Improving Pushdown System Model Checking

Akash Lal
Thomas Reps

Technical Report #1552

February 2006

Improving Pushdown System Model Checking

Akash Lal and Thomas Reps

University of Wisconsin, Madison, Wisconsin 53706
{akash, reps}@cs.wisc.edu

Abstract. In this paper, we reduce pushdown system (PDS) model checking to a graph-theoretic problem, and apply a fast graph algorithm to improve the running time for model checking. We use *weighted* PDSs as a generalized setting for PDS model checking, and show how various PDS model checkers can be encoded using weighted PDSs. We also give algorithms for witness tracing, differential propagation, and incremental analysis, each of which benefits from the fast graph-based algorithm.

1 Introduction

Pushdown systems (PDSs) have served as an important formalism for program analysis and verification because of their ability to concisely capture interprocedural control flow in a program. Various tools [1–6] use pushdown systems as an abstract model of a program and use reachability analysis on these models to verify program properties. Using PDSs provides an infinite-state abstraction for the control state of the program. Some of these tools [1, 2, 6], however, can only verify properties that have a finite-state data abstraction. Other tools [4, 5, 3] are based on the more generalized setting of weighted pushdown systems (WPDSs) [7] and are capable of verifying infinite-state data abstractions as well.

At the heart of all these tools is a PDS reachability-analysis algorithm that uses a chaotic-iteration strategy to explore all reachable states [8–10]. Even though there has been work to address the worst-case running time of this algorithm [11], to our knowledge, no one has addressed the issue of giving direction to the chaotic-iteration scheme to improve the running time of the algorithm in practice. In this paper, we try to improve the worst-case running time as well the running-time observed in practice. To provide a common setting to discuss most PDS model checkers, we use WPDSs to describe our improvements to PDS reachability.

An interprocedural control flow graph (ICFG) is a set of graphs, one per procedure, connected via special call and return edges [12]. A WPDS with a given initial query can also be decomposed into a set of graphs whose structure is similar. (When the underlying PDS is obtained by the standard encoding of an ICFG as a PDS for use in program analysis, these decompositions coincide.) Next, we use a fast graph algorithm, namely the Tarjan path-expression algorithm [13] to represent each graph as a regular expression. WPDS reachability can then be reduced to solving a set of regular equations. When the underlying PDS is obtained from a structured (reducible) control flow graph, the regular expressions can be found and solved very efficiently. Even when the control flow is not structured, the regular expressions provide a fast iteration strategy that improves over the standard chaotic-iteration strategy.

Our work is inspired by previous work on dataflow analysis of single-procedure programs [14]. There it was shown that a certain class of dataflow analysis problems can take advantage of the fact that a (single-procedure) CFG can be represented using a regular expression. We generalize this observation to multi-procedure programs, as well as to WPDSs. The contributions of this paper can be summarized as follows:

- We present a new reachability algorithm for WPDSs that improves on previously known algorithms for PDS reachability. The algorithm is asymptotically faster when the PDS is *regular* (decomposes into a single graph) and offers substantial improvement in the general case as well.
- The algorithm is completely demand driven and computes only that information needed for answering a particular user query. The algorithm can be easily parallelized (unlike the chaotic-iteration strategy) to take advantage of multiple processors, making it attractive to run on the coming generations of CMPs.
- We show that several PDS analysis questions and techniques carry over to the new approach. In particular, we describe how to perform witness tracing, differential propagation, and incremental analysis.

The rest of the paper is organized as follows: §2 provides background on PDSs and WPDSs. §3 presents the previously known algorithm and our new algorithm for solving reachability queries on WPDSs. In §4, we describe algorithms for witness tracing, differential propagation, and incremental analysis. §5 presents experimental results. §6 describes related work.

2 PDS Model Checking

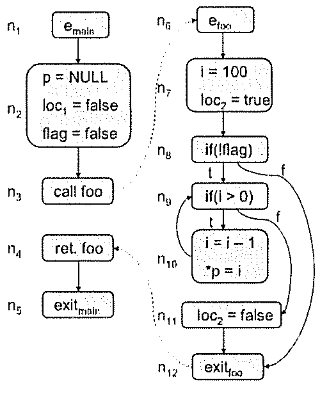
In this section, we review existing pushdown system model checkers, as well as weighted pushdown systems. We also show how the model checkers can be encoded using WPDSs.

2.1 Pushdown Systems

Definition 1. *A* **pushdown system** *is a triple $\mathcal{P} = (P, \Gamma, \Delta)$ where P is the set of states or control locations, Γ is the set of stack symbols and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is the set of pushdown rules. A* **configuration** *of $\mathcal{P}$ is a pair $\langle p, u\rangle$ where $p \in P$ and $u \in \Gamma^*$. A rule $r \in \Delta$ is written as $\langle p, \gamma\rangle \hookrightarrow_{\mathcal{P}} \langle p', u\rangle$ where $p, p' \in P$, $\gamma \in \Gamma$ and $u \in \Gamma^*$. These rules define a transition relation $\Rightarrow_{\mathcal{P}}$ on configurations of $\mathcal{P}$ as follows: If $r = \langle p, \gamma\rangle \hookrightarrow_{\mathcal{P}} \langle p', u\rangle$ then $\langle p, \gamma u'\rangle \Rightarrow_{\mathcal{P}} \langle p', uu'\rangle$ for all $u' \in \Gamma^*$. The subscript $\mathcal{P}$ on the transition relation is omitted when it is clear from the context. 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 $post^*(C) = \{c' \mid \exists c \in C : c \Rightarrow^* c'\}$, which are just backward and forward reachability under the transition relation $\Rightarrow$.*

We restrict the pushdown rules to have at most two stack symbols on the right-hand side. This means that for every rule $r \in \Delta$ of the form $\langle p, \gamma\rangle \hookrightarrow_{\mathcal{P}} \langle p', u\rangle$, we have $|u| \leq 2$. This restriction does not decrease the power of pushdown systems because by increasing the number of stack symbols by a constant factor, an arbitrary pushdown system can be converted into one that satisfies this restriction [15, 16, 10].

The standard approach for modeling program control flow is as follows: Let $(\mathcal{N}, \mathcal{E})$ be an ICFG where each *call* node is split into two nodes: one has an interprocedural edge going to the entry node of the procedure being called; the second has an incoming edge from the exit node of the procedure. $\mathcal{N}$ is the set of nodes in this graph and $\mathcal{E}$ is the set of control-flow edges. Fig. 1(a) shows an example of an ICFG, Fig. 1(b) shows the pushdown system that models it. The PDS has a single state p, one stack symbol for each node in $\mathcal{N}$, and one rule for each edge in $\mathcal{E}$. We use rules with one stack symbol on the right-hand side to model intraprocedural edges, rules with two stack symbols on the right-hand side for *call* edges, and rules with no stack symbols on the right-hand side for *return* edges. It is easy to see that a valid path in the program corresponds to a path in the pushdown system's transition system, and vice versa. Thus, PDSs can encode ordinary control flow graphs, but they also provide a convenient mechanism for modeling certain kinds of non-local control flow. For example, we can model setjmp/longjmp in C programs. At setjmp, we push a special symbol on the stack, and at a longjmp with the same environment variable (identified using some preprocessing) we pop the stack until that symbol is reached. The longjmp value can be passed using the state of the PDS.



