Improved Algorithm Complexities for Linear Temporal Logic Model Checking of Pushdown Systems*

Katia Hristova** and Yanhong A. Liu

Computer Science Department, State University of New York,
Stony Brook, NY 11794
katia@cs.sunysb.edu

Abstract. This paper presents a novel implementation strategy for linear temporal logic (LTL) model checking of pushdown systems (PDS). The model checking problem is formulated intuitively in terms of evaluation of Datalog rules. We use a systematic and fully automated method to generate a specialized algorithm and data structures directly from the rules. The generated implementation employs an incremental approach that considers one fact at a time and uses a combination of linked and indexed data structures for facts. We provide precise time complexity for the model checking problem; it is computed automatically and directly from the rules. We obtain a more precise and simplified complexity analysis, as well as improved algorithm understanding.

1 Introduction

Model checking is a widely used technique for verifying that a property holds for a system. Systems to be verified can be modeled accurately by pushdown systems (PDS). Properties can be modeled by linear temporal logic (LTL) formulas. LTL is a language commonly used to describe properties of systems [12,13,21] and is sufficiently powerful to express many practical properties. Examples include many dataflow analysis problems and various correctness and security problems for programs.

This paper focuses on LTL model checking of PDS, specifically on the global model checking problem [15]. The model checking problem is formulated in terms of evaluation of a Datalog program [5]. Datalog is a database query language based on the logic programming paradigm [11,1]. The Büchi PDS, corresponding to the product of the PDS and the automaton representing the inverse of the property, is expressed in Datalog facts, and a reach graph — an abstract representation of the Büchi PDS, is formulated in rules. The method described in [18] generates specialized algorithms and data structures and complexity formulas for the rules. The generated algorithms and data structures are such that

* This work was supported in part by NSF under grants CCR-0306399 and CCR-0311512 and ONR under grants N00014-04-1-0722 and N00014-02-1-0363.

** Corresponding author

given a set of facts, they compute all facts that can be inferred. The generated implementation employs an incremental approach that considers one fact at a time and uses a combination of linked and indexed data structures for facts. The running time is optimal, in the sense that each combination of instantiations of hypotheses is considered once in `O(1)` time.

Our main contributions are:

- A novel implementation strategy for the model checking problem that combines an intuitive definition of the model checking problem in rules [5] and a systematic method for deriving efficient algorithms and data structures from the rules[18].
- A precise and automatic time complexity analysis of the model checking problem. The time complexity is calculated directly from the Datalog rules, based on a thorough understanding of the algorithms and data structures generated, reflecting the complexities of implementation back into the rules.

We thus develop a model checker with improved time complexity guarantees and improved algorithm understanding.

The rest of this paper is organized as follows. Section 2 defines LTL model checking of PDS. Section 3 expresses the model checking problem by use of Datalog rules. Section 4 describes the generation of a specialized algorithm and data structures from the rules and analyzes time complexity of the generated implementation. Section 5 discusses related work and concludes.

2 Linear Temporal Logic Model Checking of Pushdown Systems

This section defines the problem of model checking PDS against properties expressed using LTL formulas, as described in [15].

2.1 Pushdown systems

A *pushdown system (PDS)* [14] is a triple (C_P, S_P, T_P), where C_P is a set of control locations, S_P is a set of stack symbols and T_P is a set of transitions. A transition is of the form $(c, s) \rightarrow (c', w)$ where c and c' are control locations, s is a stack symbol, and w is a sequence of stack symbols; it denotes that if the PDS is in control location c and symbol s is on top of the stack, the control location changes to c', s is popped from the stack, and the symbols in w are pushed on the stack, one at a time, from left to right. A *configuration* of a PDS is a pair (c, w) where c is a control location and w is a sequence of symbols from the top of the stack. If $(c, s) \rightarrow (c', w) \in T_P$ then for all $v \in {S_P}^*$, configuration (c, sv) is said to be an *immediate predecessor* of (c', wv). A *run* of a PDS is a sequence of configurations $conf_0, conf_1, ..., conf_n$ such that $conf_i$ is an immediate predecessor of $conf_{i+1}$, for $i = 0, \ldots, n-1$.

We only consider PDSs where each transition $(c, s) \rightarrow (c', w)$ satisfies $|w| \leq 2$. Any given PDS can be transformed to such a PDS. Any transition $(c, s) \rightarrow$

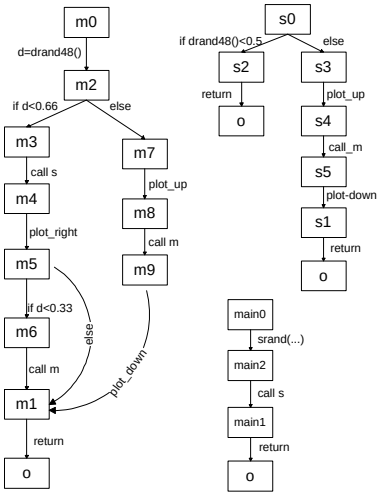
(c', w), such that $|w| > 2$, can be rewritten into $(c, s) \rightarrow (c', w_{hd}\ s')$ and $(c', s') \rightarrow (c, w_{tl})$, where w_{hd} is the first symbol in w, w_{tl} is w without its first symbol, and s' is a fresh symbol. This step can be repeated until all transitions have $|w| \leq 2$. This replaces each transition $(c, s) \rightarrow (c', w)$, where $|w| > 2$, with $|w| - 1$ transitions and introduces $|w| - 1$ fresh stack symbols.

