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. In this paper, we extend model-checking technology with the notion of an *error projection*. Given a program abstraction, an error projection divides the program into two parts: the part outside the error projection is guaranteed to be correct, while the part inside the error projection can have bugs. Subsequent automated or manual verification effort need only be concentrated on the part inside the error projection. We present novel algorithms for computing error projections using *weighted pushdown systems* that are sound and complete for the class of Boolean programs and discuss additional applications for these algorithms.

1 Introduction

Software model checkers extract a model from a program using a finite abstraction of data states and then perform reachability analysis on the model. If a property violation is detected, it reports the result back to the user, usually in the form of a counterexample on a failed run, or goes on to refine its abstraction and check again. This technique has been shown to be 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 [2], BLAST [11], and MAGIC [6]. Our goal is to extend the capabilities of model checkers to make maximum possible use of a given abstraction during the reachability check for helping subsequent analysis.

We accomplish this by computing *error projections* and *annotated error projections*. An error projection is the set of program nodes N such that for each node $n \in N$, there exists an error path that starts from the entry point of the program and passes through n. By definition, an error projection describes all of the nodes that are members of paths that lead to a specified error in the model, and no more. 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). Model checkers such as SLAM can then focus their refinement effort on the part of the program inside the projection.

Annotated error projections are an extension of 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 (memory-configuration descriptors) 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.

From a theoretical standpoint, an error projection solves a combination of forward and backward analyses. The forward analysis computes the set of program states S_{fwd} that are reachable from program entry; the backward analysis

computes the set of states S_{bck} that can reach an error at certain pre-specified nodes. Under a sound abstraction of the program, each of these sets provides a strong guarantee: only the states in S_{fwd} can ever arise in the program, and only the states in S_{bck} can ever lead to error. Error projections ask the natural question of combining these guarantees to compute the set of states $S_{\text{err}} = S_{\text{fwd}} \cap S_{\text{bck}}$ containing all states that can both arise during program execution, and lead to error. In this sense, an error projection is making maximum use of the given abstraction—by computing the smallest envelope of states that may contribute to program failure.

Computation of this intersection turns out to be non-trivial because the two sets may be infinite. In §4 and §5, we show how to compute this set efficiently and precisely for common abstractions used for model checking. We use weighted pushdown systems (WPDSs) [5, 21] as the abstract model of a program, which can, among other abstractions, faithfully encode Boolean programs [22]. The techniques that we use seem to be of general interest, and apart from the application of finding error projections, we discuss additional applications in §7.

The contributions of this paper can be summarized as follows:

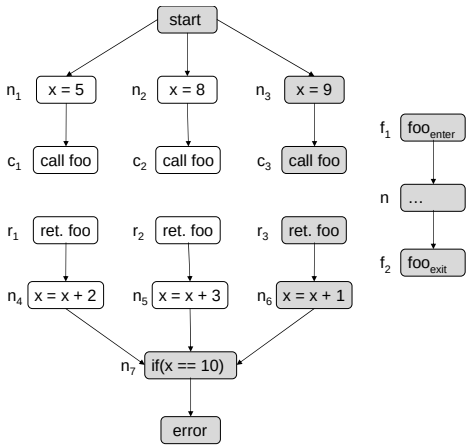
- We define the notions of error projection and annotated error projection. These projections divide the program into a correct and an incorrect part such that further analysis need only be carried out on the incorrect part.
- We give a novel combination of forward and backward analyses for multi-procedural programs using weighted automata and use it for computing (annotated) error projections (§4 and §5). We also show that our algorithms can be used for solving various problems in model checking (§7).
- Our experiments show that we can efficiently compute error projections (§6).

The remainder of the paper is organized as follows: §2 motivates the difficulty in computing (annotated) error projections and illustrates their utility. §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 Examples

Consider the program shown in Fig. 1. Here `x` is a global unsigned integer variable, and assume that procedure `foo` does not change the value of `x`. Also assume that the program abstraction is a Boolean abstraction in which integers (only `x` in this case) are modeled using 8 bits, i.e., the value of `x` can be between 0 and 255 with saturated arithmetic. This type of an abstraction is used by MOPED [22], and happens to be a precise abstraction for this example.