(a)

(1)	$\langle p, n_1 \rangle \hookrightarrow \langle p, n_2 \rangle$
(2)	$\langle p, n_2 \rangle \hookrightarrow \langle p, n_3 \rangle$
(3)	$\langle p, n_3 \rangle \hookrightarrow \langle p, n_6\, n_4 \rangle$
(4)	$\langle p, n_4 \rangle \hookrightarrow \langle p, n_5 \rangle$
(5)	$\langle p, n_5 \rangle \hookrightarrow \langle p, \varepsilon \rangle$
(6)	$\langle p, n_6 \rangle \hookrightarrow \langle p, n_7 \rangle$
(7)	$\langle p, n_7 \rangle \hookrightarrow \langle p, n_8 \rangle$
(8)	$\langle p, n_8 \rangle \hookrightarrow \langle p, n_9 \rangle$
(9)	$\langle p, n_8 \rangle \hookrightarrow \langle p, n_{12} \rangle$
(10)	$\langle p, n_9 \rangle \hookrightarrow \langle p, n_{10} \rangle$
(11)	$\langle p, n_9 \rangle \hookrightarrow \langle p, n_{11} \rangle$
(12)	$\langle p, n_{10} \rangle \hookrightarrow \langle p, n_9 \rangle$
(13)	$\langle p, n_{11} \rangle \hookrightarrow \langle p, n_{12} \rangle$
(14)	$\langle p, n_{12} \rangle \hookrightarrow \langle p, \varepsilon \rangle$

(b)

Fig. 1. (a) An interprocedural control flow graph. The e and *exit* nodes represent entry and exit points of procedures, respectively. `flag` is a global variable, `loc1` and `loc2` are local variables of `main` and `foo`, respectively. Dashed edges represent interprocedural control flow. (b) A pushdown system that models the control flow of the graph shown in (a).

A rule $r = \langle p, \gamma \rangle \hookrightarrow \langle p', u \rangle$ is called a *pop* rule if $|u| = 0$, and a *push* rule if $|u| = 2$.

Because the number of configurations of a pushdown system 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 called* **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 [10, 8, 11, 9].

2.2 Verifying Finite-State Properties

In this section, we describe two common approaches to verifying finite-state properties using pushdown systems. The first approach tries to verify safety properties on programs. The property is supplied as a finite-state automaton that performs transitions on ICFG nodes. The automaton has a designated error state, and runs (i.e., ICFG paths) that drive it to the error state are reported as potentially erroneous program executions. The automaton shown in Fig. 2 can be used to verify the absence of null-pointer dereferences (for pointer p in the program) by matching automaton edge labels against ICFG nodes. For example, we would associate p = NULL with node n_2, $\star$p with node n_{10} etc.

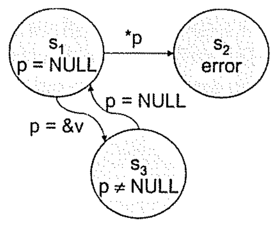


Fig. 2. A finite-state machine for checking null-pointer dereferences in a program. The initial state of the machine is s_1. The label "p = &v" stands for the assignment of a non-null address to the pointer p. We assume that the machine stays in the same state when it has to transition on an undefined label.

Actual program executions are modeled using a PDS as constructed in the previous section from an ICFG. The safety property is verified using the *cross-product* of the automaton with the PDS, which is constructed as follows: For each rule $\langle p, \gamma\rangle \hookrightarrow \langle p', u\rangle$ in the PDS and transition $s_1 \xrightarrow{\gamma} s_2$ in the automaton, add the rule $\langle (p, s_1), \gamma\rangle \hookrightarrow \langle (p', s_2), u\rangle$ to a new PDS. If we can reach a configuration in the new PDS where the automaton error state appears in the second component of the stack then the program can have invalid executions.

The second PDS model-checking approach is used for assertion checking in Boolean programs. In this approach, the PDS state and stack alphabet are expanded to encode valuations of Boolean variables. The state space is expanded to include valuations of global variables, and the stack alphabet is expanded to include valuations of local variables. We illustrate this approach using the program shown in Fig. 1. It has three Boolean variables, flag, which is a global variable, and $\mathtt{loc}_i$, $i = 1, 2$, which are local variables. A valuation for these variables can be described by a pair of bits (a, b), standing for flag $= a$ and $\mathtt{loc}_i = b$, where i indicates the procedure from which the valuation was chosen. Each ICFG edge is associated with a transformer. A transformer is simply a relation between valuations that encodes (an over-approximation of) the effect of following that ICFG edge. For example, the edge (n_2, n_3), which changes the value of flag to 0, can be associated with the relation $\{((a, b), (0, b)) \mid (a, b) \in \{0, 1\}^2\}$. A PDS for the program is constructed from the one describing its control flow as follows: For an intraprocedural rule $\langle p, \gamma\rangle \hookrightarrow \langle p, \gamma'\rangle$ that describes control flow, if R is the transformer associated with edge (γ, γ'), then add rules $\langle a, (\gamma, b)\rangle \hookrightarrow \langle c, (\gamma', d)\rangle$ for each $((a, b), (c, d)) \in R$ to the new PDS. For a push rule $\langle p, \gamma\rangle \hookrightarrow \langle p, \gamma'\gamma''\rangle$, add rules $\langle a, (\gamma, b)\rangle \hookrightarrow \langle c, (\gamma', e)(\gamma'', d)\rangle$ for each $((a, b), (c, d)) \in R$ and $e \in \{0, 1\}$ (assuming that local variables are not initialized on procedure entry). For pop rules $\langle p, \gamma \rangle \hookrightarrow \langle p, \varepsilon \rangle$, add rules $\langle a, (\gamma, b) \rangle \hookrightarrow \langle a, \varepsilon \rangle$ for each $(a, b) \in \{0,1\}^2$.[1]

The PDS obtained from such a construction serves as a faithful model of the Boolean program. Reachability analysis in this PDS can be used for verifying assertions in the program. For example, to see if node n can ever be reached in a program execution, we can ask if a configuration $\langle a, (n, b)\ u \rangle$ is reachable in the PDS for some values of $a, b \in \{0, 1\}$ and $u \in \Gamma^*$.

Note that the two approaches described above are complementary and can be used together to verify safety properties on Boolean programs.

2.3 Weighted Pushdown Systems

A weighted pushdown system is obtained by supplementing a pushdown system with a weight domain that is a bounded idempotent semiring [17, 18]. Such semirings are powerful enough to encode infinite-state data abstractions, such as copy-constant propagation and affine-relation analysis [3].

