PuMoC: A CTL Model-Checker for Sequential Programs

Fu Song
LIAFA, CNRS and Univ. Paris Diderot, France
song@liafa.univ-paris-diderot.fr

Tayssir Touili
LIAFA, CNRS and Univ. Paris Diderot, France
touili@liafa.univ-paris-diderot.fr

ABSTRACT

In this paper, we present PuMoC, a CTL model checker for Pushdown systems (PDSs) and sequential C/C++ and Java programs. PuMoC allows to do CTL model-checking w.r.t simple valuations, where the atomic propositions depend on the control locations of the PDSs, and w.r.t. regular valuations, where atomic propositions are regular predicates over the stack content. Our tool allowed to (1) check 500 randomly generated PDSs against several CTL formulas; (2) check around 1461 versions of 30 Windows drivers taken from SLAM benchmarks; (3) check several C and Java programs; and (4) perform data flow analysis of real-world Java programs. Our results show the efficiency and the applicability of our tool.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Verification

Keywords

Model-Checking, Pushdown Systems, Branching-time Temporal Logic, Software Model-Checking

1. INTRODUCTION

In this paper, we present PuMoC, a CTL model checker for Pushdown systems (PDSs) and sequential programs. PuMoC allows to check CTL properties for PDSs w.r.t. *simple valuations*, where the atomic propositions depend on the control locations of the PDSs, and w.r.t. *regular valuations*, where atomic propositions are regular predicates over the stack content. Indeed, since a configuration of a PDS has a control state and a stack content, it is natural that the validity of an atomic proposition in a configuration depends on both the control state *and the stack*. PuMoC is based on the model checking algorithms of [8], where CTL model checking for PDSs is reduced to emptiness checking for Alternating Büchi Pushdown Systems (ABPDSs).

PuMoC allows also to perform CTL model checking for sequential boolean, C/C++ and Java programs. Indeed, sequential boolean programs can naturally be modeled by PDSs. To translate C/C++ sequential programs into boolean programs, we use Satabs [3], whereas to extract a PDS from a Java program, we use JimpleToPDSolver [6]. The PDSs generated by JimpleToPDSolver contain def/use informations. We use these informations to perform data flow analysis for Java sequential programs (such as reaching definitions, live variables, very busy expressions and available expressions). We express such properties using CTL formulas. To our knowledge, PuMoC is the first tool to do CTL model checking for PDSs. The only other tool that can check branching time properties for PDSs is PDSolver [6]. Our experimental results show that our tool is much more efficient. Furthermore, PuMoC is integrated with Moped [5]. This offers a toolkit that can do both LTL and CTL model-checking for PDSs and sequential programs.

Our tool allowed to (1) check 500 randomly generated PDSs against several CTL formulas; (2) check around 1461 versions of 30 Windows drivers taken from SLAM benchmarks, (3) check several C and Java programs; and (4) perform data flow analysis of real-world Java programs. Our results show the efficiency and the applicability of our tool. PuMoC is downloadable at `http://www.liafa.jussieu.fr/~song/PuMoC`. Instructions on how to use our tool can be found in this webpage.

Related Work. *Bebop* and SLAM [1] are model-checkers for sequential boolean programs. They deal only with reachability properties. *Moped* [5] can check reachability and LTL formulas for PDSs and boolean programs. These tools cannot deal with CTL model-checking. *PDSolver* [6] is a μ-calculus model checker for pushdown systems. Given a CTL formula, we can translate it into a μ-calculus formula and then run *PDSolver*. However, this is not efficient. Indeed, our experiments show that our tool PuMoC outperforms *PDSolver* for CTL formulas. [4] can only check the universal fragment of CTL for non-recursive programs, whereas our tool can deal with all CTL and with recursive programs.

