

Proof Slicing with Application to Model Checking Web Services

Hai Huang Wei-Tek Tsai Raymond Paul*

Dept. Computer Sci. & Eng., Arizona State University, Tempe, AZ, 85287-8809
{hai, wtsai}@asu.edu

*OSD NII, Department of Defense
raymond.paul@osd.mil

Abstract

Web Services emerge as a new paradigm for distributed computing. Model checking is an important verification method to ensure the trustworthiness of composite WS. Boolean abstraction and counterexample driven refinement are major techniques for model checking software and WS. In most of the literature, the refinement is governed by the precision of the abstraction. In this paper, we present an innovative technique to distribute the precision information among proof slices, which can be selectively reused by future proofs and hence improve the performance by reducing excessive invocations of theorem provers. Moreover, the reuse approach is flexible for virtually arbitrary future extension. Our theoretical framework subsumes several existing abstraction-based model checking techniques, e.g., lazy abstraction. Besides the correctness and termination proofs, we also conducted theoretical analysis on the performance of the proof slicing algorithm.

1 Introduction

Web Services (WS) promise flexible design and development of distributed computing appliances. Composite WS can be dynamically constructed or even reconfigured on WS components provided by various vendors [23]. The flexibility of Service-Oriented Architecture (SOA) imposes challenges on efficiently ensuring the trustworthiness of composite WS. A fully automated composite WS verification and validation (V&V) framework was proposed in previous research [18, 22], which utilized BLAST [16] to model check whether the composite WS satisfies specific properties. This paper proposes an innovative model checking technique, *proof slicing*, to substitute *lazy abstraction* in BLAST to verify composite WS efficiently.

The proposed V&V framework adopts OWL-S (Web Ontology Language for Web Services) process model [1]

to model composite WS. OWL-S process model, which is migrating to be more object-oriented, describes the control flow of the composite WS as well as the inputs, outputs, preconditions, and effects (IOPE) of each WS component. The V&V framework is shown in Figure 1, consisting of the following steps: (1) converting OWL-S model to *control flow automata* (CFA); (2) embedding to-be-checked properties into the CFA; (3) model checking and positive test case generation; and (4) negative test case generation. A CFA is essentially a control flow diagram [2], which consists of two types of edges denoting *assignment transition* and *assumption transition* respectively. Assignment transition means that when the control flows through the corresponding edge, a specific value is assigned to a variable. Assumption transition means that the control flows through the corresponding edge only when specific conditions are satisfied.

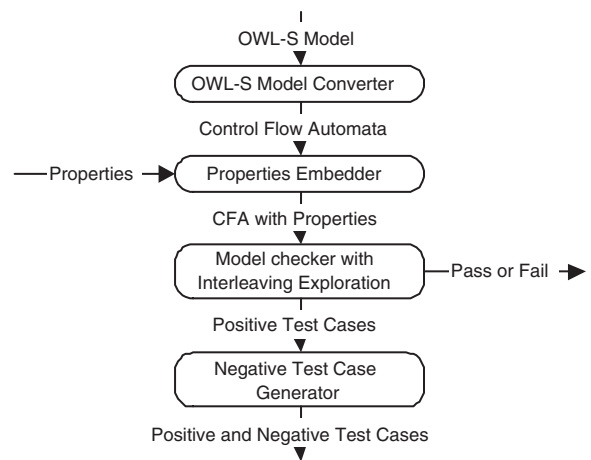


Figure 1. Automated V&V framework

The to-be-checked properties such as temporal properties are embedded into the CFA as **error** statements in step 2. The model checker exhausts all possible execution paths and checks whether any **error** statements are hit. If so, the

CFA violates the properties and the model checking fails. A *counterexample*, which is an execution path that hit an **error** statement, will be returned to indicate where and how the properties is violated.

Most structural constructs of OWL-S process model can be naturally converted to control flow automata, excepts the concurrency constructs *split + join*, which is handled by the interleaving explorer in step 3. The interleaving explorer exhausts all meaningful interleaving of atomic WS within concurrent threads and exploits partial order reduction to reduce the complexity. It enables a model checker designed for sequential programs to handle concurrent threads. Thus it is feasible to assume hereafter that all CFA are sequential. In step 4, positive and negative test cases are selectively generated by the techniques in [24].

Abstraction [4, 9, 15, 16] is one major model checking technique. Some abstraction-based model checkers, such as SLAM [3] and BLAST [16], apply directly to C source code. Particularly in lazy abstraction [16], states, which are conjunctions of predicates, are abstracted into *regions* to reduce the complexity of the state space. Lazy abstraction proceeds in three phases. In the forward exploration phase, a coarse *precision* of the regions is adopted until a counterexample is encountered. The process then switches to backward exploration phase, which back tracks the execution paths to figure out whether the counterexample is *infeasible*, i.e., whether it can be ruled out by improving the precision. If so, the process switches to refinement phase to refine the precision properly, usually through a theorem prover. Otherwise, the counterexample is real and returned to indicate a violation is detected. The process is usually named as *counterexample driven refinement*.

Calling theorem provers might be costly and degrade the performance of model checkers. We propose proof slicing technique to reduce the excessive calls to theorem provers. Instead of consulting a theorem prover to determine the proper refinement, proofs are broken into *proof slices* which can be selectively reused when proving the feasibility of future counterexamples. Roughly speaking, a proof slice is an execution path on a predicate (instead of a region, which is usually a logical combination of predicates). Proof slicing also reduces the complexity of backward exploration by concatenating to existing proof slices.