The procedure calls and returns in a program correspond to a PDS [16]. First, we construct a control flow graph (CFG) [2] of the program. Then, we set up one control location, say called c. Each CFG vertex is a stack symbol. Each CFG edge (s, s') corresponds to a transition (i) $(c, s) \rightarrow (c, \epsilon)$, where ϵ stands for the empty string, if (s, s') is labeled with a return statement; (ii) $(c, s) \rightarrow (c, s' f_0)$, if (s, s') is labeled with a call to procedure f, and f_0 is f's entry point; (iii) $(c, s) \rightarrow (c, s')$, otherwise. A run of the program corresponds to a PDS run.

```
void m()
  double d = drand48();
  if (d < 0.66):
    s(); plot_right();
    if (d < 0.33): m();
    else:
  else:
    plot_up(); m(); plot_down();

void s()
  if (drand48() < 0.5): return;
  else:
    plot_up(); m(); plot_down();

main()
  srand48(time(NULL)); s();
```



(a) Example program. (b) Corresponding CFG.

Fig. 1: Example program and corresponding CFG.

Figure 1 shows an example program and its CFG [15]. The program creates random bar graphs using the commands `plot_up`, `plot_right`, and `plot_down`. The corresponding PDS is:

$$
\begin{aligned}
C_P &= \{c\} \\
S_P &= \{m0, m1, m2, m3, m4, m5, m6, m7, m8, m9, s0, s1, s2, s3, s4, s5, \\
&\quad main0, main1, main2\} \\
T_P &= \{(c, m3) \rightarrow (c, m4s0), (c, m6) \rightarrow (c, m1m0), (c, m8) \rightarrow (c, m9m0), \\
&\quad (c, m1) \rightarrow (c, \epsilon), (c, s2) \rightarrow (c, \epsilon), (c, s4) \rightarrow (c, s5m0), \\
&\quad (c, s1) \rightarrow (c, \epsilon), (c, main2) \rightarrow (c, main1s0), (c, main1) \rightarrow (c, \epsilon)\}
\end{aligned}
$$

2.2 Linear temporal logic formulas

Linear temporal logic (LTL) formulas [12,13,21] are evaluated over infinite sequences of symbols. The standard logic operators are available; if f and g are formulas, then so are $\neg f$, $f \wedge g$, $f \vee g$, $f \rightarrow g$. The following additional operators are available: $X\ f$: f is true in the next state; $F\ f$: f is true in some future state; $G\ f$: f is true globally, i.e. in all future states; $g\ U\ f$: g is true in all future states until f is true in some future state.

A LTL formula can be translated to a Büchi automaton, a finite state automaton over infinite words. The automaton accepts a word if on reading it a good state is entered infinitely many times. Formally, a *Büchi automaton* (BA) is a tuple $(C_B, L_B, T_B, C0_B, G_B)$ where C_B is a set of states, L_B is a set of transition labels, T_B is a set of transitions, $C0_B \subseteq C_B$ is a set of starting states, and $G_B \subseteq C_B$ is a set of good states. A transition is of the form (c, l, c'), where $c, c' \in C_B$ and $l \in L_B$. The label of a transition is a condition that must be met by the current symbol in the word being read, in order for the transition to be possible. A label _ denotes an unconditional transition. An *accepting run* of a Büchi automaton is an infinite sequence of transitions $(c_0, l_0, c_1), (c_1, l_1, c_2), \ldots, (c_{n-1}, l_{n-1}, c_n)$, where a state $c_i \in G_B$ appears infinitely many times.

To specify a program property using an LTL formula, the program's CFG edges are used as atomic propositions. LTL formulas are defined with respect to infinite runs of the program. The corresponding BA accepts an infinite sequence of CFG edges, if on reading it, the automaton enters a good state infinitely many times. For example, the property that plotting up is never immediately followed by plotting down is expressed by the LTL formula $F = G(plot\_up \rightarrow X(\neg plot\_down))$. The BA[1] corresponding to $\neg F$ is shown in Figure 2. In the diagram nodes correspond to states and edges correspond to transitions of the BA; double circles mark good states and a square marks the start state.

2.3 LTL Model checking of PDS

Given a system expressed as a PDS P, and a LTL formula F, the formula F holds for P if it holds for every run of P. We check whether F holds for P as follows [15]. First, we construct B — the BA corresponding to $\neg F$. Second, we construct BP — a Büchi PDS that is the product of P and B, and make sure BP has no accepting run. A *Büchi PDS* (BPDS) is a tuple (C, S, T, C_0, G), where C

[1] The Büchi automaton was generated with the tool LBT that translates LTL formulas to Büchi automata (http://www.tcs.hut.fi/Software/maria/tools/lbt/).

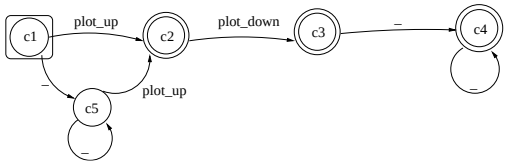


Fig. 2: Büchi automaton corresponding to $\neg G(plot\_up \rightarrow X(\neg plot\_down))$.

is a set of control locations, S is a set of stack symbols, T is a set of transitions, $C_0 \subseteq C$ is the set of starting control locations, $G \subseteq C$ is the set of good control locations. Transitions are of the form $((C * S) * (C * S^*))$. The concepts *configuration*, *predecessor*, and *run* of a BPDS are analogous to those of a PDS. An *accepting run* of the BPDS is an infinite sequence of configurations in which configurations with control locations in G appear infinitely many times. The product BPDS BP of $P = (C_P, S_P, T_P)$ and $B = (C_B, L_B, T_B, C0_B, G_B)$ is the five-tuple $((C_P * C_B), S_{BP}, T_{BP}, C0_{BP}, G_{BP})$, where $(((c_P, c_B), s), ((c'_P, c'_B), w)) \in T_{BP}$ if $(c_P, s) \rightarrow (c'_P, w) \in T_P$, and there exists f such that $(c_B, f, c'_B) \in T_B$, and f is true at configuration $((c_P, c_B), s)$; $(c_P, c_B) \in C0_{BP}$ if $c_B \in C0_B$; $(c_P, c_B) \in G_{BP}$ if $c_B \in G_B$.

