Computer Sciences Department

Abstract Error Projection

Akash Lal
Nicholas Kidd
Thomas Reps
Tayssir Touili

Technical Report #1579

September 2006

UNIVERSITY OF WISCONSIN MADISON

Abstract Error Projection

Akash Lal[1], Nicholas Kidd[1], Thomas Reps[1], and Tayssir Touili[2]

[1] University of Wisconsin, Madison, Wisconsin, USA. {akash,kidd,reps}@cs.wisc.edu
[2] LIAFA, CNRS & University of Paris 7, Paris, France. touili@liafa.jussieu.fr

Abstract. To improve the reporting of results from model checking and program-analysis systems, we introduce the notion of an error projection and annotated error projection. An *error projection* is a set of program nodes N such that for each node $n \in N$ there exists an (abstract) error path from the program entry s through n to a specified target node t. An *annotated error projection* associates with each node n in the error projection an (abstract) counterexample that validates the error along with an *abstract store*, whose presence at n induces the error. We present novel algorithms for computing (annotated) error projections and discuss additional applications for these algorithms. Our experiments show that error projections can be computed efficiently.

1 Introduction

Model checking is a popular technique for program verification. From 10,000 feet, a model checker 1) extracts a program model using a finite abstraction; 2) performs reachability on the model; and 3) reports back the results to the user, usually in the form of a counterexample on a failed run. This technique has been shown useful both for finding program errors and for verifying certain properties of programs. It has been implemented in a number of model checkers including SLAM [1], BLAST [9], and MAGIC [4]. Our goal is to extend the capabilities of model checkers to return all (or at least more) of the relevant information found during the reachability check.

We accomplish this by reporting *error projections*. An error projection is a set of original program nodes N such that for each node $n \in N$, there exists an abstract error path in the program model from an entry point that passes through an abstract node corresponding to n. An error projection is *sound* with respect to the program's abstract model. This is an important feature of error projections. That is, an error projection describes *all of the nodes* that are members of paths that lead to a specified error point in the model. This allows an automated program-analysis tool or human debugger to focus their efforts on only the nodes in the error projection: every node not in the error projection is correct (with respect to the property being verified). One can view an error projection as dividing the program into two parts, one correct and one incorrect with respect to the verification property.

Annotated error projections are an extension to error projections. An *annotated error projection* adds to each node n in the error projection two annotations: 1) A counterexample that passes through n; 2) a set of abstract stores that describes the conditions necessary at n for the program to fail. The goal is to give back to the user—either an automated tool or human debugger—more of the information discovered during the model-checking process.

Note that we are not saying counterexamples are useless for (abstract) debugging—merely that, in some cases, the user might need more information

than can be provided by a few counterexamples. For example, if a programmer is trying to debug a large piece of software, most of which is unfamiliar code, a few counterexamples may not be enough to indicate the parts of the code that need to be considered when fixing the bug. The algorithms that we present provide a programmer the ability to ask for a counterexample that passes through a specific program node (using annotated error projections), rather than be forced to consider the counterexamples chosen (arbitrarily) by the tool.

From a theoretical standpoint, an error projection solves a combination of forward and backward analyses. The forward analysis computes the set of program nodes and program states that are reachable from program entry; the backward analysis computes the set of program nodes and states that can reach an error at certain pre-specified nodes. The error projection is an intersection of sets computed by these analyses. In this paper, we give an algorithm for computing interprocedural error projections using a novel automata-theoretic construction. The algorithm is based on weighted pushdown systems (WPDSs) [3, 21].

The contributions of this paper can be summarized as follows:

- We define the notions of error projection and annotated error projection. We prove they are sound with respect to the program abstraction used for model checking.
- We give a novel combination of forward and backward analyses for multi-procedural programs and use it for computing error projections.
- We show that our algorithm is applicable in other settings—specifically, for model checking concurrent programs under certain restricted, but useful conditions.
- Our experiments show that we can efficiently compute error projections.

The remainder of the paper is organized as follows: §2 motivates the difficulty in computing (annotated) error projections. §3 presents the definitions of weighted pushdown systems and weighted automata. §4 and §5 give the algorithms for computing error projections and annotated error projections, respectively. §6 presents our initial experiments. §7 covers other applications of our algorithms. §8 discusses related work.

2 The Correlation Problem

We motivate our work with a discussion of the *correlation problem*. In Fig. 1, the nodes labeled s and t are the source and target nodes, respectively. The set of nodes labeled C_n represents the reachable configurations (defined in §3) of program node (a.k.a. program counter) n from s and to t. The labeled edges represent the abstract transformers that summarize the paths $s \Rightarrow^* n$ and $n \Rightarrow^* t$. We assume that the program abstraction can rule out (some) invalid paths due to correlated branches that cause the path to be non-executable. For the purpose of this discussion, let us assume that $\tau_4 \circ \tau_2 = \bot = \tau_3 \circ \tau_1$. Therefore, n should not be included in the error projection because neither of the two configurations in C_n lie on an executable path from s to t.

A naive approach for computing an error projection is to compute the join-over-all-paths (JOP) from s to n ($\tau_1 \sqcup \tau_2$) composed with JOP from n to t ($\tau_3 \sqcup \tau_4$). Using this approach on Fig. 1 would yield $(\tau_3 \sqcup \tau_4) \circ (\tau_1 \sqcup \tau_2) =$

(a) Counterexample	(b) Error Projection	(c) Program Slice

int foo(int i) {
assert(i > 0);
i = complex(i);
return i;}
	int foo(int i) {
	assert(i > 0);
	i = complex(i);
	return i;}
		int foo(int i) {
		assert(i > 0);
		i = complex(i);
		return i;}

int bar(int len,int* arr) {
int r = 0;
switch(f(len,arr)) {
case 3: r += foo(arr[2]);
case 2: r += foo(arr[1]);
case 1: r += foo(arr[0]);
return r; }
	int bar(int len,int* arr) {
	int r = 0;
	switch(f(len,arr)) {
	case 3: r += foo(arr[2]);
	case 2: r += foo(arr[1]);
	case 1: r += foo(arr[0]);
	return r; }
		int bar(int len,int* arr) {
		int r = 0;
		switch(f(len,arr)) {
		case 3: r += foo(arr[2]);
		case 2: r += foo(arr[1]);
		case 1: r += foo(arr[0]);
		return r; }

int main() { ...;bar(3,{1,-1,-2}); }	int main() { ...;bar(3,{1,-1,-2}); }	int main() { ...;bar(3,{1,-1,-2}); }

Table 1. `bar` is called with an abstract array $\{+, -, -\}$. Column (a) highlights a possible counterexample returned from a model checker. Column (b) highlights the error projection. Column (c) highlights the program statements obtained from a backward slice from the `assert` statement. (Error projection is compared with program slicing in §8.)

$(\tau_3 \circ \tau_1) \sqcup (\tau_3 \circ \tau_2) \sqcup (\tau_4 \circ \tau_1) \sqcup (\tau_4 \circ \tau_2)$, which includes the "cross terms" $\tau_4 \circ \tau_1$ and $\tau_3 \circ \tau_2$. If (in the abstract domain) either of these is not $\bot$, then this approach erroneously concludes that n should be in the error projection. A precise solution to the correlation problem does not allow for these over-approximate transformers. That is, it computes exactly the transformers from s to t that pass through a program node n.