The program has an error if node `error` is reached. The error projection is shaded in the figure. The paths on the left that set the value of `x` to 5 or 8 are correct paths. An error projection need not be restricted to a single trace (which would be the case if `foo` had multiple paths). An annotated error projection will additionally tell us that the value of `x` at node `n` inside `foo` has to be 9 on an error path passing through this node. Note that the value of `x` can be 5 or 8 on other paths that pass through `n`, but they do not lead to the error node.



(a)

(1)	$\langle p, \text{start}\rangle \hookrightarrow \langle p, n_1\rangle$	id
(2)	$\langle p, n_1\rangle \hookrightarrow \langle p, c_1\rangle$	$\{(\_, 5)\}$
(3)	$\langle p, c_1\rangle \hookrightarrow \langle p, f_1\ r_1\rangle$	id
(4)	$\langle p, r_1\rangle \hookrightarrow \langle p, n_4\rangle$	id
(5)	$\langle p, n_4\rangle \hookrightarrow \langle p, n_7\rangle$	$\{(i, i+2)\}$
(6)	$\langle p, \text{start}\rangle \hookrightarrow \langle p, n_2\rangle$	id
(7)	$\langle p, n_2\rangle \hookrightarrow \langle p, c_2\rangle$	$\{(\_, 8)\}$
(8)	$\langle p, c_2\rangle \hookrightarrow \langle p, f_1\ r_2\rangle$	id
(9)	$\langle p, r_2\rangle \hookrightarrow \langle p, n_5\rangle$	id
(10)	$\langle p, n_5\rangle \hookrightarrow \langle p, n_7\rangle$	$\{(i, i+3)\}$
(11)	$\langle p, \text{start}\rangle \hookrightarrow \langle p, n_3\rangle$	id
(12)	$\langle p, n_3\rangle \hookrightarrow \langle p, c_3\rangle$	$\{(\_, 9)\}$
(13)	$\langle p, c_3\rangle \hookrightarrow \langle p, f_1\ r_3\rangle$	id
(14)	$\langle p, r_3\rangle \hookrightarrow \langle p, n_6\rangle$	id
(15)	$\langle p, n_6\rangle \hookrightarrow \langle p, n_7\rangle$	$\{(i, i+1)\}$
(16)	$\langle p, n_7\rangle \hookrightarrow \langle p, \text{error}\rangle$	$\{(10, 10)\}$
(17)	$\langle p, f_1\rangle \hookrightarrow \langle p, n\rangle$	id
(18)	$\langle p, n\rangle \hookrightarrow \langle p, f_2\rangle$	id
(19)	$\langle p, f_2\rangle \hookrightarrow \langle p, \varepsilon\rangle$	id

(b)

Fig. 1. (a) An example program and (b) its corresponding WPDS. Weights, shown in the last column, are explained in §3.

It is non-trivial to conclude the above value of `x` for node `n`. An interprocedural forward analysis starting from `start` will show that the value of `x` is in the set $\{5, 8, 9\}$ at node `n`. A backward interprocedural analysis starting from `error` concludes that the value of `x` at `n` has to be in the set $\{7, 8, 9\}$. Intersecting the sets obtained from forward and backward analysis only gives an over-approximation of the annotated error projection values. In this case, the intersection is $\{8, 9\}$, but `x` can never be 8 on a path leading to `error`. The over-approximation occurs because, in the forward analysis, the value of `x` is 8 only when the call to `foo` occurs at call site c_2, but in the backward analysis the path starting at `n` with $\texttt{x} = 8$ and leading to `error` must have had the call to `foo` from call site c_1.