Next we construct a *reach graph* — a finite graph that abstracts BP. The nodes of the graph are configurations of BP. An edge $((c, s), (c', s'))$ in the reach graph corresponds to a run that takes BP from configuration (c, s) to configuration (c', s'). If a good control location in BP is visited in the run corresponding to an edge, the edge is said to be *good*. A path in the reach graph is a sequence of edges. Cycles in the reach graph correspond to infinite runs of BP. Paths containing cycles with good edges in them correspond to accepting runs of BP and are said to be *good*. If the reach graph corresponding to BP has no good paths, BP has no accepting runs and F holds for P. Otherwise, the good paths in the reach graph are counterexamples showing that F does not hold for P.

3 Specifying the Reach Graph in Rules and Detecting Good Paths

This section expresses the reach graph using Datalog rules and employs an algorithm for detecting good paths in the reach graph as presented in [5].

A Datalog program is a finite set of relational rules of the form

$$p_1(x_{11}, ..., x_{1a_1}) \ \wedge ... \wedge \ p_h(x_{h1}, ..., x_{ha_h}) \rightarrow q(x_1, ..., x_a)$$

where h is a natural number, each p_i (respectively q) is a relation of a_i (respectively a) arguments, each x_{ij} and x_k is either a constant or a variable, and variables in x_k's must be a subset of the variables in x_{ij}'s. If $h = 0$, then there are no p_i's or x_{ij}'s, and x_k's must be constants, in which case $q(x_1, ..., x_a)$ is called a *fact*. The meaning of a set of rules and a set of facts is the smallest set

of facts that contains all the given facts and all the facts that can be inferred, directly or indirectly, using the rules.

Expressing the Büchi PDS. The BPDS is expressed by the relations `loc`, `trans0`, `trans1`, and `trans2`. The `loc` relation represents the control locations of the BPDS; its arguments are a control location and a boolean argument indicating whether the control location is good. One instance of the relation exists for each control location. The three relations `trans0`, `trans1`, and `trans2` express transitions. The facts `trans0(c1,s1,c2)`, `trans1(c1,s1,c2,s2)`, and `trans2(c1,s1,c2,s2,s3)`, where `ci`'s are control locations and `si`'s are stack symbols, denote transitions of the form of the form $((c, s), (c, w))$ such that, $w \in S^*_{BP}$ and $|w| = 0$, $|w| = 1$, and $|w| = 2$, respectively. `or` is a relation with three boolean arguments; in the fact `or(x1,x2,r)`, the argument `r` is the value of the logical *or* of the arguments `x1` and `x2`.

Expressing the edges of the reach graph. The reach graph is expressed by relations `erase` and `edge`. The fact `erase(c1,s1,g,c2)` denotes a run of BP from configuration $(c1, s1)$ to configuration $(c2, \epsilon)$. The third element in the tuple is a boolean value that indicates whether the corresponding run goes through a good control location. The `edge` relation represents the reach graph edges. `edge(c1,s1,g,c2,s2)` denotes an edge between nodes $(c1, s1)$ and $(c2, s2)$; `g` is a boolean argument indicating whether the edge is good. For a BPDS $(C_{BP}, S_{BP}, T_{BP}, C0_{BP}, G_{BP})$, `erase` and `edge` are the relation satisfying:

i. $(c1, s, g, c2) \in$`erase` if $(c1, s) \rightarrow (c2, \epsilon) \in T_{BP}$, and $g = true$ if $c1 \in G_{BP}$ and $false$ otherwise
ii. $(c1, s1, g1 \vee g2, c3) \in$`erase` if $(c1, s1) \rightarrow (c2, s2) \in T_{BP}$, and $(c2, s2, g2, c3) \in$ `erase`, and $g1 = true$ if $c1 \in G_{BP}$ and $false$ otherwise
iii. $(c1, s1, g1 \vee g2 \vee g3, c4) \in$`erase` if $(c1, s1) \rightarrow (c3, s2s3) \in T_{BP}$, $(c2, s2, g2, c3) \in$`erase`, and $(c3, s3, g3, c4) \in$`erase`, and $g1 = true$ if $c1 \in G_{BP}$ and $false$ otherwise

and

i. $(c1, s1, g, c2, s2) \in$`edge` if $(c1, s1) \rightarrow (c2, s2) \in T_{BP}$, and $g = true$ if $c1 \in G_{BP}$ and $false$ otherwise
ii. $(c1, s1, g, c2, s2) \in$`edge` if $(c1, s1) \rightarrow (c2, s2s3) \in T_{BP}$, $g = true$ if $c1 \in G_{BP}$ and $false$ otherwise
iii. $(c1, s1, g1 \vee g2, c3, s3) \in$`edge` if $(c1, s1) \rightarrow (c2, s2s3) \in T_{BP}$, $(c2, s2, g2, c3) \in$ `erase`, and $g = true$ if $c1 \in G_{BP}$ and $false$ otherwise

In model checking of programs, the relation `erase` summarizes the effects of procedures. The three parts of the above definition correspond to the program execution exiting, proceeding within, or entering a procedure.

The definitions of the `erase` and `edge` relations can be readily written as rules. These rules are shown in Figure 3.