Proof slicing is future-oriented instead of past-oriented. Reusable proof slices will continue to expand when being picked up by future counterexamples, while others will cease to grow to reduce the time and space complexity. Proof slicing explores necessary and sufficient proof slices for future counterexamples, which is hard to achieve if the precision is determined by past counterexamples.

The structure of the rest of the paper is as follows. Section 2 is related work. Section 3 illustrates how proof slicing works with an example. Section 4 forms the theoretical

basis of our technique. Section 5 presents the algorithmic framework of proof slicing, which subsumes some other approaches [16, 15, 9]. Theoretical results, correctness proof, termination proof, and performance analysis, are also presented in Section 5. Section 6 presents some experimental data. Section 7 concludes this paper.

2 Related work

Model checking is developed for both hardware and software verification [10]. Abstraction-based approaches received significant attention recently for model checking source code [4, 9, 15, 16]. Ball, Podelski, and et al. introduced Boolean and Cartesian abstraction [4], which utilized a *three-valued model* [12]. Ball and Rajamani discussed counterexample driven refinement in [6] via *strongest post-conditions*. They also showed that temporal safety properties are checkable with abstraction-based approaches [5]. Henzinger, Jhala, and et al. introduced lazy abstraction [16] where the abstraction is nonuniform among execution traces. The *weakest preconditions* are used for backward reasoning in [16]. Lazy abstraction employs a theorem prover [21] to determine a small set of predicates for the proof. There are other SAT solvers available, e.g. [20]. The model used by lazy abstraction is control flow automata, which is essentially a control flow diagram [2]. Craig interpolation is integrated into lazy abstraction [15] to eliminate monotonicity of abstraction within a counterexample. Later lazy abstraction is enhanced for model checking concurrent execution of identical copies of threads [14], and automatic test cases generation [7]. Chaki, Clarke, and et al. proposed an approach to minimize the number of predicates for predicate abstraction [9]. Examples of abstraction-based model checkers that apply directly to source code are SLAM [3] and BLAST [16].

Proof slicing is inspired by the classical program slicing techniques [25, 17]. The technical basis of program slicing and proof slicing is fundamentally different from each other in the sense that: (1) program slicing is applied to source code, while proof slicing is applied to proofs; and (2) most program slicing are statically done based on *dependency graphs*, while proof slicing is dynamically tuned up to counterexamples during the model checking process.

Program slicing is a classical technique to reduce the complexity of control flow. Hatcliff, Corbett, and et al. established *bisimulation* relation between Java programs and their slices [13]. Recently program slicing was integrated into a formal verification framework, Bandera [11]. Brat and Visser combined static program slicing and model checking for software analysis [8].

3 Overview

This section presents an example to illustrate how proof slicing works. The WS V&V framework [18] can convert an OWL-S process model and embed to-be-checked properties to a C-like scenario code. Table 1 lists the sample code.

Table 1. Sample code

```

1:  a = 0; b = 0;
2:  if ( a != 0 || b != 0 )
3:      error;
4:  b = 1;
5:  if ( a != 0 )
6:      error;
7:  return;

```

The forward exploration starts from the entry node (statement) node 1 of the program, picks and follows a transition. Suppose it takes the path (1, 2, 3). At node 3, an **error** state is encountered. The path (1, 2, 3) is then a counterexample. The process switches to backward exploration to rule out the counterexample. Consider edge (2, 3). The necessary condition that allows the execution to reach node 3 is the condition $a \neq 0 \vee b \neq 0$. We split it to form the *heads* of two proof slices, $a \neq 0$ and $b \neq 0$. Keeping exploring backwards gives two proof slices $3 \xrightarrow{a \neq 0} 2 \xrightarrow{0 \neq 0} 1$, and $3 \xrightarrow{b \neq 0} 2 \xrightarrow{0 \neq 0} 1$. Note that in the last step $2 \xrightarrow{0 \neq 0} 1$, the assignment $a \leftarrow 0$ (or $b \leftarrow 0$) results in the predicate $0 \neq 0$ by substituting a (or b) with 0 in the predicate $a \neq 0$ (or $b \neq 0$). Because $0 \neq 0 \vee 0 \neq 0$ is a contradiction, the counterexample at node 3 is ruled out.

When proceeding to node 6 following the path (1, 2, 4, 5, 6), the assumption transition on edge (5, 6) is $a \neq 0$. Backward exploration will pick up the existing proof slice $6 \xrightarrow{a \neq 0} 5 \xrightarrow{0 \neq 0} 4$ when arrives at node 2. The proof slice expands to be $6 \xrightarrow{a \neq 0} 5 \xrightarrow{a \neq 0} 4 \xrightarrow{a \neq 0} 2 \xrightarrow{0 \neq 0} 1$ and rules out counterexample at node 6. The other existing proof slice $3 \xrightarrow{b \neq 0} 2 \xrightarrow{0 \neq 0} 1$ ceases to expand beyond node 2 because current counterexample does not involve the proof slice headed by $b \neq 0$.

Lazy abstraction [16] operates in a different way. At node 3, it will consult a theorem prover to compute a small (ideally minimum) set of predicates that is sufficient to rule out the counterexample at node 3. In this particular example, the minimum set of predicates is $\{a \neq 0, b \neq 0\}$. The set defines the precision of the abstraction after node 2 and further process will keep the precision. Therefore, when the forward exploration proceeds to node 4, it needs to take into account of the assignment transition $b \leftarrow 1$ to keep track of the predicate $b \neq 0$.

