Differential Testing of Pushdown Reachability with a Formally Verified Oracle

Anders Schlichtkrull Morten Konggaard Schou Jiří Srba
Department of Computer Science
Aalborg University
Aalborg, Denmark
{andsch,mksc,srba}@cs.aau.dk

Dmitriy Traytel
Department of Computer Science
University of Copenhagen
Copenhagen, Denmark
traytel@di.ku.dk

***Abstract*—Pushdown automata are an essential model of recursive computation. In model checking and static analysis, numerous problems can be reduced to reachability questions about pushdown automata and several efficient libraries implement automata-theoretic algorithms for answering these questions. These libraries are often used as core components in other tools, and therefore it is instrumental that the used algorithms and their implementations are correct. We present a method that significantly increases the trust in the answers provided by the libraries for pushdown reachability by (i) formally verifying the correctness of the used algorithms using the Isabelle/HOL proof assistant, (ii) extracting executable programs from the formalization, (iii) implementing a framework for the differential testing of library implementations with the verified extracted algorithms as oracles, and (iv) automatically minimizing counter-examples from the differential testing based on the delta-debugging methodology. We instantiate our method to the concrete case of PDAAAL, a state-of-the-art library for pushdown reachability. Thereby, we discover and resolve several nontrivial errors in PDAAAL.**

I. Introduction

In 1964, Büchi [7] proved that the possibly infinite set of all reachable pushdown configurations (from a given initial configuration) can be effectively described by a regular language. In fact, even for a given regular set of pushdown configurations, its $post^*$ and pre^* closures (representing all forward and backward reachable configurations from a given set of configurations) are also regular. Büchi's *automata-theoretic approach* gave rise to a rich theory of pushdown reachability with numerous algorithms and applications to, e.g., interprocedural control-flow analysis of recursive programs [9], [11], model checking [4], [13], [45], [46], communication network analysis [10], [21], [22] and others. A number of tools have been developed to support the theory, including Moped [45], [46], WALi [25], and PDAAAL [23] with applications ranging from the static analysis of Java [46] and C/C++ code [26], [43] to the analysis of MPLS communication protocols [22].

Even though the automata-theoretic approach for pushdown reachability is based on relatively simple saturation procedures, the proofs of correctness are nontrivial and the implementation of the algorithms in the different tools often includes numerous performance optimizations as well as additional improvements to the theory itself [23]. To be able to rely on the output of model checking tools and other applications of pushdown reachability, it is important that the theory is not only sound but also correctly implemented. A positive reachability answer is typically accompanied by a finite evidence (trace) that can function as an efficiently checkable certificate. A negative answer is, on the other hand, much harder to check, and designing a finite evidence for non-reachability is difficult, primarily because the number of reachable pushdown configurations can be infinite. One approach is to establish an invariant that (i) includes the initial configuration(s) of the system, (ii) is maintained by the transition relation and (iii) has an empty intersection with the set of undesirable configurations. Such approaches have been studied [16], [17] but are usually incomplete and require another complex tool (that can be error-prone, too) to verify such invariants.

We instead use a proof assistant, Isabelle/HOL [37] (§II), to formally verify the correctness of the pushdown reachability algorithms $post^*$ (forward search), pre^* (backward search), and $dual^*$ (bi-directional search) (§III) that lie at the heart of the automata-theoretic analysis of pushdown systems [4], [23], [44]. From the formalization of pre^*, we extract an executable program with strong correctness guarantees (§IV). For a given input, the extracted program's output can be compared with the output of other, unverified but optimized tools solving the same problem (§V). This approach is known as differential testing [14], [18], [34] with a twist that the testing oracle has been formally verified and thus is extremely trustworthy. When testing reveals a disagreement between a verified and an unverified algorithm, we know who is to blame. To help localize errors in unverified algorithms, we minimize the tests causing disagreement using the delta-debugging technique [51]. Our main contributions are as follows.

- The formalization of $post^*$, pre^* and $dual^*$ algorithms in Isabelle/HOL and verification of their correctness based on the proofs provided by Schwoon [44] for $post^*$ and pre^*, and following Jensen et al. [23] for $dual^*$.
- The refinement to and the extraction of an executable program of the formalized pre^* algorithm that serves as the verified oracle for differential testing.
- The automatic minimization of the input automata in cases where an unverified tool disagrees with the oracle.

This research was supported by the Independent Research Fund Denmark (DFF project QASNET) and by Novo Nordisk Fonden (NNF20OC0063462).

https://doi.org/

– The application of our method to a modern state-of-the-art library for pushdown reachability, PDAAAL [23], and the identification, localization (using the minimized counter-examples), and correction of three, previously unknown, implementation errors (§VI). The corrected implementation passes all differential tests successfully.

Our Isabelle formalization as well as the case study are publicly available [40].

a) **Related work***:* Differential testing with a verified oracle has been used in the context of runtime verification and automatic theorem proving. The runtime monitor VeriMon [2], [41] served as the verified oracle used to detect errors in unverified monitors. Compared to our approach, VeriMon's differential testing case study is from a different application domain, does not include exhaustive test generation for small input sizes (which is difficult in runtime monitoring) and does not minimize the tests automatically. To assess its performance but also to evaluate the benchmark's correctness, the verified first-order prover RPx [39] was evaluated on a standard benchmark for first-order logic problems. RPx's answers have in all cases coincided with the expected ones recorded in the benchmark.

The verified C compiler CompCert [32] and several verified distributed systems [20], [33], [48] have been themselves put onto the testbed [15], [50]. A few errors in these tools' unverified parts or in scenarios violating the verification assumptions were found, but none in the verified components themselves.

Many works extract efficient executable code from formalizations, but do not use it as an oracle in testing. Examples include verified model checkers for LTL [12] and timed automata [49] and verified algorithms for finite automata [3], [6], [24], [31] and context-free grammars [35], [38].

The only formalization of pushdown automata we are aware of is part of Lammich et al.'s work on dynamic pushdown networks (DPN) [30]. Lammich describes the Isabelle formalization of an executable pre^* algorithm for DPNs stemming from this work in an unpublished technical report [29]. DPNs generalize pushdown automata, but their $post^*$ is not regular [5] and so we cannot extend this work for our purposes. Moreover, Lammich's formalization does not support ε-transitions in the underlying automata, an essential component needed for our formalization of $post^*$ and $dual^*$.

b) **Background definitions***:* Let P be a finite set of control locations and Γ a finite stack alphabet. A *pushdown system* (*PDS*) is a tuple (P, Γ, Δ), where $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of *rules*, written $(p, \gamma) \hookrightarrow (q, w)$ whenever $((p, \gamma), (q, w)) \in \Delta$. Without loss of generality, we assume $|w| \leq 2$, so that $w = \varepsilon$ represents a *pop* operation that removes the topmost stack symbol, $|w| = 1$ is a *swap* that replaces the topmost symbol with another one, and $|w| = 2$ is a *push* that incorporates a *swap* followed by adding a new symbol on top.

A *configuration* of a pushdown system is a pair (p, w) of the current control location $p \in P$ and the current stack content $w \in \Gamma^*$ where we assume that the top of the stack is on the left. The set of all configurations is denoted by $\mathcal{C}$. A PDS can take a *computation step* $(p, \gamma w') \Rightarrow (q, ww')$ between configurations whenever $(p, \gamma) \hookrightarrow (q, w)$ and $w' \in \Gamma^*$. For a given $C \subseteq \mathcal{C}$, we define $post^*(C) = \{c' \in \mathcal{C} \mid c \Rightarrow^* c' \text{ for some } c \in C\}$ and $pre^*(C) = \{c \in \mathcal{C} \mid c \Rightarrow^* c' \text{ for some } c' \in C\}$.

The *reachability problem* for PDSs is to decide whether $c \Rightarrow^* c'$ for configurations c and c', and it is equivalent to asking whether $c' \in post^*(\{c\})$ or equivalently whether $c \in pre^*(\{c'\})$. Büchi [7] showed that for any regular set $C \subseteq \mathcal{C}$, the sets $post^*(C)$ and $pre^*(C)$ are also regular.