2. BACKGROUND

2.1 Pushdown Systems

A *pushdown system* (PDS) $\mathcal{P}$ is a tuple (P, Γ, Δ), where P is a finite set of control locations, Γ is the stack alphabet, $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of transition rules. A configuration of $\mathcal{P}$ is a pair $\langle p, \omega \rangle$ where $p \in P$ and $\omega \in \Gamma^*$. The successor relation $\rightsquigarrow_{\mathcal{P}} \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ is defined as follows: if $(p, \gamma, q, \omega) \in \Delta$, then $\langle p, \gamma\omega' \rangle \rightsquigarrow_{\mathcal{P}} \langle q, \omega\omega' \rangle$ for every $\omega' \in \Gamma^*$.

ASE '12, September 3-7, 2012, Essen, Germany

An *Alternating Büchi Pushdown System* (ABPDS) $\mathcal{BP} = (P, \Gamma, \Delta, F)$ is a kind of PDS with *alternating* transition rules (i.e., $\Delta \subseteq (P \times \Gamma) \times 2^{P \times \Gamma^*}$) and a finite set of Büchi accepting control locations F. A run of $\mathcal{BP}$ is a tree rooted by the initial configuration such that for each node $\langle p, \gamma\omega' \rangle$ whose children are $\langle p_1, \omega_1\omega' \rangle, ..., \langle p_n, \omega_n\omega' \rangle$, then, necessarily, $((p\gamma), \{(p_1\omega_1), ..., (p_n\omega_n)\}) \in \Delta$. A run is accepting iff all its paths infinitely often visit some accepting locations in F.

Multi-Automata [2] are used to represent (infinite) sets of configurations of PDSs. Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, a Multi-Automaton $\mathcal{A}$ is a finite automaton having P as set of initial states and Γ as alphabet. Then a configuration $\langle p, \omega \rangle$ is accepted (or recognized) by $\mathcal{A}$ iff $\mathcal{A}$ has an accepting run that starts at the initial state p, ends at a final state, and is labeled by ω. A set of configurations is regular if it can be recognized by a multi-automaton. Regular expressions over Γ^* can also be used to represent regular configurations. E.g., $p \cdot (\gamma_1 + \gamma_2)^* \gamma_3^*$ represents the regular set of configurations that are in the control location p, and whose stack contains an arbitrary number of γ_1's and γ_2's, followed by an arbitrary number of γ_3's.

2.2 The Temporal Logic CTL

We consider the standard branching-time temporal logic CTL. For technical reasons, we suppose w.l.o.g. that formulas are given in positive normal form, i.e., negations are applied only to atomic propositions. Indeed, each CTL formula can be written in positive normal form by pushing the negations inside. Moreover, we use the operator R as a dual of the until operator for which the stop condition is not required to occur. Then, standard CTL operators can be expressed as follows: $EF\psi = E[trueU\psi]$, $AF\psi = A[trueU\psi]$, $EG\psi = E[falseR\psi]$ and $AG\psi = A[falseR\psi]$.

More precisely, let $AP = \{a, b, c, ...\}$ be a finite set of atomic propositions. The set of CTL formulas is given by (where $a \in AP$):

$$\varphi ::= a \mid \neg a \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid AX\varphi \mid EX\varphi \mid A[\varphi U \varphi] \mid E[\varphi U \varphi] \mid A[\varphi R \varphi] \mid E[\varphi R \varphi].$$

The semantics of CTL formulas are given w.r.t. simple valuations or regular valuations. A *simple valuation* is a function $\lambda_s : AP \longrightarrow 2^P$. A configuration $\langle p, \omega \rangle$ satisfies an atomic proposition $a \in AP$ w.r.t. this simple valuation λ_s iff $p \in \lambda_s(a)$. A *regular valuation* is a function $\lambda_r : AP \longrightarrow 2^{P \times \Gamma^*}$ s.t. for every $a \in AP$, $\lambda_r(a)$ is a regular set of configurations represented by a regular expression corresponding to a multi automaton. These regular expressions are called *regular predicates*. In this case, a configuration $\langle p, \omega \rangle$ satisfies an atomic proposition a w.r.t. this regular valuation λ_r iff the multi-automaton corresponding to $\lambda_r(a)$ recognizes $\langle p, \omega \rangle$. Note that CTL with simple valuations corresponds to standard CTL, as considered in the literature for PDSs.

