf{A}', \mathsf{i}', \mathsf{Exp}')$ for some expressions A', i' and Exp', where A' has the same array-type as A, i' has an index type, and Exp' has the base type of A. For simplicity of exposition, we assume that all arrays are either completely initialized or completely uninitialized at the start of symbolic simulation. The invalid bit in case (i) is then easily seen to be true if A_{init} denotes an uninitialized array, and false otherwise. In case (ii), let v denote $\mathsf{read}(\mathsf{A}, \mathsf{i})$. The invalid bit of $v[q : p]$ can then be approximated as:

$$\mathsf{inv}(v[q:p]) = \mathsf{inv}(\mathsf{i}) \vee \mathsf{inv}(\mathsf{i}') \vee \mathsf{ite}\left(\mathsf{val}(\mathsf{i}) = \mathsf{val}(\mathsf{i}'), \mathsf{inv}(\mathsf{Exp}'[q:p]), temp\right), \text{where} \quad (8)$$
$$temp = \mathsf{inv}(\mathsf{read}(\mathsf{A}', \mathsf{i})[q:p]).$$

Proposition 6 *Equation (8) gives a sound approximation of* $\mathsf{inv}(\mathsf{read}(\mathsf{A}, \mathsf{i})[q : p])$.

Proof If neither i nor i' is $\mathbf{X}$, i.e. the corresponding invalid bits are false, there are two cases to consider.

- If $\mathsf{val}(\mathsf{i}) = \mathsf{val}(\mathsf{i}')$, then $\mathsf{read}(\mathsf{update}(\mathsf{A}', \mathsf{i}', \mathsf{Exp}'), \mathsf{i}) = \mathsf{Exp}'$. Hence, the required invalid bit is $\mathsf{inv}(\mathsf{Exp}'[q : p])$.
- If $\mathsf{val}(\mathsf{i}) \neq \mathsf{val}(\mathsf{i}')$, then $\mathsf{read}(\mathsf{update}(\mathsf{A}', \mathsf{i}', \mathsf{Exp}'), \mathsf{i}) = \mathsf{read}(\mathsf{A}', \mathsf{i})$. Hence, the required invalid bit is $\mathsf{inv}(\mathsf{read}(\mathsf{A}', \mathsf{i})[q : p])$.

Hence, $\mathsf{SoundCond}$ is satisfied in this case. □

Example 8 Let A be a 1-dimensional array of size 4, where each element of A is a 4-bit word. Let $b_0 = (00, \mathsf{false})$ and $b_1 = (11, \mathsf{true})$ be two 2-bit atoms, and let $b = b_1 b_0$ be the 4-bit atom obtained by concatenating b_1 and b_0 in the usual way. Suppose A has been updated twice, and its symbolic expression is $\mathsf{update}(\mathsf{update}(A_{\text{init}}, 00, b), 01, 0000)$. Thus, the least pessimistic value of some array elements (e.g. $A[00]$) are $\mathbf{X}$, while that for some other array elements (e.g. $A[01]$) are non-$\mathbf{X}$. Let $i = (00, \mathsf{false})$ be a 2-bit atom representing an array index. Clearly, the least pessimistic value for $\mathsf{read}(A, i)[1 : 0]$ is 00. Using equation (8), we obtain $\mathsf{inv}(\mathsf{read}(A, i)[1 : 0]) = \mathsf{inv}(\mathsf{read}(\mathsf{update}(A_{\text{init}}, 00, b), i)[1 : 0]) = \mathsf{inv}(b_0) = \mathsf{false}$. Similarly, $\mathsf{val}(\mathsf{read}(A, i)[1 : 0])$ can be seen to be 00 from Lemma 2. Thus, the invalid-bit encoded value of $\mathsf{read}(A, i)[1 : 0]$ is $(00, \mathsf{false})$. This matches exactly the least pessimistic value of $\mathsf{read}(A, i)[1 : 0]$, i.e. 00.

We now provide an example that illustrates why equation (8) may yield true, even when the least pessimistic value of $\mathsf{read}(\mathsf{A}, \mathsf{i})[q : p]$ is non-$\mathbf{X}$.

Example 9 Let A be a 1-dimensional array of size 4, where each element of A is a 4-bit word. Suppose A has been updated twice, and its symbolic expression is $\mathsf{update}(\mathsf{update}(A_{\text{init}}, 00, 0000), 01, 0000)$. Thus, the array elements at indices 00 and 01 have the same value, i.e. 0000. Let $i_0 = (1, \mathsf{true})$ and $i_1 = (0, \mathsf{false})$, and let $i = i_1 i_0$ be a 2-bit atom representing an array index. Clearly, the least pessimistic value for $\mathsf{read}(A, i)[3 : 0]$ is 0000. However, since $\mathsf{inv}(i_0) = \mathsf{true}$, we have $\mathsf{inv}(i) = \mathsf{true}$. Therefore, equation (8) gives $\mathsf{inv}(\mathsf{read}(\mathsf{A}, \mathsf{i})[3 : 0]) = \mathsf{true}$.

If the RTL design has multi-dimensional arrays, we simply treat them as arrays of arrays, and apply the same reasoning as above. For example, if B is a two-dimensional array, the RTL statement "`B[i][j] = Exp;`" updates the symbolic value of array B to $\mathsf{update}(\mathsf{B}_{\mathrm{orig}}, \mathsf{i}, \mathsf{update}(\mathsf{read}(\mathsf{B}_{\mathrm{orig}}, \mathsf{i}), \mathsf{j}, \mathsf{Exp}))$, where $\mathsf{B}_{\mathrm{orig}}$ is the symbolic expression for B prior to simulating the RTL statement. Similarly, the RTL statement "`x = B[i][j];`"updates the symbolic value of x to $\mathsf{read}(\mathsf{read}(\mathsf{B}, \mathsf{i}), \mathsf{j})$.

Shift operations: We discuss below the left-shift operation; the case of the right-shift operation can be analyzed similarly. A shift operation can specify either a constant number of bit positions to shift, or a variable number of positions to shift. We analyze these two cases separately since shifting by a variable number of positions does not allow us to statically identify the operand's bit-slices of interest. In either case, we assume that a left shift operation pads 0s in the least significant shifted positions. Let $\ll_k$ denote a unary left-shift operator of the first kind, where k is a positive integer constant, and let $\ll$ denote a binary left-shift operator of the second kind. Let a, b, c, d be m-bit words such that $b = \ll_k a$ and $c = a \ll d$. For simplicity of presentation, we assume no wrap-around in shifting; the case of wrap-around can be analyzed in a similar way. The following equations give $\mathsf{inv}(b[q:p])$ and a sound approximation of $\mathsf{inv}(c[q:p])$, where $0 \le p \le q \le m-1$.

$$\mathsf{inv}(b[q:p]) = \mathsf{ite}(p \ge k, \mathsf{inv}(a[q-k:p-k]), temp), \text{ where} \tag{9}$$
$$temp = \mathsf{ite}(q \ge k,\ \mathsf{inv}(a[q-k:0]),\ \mathsf{false})$$
$$\mathsf{inv}(c[q:p]) = temp_1 \ \wedge\ temp_2, \text{where} \tag{10}$$
$$temp_1 = \mathsf{inv}(d) \vee (\mathsf{inv}(a[q:0]) \wedge (\mathsf{val}(d) \le q)),$$
$$temp_2 = \mathsf{inv}(a[q:0]) \vee (\mathsf{val}(a[q:0]) \neq \mathbf{0})$$

Proposition 7 *Equation (9) computes* $\mathsf{inv}((\ll_k a)[q:p])$ *exactly, and equation (10) gives a sound approximation of* $\mathsf{inv}((a \ll d)[q:p])$.

Proof If $p \ge k$, then clearly $(\ll_k a)[q:p] = a[q-k:p-k]$. If $p < k$, there are two cases to consider.

