Approximate Probabilistic Model Checking for Programs

Jérôme Darbon
LRDE/EPITA

Richard Lassaigne
Equipe de Logique
Université Paris VII

Sylvain Peyronnet
LRDE/EPITA

Abstract

In this paper we deal with the problem of applying model checking to real programs. We verify a program without constructing the whole transition system using a technique based on Monte-Carlo sampling, also called "approximate model checking". This technique combines model checking and randomized approximation. Thus, it avoids the so-called state space explosion phenomenon. We propose a prototype implementation that works directly on C source code. It means that, contrary to others approaches, we do not need to use a specific language nor specific data structures in order to describe the system we wish to verify. Finally, we present experimental results that show the effectiveness of the approach applied to finding bugs in real programs.

1 Introduction

Model checking [4] is an algorithmic method whose goal is to verify the correctness of systems generally represented as transition systems. The method proceeds by constructing a large data structure (e.g. a transition graph) that represents the behavior of the system. Even for simple programs and protocols, data structures are so large that the verification becomes intractable. This is called the *state space explosion phenomenon*. In the last decades, most of the research in the field has focused on fighting against this phenomenon. Indeed, symbolic methods [18, 7] have been designed to verify larger classical or probabilistic systems, now permitting to verify large industrial protocols. But there still remains a challenge: the verification of programs directly from their source code. The main problem with this issue remains the state space explosion phenomenon since even the smallest program contains variables that range from large sets of values (like `double` in C), thus inducing large data structures in the verification process. For example in [24], it is stated that *"applying model checking by itself to programs will not scale to programs of much more than 10k lines"*. In parallel to the design of more compact data structures, a lot of work has been done on specific methods for considering larger systems: data abstractions[5], static analysis, abstract interpretation[3].

In the last years, methods have emerged that can be used for the automatic verification of programs. All these methods use random sampling and are based on an approach that borrows ideas from model checking and program testing. With this kind of method, the verification is approximated in the sense that there are two specific parameters: ε gives the quality of the output of the algorithm, and δ gives the error probability of the algorithm. These parameters can be lowered at will. These methods are called Monte-Carlo Model Checking [10], Approximate Probabilistic model checking [17, 15] and statistical hypothesis testing [25].

In this paper, we explain how to use one of these methods, namely approximate probabilistic model checking [17, 15], in order to verify real programs written in ANSI C and/or C++. This Monte-Carlo method is based on the random sampling of bounded paths of the transition system. So the output of the algorithm will be the probability (over the probabilistic choices of the system) of a given property to be satisfied by the system. It means that we have to sample bounded executions of the program. In order to do that, we encapsulate automatically the program into a C++ program that provides an easy way to access the values of the variables of the program at any time of the execution.

Our main result is to show the effectiveness of the approach since using this technique, we are able to find bugs in C programs that act on integers, floats or variables of any type (except for variables dynamically allocated) and that contains operations known to be intractable for classical model checking. The main differences between our approach and other methods are the following:

- Due to the use of the Monte-Carlo method we never construct the whole transition system underlying the program. It means that our tool is not subject to the state space explosion phenomenon (we store in memory only a path of fixed length).
- The method is fully automatic (nothing to do except for writing the property we want to verify).

- The method works directly on the C or C++ source code of the program (no modelling phase).

The structure of the paper is the following. First, we review the work done by other research groups in the field of program verification and the field of approximation-based methods for model checking. Then, in section 3, we recall the theory designed in [17, 15] and explain how to use it in the framework of program verification. The section 4 is dedicated to the presentation of our prototypical implementation and some experimental results. Finally, in section 5, we discuss the implementation and give clues on how to design a complete tool that can work with more complicated programs.

2 Related work

In this section, we only consider work about software verification, that is, model checking of source code (following [13]). Compared to the huge amount of literature about "classical model checking" (that is, model checking of systems described using specific modelling languages), there are only few papers and tools that work directly from the source code of the programs. Since our prototype works on C and C++ source code, the most related work is the research on CMC [20] and VeriSoft [9]. There are others tools that works on C or C++ programs but not directly. For example, SLAM [1] and BLAST [14] are used to abstract program code and verify the abstract program instead of the actual program. C Wolf [8] is a toolset for extracting transition systems from C code. It means that it is not a full model checker but an additional tool that allows classical model checkers to verify C programs.