Compared to lazy abstraction, proof slicing saves the cost of calling a theorem prover at node 3 to determine a small set of predicates. Computing a minimum precision is not necessary because the counterexample at node 6 does not require such information to determine which proof slice to pick up. Proof slicing also saves the computation that forward exploration spends at node 5 because the counterexample at node 6 does not involve predicate $b \neq 0$.

4 Abstract trace systems

4.1 Trace systems

This section introduces the formal notations of *trace systems*. The formalism of the *labeled transition systems* (LTS) resembles that in [16]. Distinctions are: (1) the concept of a CFA hosting a LTS, (2) more general nondeterministic CFA, and (3) more general trace systems rather than the reachability trees in [16].

Let S be a set of states, Σ be a set of *labels* representing program statements, $\rightarrow \subseteq S \times \Sigma \times S$ is a *labeled transition relation* that defines legal transitions. Define a LTS \mathcal{S} as the tuple (S, Σ, \rightarrow) . A transition $(s, l, s') \in \rightarrow$ is denoted as $s \xrightarrow{l} s'$.

A LTS \mathcal{S} hosted by a CFA \mathcal{C} is defined as the tuple $(\mathcal{S}, V, E, h_V, h_E)$, where (V, E) is a directed (multi-)graph representing the CFA, $h_V : V \rightarrow 2^S$ is a total *node hosting* function, and $h_E : E \rightarrow \Sigma$ is a total *edge labeling* function. By including program counter pc in the state, we restrict that a state $s \in S$ belongs to at most one node $v \in V$. The unique node with no incoming edge is the *entry* point and nodes with no outgoing edge are *exit* points.

The transitions on the edges are categorized into two classes: the *assignment* and the *assumption*. An assignment is $l : x = e$, where x is a variable and e is an expression. An assumption is $l : \mathbf{assume} \ \psi$, where ψ is a formula. The meaning of assumption is that the transition is taken only when ψ satisfies. We allow multiple outgoing assignment edges from one node. If there are multiple outgoing assumption edges from one node, we allow $\psi \wedge \psi' \neq \perp$, where ψ and ψ' are the formulae associated to any two assumption edges. That is, the CFA is *nondeterministic*.

To abstract states in S , define $R \supset S$ as a set of regions. Define $\llbracket \cdot \rrbracket : R \rightarrow 2^S$ as a total *extension* function that abstracts a set of states as a region $r \in R$. $S \subset R$ is explained by $\forall s \in S, \llbracket s \rrbracket = \{s\}$. Let $\perp \in R$ such that $\llbracket \perp \rrbracket = \emptyset$, and $\top \in R$ denoting all possible states hosted by a node. Introduce binary relations on regions $r \equiv r'$ iff $\llbracket r \rrbracket = \llbracket r' \rrbracket$, and $r \sqsubseteq r'$ if $\llbracket r \rrbracket \subseteq \llbracket r' \rrbracket$. Introduce binary operators on regions $\llbracket r \sqcap r' \rrbracket = \llbracket r \rrbracket \cap \llbracket r' \rrbracket$, and $\llbracket r \sqcup r' \rrbracket = \llbracket r \rrbracket \cup \llbracket r' \rrbracket$. To track the change of regions along transitions, define $pre : R \times \Sigma \rightarrow R$ by $\llbracket pre(r, l) \rrbracket = \{s' \in S \mid \exists s \in \llbracket r \rrbracket, s' \xrightarrow{l} s\}$, and $post :$

$R \times \Sigma \rightarrow R$ by $\llbracket \text{post}(r, l) \rrbracket = \{s' \in S \mid \exists s \in \llbracket r \rrbracket, s \xrightarrow{l} s'\}$. The tuple $\mathcal{R} = (\mathcal{C}, R, \text{pre}, \text{post}, \widehat{\text{post}}, \llbracket \cdot \rrbracket, \sqcup, \sqcap, \equiv, \sqsubseteq)$ is defined as an *abstract region structure* over a CFA, where $\widehat{\text{post}}$ is required to satisfy $\text{post}(r, l) \sqsubseteq \widehat{\text{post}}(r', l)$, for all $r \equiv r'$.

A *trace* is a sequence $\pi = r_0 l_1 r_1 \dots l_n r_n$ such that $r_i = \widehat{\text{post}}(r_{i-1}, l_i)$, where $r_i \in R$, $i \in [n] \cup \{0\}$ are regions, $l_i \in \Sigma$, $i \in [n]$ are labels, and n is the *length*. A trace is a *concrete trace* if $\forall i \in [n] \cup \{0\}, r_i \in S$. Let Π be the collection of all traces generated by \mathcal{C} over \mathcal{R} . Define $\pi^{[i,j]}$ is the inclusive subsequence of π from r_i to r_j . Define $\sigma = l_1 \dots l_n$ as the *generating transition sequence* of a trace π , and $\sigma^{[i,j]}$ as the generating transition sequence of $\pi^{[i,j]}$. Introduce binary relation on traces $\pi \sqsubseteq \pi'$, if $r_i \sqsubseteq r'_i$, $i \in [n] \cup \{0\}$ and $\sigma = \sigma'$. A *trace system* is defined as the tuple $\mathcal{T} = (\mathcal{R}, \Pi, \sqsubseteq)$.