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

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 $f : \Delta \longrightarrow D$ 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 we define $v(\sigma) \overset{\text{def}}{=} f(r_1) \otimes \ldots \otimes f(r_k)$. Moreover, for any two configurations c and c' of $\mathcal{P}$, we use $path(c, c')$ to denote the set of all rule sequences $[r_1, \ldots, r_k]$ that transform c into c'. Reachability problems on pushdown systems are generalized to weighted pushdown systems as follows.

Definition 5. *Let $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ be a weighted pushdown system, where $\mathcal{P} = (P, \Gamma, \Delta)$, and let $C \subseteq P \times \Gamma^*$ be a regular set of configurations. The* **generalized pushdown predecessor** (GPP) **problem** *is to find for each $c \in P \times \Gamma^*$:*

$$\delta(c) \overset{\text{def}}{=} \bigoplus \{\, v(\sigma) \mid \sigma \in path(c, c'), c' \in C \,\}$$

The **generalized pushdown successor** (GPS) **problem** *is to find for each $c \in P \times \Gamma^*$:*

$$\delta(c) \overset{\text{def}}{=} \bigoplus \{\, v(\sigma) \mid \sigma \in path(c', c), c' \in C \,\}$$

[1] In this construction, we ignore the single state p of the original PDS because a single state does not provide any useful information.

Weighted pushdown systems can perform finite-state verification by designing an appropriate weight domain. For verification of safety properties, let S be the set of states of a property automaton A. Then define a weight domain $(2^{S \times S}, \cup, \circ, \emptyset, id)$, where a weight is a binary relation on S, combine is union, extend is composition of relations, $\overline{0}$ is the empty relation, and $\overline{1}$ is the identity relation. A WPDS can now be constructed as follows[2]: If $\langle p, \gamma \rangle \hookrightarrow \langle p, u \rangle$ is a PDS rule that describes control flow, then associate it with the weight $\{(s_1, s_2) \mid s_1 \xrightarrow{\gamma} s_2 \text{ in } A\}$. If we solve GPS on this WPDS using the singleton set that consistings of the program's starting configuration as the initial WPDS configuration, then safety can be guaranteed by checking if $(s_{\text{init}}, \text{error}) \in \delta(c)$ for some configuration c where s_{init} is the starting state of A.[3]

Boolean programs can also be encoded as WPDSs.[4] Assume that a program has only global variables. We defer discussion about local variables to §3.3. Then a transformer for an ICFG node is simply a relation on valuations of global variables. If G is the set of all valuations of global variables, then use the weight domain $(2^{G \times G}, \cup, \circ, \emptyset, id)$. For a PDS rule $\langle p, \gamma \rangle \hookrightarrow \langle p, u \rangle$, associate it with the transformer of the corresponding ICFG edge. Assertion checking can be performed by seeing if a configuration c (or a set of configurations) can be reached with non-zero weight, i.e., $\delta(c) \neq \overline{0}$.

3 Solving Reachability Problems

In this section, we review the existing algorithm for solving generalized reachability problems on WPDSs [7], which is based on chaotic iteration, and present our new algorithm, which uses Tarjan's path-expression algorithm [13]. We limit our discussion to GPP; GPS is similar but slightly more tedious.

3.1 Solving GPP using Chaotic Iteration

Let $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ be a WPDS where $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system and $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ is the weight domain. Let C be a regular set of configurations that is recognized by $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$. We assume, without loss of generality, that $\mathcal{A}$ has no transitions that lead into an initial state, and does not have any ε-transitions as well. GPP is solved by saturating this automaton with new weighted transitions (each transition has a weight label), to create automaton $\mathcal{A}_{pre^*}$, such that $\delta(c)$ can be read-off efficiently from $\mathcal{A}_{pre^*}$: $\delta(\langle p, u \rangle)$ is the combine of weights of all accepting paths for u starting from p, where the weight of a path is the extend of the weight-labels of the transitions in the path, in order. We present the algorithm for building $\mathcal{A}_{pre^*}$ based on its abstract grammar problem.

Definition 6. *[7] Let $(S, \sqcap)$ be a meet semilattice. An* **abstract grammar** *over $(S, \sqcap)$ is a collection of context-free grammar productions, where each production θ has the form*

$$X_0 \rightarrow g_\theta(X_1, \ldots, X_k).$$

[2] This construction is due to David Melski, and was used in an experimental version of the Path Inspector [4].

[3] If we add a self loop on the error state that matches with (the action of) every ICFG node, then we just need to check $\delta(c)$ for the exit node of the program.

[4] A similar encoding is given in [1].

Parentheses, commas, and g_θ (where θ is a production) are terminal symbols. Every production θ is associated with a function $g_\theta\colon S^k \to S$. Thus, every string α of terminal symbols derived in this grammar denotes a composition of functions, and corresponds to a unique value in S, which we call $val_G(\alpha)$ (or simply $val(\alpha)$ when G is understood). Let $L_G(X)$ denote the strings of terminals derivable from a nonterminal X. The **abstract grammar problem** *is to compute, for each nonterminal X, the value*

$$\mathrm{MOD}_G(X) := \prod_{\alpha \in L_G(X)} val_G(\alpha).$$

The value $\mathrm{MOD}_G(X)$ is called the **meet-over-all-derivations** *value for nonterminal X.*

We define abstract grammars over the meet semilattice $(D, \oplus)$, where D is the set of weights as given above. An example is shown in Fig. 3. The non-terminal t_3 can derive the string $\alpha = g_4(g_3(g_1))$ and $val(\alpha) = w_4 \otimes w_3 \otimes w_1$.

$$\begin{array}{ll} t_1 \to g_1(\epsilon) & g_1 = w_1 \\ t_1 \to g_2(t_2) & g_2 = \lambda x.w_2 \otimes x \\ t_2 \to g_3(t_1) & g_3 = \lambda x.w_3 \otimes x \\ t_3 \to g_4(t_2) & g_4 = \lambda x.w_4 \otimes x \end{array}$$

Fig. 3. A simple abstract grammar with four productions.

Production	for each
(1) $PopSeq_{(q,\gamma,q')} \to g_1(\epsilon)$ $g_1 = \overline{1}$	$(q, \gamma, q') \in \to_0$
(2) $PopSeq_{(p,\gamma,p')} \to g_2(\epsilon)$ $g_2 = f(r)$	$r = \langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$
(3) $PopSeq_{(p,\gamma,q)} \to g_3(PopSeq_{(p',\gamma',q)})$ $g_3 = \lambda x.f(r) \otimes x$	$r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \Delta, q \in Q$
(4) $PopSeq_{(p,\gamma,q)} \to g_4(PopSeq_{(p',\gamma',q')}, PopSeq_{(q',\gamma'',q)})$ $g_4 = \lambda x.\lambda y.f(r) \otimes x \otimes y$	$r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle \in \Delta, q, q' \in Q$

Fig. 4. An abstract grammar problem for solving GPP.