In this paper, we use the method of [15]. This method, called Approximate Probabilistic Model Checking, is used for the verification of linear temporal property against discrete time Markov chains. We describe it precisely in the following section. Other approximation methods are available. In [10], a randomized algorithm for the approximate model checking of safety properties is given. This approximation method uses the optimal approximation algorithm of [6]. Since then, this method was used for the verification of programs via the use of a modified version of GCC [11].

In the field of approximate verification for probabilistic systems, there are methods based on statistical hypothesis testing such as the methods of [25] and [22]. Using these methods one can model check black-box probabilistic systems against specifications given in a sub-logic of Continuous Stochastic Logic (CSL). These approaches differ from the one of [15] and [10] by using statistical hypothesis testing instead of randomized approximation schemes.

Finally, we want to mention the work of [19], where both random testing and abstract interpretation are used for the verification of C programs. The main advantage of this method is that it applies to systems that contain non-deterministic choices.

3 Theoretical foundations

To implement our prototype for the verification of real C programs, we use the approximate model checking method of [15]. Originally, the purpose of this method is the efficient (w.r.t. memory) but approximate verification of monotone temporal properties of discrete time Markov chains (DTMC). Here, we use this method in a somehow different framework. First there are programs with non-deterministic choices. When there is such a choice, we replace the non-determinism by a probabilistic uniform choice. Second, since we only consider bounded temporal properties, we don't need to be careful about the monotonicity of the properties.

We now describe the Monte-Carlo method of [15] to approximate satisfaction probabilities of Linear Temporal Logic (LTL) monotone properties over fully probabilistic transitions systems (DTMCs). LTL formulas are built over a set of atomic propositions using temporal operators. We first define the kind of system we will study:

Definition 1 *A DTMC is a tuple $\mathcal{M} = (S, \overline{s}, P)$ where S is a set of states, $\overline{s}$ is the initial state, and P is a function that gives the probability of transition from one state to another one.*

If we modify the non deterministic choices as above, every program can be modelled as a DTMC. Then, using the method of [15], we can verify the program without constructing explicitly the DTMC. For a program, a state will be a vector of the actual values of all program variables.

We denote by $Path(s)$ the set of paths whose first state is s. The length of a path π is the number of states in the path and is denoted by $|\pi|$, this length can be infinite but we will only consider here bounded length paths. The probability measure $Prob$ over the set $Path(s)$ is defined in a classical way. We denote by $Prob[\phi]$ the measure of the set of paths $\{\pi \mid \pi(0) = s \text{ and } \mathcal{M}, \pi \models \phi\}$ (see [23]). Let $Path_k(s)$ be the set of all paths of length $k > 0$ starting at s in a PTS. The probability of an LTL formula ϕ on $Path_k(s)$ is the measure of paths satisfying ϕ in $Path_k(s)$ and is denoted by $Prob_k[\phi]$.

In order to estimate the probabilities of bounded properties with a simple randomized algorithm, we generate random paths in the probabilistic space underlying the DTMC structure of depth k and compute a random variable which estimates $Prob_k[\psi]$. Our approximation is good with confidence $(1 - \delta)$ after a number of samples polynomial in $\frac{1}{\varepsilon}$ and $\log \frac{1}{\delta}$. The main advantage is that, in order to design a path generator, we need only to execute the program and

store the values of all variable at each new assignment of a variable. It means that we consider that an atomic step of the program is the modification of the value of a variable.

Our approximation problem is defined by giving as input x a program, a formula and a positive integer k. The program is used to generate a set of execution paths of length k. A randomized approximation scheme is a randomized algorithm which computes with high confidence a good approximation of the probability measure $\mu(x)$ of the set of execution paths satisfying the formula ϕ.