Detecting good paths. Checking that the BPDS accepts the empty language amounts to checking that the resulting reach graph has no good paths. To find

```
trans0(c1,s1,c2)∧loc(c1,g)→erase(c1,s1,g,c2)
trans1(c1,s1,c2,s2)∧erase(c2,s2,g2,c3)∧loc(c1,g1)∧or(g1,g2,g)
    →erase(c1,s1,g,c3)
trans2(c1,s1,c2,s2,s3)∧erase(c2,s2,g2,c3)∧erase(c3,s3,g3,c4)∧
    loc(c1,g1)∧or(g1,g2,g4)∧or(g4,g3,g)→erase(c1,s1,g,c4)
trans1(c1,s1,c2,s2)∧loc(c1,g)→edge(c1,s1,g,c2,s2)
trans2(c1,s1,c2,s2,s3)∧loc(c1,g)→edge(c1,s1,g,c2,s2)
trans2(c1,s1,c2,s2,s3)∧erase(c2,s2,g2,c3)∧loc(c1,g1)∧or(g1,g2,g)
    →edge(c1,s1,g,c3,s3)
```

Fig. 3: Rules corresponding to the `erase` relation used to construct the reach graph, and the `edge` relation of the reach graph.

good paths in the reach graph we use the algorithm presented in [5, Figure 4] but ignore consideration of resource labels by the algorithm. The algorithm uses depth first search and is linear in the number of edges in the reach graph.

4 Efficient Algorithm for Computing the Reach Graph

This section describes the generation of a specialized algorithm and datastructures for computing the reach graph from the rules shown in the previous section, as well as analyzing precisely the time complexity for computing the reach graph and expressing the complexity in terms of characterizations of the facts—the parameters characterizing the BPDS.

4.1 Generation of efficient algorithms and data structures

Transforming the set of rules into an efficient implementation uses the method in [18]. We first transform each rule with more than two hypotheses into multiple rules with two hypotheses each and then carry out three key steps. Step 1 transforms the least fixed point (LFP) specification of the rule set to a `while`-loop. Step 2 transforms expensive set operations in the loop into incremental operations. Step 3 designs appropriate data structures for each set, so that operations on it can be implemented efficiently. These three steps correspond to dominated convergence [10], finite differencing [20], and real-time simulation [19], respectively, as studied by Paige et al.

Auxiliary relations. For each rule with more than two hypotheses, we transform it to multiple rules with two hypotheses each. The transformation introduces auxiliary relations with necessary arguments to combine two hypotheses at a time. We repeatedly apply the following transformations to each rule with more than two hypotheses until only rules with at most two hypotheses are left. We replace any two hypotheses of the rule, say $P_i(X_{i1}, ..., X_{ia_i})$ and $P_j(X_{j1}, ..., X_{ja_j})$ by a new hypothesis, $Q(X_1, ..., X_a)$, where Q is a fresh relation, and X_k's are

variables in the arguments of P_i or P_j that occur also in the arguments of other hypotheses or the conclusion of this rule. We add a new rule:
$P_i(X_{i1}, ..., X_{ia_i}) \wedge P_j(X_{j1}, ..., X_{ja_j}) \rightarrow Q(X_1, ..., X_a)$.

```
1.  loc(c1,g)∧trans0(c1,s1,c2)→erase(c1,s1,g,c2)
2.  loc(c1,g1)∧trans1(c1,s1,c2,s2)→gtrans1(c1,g1,s1,c2,s2)
3.  gtrans1(c1,g1,s1,c2,s2)∧erase(c2,s2,g2,c3)→gtrans1e(c1,s1,c3,g1,g2)
4.  gtrans1e(c1,s1,c3,g1,g2)∧or(g1,g2,g)→erase(c1,s1,g,c3)
5.  loc(c1,g1)∧trans2(c1,s1,c2,s2,s3)→gtrans2(c1,g1,s1,c2,s2,s3)
6.  gtrans2(c1,g1,s1,c2,s2,s3)∧erase(c2,s2,g2,c3)
        →gtrans2e(c1,s1,s2,c3,g1,g2)
7.  gtrans2e(c1,s1,s2,c3,g1,g2)∧erase(c3,s2,g3,c4)
        →gtrans2ee(c1,s1,c4,g1,g2,g3)
8.  gtrans2ee(c1,s1,c4,g1,g2,g3)∧or(g1,g2,g4)
        →gtrans2ee_or(c1,s1,c4,g3,g4)
9.  gtrans2ee_or(c1,s1,c4,g3,g4)∧or(g4,g3,g)→ erase(c1,s1,g,c4)
10. gtrans1(c1,g,s1,c2,s2)→ edge(c1,s1,g,c2,s2)
11. gtrans2(c1,g,s1,c2,s2,s3)→ edge(c1,s1,g,c2,s2)
12. gtrans2e(c1,s1,s2,c2,g1,g2)∧or(g1,g2,g)→edge(c1,s1,g,c2,s2)
```

Fig. 4: The reach graph expressed in rules with at most two hypotheses.