To represent regular sets of pushdown configurations, we use *P-automata* [44], which are nondeterministic finite automata with multiple initial states for each of the control locations from the set P. Formally, let N be a finite set of noninitial states and $F \subseteq P \cup N$ a finite set of final states. A *P-automaton* is a tuple $\mathcal{A} = (P \cup N, \rightarrow, P, F)$ with the transition relation $\rightarrow \subseteq (P \cup N) \times \Gamma \times (P \cup N)$ so that $P \cup N$ is the set of its states and the pushdown alphabet Γ is the input alphabet of the automaton. The language $L(\mathcal{A})$ of P-automaton $\mathcal{A}$ contains the pushdown configurations accepted by $\mathcal{A}$: a configuration $(p, w) \in P \times \Gamma^*$ is accepted if and only if there is a path from p to q for some $q \in F$ in the P-automaton (defined via the transition relation $\rightarrow$) labelled with w. The *reachability problem for P-automata* is as follows: given a PDS (P, Γ, Δ) and P-automata $\mathcal{A}_1$ and $\mathcal{A}_2$, does there exist $c \in L(\mathcal{A}_1)$ and $c' \in L(\mathcal{A}_2)$ such that $c \Rightarrow^* c'$ using the rules Δ?

II. Isabelle/HOL

Isabelle/HOL [37] is a proof assistant based on classical higher-order logic (HOL), a simply typed lambda calculus with Hilbert choice, axiom of infinity, and rank-1 polymorphism. We present our formalization using HOL's syntax, which mixes functional programming and mathematical notation.

Types are built from type variables $'a$, $'b$, ... and type constructors like pairs $\_ \times \_$ and functions $\_ \Rightarrow \_$ (both written infix) and sets $\_\ set$ (written postfix). Type constructors can also be nullary, e.g., the Boolean type $bool$. Type variables can be restricted by type classes: $'a :: finite$ is a type variable $'a$ that can only be instantiated with finite types (i.e., types with finitely many inhabitants). New type constructors are introduced as abbreviations for complex type expressions and as inductive datatypes using commands **type_synonym** and **datatype** respectively, e.g., the types of transitions **type_synonym** $('state, 'label)\ transition = 'state \times 'label \times 'state$ and finite lists **datatype** $'a\ list = [] \mid 'a \# ('a\ list)$.

Terms are built from variables x, y, ..., constants c, d, ..., lambda abstractions $\lambda x.\ t$ and applications written as juxtaposition $f\ x$. Isabelle includes many constants and syntax for them, e.g., infix operators $\wedge$, $\vee$, $\longrightarrow$, $\longleftrightarrow$, $\in$, $\cup$, $\cap$, unbounded and bounded quantifiers $\exists x.\ P\ x$ and $\forall y \in A.\ Q\ y$, and set comprehensions $\{x.\ P\ x\}$. Non-recursive functions are defined and given readable syntax using the **definition** command:

definition image (**infix** ‘) **where**
f ‘ $A = \{y.\ \exists x \in A.\ y = f\ x\}$

Type annotations like $\mathsf{image} :: ('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'b\ set$ can be omitted as they are inferred. Recursive definitions are supported using the **fun** command:

fun append (**infix** @) **where**
$[] \ @\ ys = ys \mid (x \# xs)\ @\ ys = x \# (xs\ @\ ys)$

```
locale LTS = fixes trans_rel :: ('state, 'label) transition set
begin
  definition step_relp (infix ⇒) where
    c ⇒ c' ⟷ (∃l. (c, l, c') ∈ trans_rel)
  definition step_starp (infix ⇒*) where
    c ⇒* c' ⟷ step_relp** c c'
  definition pre_star C = {c'. ∃c ∈ C. c' ⇒* c}
  definition post_star C = {c'. ∃c ∈ C. c ⇒* c'}
  definition srcs = {p. ∄q γ. (q, γ, p) ∈ trans_rel}
  definition sinks = {p. ∄q γ. (p, γ, q) ∈ trans_rel}
  inductive_set trans_star where
    (p, [], p) ∈ trans_star
  | (p, γ, q') ∈ trans_rel ⟶ (q', w, q) ∈ trans_star ⟶
    (p, γ#w, q) ∈ trans_star
end
```

Fig. 1: The locale for labeled transition systems

Internally, **fun** performs an automatic termination proof. More complex recursion schemes may require a manual proof.

Another way to define a function is as Prolog-style monotone rules. The **inductive** command allows such definitions as least fixed points. Take, e.g., the reflexive transitive closure:

```
inductive rtranclp (_**) where
  R** x x | R x y ⟶ R** y z ⟶ R** x z
```

Theorems and lemmas are terms of type *bool* that have been proved to be equivalent to True. All proofs pass through Isabelle's kernel, which relies only on a few well-understood reasoning rules such as modus ponens. We refer to a textbook [36] for a practical introduction to proving in Isabelle.

Structures and assumptions common to many theorems can be organized via locales [1]—Isabelle's module mechanism for fixing parameters and stating and assuming their properties. In the context of a locale, the parameters are available as constants and the assumptions as facts. Locales can be interpreted, which involves instantiating the parameters and proving the assumptions. As the result, one obtains the (instantiated) theorems proved in the context of the locale.

Consider our locale for labeled transition systems (LTSs) in Fig. 1. It fixes the parameter trans_rel, and its context consists of the definitions between the **begin** and **end** keywords. All definitions should be self-explanatory except perhaps trans_star: the set of triples (p, w, q) for which the LTS can move from p to q by consuming word w. This relation is defined inductively, first for the empty sequence and then extending it by one more symbol—here we use in conjunction two assumptions on the symbol γ and sequence w. (Following an Isabelle convention, we formalize it equivalently as two implications.) In the formalization, the locale has more definitions than shown here and a number of lemmas. Outside LTS's context, we can access its definitions, e.g., pre_star is available under the name LTS.pre_star and can be applied to any transition relation A and a set of states C as follows: LTS.pre_star A C.

```
datatype 'label op = pop | swap 'label | push 'label 'label
type_synonym ('ctr_loc, 'label) rule =
  ('ctr_loc × 'label) × ('ctr_loc × 'label op)
type_synonym ('ctr_loc, 'label) conf = 'ctr_loc × 'label list

locale PDS = fixes Δ :: ('ctr_loc, 'label:: finite) rule set
begin
  fun lbl where
    lbl pop = [] | lbl (swap γ) = [γ] | lbl (push γ γ') = [γ, γ']
  definition is_rule (infix ↪) where
    (p, γ) ↪ (p', w) ⟷ ((p, γ), (p', w)) ∈ Δ
  inductive_set step where
    (p, γ) ↪ (p', w) ⟶
      ((p, γ # w'), (), (p', lbl w @ w')) ∈ step
  interpretation LTS step .
end

datatype ('ctr_loc, 'noninit) state =
  Init 'ctr_loc | Noninit 'noninit

locale PDS_with_finals = PDS Δ
    for Δ :: ('ctr_loc :: enum, 'label :: finite) rule set +
    fixes F_inits :: 'ctr_loc set and F_noninits :: 'noninit set
begin
  definition finals = Init ` F_inits ∪ Noninit ` F_noninits
  definition inits = {q. ∃p. q = Init p}
  definition accepts A (p, w) =
    (∃q ∈ finals. (Init p, w, q) ∈ LTS.trans_star A)
  definition lang A = {c. accepts A c}
end
```

Fig. 2: The types and locales for pushdown systems

III. Pushdown Reachability

We formalize pushdown systems (PDSs) and saturation algorithms for calculating pre^* and $post^*$ following Schwoon [44] and $dual^*$ following Jensen et al. [23].

