Analysing Mu-Calculus Properties of Pushdown Systems

M. Hague and C.-H. L. Ong

Oxford University Computing Laboratory
Matthew.Hague@comlab.ox.ac.uk Luke.Ong@comlab.ox.ac.uk

Abstract. Pushdown systems provide a natural model of software with recursive procedure calls. We provide a tool (PDSolver) implementing an algorithm for computing the winning regions of a pushdown parity game and its adaptation to the direct computation of modal μ-calculus properties over pushdown systems. We also extend the algorithm to allow backwards, as well as forwards, modalities and allow the user to restrict the control flow graph to configurations reachable from a designated initial state. These extensions are motivated by applications in dataflow analysis. We provide two sets of experimental data. First, we obtain a picture of the general behaviour by analysing random problem instances. Secondly, we use the tool to perform dataflow analysis on real-world Java programs, taken from the DaCapo benchmark suite.

1 Introduction

Pushdown systems — finite-state transition systems equipped with a stack — have received a lot of interest in the software verification community. They accurately model the control flow of recursive programs (such as C and Java), and are ideal for algorithmic analysis. Pushdown systems have played a key role in the automata-theoretic approach to software model checking [1, 10, 14]. Considerable progress has been made in the implementation of scalable model checkers of pushdown systems. These tools (e.g. Bebop [11] and Moped [10]) are an essential back-end component of such model checkers as SLAM [12].

The modal μ-calculus is a highly expressive specification language (subsuming all standard temporal logics). Walukiewicz showed that modal μ-calculus model checking of pushdown systems is EXPTIME-complete [5], although no tools have been implemented. Previously, we gave the first algorithm that does not always suffer from an exponential explosion [4]. We introduce a tool (PDSolver) providing the first implementation of this algorithm. We also extend the technique to allow backwards modalities — which are needed for certain dataflow properties — and allow the analysis to be restricted to reachable configurations.

We provide two sets of experimental data in support of the tool. First, we give a picture of the behaviour of the tool on arbitrary inputs by analysing randomly generated problem instances. Secondly, we consider a specific application to dataflow analysis of pushdown systems extracted from real-world examples from the DaCapo Benchmarks [9]. This second set of experiments is an application of Steffen's work on dataflow analysis and model checking [3]. The tool can be downloaded from http://web.comlab.ox.ac.uk/people/Matthew.Hague/pdsolver.html.

Related Work There are several pushdown reachability checkers available, e.g., Bebop [11] and Moped [10]. Reps *et al.* have developed dataflow analysis tools based on *weighted* pushdown systems [14]. To the best of our knowledge, ours is the first tool for evaluating modal μ-calculus with backwards modalities. Steffen *et al.* have implemented dataflow analysis as model checking in jABC [2], although no work has been published extending jABC to pushdown systems.

2 Preliminaries

A ***pushdown system*** is a triple $\mathbb{P} = (\mathcal{P}, \mathcal{D}, \Sigma_\bot)$ where $\mathcal{P}$ is a set of control states, $\Sigma_\bot := \Sigma \cup \{\bot\}$ is a finite stack alphabet (we assume $\bot \notin \Sigma$) and $\mathcal{D} \subseteq \mathcal{P} \times \Sigma_\bot \times \mathcal{P} \times \Sigma_\bot^*$ is a set of rules. As is standard, we assume the bottom-of-stack symbol $\bot$ is neither pushed onto, nor popped from, the stack. A *configuration* is a pair $\langle p, w\rangle$ with $p \in \mathcal{P}$ and $w \in \Sigma^* \bot$. We have $\langle p, aw\rangle \to \langle p', w'w\rangle$ whenever $(p, a, p', w') \in \mathcal{D}$. Let $\mathcal{C}$ be the set of all pushdown configurations.

For a set AP of atomic propositions and a disjoint set $\mathcal{Z}$ of variables, formulas of the ***modal μ-calculus*** are (with $x \in AP$, $\Lambda \subseteq AP$ and $Z \in \mathcal{Z}$):

$$\varphi \;:=\; x \mid \neg x \mid Z \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid [\Lambda]\varphi \mid \langle\Lambda\rangle\varphi \mid \overline{[\Lambda]}\varphi \mid \overline{\langle\Lambda\rangle}\varphi \mid \mu Z.\varphi \mid \nu Z.\varphi \;.$$

Thus we assume formulas are in *positive form*. The semantics of a formula φ are given with respect to a *valuation* of free variables $V : \mathcal{Z} \to \mathcal{P}(\mathcal{C})$ and atomic propositions $\rho : AP \to \mathcal{P}(\mathcal{C})$. The *denotation* $[\![\varphi]\!]^{\mathbb{P}}_{V,\rho}$ of a formula is the set of all satisfying configurations. A configuration c satisfies $\langle\Lambda\rangle\varphi$ iff $\exists x \in \Lambda.c \in \rho(x) \wedge \exists c'.c \to c' \wedge c' \in [\![\varphi]\!]^{\mathbb{P}}_{V,\rho}$ and c satisfies $[\Lambda]\varphi$ iff $(\exists x \in \Lambda.c \in \rho(x)) \Rightarrow \big(\forall c'.c \to c' \Rightarrow c' \in [\![\varphi]\!]^{\mathbb{P}}_{V,\rho}\big)$. The operators $\overline{[\Lambda]}$ and $\overline{\langle\Lambda\rangle}$ are their backwards time counterparts. The μ and ν operators specify greatest and least fixed points (for details we refer the reader to Bradfield and Stirling [6]). We may also interpret propositions as actions: a configuration satisfies a proposition if leaving the configuration executes the action. Hence $[\Lambda]\varphi$ holds if φ holds after all Λ actions.