Some states are designated as *error states*. A trace hits an error state or a region containing an error state is a counterexample. For a to-be-checked property φ , corresponding **error** statements are embedded into the CFA. We denote $\pi \models \varphi$ if π hits no error states or error regions. If all concrete traces hit no error state, we conclude that property φ holds universally, denoted as $\mathcal{C} \models \varphi$.

We say π' *preserves concrete traces* of π if for all concrete trace $\pi'' \sqsubseteq \pi$, $\pi'' \sqsubseteq \pi'$. Preservation of concrete traces guarantees the soundness of trace systems as stated by Theorem 1.

Theorem 1. $\mathcal{C} \models \varphi$ if $\forall \pi \in \Pi$, either $\pi \models \varphi$, or $\exists \pi' \in \Pi$ such that $\pi' \models \varphi$ and π' preserve concrete traces of π .

Lemma 1 states a sufficient condition for preserving concrete traces.

Lemma 1. Let π and π' be two traces with the same generating transition sequence, r_k and r'_k be their k -th regions, respectively. If $\pi^{[0,k-1]} = \pi'^{[0,k-1]}$, $r_k \sqsubseteq r'_k$, then π' preserves concrete traces of π .

4.2 Predicate abstraction

In what follows we present a *four-valued model* over predicates for representing regions. Let $\Gamma = \{p_1, \dots, p_n\}$ be a set of predicates. Consider four distinct values 0, 1, *, and \times . Let $W = \{0, 1, *, \times\}^n$, and $w = \langle w_1, \dots, w_n \rangle \in W$ is a *quadrivector*. A quadrivector represents an *atomic region* as $a = \bigwedge w_i \cdot p_i$, where $0 \cdot p_i = \neg p_i$, $1 \cdot p_i = p_i$, and $* \cdot p_i = \times \cdot p_i = \top$. Note that \top effectively remove the presence of p_i from a . A region is the disjunction of atomic regions, $r = \bigvee a_j$. Intuitively, value * serves as “don’t care” value; that is, the value could be either 0 or 1. Value \times serve as the *precision mask* that prohibits $\widehat{\text{post}}$ from resolving the precision carried by the corresponding predicate. Define \sqsubseteq on $\{0, 1, *, \times\}$ by $0 \sqsubseteq *, 1 \sqsubseteq *, * \sqsubseteq \times$, and $\times \sqsubseteq *$. The \sqsubseteq on $\{0, 1, *, \times\}$ defines \sqsubseteq on atomic regions and regions.

We compute *pre* and hence $\widehat{\text{post}}$ by the *weakest precondition* [16, 9]. For a predicate p_i , the weakest precondition $wp(p_i, l)$ is defined as the weakest formula, whose truth before a transition l entails p_i after l . For an atomic region $a = \bigwedge p_i$, $wp(a, l) = \bigwedge w_i \cdot wp(p_i, l)$, and for a region $r = \bigvee a_j$, $wp(r, l) = \bigvee wp(a_j, l)$. Define $pre(r, l) = wp(r, l)$.

Operator $\widehat{\text{post}}$ is the inverse of *pre* with exceptions that some predicates may be masked out by precision masks. A *precision vector* is $\omega = \langle \omega_1, \dots, \omega_n \rangle \in \Omega = \{*, \times\}^n$. Define \oplus to be a binary operator on $\{*, \times\} \times \{0, 1, *, \times\}$ by $\forall \alpha \in \{0, 1, *, \times\}, * \oplus \alpha = \alpha$, and $\times \oplus \alpha = \times$. For a given precision vector ω , define $\widehat{\text{post}}(a, l, \omega) = \bigwedge (\omega_i \oplus w'_i) \cdot p_i$, where $a = pre(\bigwedge w'_i \cdot p_i, l)$. Define $\widehat{\text{post}}(r, l, \{\omega_j\}) = \bigvee \widehat{\text{post}}(a_j, l, \omega_j)$, where $r = \bigvee a_j$, and $\{\omega_j\}$ is a set of precision vectors. This definition allows nonuniform precision within a region.

Precision of regions changes along a trace: either the precision can be set arbitrarily for the purpose of the model checking algorithms (detailed in Section 5.2); or transitions may impose precision switch naturally. Recall that we have assignment transitions and assumption transitions. For an assignment $l : x = e$, $wp(p_i, l) = p_i[e/x]$, where $p_i[e/x]$ denotes p_i with all occurrences of x replaced by e . Hence $wp(p_i, l)$ is still a predicate. Assignments involving pointers are handled by Morris’ general axiom of assignment [19]. For an assumption $l : \mathbf{assume} \psi$, $wp(p_i, l) = p_i \wedge \psi$. By splitting **assume** $\psi \vee \psi'$ to two alternative transitions **assume** ψ or **assume** ψ' and allowing multiple edges between two nodes in the CFA, we can safely assume that $wp(p_i, l)$ is an atomic region. To summarize, three situations may occur on a predicate after the computation of wp :

- (1) p_i is replaced by another predicate q , as in assignment $l : x = e$ where x occurs in p_i , or
- (2) p_i preserves its presence, as either in assignment $l : x = e$ where x does not occur in p_i , or in assumptions, or
- (3) q is newly introduced, as in assumption $l : \mathbf{assume} \psi$ where q forms a part of ψ , and $q \neq p_i$.