In the *intra*procedural case, there is no correlation problem because program node n only has a single configuration. However, in the *inter*procedural case, the number of configurations may be infinite. Therefore, enumeration does not provide a solution; solving the correlation problem requires the use of symbolic techniques.

The key technical contribution of our work is a symbolic approach that solves the correlation problem. We use standard techniques as our starting point: the program's control flow is modeled via a pushdown system, and certain possibly-infinite sets of configurations are represented using finite automata. We incorporate dataflow information by augmenting the pushdown system with a weight domain, and hence the automata that we use are weighted. The desired result is a kind of intersection of two weighted automata, which represent the forward and backward reachability relations from s and t, respectively. To our knowledge,

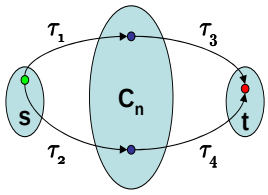

Fig. 1. The correlation problem

there was no previously known algorithm for solving this problem.[3] We solve this problem in §4, using a novel construction of an appropriate automaton.

3 Preliminary Definitions

We compute error projections using weighted pushdown systems (WPDSs). This section presents background material on WPDSs.

3.1 Pushdown Systems

Definition 1. *A **pushdown system** (PDS) is a triple $\mathcal{P} = (P, \Gamma, \Delta)$ where P is a finite set of states, Γ a finite stack alphabet, and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ a finite set of rules. A **configuration** c is a pair $\langle p, u \rangle$ where $p \in P$ and $u \in \Gamma^*$. The pushdown rules define a transition relation $\Rightarrow$ on configurations as follows: If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \Delta$, then $\langle p, \gamma u \rangle \Rightarrow \langle p', \gamma' u \rangle$ for all $u \in \Gamma^*$. The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. For a set of configurations C, we define $pre^*(C) = \{c' \mid \exists c \in C : c' \Rightarrow^* c\}$ and $post^*(C) = \{c' \mid \exists c \in C : c \Rightarrow^* c'\}$.*

Without loss of generality, we restrict PDS rules to have at most two stack symbols on the right-hand side.

A PDS is capable of encoding control flow in a program with procedures. The stack of the PDS simulates the run-time stack of the program, which stores return addresses of unfinished procedure calls, with the current program location on the top of the stack. A procedure call is modeled by a PDS rule with two stack symbols on the right-hand side: it pushes the return address on the stack before giving control to the called procedure. Procedure return is modeled by a PDS rule with no stack symbols on the right-hand side: it pops off the top of the stack and returns control to the address on the top of the stack. With such a PDS, the transition relation $\Rightarrow^*$ captures paths in the program with matched calls and returns. More details on encoding programs as PDSs can be found in [21, 22].

Because the number of configurations of a PDS is unbounded, it is useful to use finite automata to describe certain infinite sets of configurations.

Definition 2. *If $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system then a $\mathcal{P}$-**automaton** is a finite automaton $(Q, \Gamma, \rightarrow, P, F)$ where $Q \supseteq P$ is a finite set of states, $\rightarrow \subseteq Q \times \Gamma \times Q$ is the transition relation, P is the set of initial states, and F is the set of final states of the automaton. We say that a configuration $\langle p, u \rangle$ is accepted by a $\mathcal{P}$-automaton if the automaton can accept u when it is started in the state p (written as $p \xrightarrow{u}{}^* q$, where $q \in F$). A set of configurations is **regular** if some $\mathcal{P}$-automaton accepts it.*

An important result is that for a regular set of configurations C, both $post^*(C)$ and $pre^*(C)$ are also regular sets of configurations [8, 2, 22]. The algorithms for computing $post^*$ and pre^* take a $\mathcal{P}$-automaton $\mathcal{A}$ as input, and if C is the set

[3] Existing literature on weighted automata [17, 18] assumes that the extend operation (composition) is commutative. This restriction is not satisfied by most program abstractions.

of configurations accepted by $\mathcal{A}$, they produce automata $\mathcal{A}_{post^*}$ and $\mathcal{A}_{pre^*}$ that accept the set of configurations $post^*(C)$ and $pre^*(C)$, respectively. In the rest of this paper, all configuration sets will be regular.

3.2 Weighted Pushdown Systems

A weighted pushdown system (WPDS) is a PDS augmented with a weight domain that is a bounded idempotent semiring [3, 21]. The weight domain describes an abstraction with certain algebraic properties.

Definition 3. *A **bounded idempotent semiring** is a quintuple $(D, \oplus, \otimes, \overline{0}, \overline{1})$, where D is a set whose elements are called **weights**, $\overline{0}$ and $\overline{1}$ are elements of D, and $\oplus$ (the combine operator) and $\otimes$ (the extend operator) are binary operators on D such that*

1. *$(D, \oplus)$ is a commutative monoid with $\overline{0}$ as its neutral element, and where $\oplus$ is idempotent. $(D, \otimes)$ is a monoid with the neutral element $\overline{1}$.*
2. *$\otimes$ distributes of $\oplus$, i.e. for all $a, b, c \in D$ we have $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$.*
3. *$\overline{0}$ is an annihilator with respect to $\otimes$, i.e. for all $a \in D$, $a \otimes \overline{0} = \overline{0} = \overline{0} \otimes a$.*
4. *In the partial order $\sqsubseteq$ defined by $\forall a, b \in D, a \sqsubseteq b \iff a \oplus b = b$, there are no infinite ascending chains.*

In abstract-interpretation terminology, weights can be thought of as abstract transformers, $\otimes$ as transformer composition, and $\oplus$ as *join*. A WPDS is a program (PDS) augmented with an abstraction (weights) and can be thought of as an abstract model of a program.

Definition 4. *A **weighted pushdown system** is a triple $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ where $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system, $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ is a bounded idempotent semiring and $f : \Delta \to D$ is a map that assigns a weight to each pushdown rule.*

Let $\sigma \in \Delta^*$ be a sequence of rules. Using f, we can associate a value to σ, i.e. if $\sigma = [r_1, \ldots, r_k]$, then $pval(\sigma) = f(r_1) \otimes \ldots \otimes f(r_k)$. Moreover, for any two configurations c and c', if σ is a rule sequence that transitions c to c' then we say $c \Rightarrow^{\sigma} c'$. Reachability problems on PDSs are generalized to WPDSs as follows:

Definition 5. *Let $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ be a WPDS, where $\mathcal{P} = (P, \Gamma, \Delta)$, and let $S, T \subseteq P \times \Gamma^*$ be regular sets of configurations. Then the **join-over-all-paths** value $JOP(S, T)$ is defined as $\bigoplus\{pval(\sigma) \mid s \Rightarrow^{\sigma} t, s \in S, t \in T\}$.*

A PDS is a WPDS with the Boolean weight domain $(\{\overline{1}, \overline{0}\}, \oplus, \otimes, \overline{0}, \overline{1})$ and $f(r) = \overline{1}$ for all rules $r \in \Delta$. ($\mathrm{JOP}(S, T) = \overline{1}$ iff a configuration in S can reach a configuration in T.) An important weight domain that we use in §5 is the set of all binary relations on a finite set:

Definition 6. *Let V be a finite set. A **relational weight domain** on V is defined as the semiring $(D, \oplus, \otimes, \overline{0}, \overline{1})$ where $D = \mathcal{P}(V \times V)$ is the set of all binary relations on V, $\oplus$ is union, $\otimes$ is relational composition, $\overline{0}$ is the empty set, and $\overline{1}$ is the identity relation on V.*

Such domains are useful for describing finite abstractions, e.g., predicate abstraction, abstraction of Boolean programs, and finite-state safety properties (a short discussion can be found in [15]). In predicate abstraction, $v \in V$ would be a fixed valuation of the predicates, which in turn represents all memory configurations in which that valuation holds. Weights are transformations on these states that represent the abstract effect of executing a program statement. As some readers might already be aware, WPDSs with such a weight domain can also be represented using PDSs by expanding the PDS state space with elements from V. However, the advantage of using WPDSs is that weights can symbolically encode transition relations. Consider predicate abstraction with 50 predicates. The set V would then have 2^{50} elements, and it would be impractical to use explicit PDS rules to specify transitions (i.e., relations) over these sets. However, they can usually be represented succinctly using BDDs to represent weights on rules. (This is the essence of Schwoon's MOPED system [22].)

3.3 Solving for the JOP value in WPDSs

There are two algorithms for finding the JOP value, called *poststar* and *prestar*, based on forward and backward reachability, respectively [21]. These algorithms operate on *weighted automata* defined as follows.

Definition 7. *Given a weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, a $\mathcal{W}$-**automaton** $\mathcal{A}$ is a $\mathcal{P}$-automaton, where each transition in the automaton is labeled with a weight. The weight of a path in the automaton is obtained by taking an extend of the weights on the transitions in the path in either a forward or backward direction, depending on the context in which the automaton is used. The automaton is said to accept a configuration $c = \langle p, u \rangle$ with weight w, written as $\mathcal{A}(c)$, if w is the combine of weights of all accepting paths for u starting from state p in the automaton. We call the automaton a **backward $\mathcal{W}$-automaton** if the weight of the path is read backwards and a **forward $\mathcal{W}$-automaton** otherwise.*

For simplicity, we call a $\mathcal{W}$-automaton a weighted automaton. The poststar algorithm takes a backward weighted automaton $\mathcal{A}$ as input and produces another backward weighted automaton $\mathcal{A}_{post^*}$, such that $\mathcal{A}_{post^*}(c) = \bigoplus\{\mathcal{A}(c') \otimes pval(\sigma) \mid c' \Rightarrow^{\sigma} c\}$. Similarly, the prestar algorithm takes a forward weighted automaton $\mathcal{A}$ as input and produces $\mathcal{A}_{pre^*}$ such that $\mathcal{A}_{pre^*}(c) = \bigoplus\{pval(\sigma) \otimes \mathcal{A}(c') \mid c \Rightarrow^{\sigma} c'\}$.[4]

An important algorithm for reading out weights from weighted automata is called *path_summary* defined as follows: $path\_summary(\mathcal{A}) = \oplus\{\mathcal{A}(c) \mid c \in P \times \Gamma^*\}$. We briefly outline this algorithm for a forward weighted automaton. It is based on a standard fixpoint-finding algorithm. It associates a weight $l(q)$ to each state q of $\mathcal{A}$: Initialize the weight of each non-initial state in $\mathcal{A}$ to $\overline{0}$ and each initial state to $\overline{1}$; add each initial state to a worklist. Next, repeatedly

[4] The prestar and poststar algorithms, as originally defined, only took unweighted automata as input, i.e., $\mathcal{A}(c) = \overline{1}$ for each configuration c accepted by $\mathcal{A}$. However, the same algorithms work with weighted automata as defined here.

remove a state, say q, from the worklist and propagate its weight forwards: i.e., if there is a transition (q, γ, q') with weight w, then update the weight of state q' as $l(q') := l(q') \oplus (l(q) \otimes w)$; if the weight on q' changes, then add it to the worklist. This is repeated until the worklist is empty. Then *path_summary*$(\mathcal{A})$ is the combine of $l(q)$ for each final state q.

Using *path_summary*, we can calculate $\mathcal{A}(C) = \bigoplus\{\mathcal{A}(c) \mid c \in C\}$ as follows: Let $\mathcal{A}_C$ be an (unweighted) automaton that accepts C. Intersect $\mathcal{A}$ and $\mathcal{A}_C$ to obtain a weighted automaton $\mathcal{A}'$.[5] Then it is easy to see that $\mathcal{A}(C) =$ *path_summary*$(\mathcal{A}')$. Using this, we can solve for JOP. Let $\mathcal{A}_S$ and $\mathcal{A}_T$ be (unweighted) automata that accept the sets S and T, respectively. Then $\text{JOP}(S, T) =$ *poststar*$(\mathcal{A}_S)(T) =$ *prestar*$(\mathcal{A}_T)(S)$.

4 Computing an Error Projection

Let us now define an error projection using WPDSs as our model of programs. Usually, a WPDS created from a program has a single PDS state. Even when this is not the case, the states can be pushed inside the weights to get a single-state WPDS. We use this to simplify the discussion: PDS configurations are just represented as stacks (Γ^*).

Also, we concern ourselves with assertion checking. We assume that we are given a target set of control configurations T such that the program model exhibits an error only if it can reach a configuration in that set. One way of accomplishing this is to convert every assertion of the form "`assert(`$\mathcal{E}$`)`" into a condition "`if(!`$\mathcal{E}$`) then goto error`" (assuming $!\mathcal{E}$ is expressible under the current abstraction), and instantiate T to be the set of configurations $(error\ \Gamma^*) = \{error\ c \mid c \in \Gamma^*\}$. We also assume that the weight abstraction has been constructed such that a path σ in the PDS is *infeasible* if and only if its weight *pval*(σ) is $\overline{0}$. Therefore, under this model, the program has an error only when it can reach a configuration in T with a path of non-$\overline{0}$ weight. We can now formally define an error projection.

Definition 8. *Given S, the set of starting configurations of the program, and a target set of configurations T, a program node $\gamma \in \Gamma$ is in the* ***error projection*** *EP(S, T) if and only if there exists a path $\sigma = \sigma_1\sigma_2$ such that pval$(\sigma) \neq \overline{0}$ and $s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t$ for some $s \in S, c \in (\gamma\ \Gamma^*), t \in T$.*

We calculate the error projection by computing a constrained form of the join-over-all-paths value, which we call a weighted chopping query.

Definition 9. *Given regular sets of configurations S (source), T (target), and C (chop); a* ***weighted chopping*** *query is to compute the following weight:*

$$WC(S, C, T) = \bigoplus\{v(\sigma_1\sigma_2) \mid s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t, s \in S, c \in C, t \in T\}$$

It is easy to see that $\gamma \in \text{EP}(S, T)$ if and only if $\text{WC}(S, \gamma\ \Gamma^*, T) \neq \overline{0}$. For short, we call a weighted chopping query simply a chopping query. We now show how to solve these queries.