The abstract grammar for solving GPP is shown in Fig. 4. The grammar has one non-terminal $PopSeq_t$ for each possible transition $t \in Q \times \Gamma \times Q$ of $\mathcal{A}_{pre^*}$. The productions describe how the weights on those transitions are computed. Let $l(t)$ be the weight label on transition t. Then we want $l(t) = \mathrm{MOD}(PopSeq_t)$. The meet-over-all-derivation value is obtained as follows [7]: Initialize $l(t) = \overline{0}$ for all transitions t. If $PopSeq_t \to g(PopSeq_{t_1}, PopSeq_{t_2})$ is a production of the grammar (with possibly fewer non-terminals on the right-hand side) then update the weight label on t to $l(t) \oplus g(l(t_1), l(t_2))$. The existing algorithm for solving GPP is a worklist-based algorithm that uses chaotic iteration to choose (i) a transition in the worklist and (ii) all productions that have this transition on the right side, and updates the weight on the transitions on the left side of the productions as described earlier. If the weight on a transition changes then it is added to the worklist. Defn. 3(5) guarantees convergence.

Such a chaotic iteration scheme is not very efficient. Consider the abstract grammar in Fig. 3. The most efficient way of saturating weights on transitions would be to start with the first production and then keep alternating between the next two productions until $l(t_1)$ and $l(t_2)$ converge before choosing the last production. Any other strategy would have to choose the last production multiple times. Thus, it is important to identify such "loops" between transitions and to stay within a loop before exiting it.

3.2 Solving GPP using Path Expressions

To find a better iteration scheme for GPP, we convert GPP into a hypergraph problem.

Definition 7. *A (directed)* ***hypergraph*** *is a generalization of a directed graph in which generalized edges, called* ***hyperedges****, can have multiple sources, i.e., the source of an edge is an ordered set of vertices. A* **transition dependence graph (TDG)** *for a grammar G is a hypergraph whose vertices are the non-terminals of G. There is a hyperedge from $\{t_1, \cdots, t_n\}$ to t if G has a production with t appearing on the left-hand side and $t_1 \cdots t_n$ are the non-terminals that appear (in order) on the right-hand side.*

If we construct the TDG of the grammar shown in Fig. 4 when the underlying PDS is obtained from an ICFG, and the initial set of configurations is $\{\langle p, \varepsilon\rangle \mid p \in P\}$ (or $\rightarrow_0 = \emptyset$), then the TDG is identical to the ICFG (with edges reversed). Fig. 5 shows an example. This can be observed from the fact that except for the PDS states in Fig. 4, the transition dependences are almost identical to the dependences encoded in the pushdown rules, which in turn come from ICFG edges, e.g., the ICFG edge (n_1, n_2) corresponds to the transition dependence $(\{t_2\}, t_1)$ in Fig. 5, and the call-return pair (n_3, n_6) and (n_{12}, n_4) in the ICFG corresponds to the hyperedge $(\{t_4, t_6\}, t_3)$.

For such pushdown systems, constructing TDGs might seem unnecessary, but it allows us to choose an initial set of configurations, which defines a region of interest in the program. Moreover, PDSs can encode much stronger properties than an ICFG, such as setjmp/longjmp in C programs. However, it is still convenient to think of a TDG as an ICFG. In the rest of this paper, we illustrate the issues using the TDG of the grammar in Fig. 4. We reduce the meet-over-all-derivation problem on the grammar to a meet-over-all-paths problem on its TDG.

Intraprocedural Iteration. We first consider TDGs of a special form: consider the intraprocedural case, i.e., there are no hyperedges in the TDG (and correspondingly no push rules in the PDS). As an example, assume that the TDG in Fig. 5 has only the part corresponding to procedure `foo()` without any hyperedges. In such a TDG, if an edge $(\{t_1\}, t)$ was inserted because of the production $t \rightarrow g(t_1)$ for $g = \lambda x.x \otimes w$ for some weight w, then label this edge with w. Next, insert a special node t_s into the TDG and for each production of the form $t \rightarrow g(\epsilon)$ with $g = w$, insert the edge $(\{t_s\}, t)$ and label it with weight w. t_s is called a source node. This gives us a graph with weights on each edge. Define the weight of a path in this graph in the standard (but reversed) way: the weight of a path is the extend of weights on its constituent edges in the reverse order. It is easy to see that

$$\mathrm{MOD}(t) = \bigoplus\{v(\eta) \mid \eta \in \mathit{path}(t_s, t)\}$$

where $\mathit{path}(t_s, t)$ is the set of all paths from t_s to t in the TDG and $v(\eta)$ is the weight of the path η. To solve for MOD, we could still use chaotic iteration, but instead we will make use of Tarjan's path-expression algorithm [13].

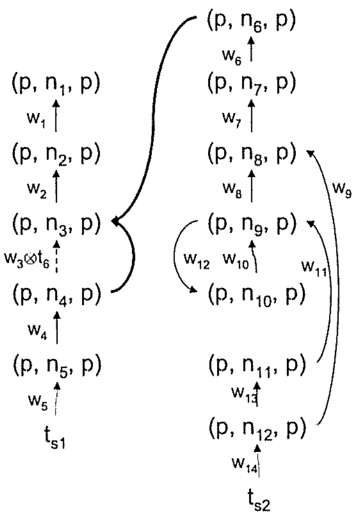


Fig. 5. TDG for the PDS shown in Fig. 1. A WPDS is obtained from the PDS by supplementing rule number i with weight w_i. Let t_j stand for the node (p, n_j, p). The thick bold arrows form a hyperedge. Nodes t_{s1} and t_{s2} are source nodes, and the dashed arrow is a summary edge. These, along with the edge labels, are explained later in §3.2.

Problem 1. Given a directed graph G and a fixed vertex s, the **single-source path expression** (SSPE) problem is to compute a regular expression that represents $path(s, v)$ for all vertices v in the graph. The syntax of the regular expressions is as follows:

$$r ::= \emptyset \mid \varepsilon \mid e \mid r_1 \cup r_2 \mid r_1.r_2 \mid r^*$$

where e stands for an edge in the graph. We say that a regular expression represents a set of paths when the language described by it is exactly those set of paths.

We can use the SSPE algorithm to compute regular expressions for $path(t_s, t)$, which gives us a compact description of the set of paths we need to consider. Also, the Kleene-star operator identifies loops in the graphs. Let $\otimes^c$ be the reverse of $\otimes$, i.e., $w_1 \otimes^c w_2 = w_2 \otimes w_1$. To compute MOD$(t)$, we take the regular expression for $path(t_s, t)$ and replace each edge e with its weight, $\emptyset$ with $\overline{0}$, ε with $\overline{1}$, $\cup$ with $\oplus$, . with $\otimes^c$ and solve the expression. The weight w^* is computed as $\overline{1} \oplus w \oplus (w \otimes w) \oplus \cdots$. Again, because of the bounded-height property of the semiring, this iteration converges. Two main advantages of using regular expressions to compute MOD(t) are: First, loops are identified in the expression, and the evaluation strategy saturates a loop before exiting it. Second, we can compute w^* faster than normal iteration could. For this, observe that

$$(\overline{1} \oplus w)^n = \overline{1} \oplus w \oplus w^2 \oplus \cdots \oplus w^n$$

where exponentiation is defined using $\otimes$, i.e., $w^0 = \overline{1}$ and $w^i = w \otimes w^{(i-1)}$. Then w^* can be computed by repeatedly squaring $(\overline{1} \oplus w)$ until it converges. If $w^* = \overline{1} \oplus w \oplus \cdots \oplus w^n$ then it can be computed in $O(\log n)$ operations. A chaotic-iteration strategy would take $O(n)$ steps to compute the same value. In other words, having a closed representation of loops provides an exponential speedup.[5]

[5] This assumes that each semiring operation takes the same amount of time.