- If $q \ge k$, then $(\ll_k a)[q:p]$ is simply $a[q-k:0]$ padded with $k-p$ 0s to the right. Hence, $\mathsf{inv}((\ll_k a)[q:p]) = \mathsf{false}$.
- If $k > q$, then $(\ll_k a)[q:p]$ consists of only the 0s padded at the right. Hence, $(\ll_k a)[q:p] = 0$.

It follows that the formula in equation (9) is semantically equivalent to that in Lemma 3.

Suppose $\mathsf{inv}(a[q:0])$ is false, i.e. $a[q:0]$ is non-**X**. If $\mathsf{inv}(d)$ is also false, i.e. d has a unique value, then then clearly $(a \ll d)[q:p]$ has no uncertain bits, regardless of the value of d. Similarly, if $\mathsf{val}(a[q:0])$ has all 0's, then once again $(a \ll d)[q:p]$ is non-**X**, regardless of the value of d. Hence, in these cases, $\mathsf{inv}\,((a \ll d)[q:p])$ is false. Finally, if $\mathsf{inv}(d)$ is false and $\mathsf{val}(d) > q$, then since the rightmost d bits of $(a \ll d)$ are always 0, we have $\mathsf{val}(a \ll d)[q:p] = 0$ and $\mathsf{inv}(a \ll d)[q:p] = \mathsf{false}$.

The above argument shows that when the right-hand side of equation (10) evaluates to false, the least pessimistic value of $\mathsf{inv}((a \ll d)[q:p])$ is indeed non-X. Hence SoundCond holds. □

Example 10 Suppose $a = (1000, \mathsf{false})$ and $d = d_1 d_0$, where $d_1 = (1, \mathsf{true})$ and $d_0 = (0, \mathsf{false})$. Clearly, d can only take values in the set $\{00, 10\}$, representing either 0 or 2. It turns out that for either of these values of d, we have $(a \ll d)[2:2] = 0$. Therefore, the least pessimistic value of $(a \ll d)[2:2]$ is 0. Let us now evaluate equation (10). Since $\mathsf{inv}(d_1) = \mathsf{true}$, we have $\mathsf{inv}(d) = \mathsf{true}$, and hence $temp_1 = \mathsf{true}$. Similarly, since $\mathsf{val}(a[2:0] = 000$, we have $temp_2 = \mathsf{false}$ as well. Therefore, $\mathsf{inv}(a \ll d)[2:2]$ is false. From Lemma 2, we also have $\mathsf{val}(a \ll d)[2:2] = 0$. Therefore, the invalid-bit encoded value of $(a \ll d)[2:2]$ is $(0, \mathsf{false})$. Note that this matches exactly the least pessimistic value obtained above.

To see why equation (10) may yield true even when the least pessimistic value of $(a \ll d)[q:p]$ is non-$\mathbf{X}$, consider the same example as above, but suppose $a = (1010, \mathsf{false})$ now. It turns out that the least pessimistic value of $(a \ll d)[2:2]$ is still 0. However, $\mathsf{inv}(a \ll d)[2:2]$, obtained from equation (10), is true in this case.

6 Word-level STE

A word-level RTL design M consists of (potentially multi-bit) inputs, outputs, memory elements and internal signals. For notational simplicity, we treat bit-level signals as 1-bit words, and uniformly talk of words. Every word in the circuit is assumed to be atomized as described in Section 3. An m-bit wide atom takes values from the set $\{\mathbf{0} \ldots \mathbf{2^m - 1}, \mathbf{X}\}$, where constant bit-vectors have been represented in bold face by their integer values. The values themselves are ordered in a lattice as discussed in Section 4. Let $\mathcal{A}$ denote the collection of all atoms in the design M, and let $\mathcal{D}_{\mathcal{A}}$ denote the collection of values of all atoms in $\mathcal{A}$. A state of the design is a mapping $s : \mathcal{A} \to \mathcal{D}_{\mathcal{A}} \cup \{\top\}$ such that if $a \in \mathcal{A}$ is an m-bit atom, then $s(a)$ is a value in the set $\{\mathbf{0}, \ldots \mathbf{2^m - 1}, \mathbf{X}, \top\}$. As discussed in Section 2, the set of all states, $\mathfrak{S}$, of the design forms a lattice that is isomorphic to the product of lattices of values of atoms in $\mathcal{A}$. In the following, we use $\sqsubseteq$ to denote the partial order in the lattice of states.

We adapt the definition of symbolic trajectory formulas given in Section 2 for purposes of word-level STE (henceforth called WSTE). Let $\mathcal{V}$ be a set of symbolic bit-vector variables. A *simple word predicate* is a predicate of the form "a **is** *vexpr*", where a is an atom in the design M, and *vexpr* is an expression in the theory of bit-vectors with (zero or more) variables in $\mathcal{V}$. The predicate "a **is** *vexpr*" is said to be *well-formed* if the bit-widths of a and *vexpr* are equal. The defining value of "a **is** *vexpr*" is the unique state in which a is set to the value of *vexpr* and all other atoms are set to $\mathbf{X}$. An assignment $\phi : \mathcal{V} \to \{0,1\}^*$ is a mapping of variables in $\mathcal{V}$ to $\{0,1\}$-vectors such that for every $v \in \mathcal{V}$, the bit-widths of v and $\phi(v)$ match.

Following the convention used in STE tools like Forte, we specify a symbolic trajectory formula as a set of tuples $(g, \mathsf{a}, vexpr, t_1, t_2)$, where g (the guard) is a quantifier-free formula in the theory of bit-vectors with free variables in $\mathcal{V}$, a is the name of an atom in the design, *vexpr* is a bit-vector expression over variables in $\mathcal{V}$ such that *vexpr* has the same bit-width as a, and t_1, t_2 are natural numbers denoting time instants such that $t_2 \geq t_1 + 1$. For convenience of exposition, let $\mathbf{N}^0\varphi$ denote φ, and let $\mathbf{N}^k\varphi$ denote k nested applications of the next-time operator on φ, for every natural number k. The tuple $f = (g, \mathsf{a}, vexpr, t_1, t_2)$ represents the symbolic trajectory formula $\varphi_f \equiv \bigwedge_{k=t_1}^{t_2-1} \mathbf{N}^k (g \to (\mathsf{a}$ **is** $vexpr))$. Note that the simple word

predicate "a **is** *vexpr*" can also be specified in the tuple notation by letting the guard g be true. A set F of tuples like the one above represents the symbolic trajectory formula $\varphi_F = \bigwedge_{f \in F}(\varphi_f)$.

Let Ant be a set of antecedent tuples, and Cons be a set of consequent tuples for the design M under verification. The pair (Ant, Cons) represents the symbolic trajectory assertion $[\varphi_{\mathsf{Ant}} \Rightarrow \varphi_{\mathsf{Cons}}]$. In WSTE, we wish to determine if for every assignment ϕ of variables in $\mathcal{V}$, we have $[\varphi_{\mathsf{Cons}}]^\phi \sqsubseteq [\![\varphi_{\mathsf{Ant}}]\!]^\phi_M$. Often, we want to restrict the set of assignments ϕ to those that satisfy some constraints. In such cases, we specify the constraints as a quantifier-free formula Constr in the theory of bit-vectors, and ask if $[\varphi_{\mathsf{Cons}}]^\phi \sqsubseteq [\![\varphi_{\mathsf{Ant}}]\!]^\phi_M$ for every ϕ such that $\phi \models \mathsf{Constr}$. If Constr is unsatisfiable, or if the defining trajectory $[\![\varphi_{\mathsf{Ant}}]\!]^\phi_M$ has $\top$ (an unachievable value) as the value of an atom at any time instant, the symbolic trajectory assertion is vacuously satisfied. We avoid such cases by requiring Constr to be satisfiable and by requiring $[\![\varphi_{\mathsf{Ant}}]\!]^\phi_M$ to not have $\top$ as the value of any atom at any time.