The three situations affect the precision of *pre*. With (1) we mask out p_i by a \times and introduce q by a $*$; with (2) we preserve p_i by a $*$; and with (3) we introduce p_i by a $*$. Consequently, the precision of $\widehat{\text{post}}$ is defined by the inverse of *pre* over the selected predicates, which correspond to the selected proof slices that compose the proof.

The precision switch achieves the same non-monotonicity within a counterexample as in [15]. The novel four-valued model also enables nonuniform precision within a region, e.g., each atomic region preserves its own precision.

5 Proof slicing

5.1 Proof slices

Without loss of generality, we assume that the *initial region*, the region embedded to the entry point of the CFA, is \top (justification is detailed in Section 5.3). Consider a trace π of length n that hits an error region ε . Let σ be the generating transition sequence of π . A *proof* of the infeasibility of the counterexample is the sequence $\theta = \top, \neg pre(\varepsilon, \sigma^{[0,n]}), \dots, \neg pre(\varepsilon, \sigma^{[n-1,n]}), \neg \varepsilon$, where $\neg pre(\varepsilon, \sigma^{[0,n]})$ is a tautology, or equivalently, $pre(\varepsilon, \sigma^{[k,n]})$ is a contradiction. For any two consecutive formulae ψ and ψ' in the sequence θ , ψ entails ψ' following the corresponding transition. The counterexample ending by ε is infeasible iff there exists a proof.

Define a *literal* q as a predicate or its negation, and include \top and \perp as special literals. A *proof slice* is a backward sequence $\rho = q_n l_n q_{n-1} \dots q_0$, where $q_i, i \in [n] \cup \{0\}$ are literals and $l_i, i \in [n]$ are transitions. We name q_n as the *head* of the proof slice, and q_0 as the *tail* of the proof slice. A proof slice is *infeasible* if its tail is \perp . We expect to have $q_{i-1} = pre(q_i, l_i)$, for $i \in [n]$, analogous to proofs. Recall the three precision switch conditions in Section 4. Equation $q_{i-1} = pre(q_i, l_i)$ holds when l_i is an assignment, or an assumption that contains only literal q_i . When l_i is an assumption **assume** ψ that contains literals other than q_i , $pre(q_i, l_i) = q_i \wedge \psi$. In this case, it *spawns* new proof slices headed by literals in ψ . Specifically, if ψ contains literals q_k^1, \dots, q_k^m , then proof slices $\rho^j = q_k^j l_k q_{k-1}^j \dots q_0^j, j \in [m]$ are spawned. We name each ρ^j as a *spawnnee*, and the original proof slice ρ as the *spawner*. Define a binary relation *spawn* on proof slices, denoted by \uparrow , as $\rho \uparrow \rho'$ if ρ spawns ρ' . We use $\rho \uparrow_k \rho'$ to denote that ρ spawns ρ' at a location $j \leq k$.

A *proof formula* is constructed over proof slices. Initially the proof formula is constructed by replacing the literals in the error region formula ε with the corresponding proof slices. Assignments do not change the proof formula. For each assumption, first compute a sub proof formula by replacing the literals in the formula of the assumption with the corresponding proof slices, and then conjunct the sub proof formula with the original one. For example, suppose the error region is $a \neq 0 \vee b \neq 0$. Initially the proof formula is $\rho_{a \neq 0} \vee \rho_{b \neq 0}$. After passing an assignment $a = 0$, the proof formula remains the same as $\rho_{a \neq 0} \vee \rho_{b \neq 0}$, though the feasibility of the proof formula does change as within the proof slice $\rho_{a \neq 0}$, the literal changes to $0 \neq 0$, i.e., \perp . After passing an assumption **assume** $c \neq 0$, the proof formula changes to be $(\rho_{a \neq 0} \vee \rho_{b \neq 0}) \wedge \rho_{c \neq 0}$. The feasibility of the proof formula can be checked at any location k along the trace by plugging in the k -th literal of every proof slice to the proof formula and evaluate the proof formula. The

checking is straightforward when expressions are linear. It is believed that most expressions in many software production systems are linear [26].

Lemma 2. *Let f be a proof formula of error region ε . If at location k , f evaluates to be \perp , then ε is infeasible, and for all $l \leq k$, f evaluates to be \perp . If at location 0, the entry point, f evaluates not to be \perp , then ε is feasible.*

The proof of Lemma 2 is straightforward. It guarantees the correctness of proof slicing.

5.2 Configurable algorithm

We present a configurable algorithm for model checking a CFA as an open framework. The configuration leaves open the choices of possible future extensions. Correctness and termination are proved on the framework and are hence ensured for all algorithms that fit in.

In Algorithm 1, the collection Π stores all the traces Algorithm 1 explores, Θ collects all the proof slices, Θ_ε stores the proof slice group for current error region ε , Θ_f stores the proof slices sufficient to prove the infeasibility of the proof formula f . The operator $last(\cdot)$ returns the last entry of a sequence. At step 3, it is trivial to test whether a trace π reaches any end points. A trace $\pi = r_0 l_1 r_1 \dots r_n$ contains a *self loop* if $\exists k < n$ such that $r_k \equiv r_n$, and the transition sub-sequences match. It follows directly that any expansion of a self loop trace by \widetilde{post} will not reach any new regions.