Such a complication also occurs while computing (non-annotated) error projections: to see this, assume that the edge leading to node `n` is predicated by the condition `if(x!=9)`. Then, node `n` can be reached from `start`, and there is a path starting at `n` that leads to `error`, but both of these cannot occur together. Formally, a node is in the error projection if and only if the associated value set computed for the annotated projection is non-empty. In this sense, computing an error projection is a special case of computing the annotated version. We still discuss error projections separately because (i) computing them is easier, as we see later (computing annotations requires one extra trick), and (ii) they can very easily be cannibalized by existing model checkers such as SLAM in their abstraction-refinement phase: when an abstraction needs to be refined, only the portion inside the error projection needs to be rechecked. We illustrate this point in more detail in the next example.

Fig. 2 shows an example program and several abstractions that SLAM might produce. This example is given in [3] to illustrate the SLAM refinement process.

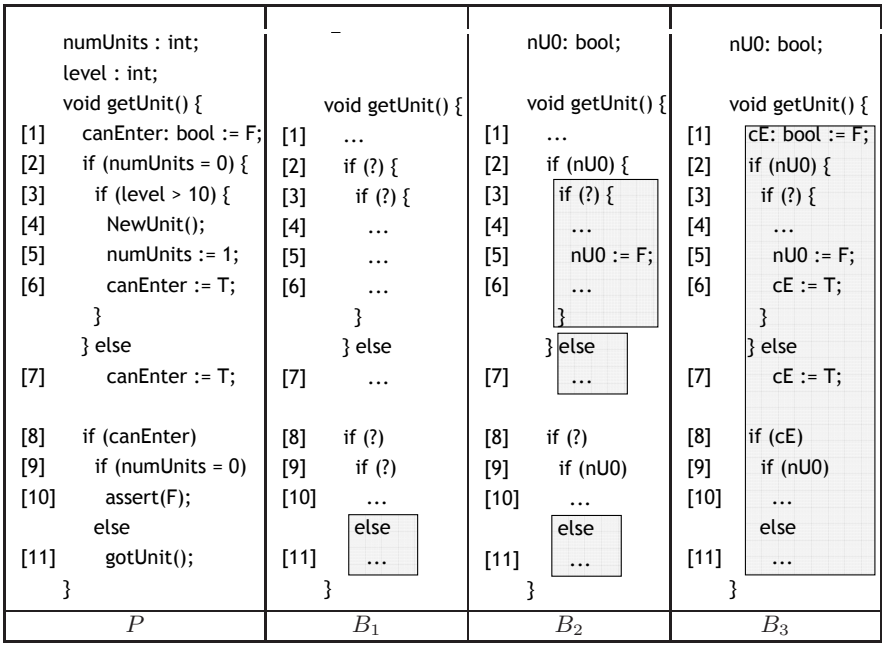


Fig. 2. An example program P and its abstractions as Boolean programs. The "$\cdots$" represents a "skip" or a no-op. The part outside the error projection is shaded in each case.

SLAM uses predicate abstraction to create Boolean programs that abstract the original program. Boolean programs are characterized as imperative programs with procedure calls and only Boolean variables (and no heap). The Boolean programs produced as a result of predicate abstraction have one Boolean variable per predicate that tracks the value of that predicate in the program. SLAM creates successive approximations of the original program by adding more predicates. We show the utility of error projections for abstraction refinement.

First, we describe the SLAM refinement process. In Fig. 2, the property of interest is the assertion on line 10. We want to verify that line 10 is never reached (because the assertion always fails). The first abstraction B_1 is created without any predicates. It only reflects the control structure of P. Reachability analysis on B_1 (assuming `getUnit` is program entry) shows that the assertion is reachable. This results in a counterexample, whose subsequent analysis reveals that the predicate $\{$`numUnits = 0`$\}$ is important. Program B_2 tracks that predicate using variable `nU0`. Reachability analysis on B_2 reveals that the assertion is still reachable. Now predicate $\{$`canEnter = T`$\}$ is added, to produce B_3, which tracks the predicate's value using variable `cE`. Reachability analysis on B_3 reveals that the assertion is not reachable, hence it is not reachable in P.