WSTE using invalid-bit encoding: We have developed a tool called STEWord that uses symbolic simulation with invalid-bit encoding to construct a formula $\xi_{\mathsf{Ant},\mathsf{Cons},M}$ in the theory of bit-vectors, such that $\xi_{\mathsf{Ant},\mathsf{Cons},M}$ is valid iff the symbolic trajectory assertion $[\varphi_{\mathsf{Ant}} \Rightarrow \varphi_{\mathsf{Cons}}]$ holds on the design M. If we wish to check the symbolic trajectory assertion only for those assignments ϕ that satisfy a specified constraint Constr, we must check the validity of $\mathsf{Constr} \rightarrow \xi_{\mathsf{Ant},\mathsf{Cons},M}$. We describe below how to construct the formula $\xi_{\mathsf{Ant},\mathsf{Cons},M}$ from the sets of tuples Ant and Cons, and from the design M. Once $\xi_{\mathsf{Ant},\mathsf{Cons},M}$ is constructed, we use an off-the-shelf SMT solver with advanced word-level reasoning, viz. Boolector [4], to check the validity of $\mathsf{Constr} \rightarrow \xi_{\mathsf{Ant},\mathsf{Cons},M}$, or equivalently, the unsatisfiability of $\mathsf{Constr} \wedge \neg\xi_{\mathsf{Ant},\mathsf{Cons},M}$.

We first describe how the defining trajectory of φ_{Ant} is computed using invalid-bit encoding. Let T denote the depth of φ_{Ant}. For every $t \in \{0, \ldots T-1\}$ and for every atom $\mathsf{a} \in \mathcal{A}$, the value of a in the defining trajectory at time t is obtained as follows. Let $Z_{\mathsf{a},t}$ be the set of all tuples of the form $(g, \mathsf{a}, \mathit{vexpr}, t_1, t_2)$ in Ant, where $t_1 \leq t < t_2$. If $Z_{\mathsf{a},t}$ is non-empty, given an assignment ϕ of variables in $\mathcal{V}$, every tuple $\tau = (g, \mathsf{a}, \mathit{vexpr}, t_1, t_2)$ in $Z_{\mathsf{a},t}$ requires that if $\phi \models g$, the value of a at time t should be set to the value of *vexpr* evaluated on the assignment ϕ. If, on the other hand, $\phi \not\models g$, the tuple τ does not constrain the value of a at time t. These two cases can be summarized by saying that the value of a at time t, as required by the tuple $(g, \mathsf{a}, \mathit{vexpr}, t_1, t_2)$, is $(\mathit{vexpr}, \neg g)$ using invalid-bit encoding. Note that $\neg g$ represents the invalid bit of the encoded value. If there are multiple tuples in $Z_{\mathsf{a},t}$, the value of a at time t, as required by the antecedent φ_{Ant}, is simply the least upper bound of the values computed for each tuple τ in $Z_{\mathsf{a},t}$. The invalid-bit encoding of the least upper bound can be computed as described in Section 5.1. In the following discussion, we use $\mathsf{a}^{\mathsf{Ant}}_t$ to refer to the value of a at time t, as required by the antecedent φ_{Ant}.

If $Z_{\mathsf{a},t}$ is empty, the value of a at time t is not constrained by φ_{Ant}. This is equivalent to having a single antecedent tuple with a false guard in $Z_{\mathsf{a},t}$. An example of such a tuple is $(\mathsf{false}, \mathsf{a}, \mathbf{0}, t, t+1)$, where $\mathbf{0}$ is an arbitrarily chosen constant vector (all 0s) with the same bit-width as a. Note that since the guard is false, we could have used any other bit-vector expression of the same bit-width as a for the value. In the following, we assume without loss of generality that $Z_{\mathsf{a},t}$ is non-empty.

If a is a primary input atom or if $t = 0$, the antecedent tuples in $Z_{\mathsf{a},t}$ effectively specify how the atom must be driven at time t. In these cases, the value of a in the defining trajectory at time t is set to $\mathsf{a}_t^{\mathsf{Ant}}$. If a is not a primary input atom and if $t > 0$, determining the value of a in the defining trajectory at time t requires us to compute the least upper bound of the value driven by the design M on a at time t, and $\mathsf{a}_t^{\mathsf{Ant}}$. The value driven by M at time t, say a_t^M, can be computed by symbolic simulation using invalid-bit encoding, as explained in Sections 5.2.1 and 5.2.2. The least upper bound of $\mathsf{a}_t^{\mathsf{Ant}}$ and a_t^M can then be computed as described in Section 5.1. The above procedure yields a unique invalid-bit encoded value for every atom a in the design M at every time instant $t \in \{0, \ldots T-1\}$. This gives us the defining trajectory of φ_{Ant} for M.

In the above discussion, we implicitly assumed that whenever a least upper bound is computed, the result is not $\top$. We now make this assumption explicit. Note that if computing a least upper bound yields $\top$ while constructing the defining trajectory, we must set the value of an atom to an unachievable over-constrained value in the defining trajectory. Clearly, this is not very meaningful, and is called *antecedent failure* in STE parlance. We collect all constraints (conditions for case (a) in Section 5.1) under which antecedent failure occurs in a set $\mathsf{AntFail}$. The assumption that no least upper bound computation yields $\top$ is equivalent to assuming that every constraint in $\mathsf{AntFail}$ is unsatisfiable. Depending on the mode of analysis, we then do one of the following.

- If we want non-vacuous satisfaction of the symbolic trajectory assertion for *all assignments* ϕ, then we check for unsatisfiability of the disjunction of constraints in $\mathsf{AntFail}$. If the disjunction is satisfiable, we conclude that there is an assignment ϕ of variables in $\mathcal{V}$ that leads to an antecedent failure. This is then viewed as a failed run of verification.
- If we want the symbolic trajectory assertion to hold only for assignments ϕ that do not cause an antecedent failure, then we negate every constraint in $\mathsf{AntFail}$ and take their conjunction to obtain a constraint, say $\mathsf{NoAntFail}$, that defines all assignments ϕ that do not lead to any antecedent failure. If Constr represents the user-provided constraint for restricting assignments ϕ, we can now check if $[\![\mathsf{Cons}]\!]^{\phi} \sqsubseteq [\![\mathsf{Ant}]\!]^{\phi}_{M}$ holds for assignments that satisfy $(\mathsf{Constr} \wedge \mathsf{NoAntFail})$.

We now describe how the defining sequence of φ_{Cons} is computed using invalid-bit encoding. As in the case of φ_{Ant}, let T denote the depth of φ_{Cons}. For every $t \in \{0, \ldots T-1\}$ and for every atom $\mathsf{a} \in \mathcal{A}$, let $Y_{\mathsf{a},t}$ be the set of all tuples of the form $(g, \mathsf{a}, \mathit{vexpr}, t_1, t_2)$ in Cons, where $t_1 \leq t < t_2$. We now have two cases to consider:

- If $Y_{\mathsf{a},t}$ is empty, φ_{Cons} imposes no requirement on the value of a at time t. In this case, the weakest admissible value of a at time t is $\mathbf{X}$, which can be encoded as $(\mathbf{0}, \mathsf{true})$ using invalid-bit encoding. Note that since the invalid bit is true, the value is an arbitrarily chosen bit-vector ($\mathbf{0}$ in this case) of the same bit-width as a.
- If $Y_{\mathsf{a},t}$ is non-empty, every consequent tuple $\tau = (g, \mathsf{a}, \mathit{vexpr}, t_1, t_2) \in Y_{\mathsf{a},t}$ specifies that given an assignment ϕ, if $\phi \models g$, then atom a must have its invalid bit set to false and value set to vexpr evaluated on the assignment ϕ, at time t. If $\phi \not\models g$, the tuple τ imposes no requirement on the value of a at time t.

The above two cases can be summarized by saying that the weakest admissible value of a at time t, as required by the tuple τ, is $(\mathit{vexpr}, \neg g)$, using invalid-bit

encoding. In case $Y_{\mathsf{a},t}$ has multiple tuples, the weakest admissible value is obtained by taking the least upper bound of the weakest admissible values of a at time t, as obtained from each tuple $\tau \in Y_{\mathsf{a},t}$. The above procedure gives a unique invalid-bit encoding of the value of every atom a at every time instant $t \in \{0, \ldots T-1\}$ in the defining sequence of φ_{Cons}.