The resulting rule set for constructing the reach graph is shown in Figure 4. Several auxiliary relations have been introduced. The relations `gtrans1` and `gtrans2` represent transitions like `trans1` and `trans2` respectively, but an extra argument indicates whether the transitions start at a good control location. The relations `gtrans1e` and `gtrans2e`, represent runs of the BPDS starting with a transition `trans1` and `trans2` respectively, followed by a run represented as a fact of the `erase` relation. The facts `gtrans1e(c1,s1,c2,g1,g2)` and `gtrans2e(c1,s1,s2,c2,g1,g2)` represent runs from configuration $(c1, s1)$ to configurations $(c2, \epsilon)$ and $(c2, s2)$ respectively, where `g1` and `g2` indicate, respectively, whether the first control location in the run is good and whether the rest of the run visits a good control location. The relation `gtrans2ee` represents runs consisting of one transition and two runs expressed as facts of the `erase` relation. The fact `gtrans2ee(c1,s1,c2,g1,g2,g3)` stands for a run from configuration $(c1, s1)$ to configuration $(c2, \epsilon)$; the arguments `g1`, `g2`, and `g3` are booleans indicating respectively, whether the first control location in the run is good, and whether the remaining two parts of the run visit a good control location. The relations `gtrans1ee_or` and `gtrans2ee_or` represents runs like `gtrans1ee` and `gtrans2ee`, except with two boolean arguments combined using logical or.

Fixed-point specification and `while`-loop. We represent a relation the form `Q(a1, a2, ... , an)` using tuples of the form `[Q,a1,a2,...,an]`. We use S `with` X and S `less` X to mean $S \cup \{X\}$ and $S - \{X\}$, respectively. We

use the notation $\{X : Y_1$ `in` $S_1, \ldots, Y_n$ `in` $S_n | Z\}$ for set comprehension. Each Y_i enumerates elements of S_i; for each combination of $Y_1, \ldots, Y_n$ if the value of boolean expression Z is true, then the value of expression X forms an element of the resulting set. If Z is omitted, it is implicitly the constant *true*.

`LFP`(S_0, F) denotes the minimum element S, with respect to the subset ordering $\subseteq$, that satisfies the condition $S_0 \subseteq S$ and $F(S) = S$. We use standard control constructs `while`, `for`, `if`, and `case`, and we use indentation to indicate scope. We abbreviate $X := X$`op`Y as X`op`$:= Y$.

We use set `bpds` for the set of facts representing the BPDS.

```
rbpds = {[loc,c1,g] : loc(c1,g) in bpds} ∪
  {[trans0,c1,s1,c2] : trans0(c1,s1,c2) in bpds} ∪
  {[trans1,c1,s1,c2,s2] : trans1(c1,s1,c2,s2) in bpds} ∪
  {[trans2,c1,s1,c2,s2,s3] : trans0(c1,s1,c2,s2,s3) in bpds },
```

Given any set of facts `R`, and a rule with rule number `n` and with relation `e` in the conclusion, let `ne(R)`, referred to as *result set*, be the set of all facts that can be inferred by rule `n` given the facts in `R`. For example,

```
2gtrans1 = {[gtrans c1 s1 g c2 s2] : [loc c1 g] in R and
                                     [trans1 c1 g s1 c2 s2] in R},
10edge = {[edge c1 s1 g c2 s2] : [gtrans1 c1 g s1 c2 s2] in R}.
```

The meaning of the give facts and the rules used to compute the reach graph is:

```
LFP({},F), where F(R) = rbpds ∪ 1erase(R) ∪ 2gtrans1(R) ∪
        3gtrans1e(R) ∪ 4erase(R) ∪ 5gtrans2(R) ∪ 6gtrans2e(R) ∪
        7gtrans2ee(R) ∪ 8gtrans2ee_or(R) ∪ 9erase(R) ∪
        10edge(R) ∪ 11edge(R) ∪ 12edge(R).
```

This least-fixed point specification of computing the reach graph is transformed into the following `while`-loop:

```
R := {}; while exists x in F(R) - R
   R with := x;
```
(1)

The idea behind this transformation is to perform small update operations in each iteration of the `while`-loop.

Incremental computation. Next we transform expensive set operations in the loop into incremental operations. The idea is to replace each expensive expression *exp* in the loop with a variable, say E, and maintain the invariant $E = exp$, by inserting appropriate initializations and updates to E where variables in *exp* are initialized and updated, respectively.

The expensive expressions in constructing the reach graph are all result sets, such as `2grtrans1(R)`, and `F(R)-R`. We use fresh variables to hold each of their respective values and maintain the following invariants:

```
Ibpds = rbpds, I1erase = 1erase(R),
I2gtrans1 = 2gtrans1(R), I3gtrans1e = 3gtrans1e(R),
I4erase = 4erase(R), I5gtrans2 = 5gtrans2(R),
I6gtrans2e = 6gtrans2e(R), I7gtrans2ee = 7gtrans2ee(R),
I8gtrans2ee_or = 8gtrans2ee_or(R), I9erase = 9erase(R),
I10edge = 10edge(R), I11edge = 11edge(R), I12edge = 12edge(R),
W = F(R) - R.
```

`W` serves as the workset. As an example of incremental maintenance of the value of an expensive expression, consider maintaining the invariant `I2gtrans1`. `I2gtrans1` is the value of the set formed by joining elements from the set of facts of the `loc` and `trans1` relations. `I2gtrans1` can be initialized to {} with the initialization `R = {}`. To update `Igtrans1` incrementally with update `R with:= x`, if `x` is of the form `[loc,c1,g]` we consider all matching tuples of the form `[trans1,c1,s1,c2,s2]` and add the tuple `[gtrans1,c1,g,s1,c2,s2]` to `I2gtrans1`. To form the tuples to add, we need to efficiently find the appropriate values of variables that occur in `[trans1,c1,s1,c1,s2]` tuples, but not in `[loc,c1,g]`, i.e. the values of `s1,c2,` and `s2`, so we maintain an auxiliary map that maps `[c1]` to `[s1,c2,s2]` in the variable `I2gtrans1_trans1` shown below. Symmetrically, if `x` is a tuple of `[trans1,c1,s1,c2,s2]`, we need to consider every matching tuple of `[loc,c1,g]` and add the corresponding tuple of `[gtrans1,c1,g,s1,c2,s2]` to `I2gtrans1_loc`. The first set of elements in auxiliar maps is referred to as the *anchor* and the second set of elements as the *nonanchor*.

```
I2gtrans1_trans1 = {[[c1], [s1,c2,s2]] :
                          [trans1,c1,s1,c2,s2] in R},
I2gtrans1_loc = {[[c1], [g]] : [loc,c1,g] in R}.
```

Thus, we are able to directly find only matching tuples and consider only combinations of facts that make both hypotheses true simultaneously, as well as consider each combination only once. Similarly, such auxiliary maps are maintained for all invariants that we maintain.