The advantage of using error projections is that the whole program need not be abstracted when a new predicate is added. Analysis on B_1 and B_2 fails to prove that the whole program is correct, but error projections may reveal that at least some part of the program is correct. The parts outside the error

projections (and hence correct) are shaded in the figure. Error projection on B_1 shows that line 11 cannot contribute to the bug, and need not be considered further. Therefore, when constructing B_2, we need not abstract that statement with the new predicate. Error projection on B_2 further reveals that lines 3 to 6 and line 7 do not contribute to the bug (the empty else branch to the conditional at line 3 still can). Thus, when B_3 is constructed, this part need not be abstracted with the new predicate. B_3, with the shaded region of B_2 excluded, reduces to a very simple program, resulting in reduced effort for its construction and analysis.

Annotated error projections can further reduce the analysis cost. Suppose there was some code between lines 1 and 2, possibly relevant to proving the program to be correct, that does not modify `numUnits`. After constructing B_2, the annotated error projection would tell us that in this region of code, `nU0` can be assumed to be *true*, because otherwise the assertion cannot be reached. This might save half of the theorem prover calls needed to abstract that region of code when using multiple predicates.

While this example did not require an interprocedural analysis, placing any piece of code inside a procedure would necessitate its use. Because Boolean programs are a common abstract model used by model checkers, we devise techniques to compute error projections precisely and efficiently on them. For this, we use weighted pushdown systems.

3 Preliminary Definitions

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 [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. 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.*

If C is a regular set of configurations then both $post^*(C)$ and $pre^*(C)$ are also regular sets of configurations [10, 4, 22]. The algorithms for computing $post^*$ and pre^* take a $\mathcal{P}$-automaton $\mathcal{A}$ as input, and if C is the set 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 are regular.

A weighted pushdown system (WPDS) is a PDS augmented with a weight domain that is a bounded idempotent semiring [5, 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 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* $f : \Delta \rightarrow D$ *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.) In §5 we use the weight domain 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.*

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 [16]). 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. They can usually be represented succinctly using BDDs. (This is the essence of Schwoon's MOPED system [22].)

For the program shown in Fig. 1 and an 8-bit integer abstraction (explained in §2), the WPDS uses a relational weight domain over the set $V = \{0, 1, \cdots, 255\}$. The weight $\{(\_, 5)\}$ is shorthand for the set $\{(i, 5) \mid i \in V\}$; $\{(i, i+1)\}$ stands for $\{(i, i+1) \mid i \in V\}$ (with saturated arithmetic); and *id* stands for the identity relation on V.

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 WPDS $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, 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. 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 *poststar*$(\mathcal{A})$, such that *poststar*$(\mathcal{A})(c) = \bigoplus\{\mathcal{A}(c') \otimes \mathit{pval}(\sigma) \mid c' \Rightarrow^{\sigma} c\}$. Similarly, the prestar algorithm takes a forward weighted automaton $\mathcal{A}$ and produces *prestar*$(\mathcal{A})$ such that *prestar*$(\mathcal{A})(c) = \bigoplus\{\mathit{pval}(\sigma) \otimes \mathcal{A}(c') \mid c \Rightarrow^{\sigma} c'\}$.

We briefly describe how the prestar algorithm works. The interested reader is referred to [21] for more details, and an efficient implementation of the algorithm. The algorithm takes a weighted automaton $\mathcal{A}$ as input, and adds weighted transitions to it until no more can be added. The addition of transitions is based on the following rule: for a WPDS rule $r = \langle p, \gamma \rangle \hookrightarrow \langle q, \gamma_1 \gamma_2 \cdots \gamma_n \rangle$ with weight $f(r)$ and transitions $(q, \gamma_1, q_1), (q_1, \gamma_2, q_2), \cdots, (q_{n-1}, \gamma_n, q_n)$ with weights $w_1, w_2, \cdots, w_n$, add the transition (p, γ, q_n) to $\mathcal{A}$ with weight $w = f(r) \otimes w_1 \otimes \cdots \otimes w_n$. If this transition already exists with weight w', change the weight to $w \oplus w'$. This algorithm is based on the intuition that if the automaton accepts configurations

c and c' with weights w and w', respectively, and rule r allows the transition $c' \Rightarrow c$, then the automaton is changed to accept c' with weight $w' \oplus (f(r) \otimes w)$. Termination follows from the fact that the number of states of the automaton does not increase (hence, the number of transitions is bounded), and that the weight domain satisfies the ascending-chain condition.