Let $\mathsf{a}_t^{\mathsf{Ant},M}$ be the invalid-bit encoded value of atom a at time t in the defining trajectory of φ_{Ant} for design M. Let $\mathsf{a}_t^{\mathsf{Cons}}$ be the invalid-bit encoded weakest admissible value of a at time t in the defining sequence of φ_{Cons}. We are now ready to define the formula $\xi_{\mathsf{Ant},\mathsf{Cons},M}$ referred to earlier in this section. Specifically, let $\xi_{\mathsf{Ant},\mathsf{Cons},M}$ be the following formula in the theory of bit-vectors with equality:

$$\bigwedge_{\mathsf{a}\in\mathcal{A}\,,\,t\in\{0,\ldots T-1\}} \left(\neg\mathsf{inv}\left(\mathsf{a}_t^{\mathsf{Cons}}\right) \rightarrow \left(\neg\mathsf{inv}\left(\mathsf{a}_t^{\mathsf{Ant},M}\right) \wedge \left(\mathsf{val}\left(\mathsf{a}_t^{\mathsf{Cons}}\right) = \mathsf{val}\left(\mathsf{a}_t^{\mathsf{Ant},M}\right)\right)\right)\right).$$

We claim that symbolic trajectory evaluation on the design M reduces to checking validity of the formula $\xi_{\mathsf{Ant},\mathsf{Cons},M}$.

Theorem 1 *The symbolic trajectory assertion* $[\varphi_{\mathsf{Ant}} \Rightarrow \varphi_{\mathsf{Cons}}]$ *holds for design* M *iff the bit-vector formula* $\xi_{\mathsf{Ant},\mathsf{Cons},M}$ *is valid.*

Proof By the central theorem of the general theory of STE [23], the symbolic trajectory assertion $[\varphi_{\mathsf{Ant}} \Rightarrow \varphi_{\mathsf{Cons}}]$ holds for design M iff $[\varphi_{\mathsf{Cons}}]^{\phi} \sqsubseteq [\![\varphi_{\mathsf{Ant}}]\!]_M^{\phi}$ holds for all assignments ϕ of variables in $\mathcal{V}$. For every atom a in M, let $bw(\mathsf{a})$ denote the bit-width of a, and let $\sqsubseteq_{bw(\mathsf{a})}$ denote the partial order in the lattice of values of a. The requirement $[\varphi_{\mathsf{Cons}}]^{\phi} \sqsubseteq [\![\varphi_{\mathsf{Ant}}]\!]_M^{\phi}$ is then equivalent to the requirement $\mathsf{a}_t^{\mathsf{Cons}} \sqsubseteq_{bw(\mathsf{a})} \mathsf{a}_t^{\mathsf{Ant},M}$, for every atom a in M and for every time instant $t \in \{0, \ldots T-1\}$. It therefore suffices to show that $\mathsf{a}_t^{\mathsf{Cons}} \sqsubseteq_{bw(\mathsf{a})} \mathsf{a}_t^{\mathsf{Ant},M}$ holds iff the formula $\left(\neg\mathsf{inv}(\mathsf{a}_t^{\mathsf{Cons}}) \rightarrow \left(\neg\mathsf{inv}(\mathsf{a}_t^{\mathsf{Ant},M}) \wedge \left(\mathsf{val}(\mathsf{a}_t^{\mathsf{Cons}}) = \mathsf{val}(\mathsf{a}_t^{\mathsf{Ant},M})\right)\right)\right)$ is valid. However, this follows directly from the interpretation of the invalid-bit encoded values $\mathsf{a}_t^{\mathsf{Cons}}$ and $\mathsf{a}_t^{\mathsf{Ant},M}$, and from the definition of $\sqsubseteq_{bw(\mathsf{a})}$. □

Note that if sound approximations of invalid bits are used (as discussed in Section 5.2.2), then the "iff" in Theorem 1 must be replaced by "if". Furthermore, if we wish to focus only on assignments ϕ that do not cause any antecedent failure and also satisfy a user specified constraint Constr, our verification goal is modified to checking the validity of $\mathsf{NoAntFail} \wedge \mathsf{Constr} \rightarrow \xi_{\mathsf{Ant},\mathsf{Cons},M}$. Recall from Section 5.2 that auxiliary variables and assumptions/constraints on these variables may need to be introduced when symbolically simulating RTL operators with invalid-bit encoding (an example being the simulation of division). Let Aux denote the conjunction of all assumptions/constraints on auxiliary variables needed during symbolic simulation. The verification goal is then refined to checking the validity of $\mathsf{NoAntFail} \wedge \mathsf{Constr} \wedge \mathsf{Aux} \rightarrow \xi_{\mathsf{Ant},\mathsf{Cons},M}$.In our implementation, we use Boolector [4], a state-of-the-art solver for bit-vectors and the theory of arrays, to check the validity of the verification conditions generated by STEWord.

7 Implementation issues

In this section, we discuss several implementation related issues that contribute to making STEWord efficient and useful in practice.

7.1 Compile once, evaluate many times

Symbolically simulating an RTL design requires taking into account details of the simulation semantics, as defined in the language reference manual, in each simulated time step. In WSTE, we are often required to simulate a design for a large number of time steps. Instead of expending the computational effort required to stay faithful to the detailed simulation semantics repeatedly, we "compile" the design by symbolically simulating it for only a single time step, but with fresh symbolic variables for the invalid-bits and values of all primary input atoms and atoms at the outputs of state-holding elements. This gives generic symbolic expressions for the invalid bits and values of all atoms in the design. In order to obtain the invalid bit and value expressions for an atom in a specific step of symbolic simulation, it suffices to evaluate (or interpret) the generic expressions obtained above for that atom with specific expressions/values of primary input and state atoms, as obtained from the simulation context.

Since no assumptions are made about the symbolic invalid bits and values of primary input and state atoms during the compilation step, any expression-level optimization done in the compilation step continues to be applicable in every step of symbolic simulation. We save ourselves the computational effort required to apply these optimizations in every step of symbolic simulation by doing it only once in the compilation step, and using the optimized expressions repeatedly. Furthermore, as discussed in Section 6, to construct the formula $\xi_{\mathsf{Ant},\mathsf{Cons},M}$, we need the symbolic expressions for the invalid bits and values of only those atoms that appear in either an antecedent tuple or a consequent tuple. The symbolic expressions for atoms at the inputs of state-holding elements are also required in order to simulate the copying of values from the inputs of state-holding elements to their outputs at appropriate cycle boundaries. The expressions for all other atoms are not needed and may be discarded after the compilation step. This can lead to significant savings in both time and space required to symbolically simulate the design, especially if a large number of time steps are involved.

7.2 Weakening atoms

In practical verification scenarios, it often becomes desirable to inject values at the outputs of state-holding elements or on internal signals of a design at specified time points. State-of-the-art bit-level STE tools like Forte allow this by permitting signals to be *weakened* at specific time points. If a signal is weakened at time t, during the course of symbolic simulation, its value at t is not determined by the simulation semantics; instead it is set to the injected value, often specified through an antecedent. We adopt the same approach for WSTE, and use *weakening tuples* to specify weakening of signals. A weakening tuple is of the form $(g, \mathsf{a}, t_1, t_2)$, where g (the guard) is a quantifier-free expression in the theory of bit-vectors with variables in $\mathcal{V}$, a is the name of an atom in the design, and $t_1, t_2 \in \mathbb{N}$ are time instants with $t_2 \geq t_1 + 1$. A weakening tuple specifies that if the guard g evaluates to true for the given assignment ϕ of variables in $\mathcal{V}$, then the atom a is weakened at all time instants $t \in \{t_1, \ldots t_2 - 1\}$, and takes its value directly from the antecedent. The complete specification of weakened signals is provided by a set, Weak, of weakening tuples.