There are three configurable operators in Algorithm 1, \widetilde{post} , \widetilde{eval} , and \widetilde{pick} . We impose the following *correctness requirements*:

$$\forall r \in R, \forall l \in \Sigma, \widetilde{post}(r, l) \sqsubseteq \widetilde{post}(r, l) \quad (1)$$

$$\widetilde{eval}(f) \equiv \perp \text{ iff } f \text{ evaluates to be } \perp \text{ at location } 0 \quad (2)$$

$$f[(\Theta_\varepsilon - \widetilde{pick}(\Theta_\varepsilon))/\top] \text{ evaluates to be } \perp \text{ at location } 0 \quad (3)$$

where $f[(\Theta_\varepsilon - \widetilde{pick}(\Theta_\varepsilon))/\top]$ is to replace all the proof slices in $\Theta_\varepsilon - \widetilde{pick}(\Theta_\varepsilon)$ by \top in the proof formula f . Formula 1 guarantees that the traces generated by \widetilde{post} preserves concrete traces. Formula 2 guarantees the correctness of \widetilde{eval} . Formula 3 ensures that \widetilde{pick} picks up sufficiently precision to rule out the infeasible counterexample.

Different configurations on \widetilde{post} , \widetilde{eval} , and \widetilde{pick} exhibits different performance. We suggest the *lazy slicing* configurations: $\widetilde{post} \equiv \top$, \widetilde{eval} is the evaluation of f at location 0, and $\widetilde{pick} \equiv I$, the identity operator that returns all proof slices in Θ_ε . Note that the configuration satisfies the correctness requirements.

Lazy abstraction [16] fits in the algorithmic framework w.r.t. the configuration $\widetilde{post} \equiv \widetilde{post}$; \widetilde{eval} starts at location n and stops at the first location k such that f evaluates to

Algorithm 1: Nondeterministic algorithm

```
1: Let  $\pi \leftarrow v_0$ , and  $\Pi \leftarrow \{\pi\}$ , where  $v_0$  is the start point
   of the CFA
2: Let  $\Theta \leftarrow \emptyset$ 
3: while  $\exists \pi \in \Pi$  such that  $\pi$  does not reach any end
   point, or  $\pi$  contains no self loop do
4:    $\Pi \leftarrow \Pi - \{\pi\}$ 
5:   foreach  $l$  do
6:     if  $\varepsilon \sqcap \widetilde{post}(last(\pi), l) \equiv \perp$ , where  $\varepsilon$  is error
       state then
7:       Let  $\pi' \leftarrow \pi, l, \widetilde{post}(last(\pi), l)$ , and
          $\Pi \leftarrow \Pi \cup \{\pi'\}$ 
8:     else
9:       #  $\pi, l, \widetilde{post}(last(\pi), l)$  is a counterexample
10:      Let  $\Theta_\varepsilon \leftarrow \emptyset$ 
11:      foreach literal  $q$  in  $\varepsilon$  do
12:        Generate  $\rho_q$  as the proof slice headed
          by  $q$ , reuse proof slices in  $\Theta$  when
          possible
13:        Let  $\Theta_\varepsilon \leftarrow \Theta_\varepsilon \cup \{\rho_q\} \cup \{\text{all proof}
          \text{ slices } \rho_q \text{ spawns}\}$ 
14:      end
15:      Construct proof formula  $f$ 
16:      Let  $\Theta \leftarrow \Theta \cup \{\Theta_\varepsilon\}$ 
17:      # Form a proof
18:      if  $\widetilde{eval}(f) \equiv \perp$  then
19:        Let  $\Theta_f \leftarrow \widetilde{pick}(\Theta_\varepsilon)$ 
20:        Defines  $\widetilde{post}$  with the precision of  $\Theta_f$ 
21:        Refine  $\pi$  by  $\widetilde{post}$ ; let  $\pi'$  be the refined
          trace; let  $\Pi \leftarrow \Pi \cup \{\pi'\}$ 
22:      else
23:        # The counterexample  $\pi, l, \varepsilon$  is feasible
24:        return  $\pi, l, \varepsilon$  is a feasible
          counterexample
25:      end
26:    end
27:  end
28: return  $\emptyset$ 
```

\perp , and \widetilde{pick} is uniformly determined by a theorem prover and hence the precision is uniform. In the Craig interpolation [15], \widetilde{pick} is still determined by a theorem prover, but the precision changes along the trace. In the minimum predicates [9], the \widetilde{pick} compute a minimum set of proof slices over all early counterexamples.

Compared with existing approaches, lazy slicing omits the calls to theorem provers in the \widetilde{pick} and gains significant performance boost. Other performance boosts are the triv-

ial \widetilde{post} operator, and the single evaluation of proof formula f only at the entry point. The storage usage deserves more explanation since Algorithm 1 keeps all the proof slices in Θ . But since a proof slice used for different counterexamples and traces is stored only once in Θ , it also gains some storage save. In other approaches, such as lazy abstraction, each path in the reachability tree stores regions separately and waste storage if two regions have only minor difference.