Given a graph with m edges (or m grammar productions in our case) and n nodes (or non-terminals), regular expressions for $path(t_s, t)$ can be computed for all nodes t in time $O(m \log n)$ when the graph is *reducible*. Evaluating these expressions will further take $O(m \log n \log h)$ semiring operations where h is the height of the semiring. Because most high-level languages are well-structured, their CFGs are mostly reducible. Even for programs in x86 assembly code, we found that the CFGs were mostly reducible. When the graph is not reducible, the running time gradually degrades to $O((m \log n + k) \log h)$ semiring operations, where k is the sum of the cubes of the sizes of *dominator-strong components* of the graph. In the worst case, k can be $O(n^3)$. In our experiments, we seldom found irreducibility to be a problem: k/n was a small constant. A pure chaotic-iteration strategy would take $O(m\ h)$ semiring operations in the worst case. Comparing these complexities, we can expect to be much faster than chaotic iteration, and the benefit will be greater as the height of the semiring increases.

Interprocedural Iteration. We now generalize our algorithm to any TDG. For each hyperedge $(\{t_1, t_2\}, t)$ delete it from the graph and replace it with the edge $(\{t_1\}, t)$. This new edge is called a *summary edge*, and node t_2 is called an *out-node*. For example, in Fig. 5 we would delete the hyperedge $(\{t_4, t_6\}, t_3)$ and replace it with $(\{t_4\}, t_3)$. The new edge is called a summary edge because it crosses a call-site (from a return node to a call node) and will be used to summarize the effect of a procedure call. Node t_6 is an out-node and will supply the procedure summary weight. The resultant TDG is a collection of connected graphs, with each graph roughly corresponding to a procedure. In Fig. 5, the transitions that correspond to procedures `main` and `foo` get split. Each connected graph is called an *intragraph*. For each intragraph, we introduce a source node as before and add edges from the source node to all nodes that have ϵ-productions. The weight labels are also added as before. For a summary edge $(\{t_1\}, t)$ obtained from a hyperedge $(\{t_1, t_2\}, t)$ with associated production function $g = \lambda x.\lambda y.w \otimes x \otimes y$, label it with $w \otimes t_2$.

This gives us a collection of intragraphs with edges labeled with either a weight or a simple expression with an out-node. To solve for the MOD value, we construct a set of *regular equations*. For an intragraph G, let t_G be its unique source node. Then for each out-node t_o in G construct the regular expression for all paths in G from t_G to t_o, i.e., for $path(t_G, t_o)$. In this expression, replace each edge with its corresponding label. If the resulting expression is r and it contains out-nodes t_1 to t_n, add the equation $t_o = r(t_1, \cdots, t_n)$ to the set of equations. Repeating this for all intragraphs, we get a set of equations whose variables correspond to the out-nodes. These resulting equations describe all hyperpaths in the TDG to an out-node from the collection of all source nodes. The MOD value of the out-nodes is the greatest[6] fixpoint of these equations. For example, for the TDG shown in Fig. 5, assuming that t_1 is also an out-node, we would obtain the following set of equations.[7]

$$t_6 = w_{14}.(w_9 \oplus w_{13}.w_{11}.(w_{12}.w_{10})^*.w_8).w_7.w_6$$
$$t_1 = w_5.w_4.(w_3 \otimes t_6).w_2.w_1$$

[6] With respect to the partial order $w_1 \leq w_2$ iff $w_1 \oplus w_2 = w_1$

[7] The equations might be different depending on how the SSPE algorithm was implemented but all such equations would have the same solution.

Here we have used . as a shorthand for $\otimes^c$. One way to solve these equations is by using chaotic iteration: start by initializing each out-node with $\overline{0}$ (the greatest element in the semiring) and update the values of out-nodes by repeatedly solving the equations until they converge. Another way is to give direction to the chaotic iteration by using regular expressions again. Each equation $t_o = r(t_1, \cdots, t_n)$ gives rise to dependences $t_i \rightarrow t_o, 1 \leq i \leq n$. Construct a dependence graph for the equations, but this time label each edge with the equation that it came from. Assume any out-node to be the source node and construct a regular expression to all other nodes using SSPE again. These expressions give the order in which equations have to be evaluated. For example, if we have the following set of equations on three out-nodes:

$$t_1 = r_1(t_1, t_3) \quad t_2 = r_2(t_1) \quad t_3 = r_3(t_2)$$

then a possible regular expression for paths from t_1 to itself would be $(r_1 \cup r_2.r_3.r_1)^*$. This suggests that to solve for t_1 we should use the following evaluation strategy: evaluate r_1, update t_1, then evaluate r_2, r_3, and r_1, and update t_1 again — repeating this until the solution converges. In our implementation, we use a simpler strategy. We take a *strongly connected component* (SCC) decomposition of the dependence graph and solve all equations in one component before moving to equations in next component (in a topological order). We chose this strategy because SCCs tend to be quite small in practice.

Each regular expression in these equations summarizes all paths in an intragraph and can be quite large. Therefore, we want to avoid evaluating them repeatedly while solving the equations. To this end, we incrementally evaluate the regular expressions: only that part of an expression is reevaluated that contains a modified out-node. A regular expression is represented using its abstract-syntax tree (AST), where leaves are weights or out-nodes and internal nodes correspond to $\oplus$, $\otimes$, or *. A possible AST for the regular expression for out-node t_1 of Fig. 5 is shown in Fig. 6. Whenever the value of out-node t_6 is updated, we only need to reevaluate the weight of subtrees at a_4, a_3, and a_1, and update the value of out-node t_1 to the weight at a_1.

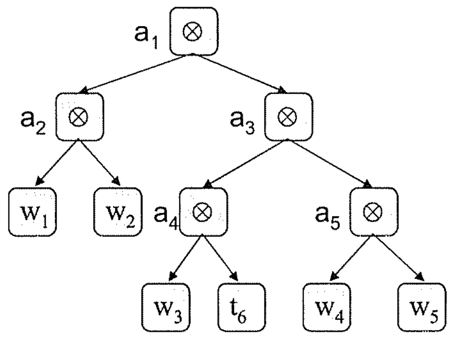


Fig. 6. An AST for $w_5.w_4.(w_3 \otimes t_6).w_2.w_1$. Internal nodes for $\otimes^c$ are converted into $\otimes$ nodes by reversing the order of its children. Internal nodes in this AST have been given names a_1 to a_5.