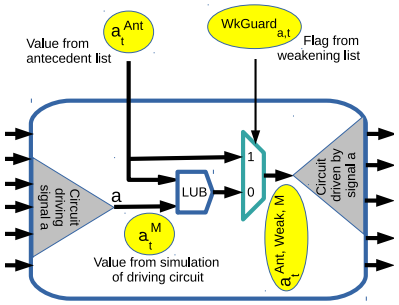

Fig. 4 Computing weakened value of signal a.

To implement weakening during symbolic simulation, we modify the symbolic expressions for weakened atoms as follows. Let a be an atom of the design M, and let T be the depth of the antecedent or consequent in the symbolic trajectory assertion under consideration. For every time instant $t \in \{0, \ldots T-1\}$, let $W_{\mathsf{a},t}$ be the set of weakening tuples of the form $(g, \mathsf{a}, t_1, t_2)$ in Weak such that $t_1 \leq t < t_2$. Let $\mathsf{wkGuard}_{a,t}$ denote the disjunction of guards g from all weakening tuples in $W_{\mathsf{a},t}$. If $W_{\mathsf{a},t}$ is empty, we let $\mathsf{wkGuard}_{a,t}$ be false. Clearly, if an assignment ϕ of variables in $\mathcal{V}$ renders $\mathsf{wkGuard}_{a,t}$ true, then a is weakened at time t. Using notation introduced in Section 6, let $\mathsf{a}_t^M = (\mathsf{val}(\mathsf{a}_t^M), \mathsf{inv}(\mathsf{a}_t^M))$ be the invalid-bit encoding of the value of a, as obtained from symbolic simulation of the design M *without considering weakening of* a. Similarly, let $\mathsf{a}_t^{\mathsf{Ant}} = (\mathsf{val}(\mathsf{a}_t^{\mathsf{Ant}}), \mathsf{inv}(\mathsf{a}_t^{\mathsf{Ant}}))$ denote the invalid-bit encoding of the value of a at time t, as required by the antecedent Ant. The invalid-bit encoding of the value of a at time t after considering weakening, denoted $\mathsf{a}_t^{\mathsf{Ant,Weak},M}$, is then obtained as follows.

$$\mathsf{val}(\mathsf{a}_t^{\mathsf{Ant,Weak},M}) = \mathsf{ite}\left(\mathsf{wkGuard}_{a,t},\ \mathsf{val}\left(\mathsf{a}_t^{\mathsf{Ant}}\right),\ \mathsf{val}\left(lub\left(\mathsf{a}_t^{M}, \mathsf{a}_t^{\mathsf{Ant}}\right)\right)\right) \quad (11)$$

$$\mathsf{inv}(\mathsf{a}_t^{\mathsf{Ant,Weak},M}) = \mathsf{ite}\left(\mathsf{wkGuard}_{a,t},\ \mathsf{inv}\left(\mathsf{a}_t^{\mathsf{Ant}}\right),\ \mathsf{inv}\left(lub\left(\mathsf{a}_t^{M}, \mathsf{a}_t^{\mathsf{Ant}}\right)\right)\right) \quad (12)$$

The computation of $\mathsf{a}_t^{\mathsf{Ant,Weak},M}$ is pictorially depicted in Fig. 4. Note that the invalid-bit encoding of $lub\left(\mathsf{a}_t^{M}, \mathsf{a}_t^{\mathsf{Ant}}\right)$ can be easily computed from the invalid-bit encodings of a_t^{M} and $\mathsf{a}_t^{\mathsf{Ant}}$, as described in Section 5.1. Equations 11 and 12 therefore give the invalid-bit encoding of the value of a at time t in the defining trajectory of Ant for the design M, with signals weakened as specified in Weak.

The above discussion and the one in Section 6 assumed that the sets $W_{\mathsf{a},t}$ and $Z_{\mathsf{a},t}$ were non-empty for every atom a, and for every time instant $t \in \{0, \ldots T-1\}$. In our implementation of STEWord, we do not augment the user-provided sets Ant and Weak to ensure this. Instead, for every atom a that appears in an antecedent tuple, we introduce an auxiliary Boolean input, named inAnt_a. Similarly, for every atom a that appears in a weakening tuple, we introduce an auxiliary Boolean input, named inWeak_a. These inputs are then assigned values from {true, false} at every time instant $t \in \{0, \ldots T-1\}$ as follows.

- If there exists a weakening tuple $(g, \mathsf{a}, t_1, t_2) \in \mathsf{Weak}$ such that $t_1 \leq t < t_2$, we set inWeak_a to true at time t; otherwise inWeak_a is set to false at time t.
- If there exists an antecedent tuple $(g, \mathsf{a}, \mathit{vexpr}, t_1, t_2) \in \mathsf{Ant}$ such that $t_1 \leq t < t_2$, we set inAnt_a to true at time t; otherwise inAnt_a is set to false at time t.

If atom a appears in some weakening tuple, and if inWeak_a is false at time t, the value of $\mathsf{wkGuard}_{a,t}$ in Equations 12 and 11 is set to false during symbolic simulation. If a doesn't appear in any weakening tuple, Equations 12 and 11 are simplified assuming $\mathsf{wkGuard}_{a,t}$ is false at all time instances t. Similarly, if a appears in some

antecedent tuple, and if inAnt_a is false at time t, the value of $\mathsf{inv}^{\mathsf{Ant}}_{\mathsf{a},t}$ in Equations 12 and 11 is set to true during symbolic simulation. Finally, if a doesn't appear in any antecedent tuple, Equations 12 and 11 are simplified assuming $\mathsf{inv}^{\mathsf{Ant}}_{\mathsf{a},t}$ is true at all time instants t.

7.3 On-the-fly simplifications of invalid-bit encoded values

Recall from Section 5 that if the invalid-bit encoding of the value of an atom is $(vexpr, \mathsf{true})$, i.e. the invalid bit in the encoding is set to true, then the expression $vexpr$ used in the first component of the encoding is of no consequence. This provides us an opportunity to simplify invalid-bit encoded values significantly on-the-fly. Specifically, if at any time during symbolic simulation, the invalid-bit encoding of an atom a becomes $(vexpr, \mathsf{true})$, we can replace $vexpr$ with an arbitrary bit-vector constant having the same bit-width as a. Since constants are among the simplest forms of symbolic expression, this can lead to significant simplifications in the subsequent simulation and reasoning.

In general, different constants can be chosen to replace $vexpr$ in different contexts to simplify subsequent symbolic computations. For example, if the atom a is an input to a word-level multiplication operator in the design M, then since 0 is the annihilator for multiplication, it is beneficial to replace $vexpr$ with $\mathbf{0}$, where $\mathbf{0}$ denotes the bit-vector of all 0s having the same bit-width as a. On the other hand, if a is an input to a bit-wise logical OR operator, then since the bit-wise OR of any bit-vector with a vector of all 1s gives a vector of all 1s, it is beneficial to replace $vexpr$ with $\mathbf{2^m - 1}$, where $\mathbf{2^m - 1}$ denotes the bit-vector of all 1s having the same bit-width (m) as a. In general, finding the optimal constant to replace $vexpr$ is an interesting problem. However, in the interests of efficiency and simplicity, our implementation of $\mathsf{STEWord}$ uses $\mathbf{0}$ for the value component of the invalid bit encoding of an atom whose invalid bit is set to true. In case this leads to a divisor in a division operation being assigned the value $(\mathbf{0}, \mathsf{true})$, we replace the divisor with $(\mathbf{1}, \mathsf{true})$.