Fig. 2 shows our modeling of PDSs. We use type variables to represent control locations ($'ctr\_loc$) and stack labels ($'label$). We introduce types for operations ($'label\ op$), rules (($'ctr\_loc, 'label$) $rule$) and configurations (($'ctr\_loc, 'label$) $conf$). A PDS is given by the locale PDS, which fixes a set of rules Δ. Each PDS gives rise to an unlabeled transition relation, which we model by an LTS step with label ()—the only element of type $unit$. The definition is a non-recursive **inductive** definition. We use the **interpretation** command to interpret LTS with step. This means that pre_star refers to LTS.pre_star step in PDS. Likewise, trans_star refers to LTS.trans_star step and similarly for other LTS definitions. The type ($'ctr\_loc, 'noninit$) $state$ represents P-automata states, where $'noninit$ is the type variable for noninitial states. The locale PDS_with_finals extends PDS with a set of final initial states F_inits and final noninitial states F_noninits. For the rest of this section, we work within the PDS_with_finals locale. In this locale, a P-automaton is a set of transitions.

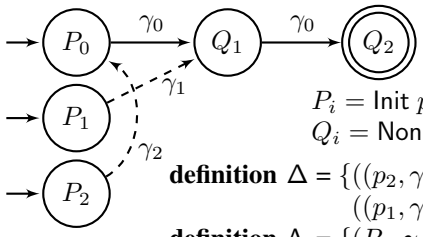

Fig. 3: Adding two transitions (dashed arrows) to a P-automaton. Initially (solid arrows) the P-automata encodes only configuration $(p_0, [\gamma_0, \gamma_0])$. After saturation, the configurations $(p_1, [\gamma_1, \gamma_0])$ and $(p_2, [\gamma_2, \gamma_0, \gamma_0])$ are also encoded.

A. Nondeterministic pre^* Saturation

Schwoon [44] presents the pre^* saturation which is a nondeterministic algorithm that given a P-automaton A returns a P-automaton whose language is pre_star (lang A). The algorithm proceeds by iteratively adding transitions to A. In each step, the algorithm nondeterministically chooses an transition to add that satisfies a number of criteria. The P-automaton is saturated when no more transitions can be added. We formalize a step of the algorithm by the relation:

inductive pre_star_rule **where**
(Init $p, \gamma, q) \notin A \longrightarrow (p, \gamma) \hookrightarrow (p', w) \longrightarrow$
(Init p', lbl $w, q) \in$ LTS.trans_star $A \longrightarrow$
pre_star_rule A $(A \cup \{(\text{Init } p, \gamma, q)\})$

The pre_star_rule relation relates two P-automata if the latter can be obtained from the former via one step of the algorithm. The criteria of the algorithm are expressed as the premises of the implication shown in pre_star_rule's definition. The last two premises are taken directly from Schwoon's definition of the algorithm and the first one ensures that the transition we add into the new P-automaton is a new one. A single P-automaton can be related to different P-automata via pre_star_rule, which captures nondeterministic choice.

Consider the PDS defined by Δ in Fig. 3, and let the P-automaton A consist of the two solid transitions in the figure. Let A′ be A$\cup\{(P_2, \gamma_2, P_0)\}$. Notice that $(P_2, \gamma_2, P_0) \notin$ A and $(p_2, \gamma_2) \hookrightarrow (p_0, \text{pop})$ and $(P_0, \text{lbl pop}, P_0) \in$ LTS.trans_star A. From pre_star_rule's definition then follows that pre_star_rule A A′. Let A″ be A′ $\cup \{(P_1, \gamma_1, Q_1\})$. From pre_star_rule's definition it follows that pre_star_rule A′ A″.

We formalize what it means for a P-automaton A to be saturated w.r.t a rule *r*, and for A' to be a saturation of A:

definition saturated $r\ A = (\nexists A'.\ r\ A\ A')$
definition saturation $r\ A\ A' = (r^{**}\ A\ A' \wedge$ saturated $r\ A')$

In our example, A″ is saturated and thus formally we have saturated pre_star_rule A″ and saturation pre_star_rule A A″.

We next prove the pre^* saturation algorithm correct. Here, we focus on the proof's most interesting aspects, especially those where we had to deviate from Schwoon's pen-and-paper proof, and refer to our formalization for full details [40].

The correctness theorem states that if a transition system A' is a saturation of a transition system A then the language of A' is indeed the pre^* closure of the language of A. Like Schwoon, we assume that the initial states are sources:

theorem *pre_star_rules_correct*:
assumes inits $\subseteq$ LTS.srcs A
and saturation pre_star_rule A A'
shows lang A' = pre_star (lang A)

Schwoon's Lemma 3.1 is used to prove the $\supseteq$ direction of the theorem's conclusion. He proves it by considering an arbitrary predecessor configuration (p', w) of a configuration (p, v) in A's language. The proof proceeds by induction on the number of $\Rightarrow$ transitions from (p', w) to (p, v). We do not keep track of this number, but we instead prove the lemma by induction on the transitive and reflexive closure of $\Rightarrow$. The formalization of the proof is written in Isabelle's structured proof language Isar (not shown) and follows Schwoon's arguments. Schwoon's Lemma 3.2 is used to prove the $\subseteq$ direction of *pre_star_rules_correct*'s conclusion. We showcase Lemma 3.2 in Schwoon's formulation, but adapted to our notation:

Lemma 3.2 If saturation pre_star_rule A A' and $(p, w, q) \in$ LTS.trans_star A' then:

(a) $(p, w) \Rightarrow^* (p', w')$ for a configuration (p', w') such that $(p', w', q) \in A$;
(b) moreover, if q is an initial state, then $w' = []$.

In his proof, Schwoon claims to prove (a) by an induction and then that (b) will follow immediately from a simple argument. However, reading his proof we notice that he uses (b) in the proof of (a). We resolve this by noticing that we can strengthen (b) to hold for any stack w and not just the one w' claimed to exist in (a). Our formulation of (b) looks as follows:

lemma *word_into_init_empty*:
assumes $(p, w, \text{Init } q) \in$ LTS.trans_star A
and inits $\subseteq$ LTS.srcs A
shows $w = [] \wedge p = \text{Init } q$

We prove (a) using the strengthened version of (b). Like Schwoon, we prove (a) by a nested induction. His outer induction is on the number of times the algorithm added transitions to the P-automaton. We instead prove the lemma by induction on the transitive reflexive closure of pre_star_rule. The inner induction is more challenging to formalize. Here, Schwoon considers a specific transition t which he defines as the ith transition added to P-automaton A. In the same context he considers a word w and two states, Init p and q, such that (Init $p, w, q) \in$ LTS.trans_star A'. He then defines j as the number of times t is used in (Init $p, w, q) \in$ LTS.trans_star A'. We may argue that this number is not well-defined, because there can be several paths from Init p to q consuming w, and on these paths t may not occur the same number of times. It turns out we can choose among these paths completely freely—any one of them will work, and so we just choose one arbitrarily. Formalizing this required us to define a variant of trans_star that keeps track of the intermediate states.

B. Nondeterministic $post^*$ Saturation

We call states with no incoming or outgoing transitions isolated. The $post^*$ saturation algorithm requires the addition

of new noninitial states that are isolated in the automaton on which the algorithm is run. Under certain conditions the algorithm adds transitions into and out of these. Each such new state corresponds to a control location and a label. We extend the datatype of states with a new constructor Isolated for these:

datatype $('ctr\_loc, 'noninit, 'label)\ state$ =
Init $'ctr\_loc$ | Noninit $'noninit$ | Isolated $'ctr\_loc$ $'label$

Moreover, we define isols $= \{q.\ \exists p.\ q =$ Isolated $p\}$.