2.3 The Model-Checking Algorithms

PuMoC is based on the model-checking algorithms of [8], where model checking CTL formulas for PDSs w.r.t. both simple and regular valuations can be reduced to the emptiness checking problem for ABPDSs. Given a CTL formula φ and a PDS $\mathcal{P}$, to check whether $\mathcal{P}$ satisfies φ, we first compute an ABPDS $\mathcal{BP}$ that can be seen as the product of the CTL formula φ and the PDS $\mathcal{P}$. The computation of $\mathcal{BP}$ depends on whether we consider simple or regular valuations. Then the formula is satisfied by $\mathcal{P}$ iff $\mathcal{BP}$ has an accepted run. PuMoC computes a multi-automaton $\mathcal{A}$ recognizing the set of configurations from which $\mathcal{BP}$ has an accepted run. $\mathcal{A}$ represents the whole set of configurations from which $\mathcal{P}$ satisfies φ. We refer to [8] for more details on the different algorithms.

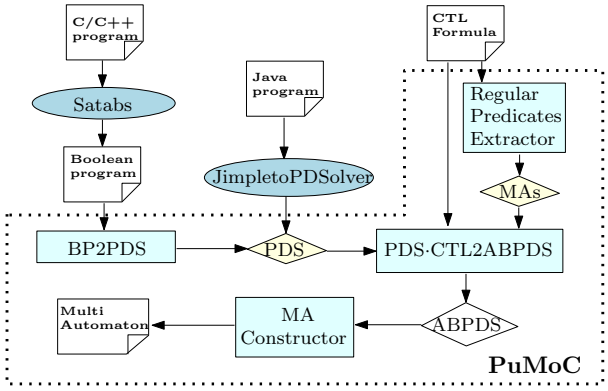

Figure 1: The architecture of PuMoC

3. ARCHITECTURE AND IMPLEMENTATION

PuMoC is implemented in C. As shown in Figure 1, it consists of four components: **BP2PDS**, **Regular Predicates Extractor**, **MA Constructor** and **PDS·CTL2ABPDS**.

BP2PDS takes as input a boolean program written in the syntax of Satabs and translates it into a PDS. This translation is made in two steps. First, a set of control flow graphs (CFGs) is extracted from the program, then these CFGs are translated into a PDS. **BP2PDS** extends the translation done by MOPED [5] in order to deal with more statements of the boolean programs. **Regular Predicates Extractor** extracts the regular expressions (regular valuations) present in the CTL formulas and computes a multi automaton corresponding to each regular expression. This component is used when regular valuations are considered. Given a CTL formula φ, a set of multi-automata corresponding to the regular predicates of the CTL formula and a PDS $\mathcal{P}$ written in the syntax of Moped or PDSolver, **PDS·CTL2ABPDS** constructs an ABPDS $\mathcal{BP}$ as described in [8]. $\mathcal{BP}$ is such that $\mathcal{P}$ satisfies φ iff $\mathcal{BP}$ has an accepting run. **MA Constructor** takes as input an ABPDS $\mathcal{BP}$ and applies the algorithm of [8] to compute a multi-automaton recognizing the set of configurations from which $\mathcal{BP}$ has an accepting run, i.e., the set of configurations from which the PDS satisfies the CTL formula. Furthermore, PuMoC is integrated with Moped [5]. This offers a toolkit that can do both LTL and CTL model-checking for PDSs and sequential programs.

4. EXPERIMENTS

All the tests were run on a Fedora 13 with a 2.4GHz CPU and 2GB of memory.