[5] Intersection of a weighted automaton with an unweighted one is carried out the same way as for two unweighted automata, except that the weights of the weighted automaton are copied over to the resultant automaton.

First, note that $\mathrm{WC}(S,C,T) \neq \mathrm{JOP}(S,C) \otimes \mathrm{JOP}(C,T)$ because of the correlation problem. A first attempt at solving weighted chopping is to use the identity $\mathrm{WC}(S,C,T) = \bigoplus\{\mathrm{JOP}(S,c) \otimes \mathrm{JOP}(c,T) \mid c \in C\}$. However, this only works when C is a finite set of configurations, which is not the case if we want to compute an error projection. We can solve this problem using the automata-theoretic constructions described in the previous section. Let $\mathcal{A}_S$ be an unweighted automaton that represents the set S, and similarly for $\mathcal{A}_C$ and $\mathcal{A}_T$. The following two algorithms, given in different columns, are valid ways of solving a weighted chopping query.

1. $\mathcal{A}_1 = \mathit{poststar}(\mathcal{A}_S)$	1. $\mathcal{A}_1 = \mathit{prestar}(\mathcal{A}_T)$
2. $\mathcal{A}_2 = (\mathcal{A}_1 \cap \mathcal{A}_C)$	2. $\mathcal{A}_2 = (\mathcal{A}_1 \cap \mathcal{A}_C)$
3. $\mathcal{A}_3 = \mathit{poststar}(\mathcal{A}_2)$	3. $\mathcal{A}_3 = \mathit{prestar}(\mathcal{A}_2)$
4. $\mathcal{A}_4 = \mathcal{A}_3 \cap \mathcal{A}_T$	4. $\mathcal{A}_4 = \mathcal{A}_3 \cap \mathcal{A}_S$
5. $\mathrm{WC}(S,C,T) = \mathit{path\_summary}(\mathcal{A}_4)$	5. $\mathrm{WC}(S,C,T) = \mathit{path\_summary}(\mathcal{A}_4)$

The running time is only proportional to the size of $\mathcal{A}_C$, not the size of the set represented by it. A proof of correctness is given in the appendix.

An error projection is computed by asking a separate weighted chopping query for each node γ in the program. This means that the source set S and the target set T remain fixed, but the chop set C keeps changing. Unfortunately, the two algorithms given above have a major shortcoming: only their respective first steps can be carried over from one chopping query to the next; the rest of the steps have to be recomputed for each node γ. As shown in §6, this approach is very slow, and the algorithm discussed next is about 3 orders of magnitude faster.

To derive a better algorithm for weighted chopping that is more suited for computing error projections, let us first look at the unweighted case (i.e., the weighted case where the weight domain just contains the weights $\overline{0}$ and $\overline{1}$). Then $\mathrm{WC}(S,C,T) = \overline{1}$ if and only if $(\mathit{post}^*(S) \cap \mathit{pre}^*(T)) \cap C \neq \emptyset$. This procedure just requires a single intersection operation for different chop sets. Computation of both $\mathit{post}^*(S)$ and $\mathit{pre}^*(T)$ have to be done just once. The main difficulty in extending this approach to the case of general weights is that there is no known algorithm for intersecting weighted automata.

To address this issue, we now introduce the key theoretical contribution of this paper. First, we need to define what we mean by intersecting weighted automata. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two weighted automata. Define their *intersection* $\mathcal{A}_1 \lhd \mathcal{A}_2$ to be a function from configurations to weights, which we later compute in the form of a weighted automaton, such that $(\mathcal{A}_1 \lhd \mathcal{A}_2)(c) = \mathcal{A}_1(c) \otimes \mathcal{A}_2(c)$.[6] Define $(\mathcal{A}_1 \lhd \mathcal{A}_2)(C) = \bigoplus\{(\mathcal{A}_1 \lhd \mathcal{A}_2)(c) \mid c \in C\}$, as before. Based on this definition, if $\mathcal{A}_{post^*} = \mathit{poststar}(\mathcal{A}_S)$ and $\mathcal{A}_{pre^*} = \mathit{prestar}(\mathcal{A}_T)$, then $\mathrm{WC}(S,C,T) = (\mathcal{A}_{post^*} \lhd \mathcal{A}_{pre^*})(C)$.

Let us give some intuition into why intersecting weighted automata is hard. For $\mathcal{A}_1$ and $\mathcal{A}_2$ as above, the intersection is defined to read off the weight from

[6] Note that the operator $\lhd$ is not commutative is general, but we still call it *intersection* because the construction of $\mathcal{A}_1 \lhd \mathcal{A}_2$ resembles the one for intersection of unweighted automata.

$\mathcal{A}_1$ first and then extend it with the weight from $\mathcal{A}_2$. A naive approach would be to construct a weighted automaton $\mathcal{A}_{12}$ as the concatenation of $\mathcal{A}_1$ and $\mathcal{A}_2$ (with epsilon transitions from the final states of $\mathcal{A}_1$ to the initial states of $\mathcal{A}_2$) and let $(\mathcal{A}_1 \triangleleft \mathcal{A}_2)(c) = \mathcal{A}_{12}(c\ c)$. However, computing $(\mathcal{A}_1 \triangleleft \mathcal{A}_2)(C)$ for a regular set C requires computing join-over-all-paths in $\mathcal{A}_{12}$ over the set of paths that accept the language $\{(c\ c) \mid c \in C\}$ because the *same* path (i.e., c) must be followed in both $\mathcal{A}_1$ and $\mathcal{A}_2$. This language is neither regular nor context-free, and we do not know of any method that computes join-over-all-paths over a non-context-free set of paths.

The trick here is to recognize that weighted automata have a direction in which weights are read off. We need to intersect $\mathcal{A}_{post^*}$ with $\mathcal{A}_{pre^*}$, where $\mathcal{A}_{post^*}$ is a backward automaton and $\mathcal{A}_{pre^*}$ is a forward automaton. If we concatenate these together but reverse the second one (reverse all transitions and switch initial and final states), then we get a purely backward weighted automaton and we only need to solve for join-over-all-paths over the language $\{(c\ c^R) \mid c \in C\}$ where c^R is c written in the reverse order. This language can be defined using a context-free grammar with production rules of the form "$X \to \gamma Y \gamma$" where X and Y are non-terminals. The following section uses this intuition to derive an algorithm for intersecting two weighted automata.