3 Algorithm and Implementation

Algorithm We present the full algorithm separately [7] and provide a summary here. Sets of configurations are represented using a kind of automata over words [1]. Broadly speaking, when representing $[\![\varphi]\!]^{\mathbb{P}}_{V,\rho}$, we have $\langle p, w\rangle \in [\![\varphi]\!]^{\mathbb{P}}_{V,\rho}$ if the word w is accepted from the state (p, φ) of the automaton.

The algorithm recurses over subformulas. For propositions x or variables Z the automaton representing its valuation is either given or already computed. For $\varphi = \varphi_1 \wedge \varphi_2$, we introduce the state (p, φ) for each control state p and add transitions combining the transitions from (p, φ_1) and (p, φ_2). Similarly for $\vee$.

We use, for $[\Lambda]$ and $\langle\Lambda\rangle$, standard backwards reachability techniques [1]. E.g., for $\langle\Lambda\rangle\varphi$, we have all configurations a step in front of $[\![\varphi]\!]^{\mathbb{P}}_{V,\rho}$. The extension to backwards modalities is similar, but uses an adaptation of forwards reachability [10]. To restrict the analysis to the reachable configurations, we first use the

efficient algorithm of Schwoon [10] to obtain the set of reachable configurations, then restrict our results by taking the conjunction with this set. The $\overline{[\Lambda]}$ modality is an exception: we compute $\overline{[\Lambda]}(\varphi \vee \neg \mathit{reachable})$. This is not needed for $[\Lambda]$ since any successor of a reachable state is itself reachable.

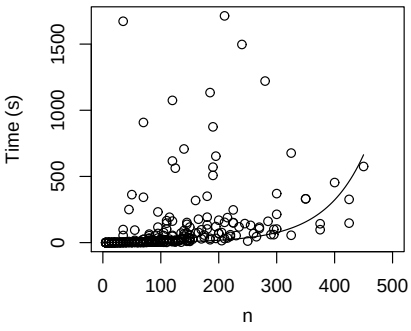
For fixed points we use a trick called *projection*, introduced by Cachat [13]. This ingenious technique allows us to compute a fixed point of φ by repeatedly computing φ. We refer to our previous work for details [4].

The algorithm is exponential in the number of control states and the sizes of the formula and the (automaton) representations of V and ρ. With backwards modalities, it is also exponential in the size of the alphabet.

Implementation We provide an explicit state implementation using OCaml and apply two main optimisations. The first is to identify subformulas whose denotation will not change when iterating the computation of a fixed point, and then store the computed value to speed up subsequent iterations. Secondly, there are cases where an automaton state should behave similarly for all characters except a few. We introduce *default* transitions that can be taken on reading a character a whenever there are no other a-transitions. This is important for backwards modalities as it greatly reduces the cost of an n^2 loop. These transitions need to be dealt with carefully by the other procedures.

4 Experimental Results[1]

Random Instances We generated 395 model checking instances. Each PDS of size n, ranging from 5 to 450, has n states and n characters (giving between 25 and 250k pairs $\langle p, a \rangle$) and between n^2 and $2n^2$ transitions ((p, a, p', w) where $|w|$ is uniformly 0, 1 or 2). Each formula has a maximum connective depth of 5, a minimum fixed point depth of 2, lengths between 7 and 22 and up to 5 literals. Each proposition has a 10% probability of holding at a given pair of control state and top of stack character. Furthermore, each bound variable occurs within at least one $[\Lambda]$ or $\langle \Lambda \rangle$ operator. Note we only use forwards modalities and $\Lambda = AP$. The figure above shows the terminating results, plotted using the tool *R*. In 120 instances the algorithm timed out after 30 minutes, with a failure rate of 23% at $0 \leq n \leq 50$ rising to 50% at $400 \leq n \leq 450$. Hence, difficult instances can occur even on small pushdown systems. Never-the-less, in the next section, we show that useful properties can be checked on large systems.



[1] All tests were run on a 2.4Ghz, quad core Intel Xeon with 12Gb of RAM.

Dataflow Analysis Steffen advocates using modal μ-calculus model checkers for dataflow analysis [3]. This approach provides real-life test cases, taken from the DaCapo Benchmarks [9] (Version 9.12). We consider the *optimal placement of computations* problem. Intuitively, all modifications of a variable should happen as early as possible, provided they are necessary. For example, take the program

```
main () { i = [value]; a(); b(); }
a() { [computation not including i] }
b() { print(i); }
```

where `a()` may cause the program to fail. The only optimal placement of the computation of `i` is just before `b()`. An earlier computation is unnecessary if `a()` fails. We tested this example with our tool, obtaining the expected result.