An important algorithm for reading out weights from weighted automata is called *path_summary* defined as follows: $\mathit{path\_summary}(\mathcal{A}) = \oplus\{\mathcal{A}(c) \mid c \in P \times \Gamma^*\}$. 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 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 $\mathit{path\_summary}(\mathcal{A})$ 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 $\mathcal{A}'$.[3] Then it is easy to see that $\mathcal{A}(C) = \mathit{path\_summary}(\mathcal{A}')$. 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 $\mathrm{JOP}(S,T) = \mathit{poststar}(\mathcal{A}_S)(T) = \mathit{prestar}(\mathcal{A}_T)(S)$. For the program shown in Fig. 1, parts of the automata produced by *poststar*($\{\texttt{start}\}$) and *prestar*(`error` Γ^*) are shown in Fig. 3 (only the part important for node `n` is shown).[4] Using these, we get $\mathrm{JOP}(\{\texttt{start}\}, \texttt{n}\ \Gamma^*) = \{(\_,5),(\_,8),(\_,9)\}$ and $\mathrm{JOP}(\texttt{n}\ \Gamma^*, \texttt{error}\ \Gamma^*) = \{(7,10),(8,10),(9,10)\}$. Here, $(\gamma\ \Gamma^*)$ stands for the set $\{\gamma\ c \mid c \in \Gamma^*\}$.

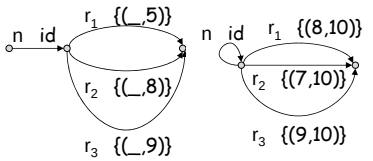


Fig. 3. Parts of the *poststar* and *prestar* automaton, respectively.

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

[3] 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.

[4] Intuitively, for the *poststar* automaton, the weight on a transition labeled with γ is the net transformer to go from the entry of the procedure containing γ to γ. For the *prestar* automaton, it is the transformer to go from γ to the exit of the procedure.

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` Γ^*). 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 $pval(\sigma)$ 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.

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 $\gamma \in \mathrm{EP}(S,T)$ if and only if $\mathrm{WC}(S, \gamma\ \Gamma^*, T) \neq \overline{0}$. We now show how to solve these queries. First, note that $\mathrm{WC}(S,C,T) \neq \mathrm{JOP}(S,C) \otimes \mathrm{JOP}(C,T)$. For example, in Fig. 1, if `foo` was not called from `c`$_3$, and $S = \{$`start`$\}, T = ($`error` $\Gamma^*), C = ($`n` $\Gamma^*)$ then $\mathrm{JOP}(S,C) = \{(\_,5),(\_,8)\}$ and $\mathrm{JOP}(C,T) = \{(7,10),(8,10)\}$, and their extend is non-empty, whereas $\mathrm{WC}(S,C,T) = \emptyset$. This is exactly the problem mentioned in §2.

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

The running time is only proportional to the size of $\mathcal{A}_C$, not the size of the language accepted by it. A proof of correctness can be found in [15].

An error projection is computed by solving 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 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 $(post^*(S) \cap pre^*(T)) \cap C \neq \emptyset$. This procedure just requires a single intersection operation for different chop sets. Computation of both $post^*(S)$ and $pre^*(T)$ have to be done just once. We generalize this approach to the weighted case.