Intersecting Weighted Automata

Let $\mathcal{A}_b = (Q_b, \Gamma, \to_b, P, F_b)$ be a backward weighted automaton and $\mathcal{A}_f = (Q_f, \Gamma, \to_f, P, F_f)$ be a forward weighted automaton. We proceed with the standard automaton-intersection algorithm: Construct a new automaton $\mathcal{A}_{bf} = (Q_b \times Q_f, \Gamma, \to, P, F_b \times F_f)$, where we identify the state $(p, p), p \in P$ with p, i.e., the P-states of $\mathcal{A}_{bf}$ are states of the form $(p, p), p \in P$. The transitions of this automaton are computed by matching on stack symbols. If $t_b = (q_1, \gamma, q_2)$ is a transition in $\mathcal{A}_b$ with weight w_b and $t_f = (q_3, \gamma, q_4)$ is a transition in $\mathcal{A}_f$ with weight w_f, then add transition $t_{bf} = ((q_1, q_3), \gamma, (q_2, q_4))$ to $\mathcal{A}_{bf}$ with weight $\lambda z.(w_b \otimes z \otimes w_f)$. We call these type of weights *functional weights* and will use the capital letter W (possibly subscripted) to distinguish them from normal weights. Functional weights are special functions on weights: given a weight w and a functional weight $W = \lambda z.(w_1 \otimes z \otimes w_2)$, $W(w) = (w_1 \otimes w \otimes w_2)$. The automaton $\mathcal{A}_{bf}$ is called a *functional automaton*.

We define extend on functional weights as reversed function composition. That is, if $W_1 = \lambda z.(w_1 \otimes z \otimes w_2)$ and $W_2 = \lambda z.(w_3 \otimes z \otimes w_4)$, then $W_1 \otimes W_2 = W_2 \circ W_1 = \lambda z.((w_3 \otimes w_1) \otimes z \otimes (w_2 \otimes w_4))$, and is therefore also a functional weight. However, the natural combine operator, defined as $W_1 \oplus W_2 = \lambda z.(W_1(z) \oplus W_2(z))$, does not preserve the form of functional weights. Hence, functional weights do not form a semiring. We show next that this is not a handicap, and we can still compute $\mathcal{A}_b \triangleleft \mathcal{A}_f$ as required.

Because $\mathcal{A}_{bf}$ is a product automaton, every path in it of the form $(q_1, q_2) \xrightarrow{c}{}^* (q_3, q_4)$ is in one-to-one correspondence with paths $q_1 \xrightarrow{c}{}^* q_3$ in $\mathcal{A}_b$ and $q_2 \xrightarrow{c}{}^* q_4$ in $\mathcal{A}_f$. Using this, it is easy to see that the weight of a path in $\mathcal{A}_{bf}$ will be a function of the form $\lambda z.(w_b \otimes z \otimes w_f)$, where w_b and w_f are the weights of the corresponding paths in $\mathcal{A}_b$ and $\mathcal{A}_f$, respectively. In this sense, $\mathcal{A}_{bf}$ is

constructed based on the intuition given in the previous section: the functional weights resemble grammar productions "$X \to \gamma Y \gamma$" for the language $\{(c\ c^R)\}$ with weights replacing the two occurrences of γ, and their composition resembles the derivation of a string in the language. (Note that in "$X \to \gamma Y \gamma$", the first γ is a letter in c, whereas the second γ is a letter in c^R. In general, the letters will be given different weights in $\mathcal{A}_b$ and $\mathcal{A}_f$.)

Formally, for a configuration c and a weighted automaton $\mathcal{A}$, define the predicate $accpath(\mathcal{A}, c, w)$ to be true if there is an accepting path in $\mathcal{A}$ for c that has weight w, and false otherwise (note that we only need the extend operation to compute the weight of a path). Similarly, $accpath(\mathcal{A}, C, w)$ is true iff $accpath(\mathcal{A}, c, w)$ is true for some $c \in C$. Then we have:

$$\begin{aligned}(\mathcal{A}_b \lhd \mathcal{A}_f)(c) &= \mathcal{A}_b(c) \otimes \mathcal{A}_f(c) \\ &= \bigoplus\{w_b \otimes w_f \mid accpath(\mathcal{A}_b, c, w_b), accpath(\mathcal{A}_f, c, w_f)\} \\ &= \bigoplus\{w_b \otimes w_f \mid accpath(\mathcal{A}_{bf}, c, \lambda z.(w_b \otimes z \otimes w_f))\} \\ &= \bigoplus\{\lambda z.(w_b \otimes z \otimes w_f)(\overline{1}) \mid accpath(\mathcal{A}_{bf}, c, \lambda z.(w_b \otimes z \otimes w_f))\} \\ &= \bigoplus\{W(\overline{1}) \mid accpath(\mathcal{A}_{bf}, c, W)\}\end{aligned}$$

Similarly, we have $(\mathcal{A}_b \lhd \mathcal{A}_f)(C) = \bigoplus\{W(\overline{1}) \mid accpath(\mathcal{A}_{bf}, C, W)\} = \bigoplus\{W(\overline{1}) \mid accpath(\mathcal{A}_{bf} \cap \mathcal{A}_C, \Gamma^*, W)\}$, where $\mathcal{A}_C$ is an unweighted automaton that accepts the set C, and this can be obtained using a procedure similar to *path_summary*. The advantage of the way we have defined $\mathcal{A}_{bf}$ is that we can intersect it with $\mathcal{A}_C$ (via ordinary intersection) and then run *path_summary* over it, as we show next.

Functional weights distribute over (ordinary) weights, i.e., $W(w_1 \oplus w_2) = W(w_1) \oplus W(w_2)$. Thus, *path_summary*$(\mathcal{A}_{bf})$ can be obtained merely by solving an intraprocedural join-over-all-paths over distributive transformers starting with the weight $\overline{1}$, which is completely standard: Initialize $l(q) = \overline{1}$ for initial states, and set $l(q) = \overline{0}$ for other states. Then, until a fixpoint is reached, for a transition (q, γ, q') with weight W, update the weight on state q' by $l(q') := l(q') \oplus W(l(q))$. Then *path_summary*$(\mathcal{A}_{bf})$ is the combine of weight on final states. Termination is guaranteed because we still have weights associated with states, and functional weights are monotonic. Because of the properties satisfied by $\mathcal{A}_{bf}$, we use $\mathcal{A}_{bf}$ as a representation for the function $(\mathcal{A}_b \lhd \mathcal{A}_f)$.

This allows us to solve $\mathrm{WC}(S, C, T) = (\mathcal{A}_{post^*} \lhd \mathcal{A}_{pre^*})(C)$. That is, after a preparation step to create $(\mathcal{A}_{post^*} \lhd \mathcal{A}_{pre^*})$, one can solve $\mathrm{WC}(S, C, T)$ for different chop sets C just using intersection with $\mathcal{A}_C$ followed by *path_summary*, as shown above.

It should be noted that this technique applies only to the intersection of a forward weighted automaton with a backward one, because in this case we are able to get around the problem of computing join-over-all-paths over a non-context-free set of paths. We are not aware of any algorithms for intersecting two forward or two backward automata; those problems remain open.

5 Computing an Annotated Error Projection

An annotated error projection adds more information to an error projection by associating each node in the error projection with (i) at least one counterexample

that goes through that node and (*ii*) the set of *abstract stores* (or memory descriptors) that may arise on a path doomed to fail in the future. To be consistent with WPDS terminology, we use the term *witness* as a synonym for counterexample. Finding witnesses also helps in accelerating the computation of an error projection.

5.1 Computing witnesses

Given source set S and target set T, previous work on WPDSs allows the computation of a finite set of paths, called *witnesses*, $\{\sigma_i \mid 1 \leq i \leq n\}$ such that $\oplus_i\{pval(\sigma_i)\} = \text{JOP}(S,T)$ [21]. The same result holds for *path_summary* on weighted as well as functional automata: we can find a finite set of paths in the automaton that justifies the weight returned by *path_summary* (we say that a set of paths justifies a weight w if the combine of their weights is equal to w). We make use of this technology in this section.