As a further optimization, all regular expressions share common subtrees, and are represented as DAGs instead of trees. The incremental algorithm we use takes care of this sharing and also identifies modified out-nodes in an expression automatically. At each DAG node we maintain two integers: `last_change` and `last_seen`; and the weight `weight` of the subdag rooted at the node. We assume that all regular expressions share the same leaves for out-nodes. We keep a global counter `update_count` that is incremented each time the weight of some out-node is updated. For a node, the counter `last_change` records the last update count at which the weight of its subdag changed, and the counter `last_seen` records the update count at which the subdag was reevaluated. Let $\odot$ stand for $\oplus$ or $\otimes$. The evaluation algorithm is shown in Fig. 7. When the weight of an out-node is changed, its corresponding leaf node is updated with that weight, `update_count` is incremented, and the out-node's counters are set to `update_count`.

```
procedure evaluate(r)
begin
  if r.last_seen == update_count then
    return
  case r = $w$, r = $t_o$ return
  case r = $r_1^*$
    evaluate($r_1$)
    if $r_1$.last_change $\not\leq$ r.last_seen then
      w = ($r_1$.weight)*
      if r.weight $\neq$ w then
        r.last_change = $r_1$.last_change
        r.weight = w
    r.last_seen = update_count
  case r = $r_1 \odot r_2$
    evaluate($r_1$)
    evaluate($r_2$)
    m = max{$r_1$.last_change, $r_2$.last_change}
    if m $\not\leq$ r.last_seen then
      w = $r_1$.weight $\odot$ $r_2$.weight
      if r.weight $\neq$ w then
        r.last_change = m
        r.weight = w
    r.last_seen = update_count
end
```

Fig. 7. Incremental evaluation algorithm for regular expressions.

Once we solve for the values of the out-nodes, we can change the out-node labels on summary edges in the intragraphs and replace them with their corresponding weight. Then the MOD values for other nodes in the TDG can be obtained as in the intraprocedural version by considering each intragraph in isolation.

The time required for solving this system of equations depends on reducibility of the intragraphs. Let S_G be the time required to solve SSPE on intragraph G, i.e., $S_G = O(m \log n + k)$ where k is $O(n^3)$ in the worst-case, but is ignorable in practice. If the equations do not have any mutual dependences (corresponding to no recursion) then the running time is $\sum_G S_G \log h$, where the sum ranges over all intragraphs, because each equation has to be solved exactly once. In the presence of recursion, we use the observation that the weight of each subdag in a regular expression can change at most h times while the equations are being solved because it can only decrease monotonically. Because the size of a regular expression obtained from an intragraph G is bounded by S_G, the worst-case time for solving the equations is $\sum_G S_G\ h$. This bound is very pessimistic and is actually worse than that of chaotic iteration. Here we did not make use of the fact that incrementally computing regular expressions is much faster than reevaluating them. For a regular expression with one modified out-node, we only need to perform semiring operations for each node from the out-node leaf to the root of the expression. For a nearly balanced regular expression tree, this path to the root can be as small as $\log S_G$. Empirically, we found that incrementally computing the expression required many fewer operations than recomputing the expression.

Unlike the chaotic-iteration scheme, where the weights of all TDG nodes are computed, we only need to compute the weights on out-nodes. The weights for the rest of the nodes can be computed lazily. For applications that just require the weight for a few TDG nodes, this gives us additional savings. Moreover, the algorithm can be executed on multi-processor machines by assigning each intragraph to a different processor. The only communication required between the processors would be the weights on out-nodes while they are being saturated.

3.3 Handling Local Variables

WPDSs were recently extended to Extended-WPDSs (EWPDSs) to provide a more convenient mechanism for handling local variables [3]. EWPDSs are similar to WPDSs, but allow for a special *merge function* to be associated with each push rule, in addition to a weight. These merge functions are binary functions on weights, and are used to merge the weight returned by a procedure with the weight at a call site of the procedure to compute the required weight at the return site. Instead of giving the formal definition of EWPDSs and merge functions, we describe how to encode Boolean programs with local variables as an EWPDS. Note that §2.3 only gave an encoding for Boolean programs without local variables. Let G be the set of all global variable valuations and L be the set of all local-variable valuations (assume that all procedures have the same number of local variables). Then each ICFG edge is associated with a transformer, which is a binary relation on $G \times L$. The weight domain is: $(2^{(G\times L)\times(G\times L)}, \cup, \circ, \emptyset, id)$. Each PDS rule is still associated with the transformer of the corresponding ICFG edge, but in addition each push rule is associated with the following merge function:

$$h(w_1, w_2) = \{(g_1, l_1, g_2, l_1') \mid (g_1, l_1, g_1', l_1') \in w_1, (g_1', l_2, g_2, l_2') \in w_2\}$$

The first argument is the weight accumulated at a call-site and the second argument is a summary of the called function. The merge function forgets the local variables of the second argument and composes the global information between the two arguments.

Reachability problems in EWPDS can also be solved using an abstract grammar. The abstract grammar for GPP on EWPDSs is shown in Fig. 8. It only differs from that of a WPDS in the last case.

To solve GPP we just require one change. For hyperedges in the TDG corresponding to case 5 in Fig. 8, if t_o is the out-node, then label the corresponding summary edge with $h_r(\overline{1}, t_o)$. Application of merge functions amounts to passing only global information between intragraphs.

Production	for each
(1) $\mathit{PopSeq}_{(q,\gamma,q')} \rightarrow g_1(\epsilon)$	$(q,\gamma,q') \in \rightarrow_0$
$g_1 = \overline{1}$	
(2) $\mathit{PopSeq}_{(p,\gamma,p')} \rightarrow g_2(\epsilon)$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\varepsilon\rangle \in \Delta$
$g_2 = f(r)$	
(3) $\mathit{PopSeq}_{(p,\gamma,q)} \rightarrow g_3(\mathit{PopSeq}_{(p',\gamma',q)})$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle \in \Delta, q \in Q$
$g_3 = \lambda x. f(r) \otimes x$	
(4) $\mathit{PopSeq}_{(p,\gamma,q)} \rightarrow g_4(\mathit{PopSeq}_{(p',\gamma',q')}, \mathit{PopSeq}_{(q',\gamma'',q)})$	
$g_4 = \lambda x. \lambda y. f(r) \otimes x \otimes y$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle \in \Delta, q \in Q, q' \in (Q-P)$
(5) $\mathit{PopSeq}_{(p,\gamma,q)} \rightarrow g_5(\mathit{PopSeq}_{(p',\gamma',q')}, \mathit{PopSeq}_{(q',\gamma'',q)})$	
$g_5 = \lambda x. \lambda y. h_r(\overline{1}, x) \otimes y$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle \in \Delta, q \in Q, q' \in P$

Fig. 8. An abstract grammar problem for GPP in an EWPDS. h_r is the merge function associated with rule r.