Steps in the $post^*$ saturation are formalized as follows:

inductive post_star_rules **where**
$(p, \gamma) \hookrightarrow (p', \mathsf{pop}) \longrightarrow (\mathsf{Init}\ p', \varepsilon, q) \notin A \longrightarrow$
$(\mathsf{Init}\ p, [\gamma], q) \in \mathsf{LTS\_}\varepsilon.\mathsf{trans\_star\_}\varepsilon\ A \longrightarrow$
post_star_rules A $(A \cup \{(\mathsf{Init}\ p', \varepsilon, q)\})$
| $(p, \gamma) \hookrightarrow (p', \mathsf{swap}\ \gamma') \longrightarrow (\mathsf{Init}\ p', \mathsf{Some}\ \gamma', q) \notin A \longrightarrow$
$(\mathsf{Init}\ p, [\gamma], q) \in \mathsf{LTS\_}\varepsilon.\mathsf{trans\_star\_}\varepsilon\ A \longrightarrow$
post_star_rules A $(A \cup \{(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', q)\})$
| $(p, \gamma) \hookrightarrow (p', \mathsf{push}\ \gamma'\ \gamma'') \longrightarrow$
$(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma') \notin A \longrightarrow$
$(\mathsf{Init}\ p, [\gamma], q) \in \mathsf{LTS\_}\varepsilon.\mathsf{trans\_star\_}\varepsilon\ A \longrightarrow$
post_star_rules A $(A \cup (\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma'))$
| $(p, \gamma) \hookrightarrow (p', \mathsf{push}\ \gamma'\gamma'') \longrightarrow$
$(\mathsf{Isolated}\ p'\ \gamma', \mathsf{Some}\ \gamma'', q) \notin A \longrightarrow$
$(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma') \in A \longrightarrow$
$(\mathsf{Init}\ p, [\gamma], q) \in \mathsf{LTS\_}\varepsilon.\mathsf{trans\_star\_}\varepsilon\ A \longrightarrow$
post_star_rules A $(A \cup \{(\mathsf{Isolated}\ p'\ \gamma', \mathsf{Some}\ \gamma'', q)\})$

The relation has one rule for pop, one for swap, and two for push. It uses LTS_ε.trans_star_ε, which is similar to LTS.trans_star but allows ε-transitions that do not consume stack symbols. The transition $(\mathsf{Init}\ p', \varepsilon, q)$ is an ε-transition and $(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', q)$ is a γ'-labeled non-ε-transition. The function lang_ε returns the language of a P-automaton with ε-transitions. We prove $post^*$ saturation correct:

theorem *post_star_rules_correct*:
assumes saturation post_star_rules A A'
and inits $\subseteq$ LTS.srcs A **and** isols $\subseteq$ LTS.isolated A
shows lang_ε $A' =$ post_star (lang_ε A)

Schwoon's definition of the $post^*$ rule has only one rule for push (in contrast to our two rules). In his rule, Schwoon *first* adds a transition $(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma')$ and *then* adds a transition $(\mathsf{Isolated}\ p'\ \gamma', \mathsf{Some}\ \gamma'', q)$. Consider his rule here presented in his formulation but our notation:

If $(p, \gamma) \hookrightarrow (p', \mathsf{push}\ \gamma'\ \gamma'')$ and
$(\mathsf{Init}\ p, \gamma, q) \in \mathsf{LTS\_}\varepsilon.\mathsf{trans\_star\_}\varepsilon\ A$,
first add $(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma')$;
then add $(\mathsf{Isolated}\ p'\ \gamma', \mathsf{Some}\ \gamma'', q)$.

We were at first surprised that he specified this *first*/*then* order, but his correctness proof actually relies on it. Specifically, the order is used in his proof of Lemma 3.4, which is the key to prove the $\supseteq$ direction of *post_star_rules_correct*. We present Lemma 3.4 in Schwoon's formulation but our notation:

Lemma 3.4 If saturation post_star_rules A A' and $(\mathsf{Init}\ p, w, q) \in \mathsf{LTS\_}\varepsilon.\mathsf{trans\_star\_}\varepsilon\ A'$ then:

(a) if $q \notin$ isols, then $(p', w') \Rightarrow^* (p, w)$ for a configuration (p', w') such that $(\mathsf{Init}\, p', w', q) \in \mathsf{LTS\_}\varepsilon.\mathsf{trans\_star\_}\varepsilon\ A$;
(b) if $q = \mathsf{Isolated}\ p'\ \gamma'$, then $(p', \gamma') \Rightarrow^* (p, w)$.

Schwoon's proof is a nested induction. The outer induction is on the number of transitions $post^*$ has added. The induction step proceeds by an inner induction on the number of times the most recently added transition t was used in $(\mathsf{Init}\ p, w, q') \in \mathsf{LTS\_}\varepsilon.\mathsf{trans\_star\_}\varepsilon\ A'$. (We resolve the ambiguity of that number's meaning in a similar way as for pre^*.) The proof then proceeds by a case distinction on which of the $post^*$ saturation rules added t. Consider the case where t was added by the "first" part of the rule for push. In this case, t has the form $(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma')$. Schwoon states that "Then since $\mathsf{Isolated}\ p'\ \gamma'$ has no transitions leading into it initially, it cannot have played part in an application rule before this step, and t is the first transition leading to it. Also, there are no transitions leading away from t so far." Had Schwoon not forced the algorithm to *first* add the transition into $\mathsf{Isolated}\ p'\ \gamma'$ and *then* add the one out of it, then he could not have claimed that there are no transition leading away from t. We capture this idea in the following two lemmas, stating that if t is not present, then $\mathsf{Isolated}\ p'\ \gamma'$ must be a source and a sink:

lemma *post_star_rules_Isolated_source_invariant*:
assumes post_star_rules** A A'
and isols $\subseteq$ LTS.isolated A
and $(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma') \notin A'$
shows $\mathsf{Isolated}\ p'\ \gamma' \in$ LTS.srcs A'

lemma *post_star_rules_Isolated_sink_invariant*:
assumes post_star_rules** A A'
and isols $\subseteq$ LTS.isolated A
and $(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma') \notin A'$
shows $\mathsf{Isolated}\ p'\ \gamma' \in$ LTS.sinks A'

Formalizing Schwoon's push rule as a single rule in post_star_rules does not capture the order in which the two transition are added to the set. This is why we split the rule in two—one adding the transition into the new noninitial state and another adding the transition out of the new noninitial state. This does not yet impose the needed *first*/*then* order. However, we can impose the order by letting the latter rule be only applicable if the transition added by the former is indeed already in the automaton. This is possible because the transition added into state $\mathsf{Isolated}\ p'\ \gamma'$ is $(\mathsf{Init}\ p', \mathsf{Some}\ \gamma', \mathsf{Isolated}\ p'\ \gamma')$, and thus we can refer to the states comprising this transition in any context where $\mathsf{Isolated}\ p'\ \gamma'$ is available, in particular, the second push rule. Note that our $post^*$ saturation algorithm is slightly more general than Schwoon's as we do not require the transition out of the new noninitial state to be added immediately after the transition into it, rather we allow this to happen at any time after.

C. Combined $dual^*$ Saturation

We now consider the recent bi-directional search approach, called $dual^*$ [23]. With $dual^*$ we can check if the configurations of one P-automaton A_2 are reachable from another P-automaton A_1 by alternating between saturating A_2 towards its pre^* closure and A_1 towards its $post^*$ closure, while simultaneously (on-the-fly) keeping track of their intersection

fun (in LTS) reach **where**
  reach p [] = {p}
| reach p ($\gamma\#w$) = ($\bigcup q' \in (\bigcup(p', \gamma', q') \in$ step.
    if $p' = p \wedge \gamma' = \gamma$ then {q'} else {}). reach q' w)

definition (in PDS) pre_star1 A = ($\bigcup((p, \gamma), (p', w)) \in \Delta$.
  $\bigcup q \in$ LTS.reach A (Init p') (lbl w). {(Init p, γ, q)})

definition (in PDS) pre_star_exec = the $\circ$ while_option
  ($\lambda s.\ s \cup$ pre_star1 $s \neq s$) ($\lambda s.\ s \cup$ pre_star1 s)