While calculating the error projection, if we find that γ is in the error projection, then we know $\text{WC}(S,C,T) = w \neq \overline{0}$, where $C = \gamma\Gamma^*$. If $\mathcal{A}_C$ is the unweighted automaton that represents C, and $\mathcal{A}_\cap = (\mathcal{A}_{post^*} \lhd \mathcal{A}_{pre^*}) \cap \mathcal{A}_C$, then *path_summary*$(\mathcal{A}_\cap) = w$. Using witness generation, we can find at least one path in $\mathcal{A}_\cap$ whose weight is not $\overline{0}$. A path in this automaton corresponds to a configuration c with $\mathcal{A}_\cap(c) \neq \overline{0}$. This, in turn, implies that $c \in C$ and there is a path in the WPDS from S to T through c with non-$\overline{0}$ weight. Again, using standard witness generation, we can find a set of witness $\{\sigma_i\}_{1\leq i\leq n}$ for $\text{JOP}(S,c)$ and a set of witnesses $\{\rho_j\}_{1\leq j\leq m}$ for $\text{JOP}(c,T)$. The concatenation of these witnesses $\{\sigma_i\rho_j\}_{1\leq i\leq n}^{1\leq j\leq m}$ justifies $\text{JOP}(S,c) \otimes \text{JOP}(c,T) = w \neq \overline{0}$. (The concatenation is a constant-time operation because a witness set is represented using a DAG.) Therefore, one of these witnesses is a path with non-$\overline{0}$ weight and serves as the desired witness for node γ. The same procedure can be repeated for each node in the error projection. Finding witnesses is not a very expensive operation, but it adds a fair amount of overhead to the execution of *poststar* and *prestar* (although their worst-case running times do not change).

One optimization that witnesses allow is that if we obtain σ as a witness for a node γ in the error projection, then for every node γ' such that a configuration $c \in \gamma'\Gamma^*$ occurs in σ, γ' must also be in the error projection. Therefore, while computing an error projection, if we find γ to be in the error projection, then we can find a witness for it and immediately include in the error projection every such γ'.

5.2 Computing abstract stores

For defining and computing the abstract stores for nodes in an error projection, we restrict ourselves to relational abstractions over a finite set. We can only compute the precise set of abstract stores under this assumption. In other cases, we can only approximate the desired set of abstract stores (the over- and under-approximation algorithms given below work for any weight domain).

Let V be a finite set of abstract stores and $(D, \oplus, \otimes, \overline{0}, \overline{1})$ the relational weight domain on V, as defined in Defn. 6. For weights $w, w_1, w_2 \in D$, define $\texttt{Rng}(w)$ to be the range of w, $\texttt{Dom}(w)$ to be the domain of w and $\texttt{Com}(w_1, w_2) =$

$\texttt{Rng}(w_1) \cap \texttt{Dom}(w_2)$. If $v \in \texttt{Com}(w_1, w_2)$ then there are some abstract stores v_s and v_t such that $v \in w_1(v_s)$ and $v_t \in w_2(v)$. For a node $\gamma \in \mathrm{EP}(S,T)$, we want to compute the following non-empty subset of V:

$$V_\gamma = \{v \in \texttt{Com}(pval(\sigma_1), pval(\sigma_2)) \mid s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t, pval(\sigma_1\sigma_2) \neq \overline{0}, s \in S, c \in \gamma\Gamma^*, t \in T\}$$

If $v \in V_\gamma$, then there must be a path in the program model that leads to an error such that the abstract store v arises at node γ. In this section, we give symbolic algorithms (i.e., based on weights) to over-approximate and under-approximate the above set, and one for verifying membership in that set. First, we show how to check if $v \in V_\gamma$. Conceptually, we place a bottleneck at node γ, using a special weight, to see if there is a feasible path that can pass through the bottleneck at γ with abstract store v, and then continue on to the error configuration. Let $w_v = \{(v, v)\}$. Note that $v \in \texttt{Com}(w_1, w_2)$ iff $w_1 \otimes w_v \otimes w_2 \neq \overline{0}$. Let $\mathcal{A}_{post^*} = poststar(\mathcal{A}_S), \mathcal{A}_{pre^*} = prestar(\mathcal{A}_T)$ and $\mathcal{A}_\triangleleft$ be their intersection. Then $v \in V_\gamma$ iff there is a configuration $c \in \gamma\Gamma^*$ such that $\mathrm{JOP}(S, c) \otimes w_v \otimes \mathrm{JOP}(c, T) \neq \overline{0}$ or, equivalently, $\mathcal{A}_{post^*}(c) \otimes w_v \otimes \mathcal{A}_{pre^*}(c) \neq \overline{0}$. To check this, we use the functional automaton $\mathcal{A}_\triangleleft$ again. It is not hard to check that the following holds for any weight w:

$$\mathcal{A}_{post^*}(c) \otimes w \otimes \mathcal{A}_{pre^*}(c) = \bigoplus\{W(w) \mid accpath(\mathcal{A}_\triangleleft, c, W)\}$$

Then $v \in V_\gamma$ if $\bigoplus\{W(w_v) \mid accpath(\mathcal{A}_\triangleleft, \gamma\Gamma^*, W)\} \neq \overline{0}$. This is, again, computable using *path_summary*, but we initialize the weight on initial states with w_v instead of $\overline{1}$.

This gives us an algorithm for computing V_γ, but its running time would be proportional to $|V|$, which might be very large. To get around this, we compute symbolic under- and over-approximations of V_γ. Also, we envision that the membership can be tested on-demand as a user asks for more information about a particular abstract store occurring at a particular node in the error slice. Moreover, we can use this algorithm to check, for a set $V' \subseteq V$, whether $V' \cap V_\gamma = \emptyset$: for this, we would use the weight $w_{V'} = \{(v, v) \mid v \in V'\}$ as the bottleneck. This would be useful (and fast), for example, to check if a predicate can hold at a node in the error projection: choose V' to be all abstract stores in which that predicate holds.

An under-approximation of V_γ can be obtained from the set of witnesses calculated for the error projection. Suppose that σ is a witness for some node in the error projection and passes through node γ; let σ_1 be the prefix of σ up to node γ and σ_2 the suffix starting from γ. Then, by definition, $\texttt{Com}(pval(\sigma_1), pval(\sigma_2)) \subseteq V_\gamma$. This can be repeated for each witness and all nodes it passes through to get an under-approximation of V_γ.

For an over-approximation of V_γ, we turn to the functional automaton again and setup a different join-over-all-paths problem than the one solved by *path_summary* on functional automata. This time, each state in the automaton will be associated with a triple of weights. The first component will be responsible for computing the weight from S to some intermediate configuration c, the second component will compute the weight from c to T, and the third weight tracks the weight of a path from S to T that goes through c. In accordance with the definition of V_γ, we drop the weight in the first two components (set them

to $\overline{0}$) if the third component becomes $\overline{0}$, i.e., the path from S to T being tracked currently is infeasible.