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 $(\mathcal{A}_1 \lhd \mathcal{A}_2)(c) = \mathcal{A}_1(c) \otimes \mathcal{A}_2(c)$. [5] 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 $\mathcal{A}_1$ 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 $(\mathcal{A}_1 \lhd \mathcal{A}_2)(c) = \mathcal{A}_{12}(c\ c)$. However, computing $(\mathcal{A}_1 \lhd \mathcal{A}_2)(C)$ 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 linear 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 automata-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

[5] 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.

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 this type of weight a *functional weight* and 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 thus also a functional weight. However, the 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 now show that this is not a handicap, and we can still compute $\mathcal{A}_b \lhd \mathcal{A}_f$ 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 fact, we get 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 the weights on the 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 $(\mathcal{A}_b \lhd \mathcal{A}_f)$.

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. Fig. 4 shows an example. For short, the weight $\lambda z.(w_1 \otimes z \otimes w_2)$ is denoted by $[w_1.z.w_2]$. Note how the weights get appropriately paired for different call sites.

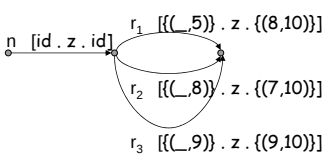


Fig. 4. Functional automaton obtained after intersecting the automata of Fig. 3.

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. Due to space constraints, we do not discuss the first part here. It can be found in [15].

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 approximation algorithms are given in [15]). Note that the value of $\mathrm{WC}(S, C, T)$ does not say anything about the required set of abstract stores at C: for Fig. 1, $\mathrm{WC}(S, \mathtt{n}\ \Gamma^*, T) = \{(\_, 10)\}$ but the required abstract store at $\mathtt{n}$ is $\{9\}$.

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 $\mathtt{Rng}(w)$ to be the range of w, $\mathtt{Dom}(w)$ to be the domain of w and $\mathtt{Com}(w_1, w_2) = \mathtt{Rng}(w_1) \cap \mathtt{Dom}(w_2)$. For a node $\gamma \in \mathrm{EP}(S, T)$, we compute the following subset of V: $V_\gamma = \{v \in \mathtt{Com}(pval(\sigma_1), pval(\sigma_2)) \mid s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t, s \in S, c \in \gamma\Gamma^*, t \in T\}$. 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.

An Explicit Algorithm. First, we show how to check for membership in the set V_γ. 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 $\mathcal{A}_{post^*} = \mathit{poststar}(\mathcal{A}_S), \mathcal{A}_{pre^*} = \mathit{prestar}(\mathcal{A}_T)$ and $\mathcal{A}_\lhd$ 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 $\mathcal{A}_\lhd$ again. It is not hard to check that the following holds for any weight w:

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

Then $v \in V_\gamma$ iff $\bigoplus\{W(w_v) \mid \mathit{accpath}(\mathcal{A}_\lhd, \gamma\Gamma^*, W)\} \neq \overline{0}$. This is, again, computable using *path_summary*: Intersect $\mathcal{A}_\lhd$ with an unweighted automaton accepting $\gamma\Gamma^*$, then run *path_summary* but 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. In the case of predicate abstraction, $|V|$ is exponential in the number of predicates, but the weights (transformers) can be efficiently encoded using BDDs. For example, the identity transformer on V can be encoded with a BDD of size $\log |V|$. To avoid losing the advantages of using BDDs, we now present a symbolic algorithm.