4 Solving other PDS Problems

4.1 Witness Tracing

For program-analysis tools, if a program does not satisfy a property, it is often useful to provide a justification of why the property was not satisfied. In terms of WPDSs, it amounts to reporting a set of paths, or rule sequences, that together justify the reported weight for a configuration. Formally, using the notation of Defn. 5, the witness tracing problem for GPP is to find, for each configuration c, a set $\omega(c) \subseteq \bigcup_{c' \in C} path(c,c')$ such that

$$\bigoplus_{\sigma \in \omega(c)} v(\sigma) = \delta(c)$$

This definition of witness tracing does not impose any restrictions on the size of the reported witness set because any compact representation of the set suffices for most applications.

Because of Defn. 3(5), it is always possible to create a finite witness set. In [7], it was shown that a witness set can be found by recording how the weight on a transition changes during the GPP saturation procedure. If the weight of a transition is updated from $l(t)$ to $w = l(t) \oplus g(l(t_1), l(t_2))$ and the latter differs from the former, then it is recorded that transition t with weight w can be created from (i) transition t with weight $l(t)$, (ii) transitions t_1 and t_2 with weights $l(t_1)$ and $l(t_2)$, and (iii) production function g (which corresponds to some WPDS rule). The witness set for a configuration can be obtained from those of individual transitions. The running time is covered by the GPP saturation procedure but it requires $O(|Q|^2\ |\Gamma|\ h)$ memory, which can be quite large.

In our new GPP algorithm, we already have a head start because we have regular expressions that describe all paths in an intragraph. In the intragraphs, we label each edge with not just a weight, but also the rule that justifies the edge. Push rules will be associated with summary edges and pop rules with edges that originate from a source node. Edges from the source node that were inserted because of production (1) in Fig. 4 are not associated with any rule (or with an empty rule sequence). After solving SSPE on the intragraphs, we can replace each edge with the corresponding rule label. This gives us, for each out-node, a regular expression in terms of other out-nodes that captures the set of all rule sequences that can create that out-node. Next, while solving the regular equations, we record the weights on out-nodes; i.e., when we solve the equation $t_o = r(t_1, \cdots, t_n)$, we record the weights on $t_1, \cdots, t_n$ — say $w_1, \cdots, w_n$ — whenever the weight on t_o changes to, say, w_o. Then the set of rule sequences to create transition t_o with weight w_o is given by the expression r (where we replace TDG edges with their rule labels) by replacing each out-node t_i with the regular expression for all rule sequences used to create t_i with weight w_i (obtained recursively). This gives a regular expression for the witness set of each out-node. Witness sets for other transitions can be obtained by solving SSPE on the intragraphs by replacing out-node labels with their witness-set expression.

Thus, we only require $O(|ON|\ h)$ space for recording witnesses where $|ON|$ is the number of out-nodes. For PDSs obtained from ICFGs and empty initial automaton, $|ON|$ is the number of procedures in the ICFG, which is very small compared to $|\Gamma|$.

4.2 Differential Propagation

The general framework of WPDSs can sometimes be inefficient for certain analysis. While executing GPP, when the weight of a transition changes from w_1 to $w_2 = w_1 \oplus w$, the new weight w_2 is propagated to other transitions. However, because the weight w_1 had already been propagated, this will do extra work by propagating w_1 again when only w (or a part of w) needs to be propagated. This simple observation can be incorporated into WPDSs when the semiring weight domain has a special subtraction operation (called *diff*, denoted by $\dot{-}$) [7]. The *diff* operator must satisfy the following properties: For each $a, b, c \in D$,

$$\begin{array}{rcl} a \oplus (b \dot{-} a) & = & a \oplus b \\ (a \dot{-} b) \dot{-} c & = & a \dot{-} (b \oplus c) \\ a \oplus b = a & \Longleftrightarrow & b \dot{-} a = \overline{0} \end{array}$$

For the weight domains presented in §2.3 for finite-state property verification, set difference (where relations are considered to be sets of tuples) satisfies all of the required properties.

We make use of the *diff* operation while solving the set of regular equations. In addition to incrementally computing the regular expressions, we also incrementally compute the weights. When the weight of an out-node changes from w_1 to w_2, we associate its corresponding leaf node with the change $w_2 \dot{-} w_1$. This change is then propagated to other nodes. If the weight of expressions r_1 and r_2 is w_1 and w_2, and they change by d_1 and d_2, then the weights of the following kinds of expressions change as follows:

$$\begin{array}{lcl} r_1 \cup r_2 & : & d_1 \oplus d_2 \\ r_1.r_2 & : & (d_1 \otimes^c d_2) \oplus (d_1 \otimes^c w_2) \oplus (w_1 \otimes^c d_2) \\ r_1^* & : & (w_1 \oplus d_1)^* \dot{-} w_1^* \end{array}$$

There is no better way of computing the change for Kleene-star (chaotic iteration suffers from the same problem), but we can use the *diff* operator to compute the Kleene-star closure of a weight as follows.

```
begin
  wstar = del = $\overline{1}$
  while del $\neq \overline{0}$
    temp = del $\otimes$ w
    del = temp $\dot{-}$ wstar
    wstar = wstar $\oplus$ temp
end
```

4.3 Incremental Analysis

The first incremental algorithm for verifying finite-state properties on ICFGs was given by Conway et al. [6]. We can use the methods presented in this paper to generalize their algorithm to WPDSs. An incremental approach to model checking has the advantage of amortizing the verification time across program development or debugging time.

We consider two cases: addition of new rules and deletion of existing ones. In each case we work at the granularity of intragraphs. When a new rule is added, the fixpoint solution of the regular equations monotonically decreases and we can reuse all of the existing computation. We first identify the intragraphs that changed (have more edges) because of the new rule. Next, we recompute the regular expression for out-nodes in those intragraphs.[8] Then we solve the regular expressions as before but set the initial weights of out-nodes to be their existing value. If new out-nodes got added, then set their initial value to $\overline{0}$.

Deletion of a rule requires more work. Again, we identify the changed intragraphs and recompute the regular expression for out-nodes in those intragraphs. These out-nodes are called *modified* out-nodes. Next, we look at the dependence graph of out-nodes as constructed in §3.2. We perform a SCC decomposition of this graph and topologically sort the SCCs. Then the weights for all out-nodes that appear before the first SCC that has a modified out-node need not be changed. We recompute the solution for other out-nodes in topological order, and stop as soon as the new values agree with previous values. We start with out-nodes in the first SCC that has a modified out-node and solve for their weights. If the new weight of an out-node is different from its previously computed weight, all out-nodes in later SCCs that are dependent on it are marked as modified. We repeat this procedure until there are no more modified out-nodes.

The advantage of doing incremental analysis in our framework is that very little information has to be stored between analysis runs. In particular, we only need to store weights on out-nodes. Moreover, because the algorithm is demand-driven, we only compute what is required by the user.