To accomplish this, we replace each functional weight $\lambda z.(w_1 \otimes z \otimes w_2)$ with $g(w_1, w_2)$, defined as a transformer on a triple of weights as follows:

$$g(w_1, w_2) = \lambda(w_3, w_4, w_5).\mathtt{SP}(w_1 \otimes w_3, w_4 \otimes w_2, w_1 \otimes w_5 \otimes w_2)$$

$$\mathtt{SP}(w_1, w_2, w_3) = \begin{cases} (w_1, w_2, w_3) & \text{if } w_3 \neq \overline{0} \\ (\overline{0}, \overline{0}, \overline{0}) & \text{if } w_3 = \overline{0} \end{cases}$$

Combine on triples is defined component-wise. It is easy to verify that the join-over-all-paths on this automaton intersected with $\mathcal{A}_C, C = \gamma\Gamma^*$, using transformers as defined above gives us the value $(w_1, w_2, \mathrm{WC}(S, C, T))$ such that $V_\gamma = \mathtt{Com}(w_1, w_2)$. However, unlike functionals, the transformers above are not distributive, i.e., $g(w_1, w_2)$ does not distribute over triples of weights with component-wise combine. They are still monotonic and the JOP value can be safely over-approximated. Termination is guaranteed because triples of weights have bounded height under component-wise combine.

6 Experiments

We test our algorithm for computing an error projection on Moped [22], a program-analysis tool that encodes Boolean programs as WPDSs and answers reachability queries on them for checking assertions in the program. The Boolean programs may be obtained after performing predicate abstraction or from integer programs with a limited number of bits to represent bounded integers. Although it uses a finite abstraction, the use of weights to encode abstract transformers as BDDs is crucial for its scalability. Because we can compute an error projection using just extend and combine, we take full advantage of the BDD encoding. Some additional experiments are given in App. B.

We measured the time needed to solve $\mathrm{WC}(S, n\Gamma^*, T)$ for all program nodes n using the algorithms from §4: one that uses functional automata and one based on running two *prestar* queries (called the double-pre^* method below). Although we report the size of the error projection, we could not validate how useful it was because only the model (and not the source code) was available to us.

The results are shown in Tab. 2. The table can be read as follows: the first five columns give the program names, the number of nodes (or basic blocks) in the program, error-projection size relative to program size, and times to compute $post^*(S)$ and $pre^*(T)$, respectively. The next two columns give the running time for solving $\mathrm{WC}(S, n\Gamma^*, T)$ for all nodes n using functionals and using double-pre^*, after the initial computation of $post^*(S)$ and $pre^*(T)$ was completed. Because the double-pre^* method is so slow, we did not run these examples to completion; instead, we report the time for solving the weighted chop query for only 1% of the blocks and multiply the resulting number by 100. The last two columns compare the running time for using functionals (column six) against the time taken to compute $post^*(S) + pre^*(T)$; and the time taken by the double-pre^* method. All running times are in seconds. The experiments were run on a 3GHz P4 machine with 2GB RAM.

As can be seen from the table, using functionals is about three orders of magnitude faster than using the double-pre^* method. Also, as shown by column

eight, computation of the error projection compares fairly well with running a single forward or backward analysis (at least for the smaller programs). To some extent, this implies that error-projection computation can be incorporated into model checkers without adding significant overhead. We wish to investigate further optimizations, such as using witnesses, as future work.

					WC(S, nI^*, T)		Functional vs.	
Prog	Nodes	Error Proj.	$post^*(S)$	$pre^*(T)$	Functional	Double pre^*	Reach	Double pre^*
iscsiprt16	4884	0%	79	1.8	3.5	5800	0.04	1657
pnpmem2	4813	0%	7	4.1	8.8	16000	0.79	1818
iscsiprt10	4824	46%	0.28	0.36	1.6	1200	2.5	750
pnpmem1	4804	65%	7.2	4.5	9.2	17000	0.79	1848
iscsi1	6358	84%	53	110	140	750000	0.88	5357
bugs5	36972	99%	13	2	170	85000	11.3	500

Table 2. MOPED results: The programs are Boolean programs provided by S. Schwoon. S is the entry point of the program, and T is the error configuration set obtained as mentioned in the beginning of §4. An error projection of size 0% means that the program is correct.

7 Additional Applications

7.1 Optimizing the CEGAR loop

The idea of *CounterExample Guided Abstraction Refinement* (CEGAR) [13, 5] is (1) starting from an initial abstract model (2) perform the verification procedure on the model. If the property is satisfied, conclude that it is also satisfied by the real program. Otherwise, a counterexample is computed. (3) If the counterexample does not correspond to an execution in the program, refine the model to eliminate this spurious trace, and go back to step (2). (4) Otherwise, return the counterexample. The refinement step is usually done by adding new predicates. The CEGAR approach has been successfully applied in many program-verification tools, including SLAM [1], BLAST [9], MAGIC [4], KISS [20] and ZING [19].

Using our techniques, CEGAR loops can be optimized in two ways:

1. Our techniques allow the model checker to consider smaller models in step (2). This is possible because the error projection is *sound*. That is, the error projection includes *all* the nodes that occur in error paths. Therefore, we are sure that no paths outside the error projection can violate the property being checked. Hence, there is no need to re-check these paths after a refinement step.
2. One might also be able to use an error projection during the refinement step to eliminate not just a single counterexample as done in current tools, but several ones. We speculate that this ability to focus the effort of the model checker will work particularly well for programs that do not violate the property of interest.

7.2 Applications to multi-threaded programs

KISS [20] is a system that can detect errors in concurrent programs that arise in at most two context switches. The two-context-switch bound enables verification using a sequential model checker. To convert a concurrent program into one

suitable for a sequential model checker, Qadeer and Wu add nondeterministic function calls to the `main` method of process 2 after each statement of process 1. Likewise they add nondeterministic function returns after each statement of process 2. They also ensure that a function call from process 1 to process 2 is only performed once. This technique essentially results in a sequential program that mimics the behavior of a concurrent program along paths of the form $\overset{1}{\rightsquigarrow}\overset{2}{\rightsquigarrow}\overset{1}{\rightsquigarrow}$.

Using our techniques, we can extend the KISS approach so that it will not only return a counterexample, but also determine all of the nodes in process 1 where a context switch can occur that leads to an error later in process 1. One way to do this is to use nondeterministic calls and returns as Qadeer and Wu do, apply a model checker, and then compute the error projection. However, due to the automata-theoretic techniques we employ, we can omit the additions. The following algorithm shows how to do this:

1. Create $\mathcal{A}_{\lhd} = \mathcal{A}_{post^*} \lhd \mathcal{A}_{pre^*}$ for process 1
2. Let $\mathcal{A}_2$ be the result of a poststar query from `main` for process 2.
3. Let $w = path\_summary(\mathcal{A}_2)$; w represents the state transformation caused by the execution steps spent in process 2.
4. For each program node n of process 1, let $A_n = \mathcal{A}_{\lhd} \cap \mathcal{A}_{n\Gamma^*}$ and compute $w_n = path\_summary(\mathcal{A}_n, w)$ where $path\_summary$ is augmented to assign w to the initial states of $\mathcal{A}_n$ instead of $\overline{1}$, i.e., the weight w is used as the bottleneck in $\mathcal{A}_n$ (see §5.2).
5. If $w_n \neq \overline{0}$ then an error can occur in the program when the first context switch occurs at node n in process 1.