A Symbolic Algorithm. Let $Y = \{y_v \mid v \in V\}$ be a set of variables. We switch our weight domain from being $V \times V$ to $V \times Y \times V$. We write weights in the new domain with superscript e. Intuitively, the triple (v_1, y, v_2) denotes the transformation of v_1 to v_2 provided the variable y is "true". Combine is still defined to be union and extend is defined as follows: $w_1^e \otimes w_2^e = \{(v_1, y, v_2) \mid (v_1, y, v_3) \in w_1^e, (v_3, y, v_2) \in w_2^e\}$. Also, $\overline{1}^e = \{(v, y, v) \mid v \in V, y \in Y\}$ and $\overline{0}^e = \emptyset$. Define a symbolic identity id_s^e as $\{(v, y_v, v) \mid v \in V\}$. Let $\texttt{Var}(w^e) = \{v \mid (v_1, y_v, v_2) \in w^e \text{ for some } v_1, v_2 \in V\}$, i.e., the set of values whose corresponding variable appears in w^e. Given a weight in $V \times V$, define $\texttt{ext}(w) = \{(v_1, y, v_2) \mid (v_1, v_2) \in w, y \in Y\}$, i.e., all variables are added to the middle dimension. We will use the middle dimension to remember the "history" when composition is performed: for weights $w_1, w_2 \in V \times V$, it is easy to prove that $\texttt{Com}(w_1, w_2) = \texttt{Var}(\texttt{ext}(w_1) \otimes \mathrm{id}_s^e \otimes \texttt{ext}(w_2))$. Therefore, $V_\gamma = \texttt{Var}(w_\gamma^e)$ where, $w_\gamma^e = \bigoplus\{\texttt{ext}(\mathit{pval}(\sigma_1)) \otimes \mathit{id}_s^e \otimes \texttt{ext}(\mathit{pval}(\sigma_2)) \mid s \Rightarrow^{\sigma_1} c \Rightarrow^{\sigma_2} t, s \in S, c \in \gamma\Gamma^*, t \in T\}$. This weight is computed by replacing all weights w in the functional automaton with $\texttt{ext}(w)$ and running *path_summary* over paths accepting $\gamma\Gamma^*$, and initializing initial states with weight id_s^e. The advantages of this algorithm are: the weight $\texttt{ext}(w)$ can be represented using the same-sized BDD as the one for w (the middle dimension is "don't-care"); and the weight id_s^e can be represented using a BDD of size $O(\log |V|)$.

For our example, the weight $w_\mathtt{n}^e$ read off from the functional automaton shown in Fig. 4 is $\{(\_, y_9, 10)\}$, which gives us $V_\mathtt{n} = \{9\}$, as desired.

6 Experiments

We added the error-projection algorithm to MOPED [22], a program-analysis tool that encodes Boolean programs as WPDSs and answers reachability queries on them for checking assertions. 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.

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. 1. 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.

					$\mathrm{WC}(S, n\Gamma^*, T)$		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 1. MOPED results: The WPDSs are models of Boolean programs provided by S. Schwoon. S is the entry point of the program, and T is the error configuration set. An error projection of size 0% means that the program is correct.

Discussion. 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 in 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.

The sizes of the error projections indicate that they might be useful in model checkers. Simple slicing, which only deals with the control structure of the program (and no weights) produced more than 99% of the program in each case, even when the program was correct.

The result for the last program `bugs5`, however, does not seem as encouraging due to the large size of the error projection. We do not have the source code for this program, but investigating the model reveals that there is a loop that calls

into most of the code, and the error can occur inside the loop. If the loop resets its state when looping back, the error projection would include everything inside the loop or called from it. This is because for every node, there is a path from the loop head that goes through the node, then loops back to the head, with the same data state, and then goes to error.

This seems to be a limitation of error projections and perhaps calls for similar techniques that only focus on acyclic paths (paths that do not repeat a program state). However, for use inside a refinement process, error projections still give the minimal set of nodes that is sound with respect to the property being verified (focusing on acyclic paths need not be sound, i.e., the actual path that leads to error might actually be cyclic in an abstract model).

7 Additional Applications

The techniques presented in §4 and §5 give rise to several other applications of our ideas in model checking. Let $\mathrm{BW}(w_{\mathrm{bot}}, \gamma)$ be the weight obtained from the functional automaton intersected with $(\gamma\ \Gamma^*)$ and bottleneck weight w_{bot} (as used in §5). This weight can be computed for all nodes γ in roughly the same time as the error projection (which computes $\mathrm{BW}(\overline{1}, \gamma)$).