In practical WSTE-based verification scenarios, often several primary input and state atoms are left unconstrained (i.e. $\mathbf{X}$), since they are not relevant to the property being verified. This can cause several internal atoms and atoms at the inputs of state-holding elements to have unconstrained values (i.e. $\mathbf{X}$) during symbolic simulation. Since values at the inputs of state-holding elements must be copied to the outputs of these elements in each simulation time step, the complexity of symbolic expressions can increase significantly if we symbolically simulate over a large number of time steps without simplifying the value components of invalid-bit encodings of state atoms that evaluate to $\mathbf{X}$. By replacing these value components by $\mathbf{0}$ whenever the invalid bit of the encoding simplifies to true, we obtain significant savings in both time and space required for symbolic simulation. Yet another opportunity for on-the-fly simplifications arises when computing the least upper bound of two invalid-bit encoded values. This was explained at the end of Section 5.1.

7.4 Managing symbolic expressions

Despite the optimizations discussed above, word-level STE spanning a large number of cycles can result in significant sizes of symbolic expressions. In order to manage and share symbolic expressions efficiently, we have developed a symbolic expression manager to work with STEWord. Our expression manager represents symbolic expressions as dynamically allocated directed acyclic graphs, and provides several useful functionalities through APIs to reason about these expressions. It also manages dynamically allocated memory efficiently during symbolic simulation. The expression manager uses structural hashing to identify equivalence of two expressions and re-uses already existing expressions whenever possible. Structural hashing ensures that no two expressions in the manager are structurally the same. This results in significant re-use of expressions during the course of symbolic simulation. Another important feature of the expression manager is a rule-based DAG rewrite facility. This allows the developer (and also verification engineer) to incrementally define complex transformation rules that can be used to simplify expressions represented as DAGs on-the-fly. Every time a new expression is created in the expression manager, a compact signature for the expression is created. This is then used to efficiently search a collection of DAG rewrite rules to determine if any rule can be used to simplify the expression. If the answer is in the affirmative, the rewrite rule is applied to simplify the expression, and the expression manager's data structures and hash tables are updated to reflect the transformation. Note that the onus of verifying the semantic equivalence of the transformed expression and the original expression lies with the developer/user specifying the transformation rules. Currently, our implementation has a core set of approximately 20 rules whose correctness has been manually verified. These rules play a very important role in simplifying symbolic expressions and keeping their sizes under control.

Table 2 presents a small sampling of our DAG rewrite rules, where `$e1, $e2` etc. represent placeholders for (not necessarily distinct) sub-DAGs. In this table, a DAG fragment rooted a node `n` is specified in the format "`(node-label-of-n child-dag-1 child-dag-2 ...)`". If each `child-dag-i` represents an expression $expr_i$ and if `node-label-of-n` is the operator `op`, then `(node-label-of-n child-dag-1 child-dag-2 ...)` is assumed to represent the expression $\texttt{op}(expr_1, expr_2, \ldots)$. Each re-write rule also has an associated condition, which must be satisfied before the re-write rule is applied. For example, the last rule in Table 2 is applied only if both `$e1` and `$e2` are DAG nodes representing constants.

Expression before re-write	**Expression after re-write**	**Condition**
`(and true $e1)`	`$e1`	None
`(not (not $e1))`	`$e1`	None
`(ite false $e1 $e2)`	`$e2`	None
`(ite true $e1 $e2)`	`$e1`	None
`(ite $e1 (ite (not $e1) $e2 $e3) $e4)`	`(ite $e1 $e3 $e4)`	None
`(select (bitnot $e1) $e2 $e3)`	`(bitnot (select $e1 $e2 $e3))`	None
`(add (mult $e1 $e2) (mult $e1 $e3))`	`(mult $e1 (add $e2 $e3))`	`$e2`, `$e3` are constants

Table 2 A subset of re-write rules for DAGs

7.5 Handling structurally cyclic combinational circuits

Combinational circuits with structural cycles are not uncommon in high-performance designs. Carefully designed cyclic circuits can enhance performance without causing instabilities on internal signals. Hence, we support reasoning about such designs in STEWord.

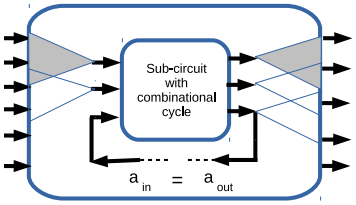

Fig. 5 Cutting an atom to break cycles.

When simulating a design with structurally cyclic combinational circuits, special care has to be taken to generate the symbolic expressions for the outputs of these circuits. In our implementation of STEWord, we proceed initially with symbolic simulation without assuming anything about the presence of structural cycles in the design. For some designs, even when a structural cycle is present, the values of different atoms in the cycle may be such that there is no cyclic dependency in the symbolic expressions generated for various atoms. Such cases do not require any special handling. However, if the expression manager detects a cyclic dependency when constructing symbolic expressions for atoms along a cycle, additional processing must be done. In such cases, STEWord continues to symbolically simulate operators along the structural cycle for a pre-determined number k of iterations (currently, $k = 2$ in our expression manager). This is equivalent to unrolling the cycle k times and then symbolically simulating the unrolled circuit. If the symbolic expression for each atom along a cycle remains unchanged in the last two of the k iterations (this can be detected through structural hashing in the expression manager), we conclude that the circuit stabilizes in at most k iterations of cyclic dependencies, and proceed with symbolically simulating other parts of the design. Otherwise, we explicitly break cycles by cutting an atom in each cycle, thereby generating new paired internal atoms. We explain this in some detail below.

Suppose a cycle is broken by cutting an internal atom a. This generates two new atoms, denoted a_{out} and a_{in}, where a_{out} is driven by some gate/operator in the circuit, and a_{in} drives some (possibly different) gate/operator in the circuit. Figure 5 depicts an example of this, where the dotted line represents an atom a that has been cut to break a cycle in a combinational sub-circuit. The symbolic simulator maintains the information that $(\mathsf{a}_{out}, \mathsf{a}_{in})$ is a paired set of internal atoms, generated by breaking a cycle. The atom a_{out} may be considered as a primary output of the design after breaking the cycle. Similarly, the atom a_{in} may be thought of as a primary input, except that its invalid-bit encoded value in each time step is not determined by the antecedents, unlike actual primary inputs of the design. Instead, the invalid bit and value expression of a_{in} are assigned new symbolic variables in every simulation time step.

Once all cycles are broken, we proceed with symbolic simulation of the transformed circuit (i.e., original circuit with cycles broken) as usual. However, for each time step t of symbolic simulation, we assert that the values of a_{out} and a_{in} are the same modulo invalid-bit encoding. Let $(\mathsf{val}(\mathsf{a}_{out,t}), \mathsf{inv}(\mathsf{a}_{out,t}))$ and $(\mathsf{val}(\mathsf{a}_{in,t}), \mathsf{inv}(\mathsf{a}_{in,t}))$ represent the invalid-bit encodings of a_{out} and a_{in} respectively at time t, during symbolic simulation of the design. Define the bit-vector formula $\mathsf{cycleEq}_{\mathsf{a},t} \equiv (\mathsf{inv}(\mathsf{a}_{out,t}) = \mathsf{inv}(\mathsf{a}_{in,t})) \wedge (\neg\mathsf{inv}(\mathsf{a}_{out,t}) \rightarrow (\mathsf{val}(\mathsf{a}_{out,t}) = \mathsf{val}(\mathsf{a}_{in,t}))$ that

asserts that the invalid bits of a_{out} and a_{in} are the same at time t, and their values are also the same if a_{out} doesn't evaluate to $\mathbf{X}$ at time t. Let CycleCuts denotes the set of atoms that are cut to break all cycles in the design. The formula $\mathsf{CycleConstr} \equiv \bigwedge_{\mathsf{a} \in \mathsf{CycleCuts},\, t \in \{0,\ldots T-1\}} \mathsf{cycleEq}_{\mathsf{a},t}$ asserts the equality of all paired internal atoms generating by breaking cycles at all time steps. Clearly, we must consider only those assignments ϕ of variables in $\mathcal{V}$ that satisfy CycleConstr during symbolic trajectory evaluation. Every such assignment ϕ gives rise to fixed points in the evaluation of structurally cyclic combinational circuits in the design. Using notation introduced in Section 6, the bit-vector formula CycleConstr must be conjoined with the user-provided set of constraints Constr and the constraints Aux on auxiliary variables, to obtain the final set of constraints restricting assignments ϕ. Thus, for designs with structurally cyclic combinational circuits, checking the validity of the symbolic trajectory assertion $[\varphi_{\mathsf{Ant}} \Rightarrow \varphi_{\mathsf{Cons}}]$ reduces to checking the validity of $\mathsf{NoAntFail} \wedge \mathsf{Constr} \wedge \mathsf{Aux} \wedge \mathsf{CycleConstr} \rightarrow \xi_{\mathsf{Ant},\mathsf{Cons},M}$