4.1 Verifying Random Pushdown Systems

We randomly generated 500 PDSs each of them equipped with a random CTL formula. The number n of control locations and stack alphabet ranges from 10 to 510. The number of transition rules ranges from n^2 to $2n^2$. The size of the CTL formulas ranges from 2 to 15. Figure 2 depicts the time consumption w.r.t. the size n. Only 5.2% (i.e. 26) of the cases run out of time (30 minutes) while the majority of cases terminated in a few seconds. Figure 3 depicts the memory consumption w.r.t. the size n. Only 6.6% (i.e. 33) of tests run out of memory with 2GB limitation. The majority of tests finished in a few MB.

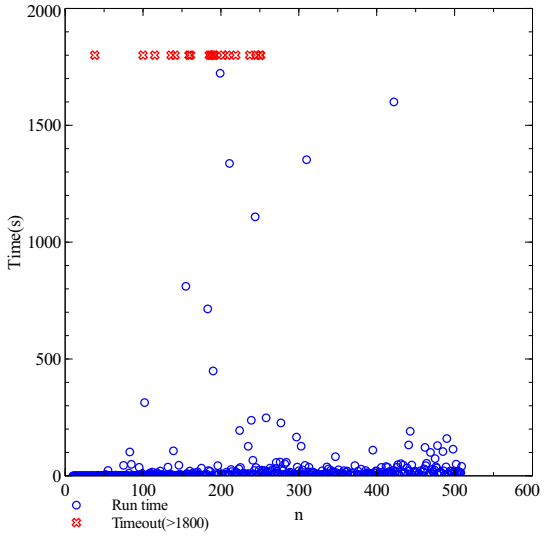

Figure 2: Time consumption

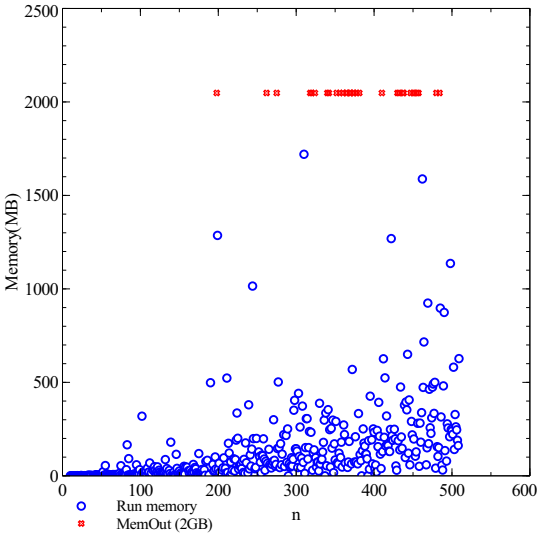

Figure 3: Memory consumption

4.2 Verifying Windows Drivers

We tested 1461 versions of 30 real-world drivers. All these versions are taken from SLAM benchmarks [1]. SLAM provides the boolean programs corresponding to these drivers. We checked 2 CTL formulas that are relevant for these drivers. The first formula r_1 ensures that the programmer uses the API functions in the good order. This formula uses regular valuations to ensure that whenever a function named *DeviceAdd* is called, the function *DeviceCreate* will be called eventually before *DeviceAdd* returns. The second CTL formula r_2 describes a locking property. Our results are described in Table 1. Column **No. Versions** gives the number of versions we considered of the corresponding driver. Columns **Avg. #LOC** denotes the average size of the programs' versions. Column **Avg. Time(s)** and **Avg. Mem(MB)** give the average time and memory in seconds and MB.

4.3 Model-Checking C and Java Programs

We also were able to check several CTL properties of several C and Java programs. We checked the C source code of two bounded model checkers (verbs and verds) [10]. We also checked 4 real-world Java programs taken from JimpleToPDSolver [6], 4 real-world Java programs taken from a Java benchmark SciMark2 [7], and 7 real-world Java programs taken from JBDD [9]. The experimental results are summarized in Table 2.