5 Experiments

We have implemented our algorithm as a back-end for WPDS++ [19], a C++ implementation of WPDSs. The interface presented to WPDS++ clients is unchanged. We refer to our implementation as FWPDS.[9] We compare FWPDS against an optimized version of WPDS++. This version, called BFS-WPDS++, can be supplied with a user priority-ordering on stack symbols that gets used by chaotic iteration to choose the transition with least priority first. In our application, we use a breadth-first ordering on the ICFG obtained by treating it as a graph. BFS-WPDS++ performed better than WPDS++ in the experiments. We do not compute witnesses in the experiments.

5.1 Basic Saturation Algorithm

We tested our algorithm on two applications. The first application, BTRACE, is for debugging [5]. It performs path optimization on C programs: given a set of ICFG nodes,

[8] There are incremental algorithms for SSPE as well, but we have not used them because solving SSPE for a single intragraph is usually very fast.

[9] F stands for "fast".

called critical nodes, it tries to find a shortest ICFG path that touches the maximum number of these nodes. The path starts at the entry point of the program and stops at a given failure point in the program. We perform GPS with the entry point of the program as the initial configuration, and compute the weight at the failure site. We measure end-to-end performance to take advantage of the lazy nature of our algorithm, and, thus, only compute the weight at the failure site. As shown in Table 1, FWPDS performs much better than BFS-WPDS++ for this application.

Prog	ICFG nodes	Procs	BFS-WPDS++	FWPDS	Improvement
gawk	86617	401	170	53	3.21
indent	28155	104	49	44	1.11
less	33006	359	46	12	3.83
make	40667	204	31	10	3.10
mc	78641	676	12	8	1.46
patch	27389	133	170	32	5.31
uucp	16973	139	10	5	2.11
wget	44575	399	800	64	12.50

Table 1. Comparison of BTRACE results. Running times are reported in seconds and improvement is given as a ratio of the running time of FWPDS versus BFS-WPDS++. The critical nodes were chosen at random from ICFG nodes and the failure site was set as the exit point of the program. The programs are common Unix utilities, and the experiments were run on P4 2.4 GHz machine with 4GB RAM.

The second application is nMoped [20], which is a model checker for Boolean programs. It uses a WPDS library for performing reachability queries. Weights are binary relations on variable valuations, and are represented using BDDs. We measure the performance of FWPDS against this library. The results, as shown in Table 2, are inconclusive. We attribute the big differences in running times to BDD-variable ordering. FWPDS performs a different sequence of weight operations, which in turn use different BDD operations. Variable ordering is crucial for these operations.

Prog	ICFG nodes	Procs	nMoped	FWPDS	Improvement
blast.rem	30	4	10.52	0.85	12.38
qsort3.rem	13	2	14.36	336	0.04
simplInv.rem	7	1	39.68	4.3	9.23
qsortIrrel.rem	31	5	2	12	0.17
intInt.rem	13	2	6.79	204	0.03
files.rem	45	5	267	6.86	38.92

Table 2. Comparison of nMoped results. Experiments were run on P4 3 GHz machine with 2GB RAM. (The programs were provided by S. Schwoon.)

5.2 Incremental Analysis

We also measure the advantage of incremental analysis for BTRACE. Similar to the experiments performed in [6], we delete a procedure from a program, solve GPS, then reinsert the procedure and look at the time that it takes to solve GPS incrementally. We compare this time with the time that it takes to compute the solution from scratch. We repeated this for all procedures in a given program, and discarded those runs that did not affect at least one other procedure. The results are shown in Table 3, which shows an average speed up by a factor of 10.

Prog	Procs	#Recomputed	Time (sec)	Improvement
less	359	91	1.66	7.25
mc	676	70	0.41	20.2
uucp	139	36	2.00	2.34

Table 3. Results for incremental analysis for BTRACE. The third column gives the average number of procedures for which the solution had to be recomputed. The last column compares the time required to compute the solution incrementally versus the time required to compute the solution from scratch, the latter of which is reported in Table 1.

6 Related Work

The basic strategy of using a regular expression to describe a set of paths has been used previously for dataflow analysis [14]. However, it has only been used for dataflow analysis of single-procedure programs. We have generalized the approach to multi-procedure programs, as well as pushdown systems.

Most other related work has already been discussed in the body of the paper. A lengthy discussion on the use of PDS model checking for finite-state property verification can be found in [10].

There has been a host of previous work on incremental model checking [21,22], as well as on interprocedural automaton-based analysis [6]. The incremental algorithm we have presented is similar to the algorithm in [6], but generalizes it to WPDSs and is thus applicable in domains other than finite-state property verification. A key difference with their algorithm is that they explore the property automaton on-the-fly as the program is explored. Our encoding into a WPDS requires the whole automaton before the program is explored. This difference can be significant when the automaton is large but only a small part of the automaton needs to be generated.

References

1. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: CAV. (2001)
2. Schwoon, S.: Moped (2002) http://www.fmi.uni-stuttgart.de/szs/tools/moped/.
3. Lal, A., Reps, T., Balakrishnan, G.: Extended weighted pushdown systems. In: CAV. (2005)
4. GrammaTech, Inc.: CodeSurfer Path Inspector (2005) http://www.grammatech.com/products/codesurfer/overview_pi.html.
5. Lal, A., Lim, J., Polishchuk, M., Liblit, B.: Path optimization in programs and its application to debugging. In: ESOP. (2006)
6. Conway, C.L., Namjoshi, K.S., Dams, D., Edwards, S.A.: Incremental algorithms for interprocedural analysis of safety properties. In: CAV. (2005)
7. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. In: SCP. Volume 58. (2005)
8. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model checking. In: CONCUR. (1997)
9. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. Electronic Notes in Theoretical Computer Science **9** (1997)
10. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, Technical Univ. of Munich, Munich, Germany (2002)
11. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: CAV. (2000)
12. Myers, E.W.: A precise interprocedural data flow algorithm. In: POPL. (1981)

13. Tarjan, R.E.: Fast algorithms for solving path problems. J. ACM **28** (1981) 594–614
14. Tarjan, R.E.: A unified approach to path problems. J. ACM **28** (1981) 577–593
15. Jha, S., Reps, T.: Analysis of SPKI/SDSI certificates using model checking. In: IEEE Comp. Sec. Found. Workshop (CSFW), IEEE Computer Society Press (2002)
16. Schwoon, S., Jha, S., Reps, T., Stubblebine, S.: On generalized authorization problems. In: Comp. Sec. Found. Workshop, Wash., DC, IEEE Comp. Soc. (2003)
17. Reps, T., Schwoon, S., Jha, S.: Weighted pushdown systems and their application to interprocedural dataflow analysis. In: SAS. (2003)
18. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: POPL. (2003)
19. Kidd, N., Reps, T., Melski, D., Lal, A.: WPDS++: A C++ library for weighted pushdown systems (2005) http://www.cs.wisc.edu/wpis/wpds++.
20. Kiefer, S., Schwoon, S., Suwimonteerabuth, D.: nMoped (2005) http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/nmoped/.
21. Sokolsky, O., Smolka, S.A.: Incremental model checking in the modal mu-calculus. In: CAV. (1994)
22. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.: Extreme model checking. In: Verification: Theory and Practice. (2003)