Fig. 4: Executable pre^*

automaton. As soon as the intersection automaton becomes nonempty, we know that there is a state in A_2 that is reachable from A_1. This is the case even if the pre^* and $post^*$ automata are not saturated. Our correctness theorem is formalized here:

theorem *dual_star_correct_early_termination*:
  assumes inits $\subseteq$ LTS.srcs A_1 **and** inits $\subseteq$ LTS.srcs A_2
  and isols $\subseteq$ LTS.isolated $A_1 \cap$ LTS.isolated A_2
  and post_star_rules** A_1 A_1' **and** pre_star_rule** A_2 A_2'
  and lang_ε_inters (inters_ε A_1' (LTS_ε_of A_2')) $\neq$ {}
  shows $\exists c_1 \in$ lang_ε A_1. $\exists c_2 \in$ lang A_2. $c_1 \Rightarrow^* c_2$

The function LTS_ε_of trivially converts a P-automaton to a P-automaton with ε-transitions. The function inters_ε calculates the intersection P-automaton with ε-transitions of two P-automata with ε-transitions using a product construction. The function lang_ε_inters gives the language of an intersection automaton. Since the $\subseteq$ directions of *pre_star_rule_correct* and *post_star_rules_correct* do not rely on A' being saturated we prove them assuming only respectively pre_star_rule** A_2 A_2' and post_star_rules** A_1 A_1' instead of saturation pre_star_rule A_2 A_2' and saturation post_star_rules A_1 A_1'. We use these more general lemmas to prove *dual_star_correct_early_termination*.

IV. Executable Pushdown Reachability

To get an executable algorithm for pre^*, we resolve the nondeterminism by defining a functional program pre_star_exec, presented in Fig. 4 (where we indicate the corresponding locale for each definition), with this characteristic property:

theorem *pre_star_exec_language_correct*:
  assumes inits $\subseteq$ LTS.srcs A
  shows lang (pre_star_exec A) = pre_star (lang A)

The function reach is trans_star's executable counterpart: for a state p and a word w, reach p w computes the set of states reachable from p via w using step (fixed in the LTS locale). In other words, we have $q \in$ reach p w iff $(p, w, q) \in$ trans_star.

The definition of pre_star_exec uses while_option, the functional while loop counterpart. Given a test predicate b, a loop body c and a loop state s, the expression while_option b c s computes the optional state Some (c ($\cdots$(c (c s)))) not satisfying b with the minimal number of applications of c, or None if no such state exists. Our specific loop keeps adding the results of a single step pre_star1 to the P-automaton comprising the loop state. We prove that our loop never returns None,

definition nonempty A P Q =
  ($\exists p \in P.\ \exists q \in Q.\ \exists w.\ (p, w, q) \in$ trans_star A)

definition inters A B =
  {((p_1, p_2), w, (q_1, q_2)). (p_1, w, q_1) $\in A \wedge$ (p_2, w, q_2) $\in B$}

definition nonempty_inter Δ A_1 F_1 F_1^{ni} A_2 F_2 F_2^{ni} =
  nonempty (inters A_1 (pre_star_exec Δ A_2))
    (($\lambda x.\ (x, x)$) ‘ inits) (finals F_1 F_1^{ni} $\times$ finals F_2 F_2^{ni})

definition check Δ A_1 F_1 F_1^{ni} A_2 F_2 F_2^{ni} =
  (if $\neg$inits $\subseteq$ LTS.srcs A_2 then None
  else Some (nonempty_inter Δ A_1 F_1 F_1^{ni} A_2 F_2 F_2^{ni})

Fig. 5: Reachability check for P-automata

i.e., it always terminates. We thus use the, defined partially as the (Some x) = x, in pre_star_exec to extract the resulting P-automaton. The step pre_star1 computes the set of all transitions that can be added by a single application of pre_star_rule.

Fig. 4's definitions are executable: Isabelle can interpret them as functional programs and extract Standard ML, Haskell, OCaml, or Scala code [19], but it is usually not possible to extract code for inductive predicates (such as trans_star or the transitive closure in saturation) or definitions involving quantifiers ranging over an infinite domain (as in saturated). The definition of pre_star_exec has an obvious inefficiency. In every iteration, pre_star1 is evaluated twice: once as a part of the loop body and once as a part of the test. Instead we use the following improved equation, which replaces while_option with explicit recursion, for code extraction.

lemma *pre_star_exec_code*[code]:
  pre_star_exec s = (let s' = pre_star1 s in
    if $s' \subseteq s$ then s else pre_star_exec ($s \cup s'$))

With the executable algorithm for pre^*, we decide the reachability problem for P-automata using the check function shown in Fig. 5. It inputs a PDS Δ along with two P-automata represented by their transition relations (A_1 and A_2), their final initial states (F_1 and F_2) and their final noninitial states (F_1^{ni} and F_2^{ni}). The computation proceeds by intersecting (inters) the initial P-automaton with the pre^* saturation of the final P-automaton and checking the result's nonemptiness (nonempty). Fig. 5 refers to functions pre_star_exec, inits, finals, and trans_star which we introduced earlier in the context of different locales, outside of the respective locale. Therefore, these functions take additional parameters that correspond to the fixed parameters of the respective locale if they are used by the function (e.g., we write pre_star_exec Δ instead of pre_star_exec for an implicitly fixed Δ).

The definition of nonempty is not executable because of the quantification over words w. We implement, but omit here, the straightforward executable algorithm that starts with the set of initial states P and iteratively adds transitions from A until it reaches Q or saturates without reaching Q, in which case the language is empty since no state in Q is reachable from P.

Overall, check returns an optional Boolean value, where None signifies a well-formedness violation on the final

P-automaton: a non-source initial state in A_2. If check returns Some b, then b is the answer to the reachability problem for P-automata. We formalize this characterization of check by the following two theorems (phrased outside of locales).

theorem *check_None*:
check Δ A_1 F_1 F_1^{ni} A_2 F_2 F_2^{ni} = None $\longleftrightarrow$
$\neg$inits $\subseteq$ LTS.srcs A_2

theorem *check_Some*:
check Δ A_1 F_1 F_1^{ni} A_2 F_2 F_2^{ni} = Some b $\longleftrightarrow$
(inits $\subseteq$ LTS.srcs A_2 $\wedge$ (b $\longleftrightarrow$
($\exists p\ w\ p'\ w'$. step_starp Δ (p, w) (p', w') $\wedge$
$(p, w) \in$ lang A_1 F_1 F_1^{ni} $\wedge$ $(p', w') \in$ lang A_2 F_2 F_2^{ni})))

V. Differential Testing

Differential testing [14], [18], [34] is a technique for finding implementation errors by executing different algorithms solving the same problem on a set of test cases and comparing the outputs. Differential testing has been effective for finding errors in a wide range of domains, from network certificate validation [47] to JVM implementations [8]. Yet, even different algorithms do not necessarily fail independently, e.g., when built from the same specification [27] or when sharing potentially faulty components, e.g., input parsers or preprocessing. To reduce the danger of missing such errors, we suggest to incorporate a formally verified implementation in differential testing. Moreover, in case of a discrepancy the verified oracle reliably tells us which of the unverified implementations is wrong.

A. Differential Testing of Pushdown Reachability

Our executable formalization of pushdown reachability allows us to perform differential testing on unverified tools for the same problem. A test case for pushdown reachability consists of a PDS with rules Δ and two P-automata $\mathcal{A}_1$ and $\mathcal{A}_2$ representing the initial and final configurations of interest. The answer to the test case is whether there exist $c \in L(\mathcal{A}_1)$ and $c' \in L(\mathcal{A}_2)$ such that $c \Rightarrow^* c'$ using the rules Δ.

To execute the formalization on a given test case, we generate an Isabelle theory file, which first defines the control locations, labels, and automata states as finite subsets of the natural numbers (their sizes depending on the specific test case), and then includes for the pushdown rules Δ and the two P-automata, each represented by its transitions A_i along with the accepting (initial and noninitial) states F_i and $\mathsf{F}_i^{\mathsf{ni}}$ for $i \in \{1, 2\}$. Fig. 3 shows a specific example of Δ and A definitions.