4.4 Data Flow Analysis of Java Programs

PuMoC can also perform Data Flow Analysis of Java programs. Given a Java program, JimpletoPDSolver translates it into a PDS with def/use informations, where the atomic proposition *def(x)* (resp. *use(x)*) of a variable x holds at a control point if its corresponding statement is an assignment (resp. a use) of the variable x[1]. Using these informations, we can write CTL formulas that can solve some data flow analysis problems like reaching definitions, live variables, very busy expressions, and available expressions. For example the formula $\psi_1 = E[\neg def(x) U\ use(x)]$ checks whether the variable x is used without being defined. This allows to do live variables analysis. Similarly, the formula $\psi_2 = AG(def(x) \implies EF\ use(x))$ states that whenever the variable x is defined, it should be used in some path. Checking such property allows to optimize programs. Indeed, the variable x does not need to be defined (assigned) if x will not be used. Table 3 summarizes the results of our experiments. We made a comparison with PDSolver [6], a μ-calculus model-checker for PDSs and Java programs (CTL formulas can be translated to μ-calculus). Our tool is more efficient.

[1]For example, variable x (resp. y and z) is assigned (resp. used) at statement $x := y + z$.

5. ACKNOWLEDGEMENT

This work is partially funded by ANR grant ANR-08-SEGI-006.

6. REFERENCES

[1] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.

[2] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model Checking. In *CONCUR'97*. LNCS 1243, 1997.

[3] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS*, pages 570–574, 2005.

[4] B. Cook, E. Koskinen, and M. Y. Vardi. Temporal property verification as a program analysis task. In *CAV*, pages 333–348, 2011.

[5] J. Esparza and S. Schwoon. A BDD-Based Model Checker for Recursive Programs. In *CAV*, pages 324–336, 2001.

[6] M. Hague and C.-H. L. Ong. Analysing Mu-Calculus Properties of Pushdown Systems. In *SPIN*, pages 187–192, 2010.

[7] R. Pozo and B. Miller. Java mark 2.0, 2012. `http://math.nist.gov/scimark2`.

[8] F. Song and T. Touili. Efficient CTL Model-Checking for Pushdown Systems. In *CONCUR*, pages 434–449, 2011.

[9] A. Vahidi. Jbdd, 2012. `http://javaddlib.sourceforge.net`.

[10] W. Zhang. Bounded model-checker verbs and verds, 2012. http://lcs.ios.ac.cn/ zwh/verds/index.html.

Table 1: Model-Checking Drivers with PuMoC

Program	No. Versions	Avg. #LOC	r_1		r_2	
			Avg. Time(s)	Avg. Mem(MB)	Avg. Time(s)	Avg. Mem(MB)
1394	10	7.9k	48.80	30.45	27.82	9.80
bluetooth	37	10.32k	67.83	34.38	28.11	10.43
SD	14	6.9k	27.42	19.57	7.10	5.93
PLX9x5x	16	13.2k	119.14	45.72	40.53	14.01
amcc5933	14	10.0k	54.50	32.26	16.84	9.40
cancel	69	3.4k	20.95	12.43	4.01	4.34
Echo	68	5.4k	28.34	19.10	7.46	5.66
event	20	4.8k	32.04	18.24	7.22	5.35
pcidrv	72	29.1k	422.58	115.56	181.52	36.58
perfcounters	16	2.2k	11.84	8.58	2.57	2.82
portio	26	4.9k	23.17	15.17	6.00	4.96
registry	35	11.4k	150.85	46.36	56.83	14.91
toaster_wdm_bus	43	9.6k	91.19	37.93	31.91	12.15
toaster_wdm_func	183	10.1k	90.05	40.44	32.97	12.99
toaster_wdm_toastmon	41	4.5k	32.15	17.58	8.24	5.70
toaster_wdm_filter	231	4.0k	26.36	15.58	6.65	5.11
toaster_kmdf	165	5.0k	19.45	15.13	5.19	4.89
tracing	62	2.8k	15.90	10.54	3.64	3.54
firefly	17	4.2k	12.96	10.21	3.14	3.45
hidmapper	29	2.5k	13.65	9.33	2.71	3.14
hidusbfx2	17	4.4k	24.83	16.62	4.48	3.95
HBtnKey	34	5.5k	48.06	21.76	13.29	6.98
hiddigi	65	9.2k	84.00	35.46	28.90	11.25
kbfiltr	15	6.5k	25.28	19.30	6.86	6.20
moufiltr	14	5.0k	13.04	11.79	3.15	3.88
vserial	9	4.2k	17.33	14.77	4.65	4.81
smscir	10	14.8k	293.78	57.25	117.33	18.50
network	59	43.8k	1283.84	171.42	594.93	52.95
serial	46	16.1k	174.61	63.55	69.19	20.56
storage	84	57.3k	923.24	224.42	401.03	69.38