5.3 Concatenation and inclusion of slices

First we justify the assumption that the initial region is \top . For any initial region r_0 , assume that $r_0 = \bigvee a_j^0$, where a_j^0 is atomic region. For each $a_j^0 = \bigwedge p_{jk}^0$, construct a path τ_j^0 of consecutive assignments $p_{jk}^0 = true$. It follows directly that after the execution of the path τ_j^0 , the end region is a_j^0 . Then we connect all the start point of each τ_j^0 to a new created node v'_0 and label all edges with assumption transition \top . We connect all the end point of each τ_j^0 to the start point v_0 of the original CFA and label all edges with assumption transition \top . Denote the constructed CFA \mathcal{C}' . Note that \mathcal{C}' is nondeterministic. It is straightforward to show that the region at v_0 is r_0 . Note the construction takes time linear to the complexity of r_0 , and increases the size of the CFA by the complexity of r_0 .

The concatenation and inclusion Lemma, Lemma 3, forms the basis of the correctness of proof slice reuse.

Lemma 3. *Let Θ be the collection of proof slices computed by Algorithm 1, and q_i be a literal, where i is the location index along the trace. Consider the generation of Θ_ε . At any time, if the generation of proof slice ρ reaches q_i such that $\exists \rho' \in \Theta$, ρ' starts with q_i , then (1) the suffix of ρ starting from q_i is equal to ρ' , (2) $\forall \rho''$, such that $\rho \uparrow_i \rho''$, $\rho'' \in \Theta$, and (3) $\forall \rho''$ such that $\rho' \uparrow \rho''$, $\rho'' \in \Theta_\varepsilon$.*

The proof is straightforward as the generation is independent from the progress of Algorithm 1, hence the generations starting with the same q_i are identical. Intuitively, Lemma 3 says that if the generation of a proof slice concatenate to an early proof slice, then we can stop the generation, append early proof slice and include in Θ_ε all proof slices it spawns. At step 14, if the early proof slice spawns no proof slices, then the proof formula keeps the same form. Otherwise, all the proof formulae corresponding to the spawnees are conjuncted to the original one.

5.4 Correctness and termination

In this section, we prove the correctness, termination, and performance boost.

Theorem 2. *Let φ be a property, and \mathcal{C} be a CFA. If Algorithm 1 terminates, then $\mathcal{C} \models \varphi$ iff Algorithm 1 returns \emptyset .*

Proof. (Sketch) If without proof slice reuse at step 11, for all concrete trace π , $\exists \pi'$ generated by Algorithm 1 such that $\pi \sqsubseteq \pi'$. This is because we exhaust all paths in the CFA, and both \widehat{post} and \widetilde{post} preserve concrete traces, due to Lemma 1. Algorithm 1 returns \emptyset iff $\forall \pi$ generated by it, $\pi \models \varphi$; and if Algorithm 1 returns a counterexample π , then $\exists \pi'$ such that π' is a concrete trace and $\pi' \sqsubseteq \pi$, due to Lemma 2. Proof slice reuse generates exactly the same Θ_ε , due to Lemma 3. \square

We follow the termination proof in [16] to formulate the termination condition. Consider $s \in S$ and $\sigma \in \Sigma^*$, we write $s \xrightarrow{\sigma}$ if $\exists s' \in S$ such that $s \xrightarrow{\sigma} s'$. Define binary relation *trace equivalent* between state s and s' if $\forall \sigma, s \xrightarrow{\sigma}$ iff $s' \xrightarrow{\sigma}$. The LTS has a *finite trace equivalence* if the trace equivalent relation has a finite index. An abstract region structure \mathcal{R} satisfies the *ascending chain condition* if there is no infinite sequence $r_0 \sqsubset r_1 \sqsubset \dots$, where $r \sqsubset r'$ iff $r \sqsubseteq r'$ and $r \neq r'$.

Theorem 3. *Assume S has a finite trace equivalence and \mathcal{R} satisfies the ascending chain condition, then for any initial region and any property, Algorithm 1 terminates if \widehat{post} , \widetilde{eval} , and \widetilde{pick} terminate.*

As discussed in Section 5.1, Algorithm 1 reduces to Algorithm LazyAbstraction [16] under a certain configuration, and hence terminates. Because \widehat{post} , \widetilde{eval} , and \widetilde{pick} does not affect the termination, Algorithm 1 under general configuration terminates. The result helps to establish a performance upper bound of Algorithm 1.

Theorem 4. *Assume lazy slicing configuration is in use and there exists constant γ such that $\forall r, r' \in R$ and $\forall l, l' \in \Sigma$, the cost of computing $pre(r, l)$ is upper bounded by γ times of that of $pre(r', l')$. For any terminative execution of Algorithm 1 with cost T , there exists a terminative execution of Algorithm LazyAbstraction with cost T' on the same CFA and properties, such that $T = O(n \cdot T')$, where n is the length of the longest trace generated by the CFA.*

The only extra cost of Algorithm 1 is that for each counterexample, it may backward search to the start point of the CFA to generate all proof slices. Let m be the number of steps of backward search in Algorithm 1, m' be that in Algorithm LazyAbstraction. It is trivial to show that $m \leq n$ and $m' \geq 1$. Hence the extra cost is bounded by a factor of $\gamma \times n$, which implies $T = O(n \cdot T')$. The assumption of the constant factor bound among costs of pre is feasible due to the simplicity of wp . The performance boost of proof slicing resides in the elimination of excessive calls to theorem-provers.

We remark that in general, the problem whether a given safety property can be witnessed by a finite set of support predicates is undecidable as proved in Theorem 4 in [16].