We generate a lemma that uses our check function, where the expected result Some True or Some False is inserted depending on the answer produced by an unverified tool under test (invoked before generating the theory on the same inputs):

lemma check Δ A_1 F_1 $\mathsf{F}_1^{\mathsf{ni}}$ A_2 F_2 $\mathsf{F}_2^{\mathsf{ni}}$ = Some True **by** eval

The eval proof method extracts Standard ML code for check and other constants in the lemma and executes the lemma statement as an expression. It succeeds iff the lemma evaluates to True. We call a test case a counter-example, if the proof method fails. One could also run the extracted code outside Isabelle, but our setup allows us to generate the inputs to check on the formalization level instead of that of the extracted code.

Input: Reachability tools *tool* and *oracle*, PDS (P, Γ, Δ), P-automata $\mathcal{A}_i = (P \cup N_i, \rightarrow_i, P, F_i)$ for $i \in \{1, 2\}$.
Output: Minimal counter-example (failing testcase)

1: $c \leftarrow \Delta \cup (\{1\} \times (\rightarrow_1 \cup F_1)) \cup (\{2\} \times (\rightarrow_2 \cup F_2))$
▷ Convert to a set of features
2: **return** DD(c, 2) ▷ returned set of features can be converted to PDS and P-automata as on lines 10-11

3: **function** DD(c, n) ▷ c is a test case, n is granularity
4: let $c_1 \uplus \cdots \uplus c_n = c$, all c_i as evenly sized as possible
5: **if** $\exists i$. BAD(c_i) **return** DD(c_i, 2)
6: **else if** $\exists i$. BAD($c \backslash c_i$) **return** DD($c \backslash c_i$, $\max(n-1, 2)$)
7: **else if** $n < |c|$ **return** DD(c, $\min(2n, |c|)$)
8: **else return** c

9: **function** BAD(c) ▷ c is a test case
10: let $\Delta' = c \cap \Delta$ ▷ extract PDS rules and P-automata
11: for $i \in \{1, 2\}$ let $\mathcal{A}_i' = (P \cup N_i, \rightarrow_i', P, F_i')$ where $\rightarrow_i' = \{t \in \rightarrow_i \mid (i, t) \in c\}$ and $F_i' = \{q \in F_i \mid (i, q) \in c\}$
12: with both tools check if $\mathcal{A}_1'$ reaches $\mathcal{A}_2'$ via (P, Γ, Δ')
13: **return** *false* if *tool* and *oracle* agree, else *true*

Algorithm 1: Specialization of delta-debugging [51] to PDS.

To efficiently check a large number of test cases, we batch multiple definitions and lemmas into one theory file, thus reducing the overhead of starting Isabelle. We run Isabelle from the command line and check the output log for any failing eval proofs, which correspond to failing test cases.

B. Automatic Counter-Example Minimization

If differential testing finds a failing test case, we use delta-debugging [51] to automatically reduce it to a minimal failing test case to help the subsequent debugging process. We use the minimizing delta debugging algorithm [51] that sees a test case as a set of features, and works by systematically testing different subsets until a minimal failing test case is found.

We use delta debugging on any discovered counter-example and fix the set of features to contain: (i) each pushdown rule, (ii) each transition in either of the P-automata, and (iii) each final state in a P-automaton (as opposed to it not being final).

States and labels are identified by unique names, and the initial P-automata states are exactly the states mentioned in any pushdown rule in the feature set. We specialize the general delta debugging algorithm to pushdown systems as shown in Algorithm 1. The algorithm first creates the set of features and calls the recursive function DD with this set of features and the granularity 2. The function then splits the set of features into a number of equally sized subsets (according to the granularity) and checks if any of these subsets or their complements still fail. If yes, then the function tries to recursively reduce the set of features further, otherwise it will increase the granularity and try again. The function BAD converts the set

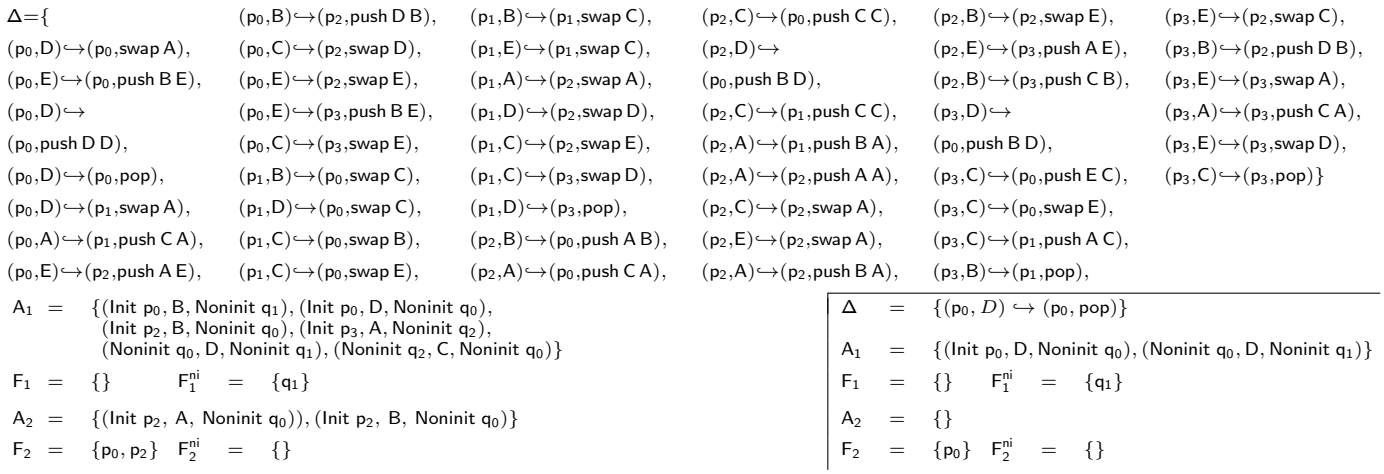

Fig. 6: Original and minimized (bottom right) counter-example

of features into a reduced pushdown system and two reduced P-automata and checks if the given tool implementation is still inconsistent with the oracle. We note that minimal failing counter-examples are only locally minimal and not necessarily unique. Yet, minimization is effective and necessary. Fig. 6 shows a real bug example we discovered by random differential testing in the PDAAAL library for pushdown reachability [23] and its minimization by Algorithm 1.

VI. Case Study: Analysis of PDAAAL

We apply differential testing with automatic counter-example minimization to PDAAAL [42], a recent C++ implementation of pushdown reachability checking, which appears to be the currently most efficient library for pushdown reachability [23]. PDAAAL implements $post^*$, pre^* and $dual^*$ [23].

These three different algorithms can be used in classical differential testing without a verified oracle, but given the large amount of shared code this is bound to miss some errors. And without a verified oracle, manual effort is needed to determine which implementation is faulty in case of discrepancies. This motivates using our verified reachability check via pre^*, and we compare the output of each unverified algorithm to the output of our trustworthy oracle on a large number of test cases.

A. Methodology of Test Case Generation

We structure our test case generation in three phases.

In phase one, we use real-world tests generated from the domain of network verification, which PDAAAL was originally built for as a backend [22]. We generate pushdown reachability problems from realistic network verification use-cases on (up to) 100 random reachability queries on each of the 260 different networks derived from the Internet Topology Zoo [28] giving a total of 25 512 test cases.

In phase two, we randomly generate valid pushdown systems and P-automata. We generate 15 000 cases of varying sizes with 4 control locations, 5 labels, up to 200 pushdown rules, and up to 13 automata transitions. Our generator writes all ingredients (pushdown system and P-automata) to a JSON file, which is then translated to the Isabelle definitions and correctness lemmas that incorporate the unverified answers.

Finally, in phase three, we exhaustively enumerate the set of all test cases up to a certain (small) size. For the pushdown systems $|P| = |\Gamma| = 2$ and $|\Delta| \leq 2$, and for P-automata $|N_1| = 2$, $|N_2| = 1$ and $|{\rightarrow}| \leq 2$. We remove symmetric cases, where swapping state names or labels gives an identical case. In total, this yields close to 27 million combinations of pushdown systems and P-automata. For the exhaustive tests, we output both JSON files and Isabelle definitions directly from the test case generator. A bash script stitches together the Isabelle definitions into a single theory file with a batch of test cases to benefit from Isabelle's parallel processing of proofs.