Table 2: Model-Checking C and Java programs with PuMoC

	Program	#LOC	Time(s)	Mem(MB)
Java Prog.	Namer	60	0.03	0.62
	cmdline	3k	5.72	32.14
	readCmdLine	78k	525.32	149.62
	usage	95k	3009.50	581.28
	FFT	1k	11.33	8.74
	Bench	3k	13.69	21.76
	HTTPPost	26k	17.95	100.28
	Applet	159k	4148.21	535.35
	Equivalence	335	0.04	1.82
	Queens	665	2.87	5.03
	Queens2	885	2.33	9.01
	Knight	1k	0.38	9.40
	DimacsSlover	1k	0.33	7.87
	interface	1k	23.86	26.17
	IQueens	109k	2440.32	411.67
	jlink	37k	2092.60	281.77
	JUnitTestRunner	26k	32.24	141.46
	DefanltDepDes	28k	1390.64	198.45
	IBiblioHelper	89k	4648.64	348.58

	Program	#LOC	Time(s)	Mem(MB)
Java Prog.	EntryPoint	11k	20.14	66.39
	ProcessDestroyer	11k	128.22	87.06
	TestProcess	1k	0.19	5.89
	IvyAuthenticator	3k	40.81	27.91
	JunitTestRunnerTest	28k	1427.90	179.40
	Diagnostics	59k	1353.17	238.09
	DirectoryScanner	17k	626.41	113.68
	IntrospectionHelper	5k	3.92	36.03
	Launcher	18k	1341.98	185.55
	KeySubst	3k	7.40	15.60
	IPlanetEjbc	22k	703.68	175.50
C Prog.	progreconstruct	4k	0.01	0.07
	cs2bool	4k	0.01	0.07
	specs	4k	0.01	0.08
	cnfsat	4k	0.01	0.07
	qmdwritecnf	8k	0.01	0.11
	treedopost	8k	128.22	87.06
	CNFspec2model	8k	0.19	5.89
	qmd2model	8k	40.81	27.91

Table 3: Data flow analysis of Java Programs with PuMoC

Program	#LOC	ψ_1			ψ_2		
		PDSolver	Our tool: PuMoC		PDSolver	Our tool: PuMoC	
		Time(s)	Time(s)	Mem(MB)	Time(s)	Time(s)	Mem(MB)
RegAction	3k	5.14	0.71	5.23	19.89	15.95	11.88
ELFDump	6k	7.63	1.43	9.19	50.62	37.27	20.64
FOP2PDF	17k	108.4	10.33	26.73	311.66	262.51	81.76
DOM2PDF	18k	54.53	11.92	29.21	167.15	254.68	88.52
DisAction	54k	458.77	134.18	87.09	>2000	1616.09	386.82
CFGAction	90k	1129.99	544.45	143.39	>2000	1734.81	514.56