All variables holding the values of expensive computations listed above and auxiliary maps are initialized together with the assignment `R := {}` and updated incrementally together with the assignment `R with:= x` in each iteration. When `R` is {}, `Ibpds = rbpds`, all auxiliary maps are initialized to {}, and `W = Ibpds`. When a fact is added to `R` in the loop body, the variables are updated. We show the update for the addition of a fact of relation `trans1` only for `I2gtrans1` invariant and `I2gtrans1_loc` auxiliary map , since other facts and updates to the variables and auxiliary maps are processed in the same way. The notation `E{Ys}`, where `E = {[Ys,Xs]}` is an auxiliary map, is used to access all matching

tuples of E and return all matching values of Xs.

```
case of x of [loc,c1,g]:
I2gtrans1 +:= {[gtrans1,c1,g,s1,c2,s2]:
    [s1,c2,s2] in I2gtrans1_trans1{c1}};
W +:= {[gtrans1,c1,g,s1,c2,s2]: [s1,c2,s2] in I2gtrans1_trans1{c1}
    |[gtrans1,c1,g,s1,c2,s2] notin R};
I2gtrans1_loc with:= {[[c1], [g]] : [loc,c1,g] in R};
```
(2)

Using the above initializations and updates, and replacing all invariant maintenance expressions with W, we obtain the following complete code:

```
initialization; R:={};
while exits x in W:
   update; W less:= x; R with:= x;
```
(3)

We next eliminate dead code and clean up the code to contain only uniform operations and set elements for data structure design. We then decompose R and W into several sets, each corresponding to a single relation that occurs in the rules. R is decomposed to Rtrans0, Rtrans1, Rtrans2, Rloc, Rerase, Rgtrans1, Rgtrans1e, Rgtrans2, Rgtrans2e, Rgtrans2ee, Rgtrans2ee_or, and Redge. W is decomposed in the same way. We eliminate relation names from the first component of tuples and transform the while-clause and case-clause appropriately. Then, we do the following three sets of transformations. We transform operations on sets into loops that use operations on set elements. Each addition of a set is transformed to a for-loop that adds the elements one at a time. For example, `I2gtrans1 +:= {[gtrans1,c1,g,s1,c2,s2]: [s1,c2,s2] in I2gtras1_trans1{c1}}` is transformed into:

```
for [s1,c2,s2] in I2gtras1_trans1{c1}:
    I2gtrans1 +:= [c1,g,s1,c2,s2];
```

We replace tuples and tuple operations with maps and map operations. We make all element addition and deletion easy by testing membership first.

Data structures. After the above transformations each firing of a rule takes a constant number of set operations. Since each of these set operations takes worst case constant time in the generated code, achieved as described below, each firing of a rule takes worst case constant time. Next we describe how to guarantee that each set operation takes worst-case constant time. The operations are of the following kinds: set initialization $S := \{\}$, computing image set $M(X)$, element retrieval `for` X `in` S and `while exists` X `in` S, membership test X `in` S, X `notin` S, and element addition S `with` X and deletion S `less` X. We use *associative access* to refer to membership test and computing image set.

A uniform method is used to represent all sets and maps, using arrays for sets that have associative access, linked lists for sets that are traversed by loops and both arrays and linked lists when both operations are needed.

The result sets, such as Rtrans0, are represented by nested array structures. Each of the result sets of, say, a components is represented using an a-level nested

array structure. The first level is an array indexed by values in the domain of the first component of the result set; the k-th element of the array is null if there is no tuple of the result set whose first component has value `k`, and otherwise is `true` if `a=1`, and otherwise is recursively an `(a-1)`-level nested array structure for remaining components of tuples of result sets whose first component has value `k`.

The worksets, such as `Wtrans0`, are represented by arrays and linked lists. Each workset is represented the same as the corresponding resultset with two additions. First, for each array we add a linked list linking indices of non-null elements of the array. Second, to each linked list we add a tail pointer. One or more records are used to put each array, linked list, and tail pointer together. Each workset is represented simply as a nested queue structure (without the underlying arrays), one level for each workset, linking the elements (which correspond to indices of the arrays) directly.

Auxiliary maps, such as `I2gtrans1_trans1` and `I2gtrans1_loc`, are implemented as follows. Each auxiliary map, say `E` for a relation that appears in a rule's conclusion uses a nested array structure as resultsets and worksets do and additionally linked lists for each component of the non-anchor as worksets do. `E` uses a nested array structure only for the anchor, where elements of the arrays of the last component of the anchor are each a nested linked-list structure for the non-anchor.

4.2 Complexity analysis of the model checking problem

We analyze the time complexity of the model checking problem by carefully bounding the number of facts actually used by the rules. For each rule we determine precisely the number of facts processed by it, avoiding approximations that use the sizes of individual argument domains.