8 Experiments

We used STEWord to verify properties of a set of SystemVerilog word-level benchmark designs. Bit-level STE tools are often known to require user-guidance with respect to problem decomposition and variable ordering (for BDD based tools), when verifying properties of designs with moderate to wide datapaths. Similarly, BMC tools need to introduce a fresh variable for each input in each time frame when the value of the input is unspecified. Our benchmarks were intended to stress bit-level STE tools, and included designs with control and datapath logic, where the width of the datapath was parameterized. Our benchmarks were also intended to stress BMC tools by providing relatively long sequences of inputs that could either be X or a specified symbolic value, depending on a symbolic condition. In each case, we verified properties that were satisfied by the system and those that were not. For comparative evaluation, we implemented word-level bounded model checking as an additional feature of STEWord itself. Below, we first give a brief description of each design, followed by a discussion of our experiments. In the following, designs 1 through 4 are obtained from our work in [8].

Design 1: Our first design was a three-stage pipelined circuit that read four pairs of k-bit words in each cycle, computed the absolute difference of each pair, and then added the absolute differences with a current running sum. Alternatively, if a reset signal was asserted, the pipeline stage that stored the sum was reset to the all-zero value, and the addition of absolute differences of pairs of inputs started afresh from the next cycle. In order to reduce the stage delays in the pipeline, the running sum was stored in a redundant format and carry-save-adders were used to perform all additions/subtractions. Only in the final stage was the non-redundant result computed. In addition, the design made extensive use of clock gating to reduce its dynamic power consumption – a characteristic of most modern designs and that significantly complicates formal verification. Because of the non-trivial control and clock gating, the STE verification required a simple datapath invariant. Furthermore, in order to reduce the complexity in specifying the correctness, we broke down the overall verification goal into six properties, and verified these properties using several datapath widths.

Design 2: Our second design was a pipelined serial multiplier that read two k-bit inputs serially from a single k-bit input port, multiplied them and made the result available on a $2k$-bit wide output port in the cycle after the second input was read. The entire multiplication cycle was then re-started afresh. By asserting and de-asserting special input flags, the control logic allowed the circuit to wait indefinitely between reading its first and second inputs, and also between reading its second input and making the result available. We verified several properties of this circuit, including checking whether the result computed was indeed the product of two values read from the inputs, whether the inputs and results were correctly stored in intermediate pipeline stages for various sequences of asserting and de-asserting of the input flags, etc. In each case, we tried the verification runs using different values of the bit-width k.

Design 3: Our third design was an implementation of the first stage in a typical (but simplified) digital camera pipeline. The design is fed as a stream of data from the output of a single CCD/CMOS sensor array whose pixels have different color filters in front of them in a Bayer mosaic pattern [21]. The design takes these values and performs a "de-mosaicing" of the image, which basically uses a fairly sophisticated interpolation technique (including edge detection) to estimate the missing color values. The circuit computes three streams of data for the computed red, green, and blue values. The challenge here was not only verifying the computation, which entailed adding a fairly large number of scaled inputs, but also verifying that the correct pixel values were used. In fact, most non-STE based formal verification engines will encounter difficulty with this design since the final result depends on a large number of 8-bit quantities. For this test case, we focused on one particular (internal) cell and verified that the computed values for the "missing" pixel for that cell was correct. For a complete verification, one would have to repeat this verification for every cell in the design.

Design 4: Our fourth design was a more general version of Design 3. Here we used the same design as in 3, but now we verified in one single STE run that all the cells in the output stream had the correct value. This was done by using a vector of Boolean variables to select the particular cell in the picture array and thus the time in the input stream when the relevant data (and the data for all its relevant neighbors) was needed, as well as when it should be produced in the output streams. Analyzing this example with BMC would require providing new inputs every cycle for over 200 cycles, leading to a blow-up in the number of variables used.

For each benchmark design, we experimented with a bug-free version, and with several buggy versions. For bit-level verification, we used both a BDD-based STE tool [24] and propositional SAT based STE tool [22]; specifically, the public-domain version of the tool Forte was used for bit-level STE. We also ran word-level BMC to verify the same properties.

In all our benchmarks, we found that Forte and STEWord successfully verified the properties within a few seconds when the bit-width was small (8 bits). However, the running time of Forte increased significantly with increasing bit-width, and for bit-widths of 16 and above, Forte could not verify the properties without serious user intervention. In contrast, STEWord required practically the same time to verify properties of circuits with wide datapaths, as was needed to verify properties of the same circuits with narrower datapaths, and required no user intervention. In fact, the word-level SMT constraints generated for a circuit with a narrow datapath

are almost identical to those generated for a circuit with a wider datapath, except for the bit-widths of atoms. This is not surprising, since once atomization is done, symbolic simulation is agnostic to the widths of various atoms. An advanced SMT solver like Boolector is often able to exploit the word-level structure of the final set of constraints and solve it without resorting to bit-blasting.

The BMC runs required adding a fresh variable in each time frame when the value of an input was not specified or conditionally specified. This resulted in a significant blow-up in the number of additional variables, especially when we have long sequences of conditionally driven inputs. This in turn adversely affected SMT-solving time, causing BMC to timeout in some cases.

To illustrate how the verification effort with STEWord compared with the effort required to verify the same property with a bit-level BDD- or SAT-based STE tool, and with word-level BMC, we present a sampling of our observations in Table 3, where no user intervention was allowed for any tool. Here "-" indicates more than 2 hours of running time, and all times are on an Intel Xeon E7520 1.86GHz CPU, using a single core. In the column labeled "Benchmark", Design$i$-Pj corresponds to verifying property j (from a list of properties) on Design i. The column labeled "Word-level latches (# bits)" gives the number of word-level latches and the total number of bits in those latches for a given benchmark. The column labeled "Sim Cycles" gives the total number of time-frames for which STE and BMC was run. The column labeled "(Partial) Atomization Profile" gives the partial profile of how words were atomized into atoms of different widths by our atomization step. An entry like $32 + 1(4)$ in this column indicates that there were 4 words of bit-width 33, each of which got atomized into a 32-bit atom and a 1-bit atom. Similarly, an entry like 32(37) indicates that there were 37 words of bit-width 32, each of which remained intact as a 32-bit atom. Designs 3 and 4 had long atomization profiles, and only part of the profile for these designs has been presented in Table 3. Clearly, atomization did not bit-blast all words, allowing us to reason at the granularity of multi-bit atoms in STEWord. In fact, for some designs, viz. Design 2, none of the words were atomized into thinner slices. This is indeed possible, since as discussed in Section 3, the atomization of a word strongly depends on how the RTL has been written.