Definition 2 *A fully polynomial randomized approximation scheme (FPRAS) for a probability problem is a randomized algorithm $\mathcal{A}$ that takes an input x, two real numbers $0 < \varepsilon, \delta < 1$ and produces a value $A(x, \varepsilon, \delta)$ such that:*

$$Prob\big[|A(x,\varepsilon,\delta) - \mu(x)| \leq \varepsilon\big] \;\geq\; 1 - \delta.$$

The running time of $\mathcal{A}$ is polynomial in $|x|$, $\frac{1}{\varepsilon}$ and $\log \frac{1}{\delta}$.

The probability is taken over the random choices of the algorithm. We call ε the *approximation parameter* and δ the *confidence parameter*. The algorithm we use is the following:

Generic approximation algorithm $\mathcal{GAA}$
Input: $program, k, \psi, \varepsilon, \delta$
Output: approximation of $Prob_k[\psi]$
$N := \ln(\frac{2}{\delta})/2\varepsilon^2$
$A := 0$
For $i = 1$ to N do
  $A := A +$ **Random Path**$(program, k, \psi)$
Return A/N

where **Random Path** is:

Random Path
Input: $program, k, \psi$
Output: samples a path π of length k and check formula ψ on π

1. Generate a random path π of length k in the program
2. If ψ is true on π then return 1 else 0

Theorem 1 *(from [15]). This approximation algorithm is a fully randomized approximation scheme for the probability $p = Prob_k[\phi]$ of an LTL formula ϕ if $p \in]0, 1[$.*

This result is obtained by using Chernoff-Hoeffding bounds [16] on the tail of the distribution of a sum of independent random variables. The time complexity of the algorithm is polynomial in $\log(1/\delta)$ and $1/\varepsilon$. The space complexity is linear in the length of execution paths. This algorithm can be deployed on a distributed computation model to distribute path generation and formula verification on a cluster of workstations. Measurements demonstrated ([12]) that this distribution scheme is scalable and provides a linear acceleration.

Note that sometimes we are only interested in finding bugs, so we just want to verify if the probability is strictly positive. It means that we don't have to generate all the paths needed to approximate the value of the probability. In order to apply our method to the framework of programs written in a real programming language such as C, we need to modify non deterministic steps (e.g. interaction with the user) into probabilistic ones (e.g. random inputs).

There is a similar probabilistic approach in the work of [10]. This approach uses an adaptive approximation algorithm with multiplicative error.

4 Implementation and results

4.1 Implementation

We designed a prototype that implements the approximation method for the verification of ANSI C and C++ programs. One of our goals was to make sure that the prototype worked without human intervention on the source code. In order to verify a program with the method described in section 3, we had two tasks to achieve:

- Generate execution paths of bounded length using the program without modification.
- Verify a given formula on each of these paths.

These two tasks are done by encapsulating the program into a larger C++ program whose general architecture is described in figure 1. Basically, we construct a modified version of the program that runs several times the original version and that computes the approximate value of the probability of the property to verify.

At the beginning of the program, we add a header that contains:

- a *tracker*: a piece of code that generates bounded execution paths,
- a rewriting of the classical types (integer, float etc.), in order to track the modifications of the values (see figure ,
- a specific declaration of formula variables: this is done automatically in order to track only the important variables.

To generate an execution path, we use the tracker (see figure 2). Basically, for each variable `var i` of the formula to check, the tracker contains two internal variables `p.var i` and `MW i`. The variable `p.var i` is in fact a pointer that links to `var i`. This means that the value of