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, KISS adds nondeterministic function calls to the `main` method of process 2 after each statement of process 1. Likewise it adds nondeterministic function returns after each statement of process 2. It also ensures 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 for two context switches.

Using our techniques, we can extend KISS to 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 KISS does and then compute the error projection. However, due to the automata-theoretic techniques we employ, we can omit the extra 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. Let $w = path\_summary(\mathcal{A}_2)$; w represents the state transformation caused by the execution steps spent in process 2.
3. For each program node γ of process 1, let $w_\gamma = \mathrm{BW}(w, \gamma)$ be the weight obtained from functional automaton $\mathcal{A}_{\lhd}$ of process 1. If $w_\gamma \neq \overline{0}$ then an error can occur in the program when the first context switch occurs at node γ in process 1.

Error reporting. The model checker SLAM [2] used a technique presented in [1] to identify error causes from counterexample traces. The main idea was to remove "correct" transitions from the error trace and the remaining transi-

tions indicate the cause of the error. These correct transitions were obtained by a backward analysis from non-error configurations. However, no restrictions were imposed that these transitions also be reachable from the entry point of the program. Using annotated error projections, we can limit the correct transitions to ones that are both forward reachable from program entry and backward reachable from the non-error configurations.

8 Related Work

The combination of forward and backward analysis has a long history in abstract interpretation, going back to Cousot's thesis [8]. It has been also been used in model checking [17] and in interprocedural analysis [13]. In the present paper, we show how forward and backward approaches can be combined precisely in the context of interprocedural analysis performed with WPDSs; our experiments show that this approach is significantly faster than a more straightforward one.

With model checkers becoming more popular, there has been considerable work on explaining the results obtained from a model checker in an attempt to localize the fault in the program [7, 1]. These approaches are complimentary to ours. They build on information obtained from reachability analysis performed by the model checker and use certain heuristics to isolate the root cause of the bug. Error projections seek to maximize information that can be obtained from the reachability search so that other tools can take advantage of this gain in precision. This paper focused on using error projections inside an abstraction refinement loop. The third application in §7 briefly shows how they can be used for fault localization. It would be interesting to explore further use of error projections for fault localization.

Such error-reporting techniques have also been used outside model checking. Kremenek et al. [14] use statistical analysis to rank counterexamples found by the `xgcc`[9] compiler. Their goal is to present to the user an ordered list of counterexamples sorted by their confidence rank.

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. These techniques have been generalized by Hong et al. [12], 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.

Mohri et al. investigated the intersection of weighted automata in their work on natural-language recognition [18, 19]. For their weight domains, the extend operation must be commutative. We do not require this restriction.

References

1. T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL*, 2003.

2. T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN*, 2001.
3. T. Ball and S. K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, 2000.
4. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *CONCUR*. Springer-Verlag, 1997.
5. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, 2003.
6. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*, 2003.
7. S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. In *FSE*, 2004.
8. 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.
9. D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.
10. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Elec. Notes in Theoretical Comp. Sci.*, 9, 1997.
11. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
12. 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.
13. B. Jeannet and W. Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. In *AMAST*, 2004.
14. T. Kremenek, K. Ashcraft, J. Yang, and D. R. Engler. Correlation exploitation in error ranking. In *SIGSOFT FSE*, 2004.
15. A. Lal, N. Kidd, T. Reps, and T. Touili. Abstract error projection. Technical Report 1579, University of Wisconsin-Madison, Jan. 2007.
16. A. Lal and T. Reps. Improving pushdown system model checking. Technical Report 1552, University of Wisconsin-Madison, Jan. 2006.
17. D. Massé. Combining forward and backward analyses of temporal properties. In *PADO*, 2001.
18. M. Mohri, F. C. N. Pereira, and M. Riley. Weighted automata in text and speech processing. In *ECAI*, 1996.
19. M. Mohri, F. C. N. Pereira, and M. Riley. The design principles of a weighted finite-state transducer library. In *Theoretical Computer Science*, 2000.
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, Tech. Univ. Munich, 2002.
23. M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.