Using this algorithm, we can determine all the nodes where a context switch must occur for an error to (eventually) arise.

8 Related Work

The combination of forward and backward analysis has a long history in abstract interpretation, going back to Cousot's thesis [6]. It has been also been used in model checking [16] and in interprocedural analysis [11]. In the present paper, we show how forward and backward approaches can be combined in the context of interprocedural analysis performed with WPDSs; we introduce a novel automata-theoretic construction, and our experiments show that this approach is significantly faster than a more straightforward one.

The goal of both program slicing [23] and our work on error projection is to compute a set of nodes that exhibit some property. In our work, the property of interest is membership in an error path, whereas in the case of program slicing, the property of interest is membership in a path along data and control dependences. Slicing and chopping have certain advantages—for instance, chopping filters out statements that do not transmit effects from source s to target t (cf. Tab. 1(b) and (c)). These techniques have been generalized by Hong et al. [10], who show how to perform more precise versions of slicing and chopping using predicate-abstraction and model checking. However, their methods are intraprocedural, whereas our work addresses interprocedural analysis.

The goal of error projections is to extend the functionality of software model checkers—particularly ones that verify safety properties using predicate abstrac-

tion. Examples include SLAM [1], BLAST [9], KISS [20], MAGIC [4], and ZING [19]. With respect to BLAST and lazy abstraction, the soundness guarantee of an error projection enables the model checker to focus its refinement step on *only* the nodes in the error projection. This can be viewed as an alternative form of lazy abstraction and appears to be a promising avenue for future work.

Kremenek et al. [12] use statistical analysis to rank counterexamples found by the `xgcc`[7] compiler. Their goal is to present to the user an ordered list of counterexamples sorted by their confidence rank.

Mohri et al. investigated the intersection of weighted automata in their work on natural-language recognition [17, 18]. For their weight domains, the extend operation must be commutative. We do not require this restriction.

References

1. T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN*, 2001.
2. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *CONCUR*. Springer-Verlag, 1997.
3. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, 2003.
4. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*, 2003.
5. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.
6. P. Cousot. Méthodes itératives de construction et d'approximation de point fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes. Thèse ès sciences mathématiques, Univ. of Grenoble, 1978.
7. D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.
8. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Elec. Notes in Theoretical Comp. Sci.*, 9, 1997.
9. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
10. H. S. Hong, I. Lee, and O. Sokolsky. Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In *SCAM*, 2005.
11. B. Jeannet and W. Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. In *AMAST*, 2004.
12. T. Kremenek, K. Ashcraft, J. Yang, and D. R. Engler. Correlation exploitation in error ranking. In *SIGSOFT FSE*, 2004.
13. R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.
14. A. Lal, J. Lim, M. Polishchuk, and B. Liblit. Path optimization in programs and its application to debugging. In *ESOP*, 2006.
15. A. Lal and T. Reps. Improving pushdown system model checking. Technical Report 1552, University of Wisconsin-Madison, Jan. 2006.
16. D. Massé. Combining forward and backward analyses of temporal properties. In *PADO*, 2001.
17. M. Mohri, F. C. N. Pereira, and M. Riley. Weighted automata in text and speech processing. In *ECAI*, 1996.

18. M. Mohri, F. C. N. Pereira, and M. Riley. The design principles of a weighted finite-state transducer library. In *Theoretical Computer Science*, 2000.
19. S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *POPL*, 2004.
20. S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *PLDI*, 2004.
21. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *SCP*, 58, 2005.
22. S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technical Univ. of Munich, July 2002.
23. M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

A Proofs

Weighted Chopping. Here we prove the correctness of the following algorithm for solving $\mathrm{WC}(S, C, T)$:

1. $\mathcal{A}_1 = \mathit{poststar}(\mathcal{A}_S)$
2. $\mathcal{A}_2 = (\mathcal{A}_1 \cap \mathcal{A}_C)$
3. $\mathcal{A}_3 = \mathit{poststar}(\mathcal{A}_2)$
4. $\mathcal{A}_4 = \mathcal{A}_3 \cap \mathcal{A}_T$
5. $\mathrm{WC}(S, C, T) = \mathit{path\_summary}(\mathcal{A}_4)$

Proof. From the definition of *poststar*, we know that

$$
\begin{aligned}
\mathcal{A}_2(c) &= \begin{cases} \overline{0} & \text{if } c \notin C \\ \bigoplus\{\mathit{pval}(\sigma_1) \mid s \Rightarrow^{\sigma_1} c, s \in S\} & \text{if } c \in C \end{cases} \\
\Rightarrow \mathcal{A}_3(t) &= \bigoplus\{\mathcal{A}_2(c) \otimes \mathit{pval}(\sigma_2) \mid c \Rightarrow^{\sigma_2} t\} \\
&= \bigoplus\{\bigoplus\{\mathit{pval}(\sigma_1) \otimes \mathit{pval}(\sigma_2) \mid s \Rightarrow^{\sigma_1} c, s \in S\} \mid c \in C, c \Rightarrow^{\sigma_2} t\} \\
&= \bigoplus\{\mathit{pval}(\sigma_1) \otimes \mathit{pval}(\sigma_2) \mid s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t, s \in S, c \in C\} \\
&= \bigoplus\{\mathit{pval}(\sigma_1\sigma_2) \mid s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t, s \in S, c \in C\} \\
\Rightarrow \mathcal{A}_3(T) &= \bigoplus\{\mathit{pval}(\sigma_1\sigma_2) \mid s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t, s \in S, c \in C, t \in T\} \\
&= \mathrm{WC}(S, C, T)
\end{aligned}
$$

B Additional Experiments

The second application we used for experiments is BTRACE, a program-debugging tool [14]. It performs a search on program paths, using WPDSs, to find the shortest one from program entry to the crash site that passes through a given set of program nodes. We do not give the details of the weight domain that it uses, but it should be noted that it is not a finite abstraction.

There is no notion of an "error" projection in this case, but we use BTRACE to benchmark the running time for computing multiple weighted chops. Results are shown in Tab. 3. As with MOPED, using functionals is much faster than using double-pre^* to solve multiple weighted chopping queries. However, unlike with the MOPED experiments, solving weighted-chop queries for all nodes is much slower than the forward or backward analysis. Further optimizations might bring down this overhead.

					$\mathrm{WC}(S, n\Gamma^*, T)$		Functional vs.	
Prog	#Blocks	Error Proj.	$post^*(S)$	$pre^*(T)$	Functional	Double pre^*	Reach	Double pre^*
uucp	4793	n/a	0.14	0.15	2.6	1400	9	538
wget	12047	n/a	0.14	0.35	14	12000	29	857
gawk	19704	n/a	0.86	0.69	210	37000	135	176
mc	33591	n/a	0.47	0.41	47	46000	53	979

Table 3. BTRACE results: The programs are common Unix utilities.