Table 2. Experimental data

	exp1	exp2	exp3	exp4	tut2.c
LOC	14	63	26	25	79
Wall Clock Time (s)					
Blade	≤ 1	≤ 1	≤ 1	≤ 1	≤ 1
BLAST	1	2	≤ 1	1	≤ 1
Craig	≤ 1	1	1	2	≤ 1
Number of Calls to Theorem Prover					
Blade	3	160	2	2	7
BLAST	23	206	46	123	23
Craig	23	199	50	123	23
Calls Weighted by Complexity of Formulae					
Blade	3	594	3	4	17
BLAST	51	1701	90	1092	55
Craig	51	998	102	992	55

6 Experimental data

We have developed a preliminary model checker “Blade” to implement Algorithm 1. This section presents experiments to illustrate its performance. Blade employs Simplify, the same theorem prover employed by BLAST. The number of calls to the theorem prover is chosen to indicate the runtime, because usually theorem prover dominates the runtime of abstraction-based model checker [3]. The complexity of the formulae fed into the theorem prover affects its runtime seriously. In the worst case, the runtime of theorem proving could be exponential to the number of the predicates in a formula, for theorem proving subsumes the SAT (Satisfiability) problem. Therefore, each call to the theorem prover is weighted by the number of predicates in the formula to roughly account for its complexity.

The same experiments are conducted with BLAST. The communication between BLAST and Simplify is hijacked to record the number of calls as well as the complexity of each call. There are four types of communication to Simplify, pure predicates, BG PUSH, BG POP, and IMPLIES. We only count IMPLIES. All the experiments are executed with the option to cache theorem prover calls. Note that the design goal of BLAST is not focused on reducing the number of calls to the theorem prover.

All the experiments are executed on a Linux box with PIII 860M CPU and 1G memory. Experimental data are listed in Table 2. The exp1, exp2, exp3, and exp4 are C-like code translated from OWL-S models, and tut2.c is from the BLAST package.

The experimental data shows that generally speaking, Blade has a far less number of calls to the theorem prover than BLAST does, with or without Craig interpolation.

7 Conclusion

A novel model checking technique proof slicing is proposed to improve the efficiency of the model checker in the automated V&V framework for ensuring the trustworthiness of composite WS. Proof slicing enables future counterexamples to selectively reuse proof slices of early counterexamples, and gain performance boost. We proved the correctness and termination for proof slicing. The algorithmic framework is open for almost arbitrary future extension.

References

- [1] OWL-S: Semantic markup for Web services. <http://www.daml.org/services/owl-s/1.1B/owl-s.pdf>, 2004.
- [2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [3] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. Slam and static driver verifier: technology transfer of formal methods inside microsoft. Technical Report MSR-TR-2004-8, Microsoft Research, Redmond, 2004.
- [4] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 268–283, 2001.
- [5] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, pages 103–122, 2001.
- [6] T. Ball and S. K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research, Redmond, 2002.
- [7] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proceedings for the 26th International Conference on Software Engineering*, pages 326–335, 2004.
- [8] G. Brat and W. Visser. Combining static analysis and model checking for software analysis. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pages 262–271, 2001.
- [9] S. Chaki, E. Clarke, A. Groce, and O. Strichman. Predicate abstraction with minimum predicates. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–244, 2004.
- [10] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2002.
- [11] J. C. Corbett, M. B. Swyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–449, 2000.
- [12] P. Godefroid and R. Jagadeesan. On the expressiveness of 3-valued modes. In *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 206–222, 2002.
- [13] J. Hatcliff, J. C. Corbett, M. B. Swyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with jvm concurrency primitives. In *Proceedings of the 6th International Symposium on Static Analysis*, pages 1–18, 1999.
- [14] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 1–13, 2004.
- [15] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–244, 2004.
- [16] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, pages 58–70, 2002.
- [17] H. Huang, W. Tsai, and S. Subramanian. Generalized program slicing for software maintenance. In *The 8th International Conference on Software Engineering and Knowledge Engineering*, pages 261–268, 1996.
- [18] H. Huang, W. T. Tsai, R. Paul, and Y. Chen. Automated model checking and testing for composite Web Services. In *Proceedings of the 8th IEEE International Symposium on Object-oriented Real-time distributed Computing*, 2005. To appear.
- [19] J. M. Morris. A general axiom of assignment. In *Theoretical Foundations of Programming Methodology*, pages 25–34, 1982.
- [20] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th Conference on Design Automation*, pages 530–535, 2001.
- [21] K. Rustan, M. Leino, and G. Nelson. An extended static checker for modula-3. In K. Koskimies, editor, *Compiler Construction: 7th International Conference*, volume 1383 of *LNCS*, pages 302–305. Springer, 1998.
- [22] W. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang. Extending WSDL to facilitate Web Services testing. In *The 7th IEEE International Symposium on High-Assurance Systems Engineering*, pages 171–172, 2002.
- [23] W. Tsai, W. Song, R. Paul, Z. Cao, and H. Hunag. Services-oriented dynamic reconfiguration framework for dependable distributed computing. In *Proceedings of the 28th Annual International Computer Software and Applications Conference*, pages 554–559, 2004.
- [24] W. Tsai, X. Wei, Y. Chen, B. Xiao, R. Paul, and H. Huang. Developing and assuring trustworthy Web Services. In *Proceedings of the 7th International Symposium on Autonomous Decentralized Systems*, 2005. To appear.
- [25] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, 1984.
- [26] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Trans. Softw. Eng.*, 6:247–257, 1980.