Calculating time complexity. We first define the size parameters used to characterize relations and analyze complexity. For a realtion `r` we refer to the number of facts of `r` that are given or can be inferred as `r`'s *size*. The parameters `#trans0`, `#trans1` and `#trans2` denote the number of transitions of the form $((c1, s1), (c2, \epsilon))$, $((c1, s1), (c2, s2))$, and $((c1, s1), (c2, s2s3))$, respectively; `#trans` denotes the total number of transitions. The parameters `#gtrans1` and `#gtrans2` denote the number of facts of relations `gtrans1` and `gtrans2`, where `#gtrans1`=`#trans1` and `#gtrans2`=`#trans2`. Parameters `#gtrans1e` and `#gtrans2e` denote the relation sizes — `#trans1` $*$ `#target_loc_trans0`, and `#trans2` $*$ `#target_loc_trans0`, respectively, and `#gtrans2ee` denotes the corresponding relation size equal to `#trans2` $*$ `#target_loc_trans0`^{2}. The parameter `#erase` denotes the number of facts in the `erase` relation; `#erase.4/123` denotes the number of different values the forth argument of `erase` can take for each combination of values of the first three arguments. In the worst case, this is the number of control locations $c2$ such that a transition of the form $((c1, s1), (c2, \epsilon))$ exists in the automaton. We use `#target_loc_trans0` to denote this number.

The time complexity for the set of rules is the total number of combinations of hypotheses considered in evaluating the rules. For each rule `r`, `r.#firedTimes` stands for the number the number of firings for the rule is a count of: (i) for rules with one hypothesis: the number of facts which make the hypothesis true; (ii) for rules with two hypotheses: the number of combinations of facts which make the two hypotheses simultaneously true. The total time complexity is time for reading the input, i.e. `O(#trans + #loc)`, plus the time for applying each rule, shown in the second column in the table of Figure 5.

rule no	time complexity	time complexity bound
1	`min(#trans0*1,#loc*#trans0.23/1)`	`#trans0`
2	`min(#loc*#trans1.234/1,#trans1*1)`	`#trans1`
3	`min(#gtrans1*#erase.4/123, #erase*#gtrans1.12/34)`	`#trans1*#target_loc_trans0`
4	`min(#gtrans1e*1, 1*#gtrans1e)`	`#trans1*#target_loc_trans0`
5	`min(#loc*#trans2.2345/1,#trans2*1)`	`#trans2`
6	`min(#gtrans2*#erase.4/123, #erase*#gtrans2.12/345)`	`#trans2*#target_loc_trans0`
7	`min(#gtrans2e*#erase.4/123, #erase*#gtrans2e.12/345)`	$\texttt{\#trans2*\#target\_loc\_trans0}^2$
8	`min(#gtrans2ee*1,1*#gtrans2ee)`	$\texttt{\#trans2*\#target\_loc\_trans0}^2$
9	`min(#gtrans2ee_or*1,1*#gtrans2ee_or)`	$\texttt{\#trans2*\#target\_loc\_trans0}^2$
10	`min(#gtrans2ee_or*1,1*#gtrans2ee_or)`	$\texttt{\#trans2*\#target\_loc\_trans0}^2$
11	`#gtrans1`	`#trans1`
12	`#gtrans2`	`#trans2`
13	`min(#gtrans2e*1,1*#gtrans2e)`	`#trans2*#target_loc_trans0`

relation	time complexity
`erase`	$\texttt{O(\#trans0 + \#trans1*\#target\_loc\_trans0 + \#trans2*\#target\_loc\_trans0}^2\texttt{)}$
`edge`	`O(#trans1 + #trans2*#target_loc_trans0)`

Fig. 5: Time complexity of computing the reach graph.

Time complexity of model checking PDS. Time complexity for processing each of the rules and computing the `erase` and `edge` relations is shown in the second table of Figure 5. After the reach graph has been computed, good cycles in the reach graph can be detected in time linear in the size of the reach graph, i.e. `O(#edge)`. Thus, the asymptotic complexity of the model checking problem is dominated by the time complexity of computing the `erase` relation.

For a BPDS, product of $P = \{C_P, S_P, T_P\}$ where $|C_P| = 1$, and $B = \{C_B, L_B, T_B, C0_B, G_B\}$, $\texttt{\#target\_loc\_trans0} \leq |C_B|$, and $\texttt{\#trans2} \leq |T_P| * |T_B|$. For such a PDS, $O(|T_P| * |T_B| * |C_B|^2)$ is the worst case time complexity of computing the `erase` relation and $O(|T_P| * |T_B| * |C_B|)$ is the worst case time

complexity for computing the `edge` relation. Since only $|T_P|$ is dependent on the size of `P`, time complexity is linear in the size of the `P` and cubic in the size of `B`.

4.3 Performance

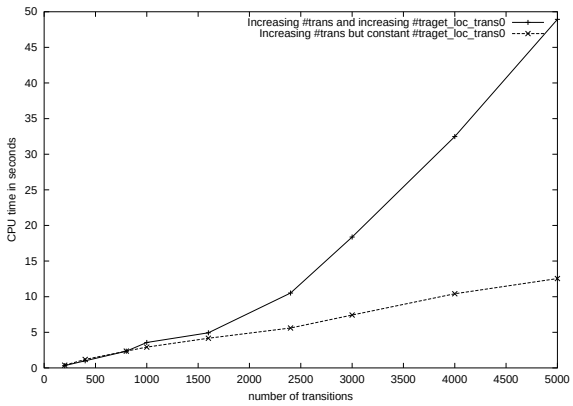


Fig. 6: Results for computing the reach graph for the BPDS.