B. Results

The real-world test cases showed no discrepancies between the verified oracle and PDAAAL. This indicates that PDAAAL has already been thoroughly tested on this type of problem instances. Isabelle ran out of memory in 30 of the 25 512 test cases. The average CPU time (on AMD EPYC 7642 processors at 1.5 GHz) per test case was 35 seconds for Isabelle, while PDAAAL used less than 0.02 seconds on most cases.

Phase two, however, resulted in 1 334 discrepancies. By applying our counter-example minimization, we noted that all these cases had a common trait: the P-automaton $\mathcal{A}_2$ accepted the empty word. This helped us find the first implementation error in the implementation of the on-the-fly automata intersection when using $post^*$. The $post^*$ algorithm can introduce ε-transitions, which were not handled correctly by the intersection implementation. In most cases, this does not matter, as for any ε-transition followed by a normal transition the $post^*$ algorithm adds a direct transition at some later point. However, in the case of an empty stack being accepted by $\mathcal{A}_2$, this does not happen, which causes the unverified algorithm to return the wrong answer False. We resolved the error and re-ran the generated tests. After that only one discrepancy remained.

This second error was found in the implementation of pre^*. The minimized counter-example helped us find the source

10: **function** ADDTRANSITION($q_i \xrightarrow{\gamma}_i q_i'$) ▷ with $i \in \{1, 2\}$
11: add $q_i \xrightarrow{\gamma}_i q_i'$ to $\mathcal{A}_i$
12: **for all** $q_{3-i}, q_{3-i}' \in Q_{3-i}$ s.t. $(q_1, q_2) \in R$ and $q_{3-i} \xrightarrow{\gamma}_{3-i} q_{3-i}'$ **do**
13: add $(q_1, q_2) \xrightarrow{\gamma} (q_1', q_2')$ to $\mathcal{A}_\cap$
14: ADDSTATE(q_1', q_2')

(a) Snippet of (correct) intersection pseudocode by Jensen et al. [23]

2 src/include/pdaaal/Solver.h

```
@@ -119,10 +119,10 @@ namespace pdaaal {
119 119                 if (res.second) { // New edge is not already in edges (rel U workset).
120 120                     _workset.emplace(from, label, to);
121 121                     if (trace != nullptr) { // Don't add existing edges
    122 +                       _automaton.add_edge(from, to, label, trace_ptr_from<W>(trace));
122 123                         if constexpr (ET) {
123 124                             _found = _found || _early_termination(from, label, to, trace_ptr_from<W>(trace));
124 125                         }
125     -                       _automaton.add_edge(from, to, label, trace_ptr_from<W>(trace));
126 126                     }
127 127                 }
128 128             };
```

(b) PDAAAL's C++ code showing the resolution of the second error

Fig. 7: Discovered second implementation error and its correct pseudocode

of the implementation error: the set of automata transitions was updated only after calling the function that performs the nonemptiness check of the intersection automaton, but it should have been updated before that call. We argue that this error is subtle, as it only causes a single failure out of 15 000 randomly generated test cases. Fig. 7a shows the correct pseudocode by Jensen et al. [23]. Fig. 7b shows PDAAAL's corresponding C++ code and the change resolving the error, where the line that needed to be moved corresponds to the pseudocode's Line 11.

For both errors, the affected test cases resulted in a correct answer for at least one of the other search strategies in PDAAAL. This is not the case for the last error, which is found in code shared by all three methods, and where PDAAAL's algorithms disagree only with Isabelle. This error is caused by a mismatch between the assumptions of the parser that builds the pushdown system and the data structure that stores the pushdown rules. The parser assumes that it can incrementally add rules to the data structure without knowing all labels in advance, but the data structure assumes to know all labels from the start to implement a memory optimization that replaces a rule that applies to all labels by a wildcard.

For the first two test phases, the program that generated Isabelle definitions also depended on this parser, so the bug was not discovered until the third phase, which has a different setup. After the three bugs were fixed, all test cases pass.

VII. CONCLUSION

We presented a methodology that increases the reliability of tools and libraries for pushdown reachability analysis. To this end, we formalized and proved in Isabelle/HOL the correctness of the essential saturation algorithms used in such tools. We extracted an executable program from our formalization and used it as a trustworthy oracle for differential testing. Putting the modern pushdown analysis library PDAAAL on the testbed, we discovered a number of implementation errors in its code, even though the library performed flawlessly in its application domain. Using our automatic counter-example minimization based on delta-debugging, we were able to identify the sources of these errors and suggested fixes to PDAAAL's implementation that now passes all the differential tests.

This process significantly increased PDAAAL's reliability and shows that with a moderate effort, the combination of proof assistants with code generation, differential testing, and delta-debugging is highly fruitful. The execution of all tests in the three phases took 303 CPU days. We executed the tests on a compute cluster with 1 536 CPU cores. The formalization work took about two person-months for experienced formalizers, creating about 4 400 nonempty lines of Isabelle definition and proofs. An additional half person-month of work was needed to implement the differential testing and counter-example minimization, set up the tests, and localize and resolve the discovered errors. This one-time effort will also benefit the future development of PDAAAL.

Too often, the race for better performance can lead to subtle implementation errors. Our methodology shows how formally verified algorithms that were not tuned for performance can be used to improve the quality of tuned but unverified algorithms.

References

[1] Ballarin, C.: Locales: A module system for mathematical theories. J. Autom. Reason. **52**(2), 123–153 (2014). https://doi.org/10.1007/s10817-013-9284-7

[2] Basin, D.A., Dardinier, T., Heimes, L., Krstic, S., Raszyk, M., Schneider, J., Traytel, D.: A formally verified, optimized monitor for metric first-order dynamic logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS, vol. 12166, pp. 432–453. Springer (2020). https://doi.org/10.1007/978-3-030-51074-9_25

[3] Berghofer, S., Reiter, M.: Formalizing the logic-automaton connection. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 147–163. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_12

[4] Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Mazurkiewicz, A.W., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer (1997). https://doi.org/10.1007/3-540-63141-0_10

[5] Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 473–487. Springer (2005). https://doi.org/10.1007/11539452_36

[6] Braibant, T., Pous, D.: Deciding Kleene algebras in Coq. Log. Methods Comput. Sci. **8**(1) (2012). https://doi.org/10.2168/LMCS-8(1:16)2012

[7] Büchi, J.R.: Regular canonical systems. Archiv für mathematische Logik und Grundlagenforschung **6**(3-4), 91–111 (1964). https://doi.org/https://doi.org/10.1007/BF01969548

[8] Chen, Y., Su, T., Su, Z.: Deep differential testing of JVM implementations. In: Atlee, J.M., Bultan, T., Whittle, J. (eds.) ICSE 2019. pp. 1257–1268. IEEE / ACM (2019). https://doi.org/10.1109/ICSE.2019.00127

[9] Conway, C.L., Namjoshi, K.S., Dams, D., Edwards, S.A.: Incremental algorithms for inter-procedural analysis of safety properties. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 449–461. Springer (2005). https://doi.org/10.1007/11513988_45

[10] van Duijn, I., Jensen, P., Jensen, J., Krøgh, T., Madsen, J., Schmid, S., Srba, J., Thorgersen, M.: Automata-theoretic approach to verification of MPLS networks under link failures. IEEE/ACM Transactions on Networking pp. 1–16 (2021). https://doi.org/10.1109/TNET.2021.3126572

[11] Esparza, J., Knoop, J.: An automata-theoretic approach to interprocedural data-flow analysis. In: Thomas, W. (ed.) FoSSaCS 1999. LNCS, vol. 1578, pp. 14–30. Springer (1999). https://doi.org/10.1007/3-540-49019-1_2

[12] Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.: A fully verified executable LTL model checker. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 463–478. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_31