Recalling Steffen, define the "always globally" operators $AG_\Lambda\varphi = \nu X.(\varphi \wedge [\Lambda]X)$ and $\overline{AG}_\Lambda\varphi = \nu X.(\varphi \wedge \overline{[\Lambda]}X)$. Fix a variable i and let M be the set of actions modifying i and U the set using i. For a set S, S^c denotes the complement. The proposition *end* marks the end of the computation. A necessary computation is one which will always be used and an optimal computation point is one at which earlier computations are unnecessary. This property is bi-directional. Let

$$\varphi_{nec} = AG_{U^c}(\neg end \wedge [M \cap U^c]false) \text{ and } \varphi_{ocp} = \varphi_{nec} \wedge \overline{[M^c]}(\overline{AG}_{M^c}\neg\varphi_{nec}) \ .$$

We chose parts of the Avrora (A) and the FOP (F) benchmarks and extracted pushdown control flow graphs with a supplementary tool based on Soot [8]. Polymorphic method calls may call any implementing procedure. We assume that all calls may result in an exception (e.g. RuntimeExceptions), hence each call is followed by an exception handling branch. Finally, data values are ignored.

For each benchmark, we chose a non-local variable with several use/define statements. Table 1 shows the results. The final columns give the size of the denotation representation. Each example had three control states. The number of control points is $|\Sigma|$. Since φ_{ocp} contains backwards modalities, the problem is exponential in $|\Sigma|$. Because $\overline{[\Lambda]}$ is computationally intensive, we evaluated $\neg\varphi_{ocp}$ rather than φ_{ocp}. In all tests, we only consider reachable configurations. Since we only consider one variable per test, program slicing could considerably increase performance. We do not perform this optimisation; instead we take the opportunity to test our tool on large systems, and obtain encouraging results.

Example	Control Points	Pushdown Rules	Time (s)	States	Transitions
RegisterTestAction (A)	3k	4k	14	509	19k
ELFDumpAction (A)	6k	7k	40	724	32k
ExampleFO2PDF (F)	17k	24k	95	1724	90k
ExampleDOM2PDF (F)	18k	25k	132	1753	95k
DisassembleAction (A)	54k	75k	1525	6215	296k
CFGAction (A)	90k	120k	3946	9429	500k

Table 1. Optimal computation point analysis of several Java examples.

5 Conclusion and Future Work

We introduced the first tool for evaluating modal μ-calculus formulas over pushdown systems. We support forwards and backwards modalities and a restriction to reachable configurations. We tested random and real-life examples, demonstrating the tool's potential as a component of a pushdown analysis framework.

For forwards reachability, Schwoon's optimisations lead to significant performance gains. We may attempt to extend these techniques to operators like AG. Applying the optimisations to the full algorithm, however, may prove difficult. We may also use BDDs to represent the transition relation of the multi-automata.

Another avenue is to develop the dataflow analysis applications of our tool, by providing an improved translation from Java and exploiting optimisations such as program slicing or counter-example guided abstraction refinement.

Since ours is the first tool of its kind, we have no comparative data. For certain specific applications, we may perform a comparison with suitable tools; however, we are unaware of any such tools for the examples considered here.

Acknowledgments. We thank Vijay D'Silva for recommending Steffen [3], Oege De Moor for suggesting DaCapo, and Georg Weissenbacher for his comments.

References

1. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR 1997*, pages 135–150.
2. A. Lamprecht, T. Margaria, and B. Steffen. Data-flow analysis as model checking within the jabc. In *CC 2006*, pages 101–104.
3. B. Steffen. Data flow analysis as model checking. In *TACS 1991*, pages 346–365.
4. M. Hague and C.-H. L. Ong. Winning regions of pushdown parity games: A saturation method. In *CONCUR 2009*, pages 384–398.
5. I. Walukiewicz. Pushdown processes: Games and model checking. In *CAV 1996*, pages 62–74.
6. J. C. Bradfield and C. P. Stirling. Modal logics and mu-calculi: An introduction. In *Handbook of Process Algebra*, pages 293–330, 2001.
7. M. Hague and C.-H. L. Ong. A saturation method for the modal mu-calculus with backwards modalities over pushdown systems. arXiv:1006.5906v1 [cs.FL], 2010
8. R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON 1999*, pages 125–135.
9. S. M. Blackburn *et al.*. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA 2006*, pages 169–190.
10. S. Schwoon. *Model-checking Pushdown Systems*. PhD thesis, Technical University of Munich, 2002.
11. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 2000*, pages 113–130.
12. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 2002*, pages 1–3.
13. T. Cachat. Symbolic Strategy Synthesis for Games on Pushdown Graphs. In *ICALP 2002*, pages 704–715.
14. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.