Our experiments indicate that when a property is not satisfied by a circuit, Boolector finds a counterexample quickly due to powerful search heuristics implemented in modern SMT solvers. BDD-based bit-level STE engines are, however, likely to suffer from BDD size explosion in such cases, especially when the bit-widths are large. In cases where there are long sequences of conditionally driven inputs (e.g., design 4) BMC performs worse compared to STEWord, presumably because of the added complexity of solving constraints with significantly larger number of variables. In other cases, the performance of BMC is comparable to that of STEWord. An important observation is that the abstractions introduced by atomization and by approximations of invalid-bit expressions do not cause STEWord to produce conservative results in any of our experiments. Thus, STEWord strikes a good balance between accuracy and performance. Another interesting observation is that for correct designs and properties, SMT solvers (all we tried) sometimes fail to verify the correctness (by proving unsatisfiability of a formula). This points to the need for further developments in SMT solving, particularly for proving unsatisfiability of complex formulas. Overall, our experiments, though limited, show that

Benchmark	STEWord	Forte (BDD & SAT)	BMC	Word-level latches (# bits)	Sim Cycles	(Partial) Atomization Profile
Design1-P1 (32 bits)	2.38s	- -	3.71s	14 latches (235 bits)	12	1(24), 1+1+1(2), 1+1+1+1(2), 32(37), 1+31(1), 33(6), 1+32(4)
Design1-P1 (64 bits)	2.77s	- -	4.53s	14 latches (463 bits)	12	1(24), 1+1+1(2), 1+1+1+1(2), 32(2), 64(35), 1+63(1), 65(6), 1+64(4)
Design2-P2 (16 bits)	1.56s	- -	1.50s	4 latches (96 bits)	6	1(7), 16(5), 32(3)
Design2-P2 (32 bits)	1.65s	- -	1.52s	4 latches (128 bits)	6	1(7), 32(7), 64(1)
Design3-P3 (16 bits)	24.06s	- -	-	54 latches (787 bits)	124	1(7), 5(1), 1+3(2), 1+1+1+1+1(7), 7(1), 4+16(4), 16(26), 32(4), 16+16+16+16(1), 16+16+16+16+16(1), ...
Design4-P4 (16 bits)	56.80s	- -	-	54 latches (787 bits)	260	1(29), 4(6), 1+3(8), 5(1) 1+1+1+1+1(7), 7(1), 16(43), 20(15), 4+16(4), 1+19(4), 16+16+16+16(1), 16+16+16+16+16(1), ...
Design4-P4 (32 bits)	55.65s	- -	-	54 latches (1555 bits)	260	1(29), 4(6), 1+3(8), 5(1) 1+1+1+1+1(7), 7(1), 32(43), 36(15), 4+32(4), 1+35(4), 32+32+32+32(1), 32+32+32+32+32(1), ...

Table 3 Comparing verification effort (time) with STEWord, Forte and BMC

word-level STE can be beneficial compared to both bit-level STE and word-level BMC in real-life verification problems.

A web-based interface to STEWord, along with a usage document and the benchmarks reported in this paper, is available for interested users. The authors may be contacted for more details about using the tool.

9 Conclusion

Increasing the level of abstraction from bits to words is a promising approach to scaling STE to large designs with wide datapaths. In this paper, we proposed a methodology and presented a tool to achieve this automatically. Our approach lends itself to a counterexample guided abstraction refinement (CEGAR) framework, where refinement corresponds to reducing the conservativeness in invalid-bit expressions, and to splitting existing atoms into finer bit-slices. We intend to build a CEGAR-style word-level STE tool as part of future work.

For our approach to work well, we rely on an effective SMT solver that does not resort to bit-blasting word-level expressions. Unfortunately, our experience with current state-of-the-art SMT solvers [10,2,4,15,9,11] has been somewhat disappointing. The primary problem has been the hit-and-miss behavior of the solvers. If a solver can successfully handle a WSTE problem (either by finding a satisfying assignment of $\neg\xi_{\mathsf{Ant},\mathsf{Cons},M}$, or by proving $\xi_{\mathsf{Ant},\mathsf{Cons},M}$ to be a tautology), it usually does so very quickly. However, if it cannot, the solvers simply never return and this behavior is far more common than what we would like to see in a practical verification system. We believe this is partly due to the lack of tuning of the heuristics to hardware examples, but it is clear that much more

work is needed on improving SMT solvers before WSTE is ready for prime time. A recent work [7] has proposed initial steps in this direction, drawing its inspiration from the difficulties encountered in using WSTE to prove properties of circuits with various multiplier implementations. Related to the above difficulty of solving SMT problems, is the question whether the style in which the RTL is written and properties specified affects the ability of the SMT solver to solve the resulting SMT problem. Clearly, if the design is described at the bit-level, it will not be efficient for a word level engine to reason about the resulting verification conditions. However, there are often many ways of capturing a design at a word level. Our experience so far indicates that there is a very big difference in the behavior of SMT solvers depending on which of these (superficially equally valid) versions the designer selected. We plan on pursuing this further. Finally, an intriguing possibility would be to create a two step verification system that would first, potentially with user guidance, create a word-level design from a bit-level design, and then apply WSTE on the resulting word-level design. The advantage with such an approach is that the "collapsing" of some bit-level constructs could be done locally and thus use very powerful, but low capacity, decision procedures.

Acknowledgements We thank Taly Hocherman and Dan Jacobi for their help and advice in designing a SystemVerilog symbolic simulator. We thank Ashutosh Kulkarni and Soumyajit Dey for their help in implementing and debugging STEWord. Rajkumar Gajavelly, Tanmay Haldankar, Dinesh Chhatani and Rakesh Mistry were supported by a research grant from Intel Corporation, which is gratefully acknowledged. Funding was provided by Intel Corporation as a research grant to IIT Bombay.

References

1. IEEE standard for SystemVerilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)*, pages 1–1315, Feb 2013.
2. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 171–177, 2011.
3. Robert Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In *Proceedings of the 22nd International Conference on Computer Aided Verification*, CAV'10, pages 24–40. Springer-Verlag, 2010.
4. R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *TACAS*, pages 174–177, 2009.
5. R. E. Bryant and C.-J. H. Seger. Formal verification of digital circuits using symbolic ternary system models. In *CAV*, pages 33–43, 1990.
6. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
7. S. Chakraborty, A. Gupta, and R. Jain. Matching multiplications in bit-vector formulas. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 131–150. Springer, 2017.
8. S. Chakraborty, Z. Khasidashvili, C.-J. H. Seger, R. Gajavelly, T. Haldankar, D. Chhatani, and R. Mistry. Word-level symbolic trajectory evaluation. In *Computer-Aided Verification (CAV)*, pages 128–143. Springer, 2015.
9. Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013.
10. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, 2008.

11. Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014.
12. Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2), 2006.
13. Niklas Eén and Niklas Sörensson. The minisat page. 2012.
14. E. A. Emerson. Temporal and modal logic. In *Hanbook of Theoretical Computer Science*, pages 995–1072. Elsevier, 1995.
15. Susmit Jha, Rhishikesh Limaye, and Sanjit A. Seshia. Beaver: Engineering an efficient SMT solver for bit-vector arithmetic. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 668–674, 2009.
16. P. Johannsen. Reducing bitvector satisfiability problems to scale down design sizes for rtl property checking. In *HLDVT*, pages 123–128, 2001.
17. R. B. Jones, J. W. O'Leary, C.-J. H. Seger, M. Aagaard, and T. F. Melham. Practical formal verification in microprocessor design. *IEEE Design & Test of Computers*, 18(4):16–25, 2001.
18. R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik. Replacing Testing with Formal Verification in Intel Core^{TM} i7 Processor Execution Engine Validation. In *CAV*, pages 414–429, 2009.
19. V. M. A. KiranKumar, A. Gupta, and R. Ghughal. Symbolic trajectory evaluation: The primary validation vehicle for next generation intel® processor graphics fpu. In *FMCAD*, pages 149–156, 2012.
20. Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.
21. H. S. Malvar, H. Li-Wei, and R. Cutler. High-quality linear interpolation for demosaicing of Bayer-patterned color images. In *ICASSP*, volume 3, pages 485–488, 2004.
22. J.-W. Roorda and K. Claessen. A new SAT-based algorithm for symbolic trajectory evaluation. In *CHARME*, pages 238–253, 2005.
23. C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995.
24. C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. F. Melham, M. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(9):1381–1405, 2005.
25. Fabio Somenzi. Cudd: Cu decision diagram package-release 2.5.0. *University of Colorado at Boulder*, 2012.
26. A. Stump, C. W. Barrett, and D. L. Dill. A decision procedure for an extensional theory of arrays. In *Logic in Computer Science (LICS)*, pages 29–37. IEEE Computer Society, 2001.