[13] Esparza, J., Schwoon, S.: A bdd-based model checker for recursive programs. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 324–336. Springer (2001). https://doi.org/10.1007/3-540-44585-4_30

[14] Evans, R.B., Savoia, A.: Differential testing: a new approach to change detection. In: Crnkovic, I., Bertolino, A. (eds.) ESEC-FSE 2007. pp. 549–552. ACM (2007). https://doi.org/10.1145/1287624.1287707

[15] Fonseca, P., Zhang, K., Wang, X., Krishnamurthy, A.: An empirical study on the correctness of formally verified distributed systems. In: Alonso, G., Bianchini, R., Vukolic, M. (eds.) EuroSys 2017. pp. 328–343. ACM (2017). https://doi.org/10.1145/3064176.3064183

[16] Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 69–87. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_5

[17] Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Bodík, R., Majumdar, R. (eds.) POPL 2016. pp. 499–512. ACM (2016). https://doi.org/10.1145/2837614.2837664

[18] Groce, A., Holzmann, G.J., Joshi, R.: Randomized differential testing as a prelude to formal verification. In: ICSE 2007. pp. 621–631. IEEE Computer Society (2007). https://doi.org/10.1109/ICSE.2007.68

[19] Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer (2010). https://doi.org/10.1007/978-3-642-12251-4_9

[20] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: Ironfleet: proving safety and liveness of practical distributed systems. Commun. ACM **60**(7), 83–92 (2017). https://doi.org/10.1145/3068608

[21] Jensen, J.S., Krøgh, T.B., Madsen, J.S., Schmid, S., Srba, J., Thorgersen, M.T.: P-Rex: fast verification of MPLS networks with multiple link failures. In: Dimitropoulos, X.A., Dainotti, A., Vanbever, L., Benson, T. (eds.) CoNEXT 2018. pp. 217–227. ACM (2018). https://doi.org/10.1145/3281411.3281432

[22] Jensen, P.G., Kristiansen, D., Schmid, S., Schou, M.K., Schrenk, B.C., Srba, J.: AalWiNes: a fast and quantitative what-if analysis tool for MPLS networks. In: Han, D., Feldmann, A. (eds.) CoNEXT 2020. pp. 474–481. ACM (2020). https://doi.org/10.1145/3386367.3431308

[23] Jensen, P.G., Schmid, S., Schou, M.K., Srba, J., Vanerio, J., van Duijn, I.: Faster pushdown reachability analysis with applications in network verification. In: Hou, Z., Ganesh, V. (eds.) ATVA 2021. LNCS, vol. 12971, pp. 170–186. Springer (2021). https://doi.org/10.1007/978-3-030-88885-5_12

[24] Jiang, D., Li, W.: The verification of conversion algorithms between finite automata. Sci. China Inf. Sci. **61**(2), 028101:1–028101:3 (2018). https://doi.org/10.1007/s11432-017-9155-x

[25] Kidd, N., Lal, A., Reps, T.: Wali: The weighted automaton library (2007), https://research.cs.wisc.edu/wpis/wpds/wali/

[26] Kincaid, Z., Breck, J., Boroujeni, A.F., Reps, T.W.: Compositional recurrence analysis revisited. In: Cohen, A., Vechev, M.T. (eds.) PLDI 2017. pp. 248–262. ACM (2017). https://doi.org/10.1145/3062341.3062373

[27] Knight, J.C., Leveson, N.G.: An experimental evaluation of the assumption of independence in multiversion programming. IEEE Trans. Software Eng. **12**(1), 96–109 (1986). https://doi.org/10.1109/TSE.1986.6312924

[28] Knight, S., Nguyen, H., Falkner, N., Bowden, R., Roughan, M.: The internet topology Zoo. IEEE Journal on Selected Areas in Comm. **29**(9), 1765 –1775 (2011)

[29] Lammich, P.: Formalization of dynamic pushdown networks in Isabelle/HOL (2009), https://www21.in.tum.de/~lammich/isabelle/dpn-document.pdf

[30] Lammich, P., Müller-Olm, M., Wenner, A.: Predecessor sets of dynamic pushdown networks with tree-regular constraints. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 525–539. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_39

[31] Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Beringer, L., Felty, A.P. (eds.) ITP 2012. LNCS, vol. 7406, pp. 166–182. Springer (2012). https://doi.org/10.1007/978-3-642-32347-8_12

[32] Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814

[33] Lesani, M., Bell, C.J., Chlipala, A.: Chapar: certified causally consistent distributed key-value stores. In: Bodík, R., Majumdar, R. (eds.) POPL 2016. pp. 357–370. ACM (2016). https://doi.org/10.1145/2837614.2837622

[34] McKeeman, W.M.: Differential testing for software. Digit. Tech. J. **10**(1), 100–107 (1998), http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[35] Minamide, Y.: Verified decision procedures on context-free grammars. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 173–188. Springer (2007). https://doi.org/10.1007/978-3-540-74591-4_14

[36] Nipkow, T., Klein, G.: Concrete Semantics - With Isabelle/HOL. Springer (2014). https://doi.org/10.1007/978-3-319-10542-0

[37] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002). https://doi.org/10.1007/3-540-45949-9

[38] Ramos, M.V.M., Almeida, J.C.B., Moreira, N., de Queiroz, R.J.G.B.: Formalization of the pumping lemma for context-free languages. J. Formaliz. Reason. **9**(2), 53–68 (2016). https://doi.org/10.6092/issn.1972-5787/5595

[39] Schlichtkrull, A., Blanchette, J.C., Traytel, D.: A verified prover based on ordered resolution. In: Mahboubi, A., Myreen, M.O. (eds.) CPP 2019. pp. 152–165. ACM (2019). https://doi.org/10.1145/3293880.3294100

[40] Schlichtkrull, A., Schou, M.K., Srba, J., Traytel, D.: Repeatability package for "Differential testing of pushdown reachability with a formally verified oracle" (2022). https://doi.org/10.5281/zenodo.6952978

[41] Schneider, J., Basin, D.A., Krstic, S., Traytel, D.: A formally verified monitor for metric first-order temporal logic. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 310–328. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_18

[42] Schou, M.K., Jensen, P.G., Kristiansen, D., Schrenk, B.C.: PDAAAL. GitHub (2021), https://github.com/DEIS-Tools/PDAAAL
[43] Schubert, P.D., Hermann, B., Bodden, E.: PhASAR: An inter-procedural static analysis framework for C/C++. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 393–410. Springer (2019). https://doi.org/10.1007/978-3-030-17465-1_22
[44] Schwoon, S.: Model checking pushdown systems. Ph.D. thesis, Technical University Munich, Germany (2002), https://d-nb.info/96638976X/34
[45] Schwoon, S.: Moped. In: http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/ (2002)
[46] Suwimonteerabuth, D., Schwoon, S., Esparza, J.: jMoped: A Java bytecode checker based on Moped. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 541–545. Springer (2005). https://doi.org/10.1007/978-3-540-31980-1_35
[47] Tian, C., Chen, C., Duan, Z., Zhao, L.: Differential testing of certificate validation in SSL/TLS implementations: An RFC-guided approach. ACM Trans. Softw. Eng. Methodol. **28**(4), 24:1–24:37 (2019). https://doi.org/10.1145/3355048
[48] Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.E.: Verdi: a framework for implementing and formally verifying distributed systems. In: Grove, D., Blackburn, S.M. (eds.) PLDI 2015. pp. 357–368. ACM (2015). https://doi.org/10.1145/2737924.2737958
[49] Wimmer, S.: Munta: A verified model checker for timed automata. In: André, É., Stoelinga, M. (eds.) FORMATS 2019. LNCS, vol. 11750, pp. 236–243. Springer (2019). https://doi.org/10.1007/978-3-030-29662-9_14
[50] Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Hall, M.W., Padua, D.A. (eds.) PLDI 2011. pp. 283–294. ACM (2011). https://doi.org/10.1145/1993498.1993532
[51] Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Software Eng. **28**(2), 183–200 (2002). https://doi.org/10.1109/32.988498