We tested the performance of our reach graph construction algorithm on two sets of BPDS consisting of BPDS with increasing `#trans`. BPDS in one set also had increasing `#target_loc_trans0`, while BPDS in the second set had constant `#target_loc_trans0`. The time complexity for computing reach graphs for BPDS in the first set is as shown in Figure 5. However, for automata in the second set time complexity should be linear — `O(#trans)`. If the PDS corresponds to a program, `#target_loc_trans0` is proportional to the total number of return points of procedures in the program. Thus, our test data corresponds to checking if a property holds on programs with an increasing number of statements and procedure calls, and programs with an number of statements, but constant number of procedures.

Results of the experiment are shown in Figure 6 and confirm our analysis. We used generated python code in which each operation on set elements is guaranteed to be constant time on average using default hashing in python. Running times are measured in seconds on a 500MHz Sun Blade 100 with 256 Megabytes of RAM, running SunOS 5.8. Running times are the average over ten runs.

5 Discussion

The problem of LTL model checking of PDS has been extensively researched, especially model checking PDS induced by CFGs of programs. The model checking

problem for context-free and pushdown processes is explored in [8]. The design and implementation of Bebop: a symbolic model checker for boolean programs, is presented in [4]. Burkart and Steffen [9] present a model checking algorithm for modal mu-calculus formulas. For a PDS with one control state, a modal-mu calculus formula of alternation depth k can be checked in time $O(n^k)$, where n is the size of the PDS. The works [17,16,15,7] describe efficient algorithms for model checking PDSs. Alur et al. [3] and Benedikt et al. [6] show that state machines can be used to model control flow of sequential programs. Both works describe algorithms for model checking PDS that have time complexity cubic in size of the BA and linear in size of the PDS; these works combine forward and backward reachability and obtain complexity estimations by exploiting this mixture. Esparza et al. [15] estimate time complexity of solving the model checking problem to be $\texttt{O(n*m}^3\texttt{)}$ for model checking PDS with one state only, where `n` is the size of the PDS and `m` is the size of the property BA [15]. While this is also linear in the size of the PDS, our time complexity analysis is more precise and automatic.

The algorithm derived in this work is essentially the same as the one in [15]. What distinguishes our work is that we use a novel implementation strategy for the model checking problem that combines an intuitive definition of the model checking problem in rules [5] and a systematic method for deriving efficient algorithms and data structures from the rules [18], and arrives at an improved complexity analysis. The time complexity is calculated directly from the Datalog rules, based on a thorough understanding of the algorithms and data structures generated, reflecting the complexities of implementation back into the rules.

An implementation of the model checking problem in logical rules is presented in [5]. The rules are evaluated using the XSB system [23]. Thus, the efficiency of the computation is highly dependent on the order of hypotheses in the given rules. Our implementation is drastically different, as it finds the best order of hypotheses in the rules automatically. We do not employ an evaluation strategy for Datalog, but generate a specialized algorithm and implementation directly from the rules.

In this paper, we presented an efficient algorithm for LTL model checking of PDS. We showed the effectiveness of our approach by using a precise time complexity analysis, along with experiments. These results show that our model checking algorithm can help accommodate larger PDS and properties. Our work is potentially a contribution not only to the model checking problem, since the idea behind the `erase` relation and the reach graph is more universal than model checking PDS. Variants of the `erase` relation are used in data flow analysis techniques, as described in [22] and related work. Applications of model checking in dataflow analysis are presented in [25,24]. It is a topic of future research to apply our method to dataflow analysis problems.

Acknowledgment. Thanks to Tom Rothamel for helping debug performance problems in the implementation.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
3. R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 207–220, London, UK, 2001. Springer-Verlag.
4. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.
5. S. Basu, K. N. Kumar, L. R. Pokorny, and C. R. Ramakrishnan. Resource-constrained model checking of recursive programs. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 236–250, London, UK, 2002. Springer-Verlag.
6. M. Benedikt, P. Godefroid, and T. W. Reps. Model checking of unrestricted hierarchical state machines. In *ICALP '01: Proceedings of the 28th International Colloquium on Automata, Languages and Programming,*, pages 652–666, London, UK, 2001. Springer-Verlag.
7. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency Theory*, pages 135–150, 1997.
8. O. Burkart, D. Caucal, F. Moller, and B. Steffen. *Verification on infinite structures*. North Holland, 2000.
9. O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. In *ICALP '97: Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, pages 419–429, London, UK, 1997. Springer-Verlag.
10. J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11(3):197–261, 1989.
11. S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.
12. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.
13. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 117–126, New York, NY, USA, 1983. ACM Press.
14. J. Edmund M. Clark, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
15. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 232–247, London, UK, 2000. Springer-Verlag.
16. J. Esparza and S. Schwoon. A bdd-based model checker for recursive programs. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 324–336, London, UK, 2001. Springer-Verlag.

17. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *Proc. 2nd Int. Workshop on Verification of Infinite State Systems (INFINITY'97)*, volume 9 of *Electronic Notes in Theoretic Comp. Sci.* Elsevier, 1997.
18. Y. A. Liu and S. D. Stoller. From datalog rules to efficient programs with time and space guarantees. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 172–183. ACM Press, 2003.
19. R. Paige. Real-time simulation of a set machine on a ram, 1989.
20. R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, 1982.
21. J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.
22. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, 1995.
23. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data SIGMOD'94*, pages 442–453, 1994.
24. B. Steffen. Generating data flow analysis algorithms from modal specifications. In *TACS'91: Selected papers of the conference on Theoretical aspects of computer software*, pages 115–139, Amsterdam, The Netherlands, 1993. Elsevier Science Publishers B. V.
25. B. Steffen, A. Classen, M. Klein, J. Knoop, and T. Margaria. The fixpoint-analysis machine. In *CONCUR '95: Proceedings of the 6th International Conference on Concurrency Theory*, pages 72–87, London, UK, 1995. Springer-Verlag.