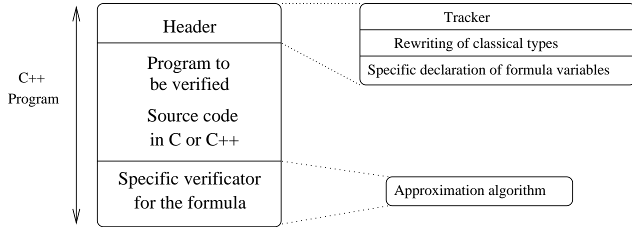

Figure 1. General scheme

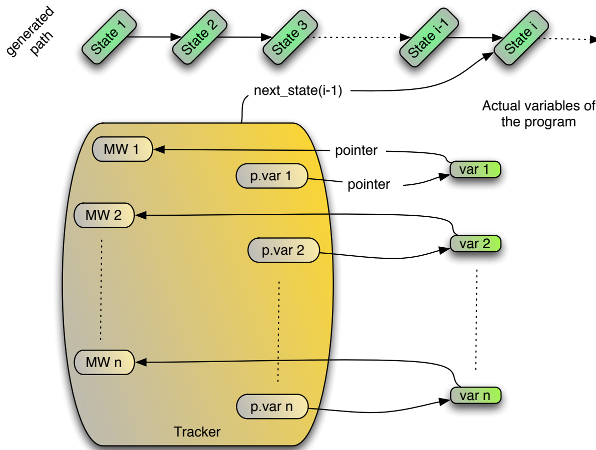

Figure 2. The tracker

`p.var i` is always the value of `var i` (even after a modification of `var i`). `MW i` (Modification Witness of `var i`) is pointed by (an automatically modified version of) `var i`. It means that each time `var i` is modified, the tracker will be aware of that. Using this mechanism, each time a variable is modified, the tracker outputs all the values of the variables of the formula as a vector `state i`. An execution path will be a list of the vectors `state i` for i ranging from 1 to the length choosen (indicated by the formula since we considered bounded temporal formulas).

For the first task, i.e. generating traces, the tracker maintains a counter of the current length of the path. When arriving at the desired length, the tracker send a `SIGUSR1` system signal to its process. The signal triggers an interruption in the program that stops the path generation and starts the path verification. Once the verification is completed, the program frees the memory (essentially the vectors that represent states), summarizes the results (i.e. computes the approximate value of the probability) and then re-launches the whole process until enough paths have been generated to compute the approximate value of the probability or enough information has been collected (for bug reporting).

For the second task, i.e. verification of the formula on one path, we just use a simple function (the *specific verifier* of the figure 1) that verifies recursively a formula using the bounded semantic of temporal logic. More information about the semantic of such a temporal logic can be found in [2]. This function also has an other goal: it sorts out the variables that are important for the verification of the formula. This has to be done since this information will be used in the header of the encapsulated program (see above).

Using this mechanism, we can output, by running the modified program, the probability that the original program satisfies the property.

4.2 Experimental results

To show the effectiveness of our approach, we made several experiments. All the experiments were done on an Intel P4 3 GHz with 1 GB RAM and 1 MB cache. For all the runs, we set $\varepsilon = 10^{-2}$ and $\delta = 10^{-5}$. We tested our prototype on two program examples. For each program, we verified the correct version of the program, and a buggy one.

Dining philosophers. We made several experiments on the dining philosophers problem [21]. Dining philosophers implement a simple mutual exclusion process. Being well studied, it allows us to validate the results of our prototype. Here, we conducted two set of experiments, one on a correct version of the algorithm and another one on a buggy version. For the correct version, we verified that the probability of a reachability property is 1. To create a buggy version we just changed one assignment of a variable in order to simulate a keystroke error while programming. The property we verified is that there exists at least an execution of the program where a philosopher does not release the forks (thus violating the fact that eventually each philosopher will eat).

The following table summarizes the results of the verification process for several values of the number of philosophers we consider.

	correct version	
# phil.	time (sec.)	memory (KB)
25	708.4	960
50	2970	1176
100	6711	1432
300	71400	4368
1000	out of reach[1]	60296

[1] The predicted computation time is around 9 days on a single workstation.

	buggy version	
# phil.	time (sec.)	memory (KB)
25	7×10^{-3}	936
50	0.02	992
100	0.04	1116
300	0.86	1592
1000	13	3272

For the correct version the computation time given is the time necessary to ensure that the probability is approximately 1. For the buggy one, this is the time to find a bug, that is the time to make sure that the probability is greater than 0. One can see that checking the correctness of the program is still a computationaly heavy task, but that finding a bug can be done in a very short time (and gives, as usual in model checking, a counter-example to the property).

Bubble sort. We also conducted the verification of a simple program that implements bubble sort. The property we verified for both versions of the program is that, at the end of the computation, the integer list given as input is well sorted. We construct the buggy version by modifying a value of a bound of a loop, basically, writing $<$ instead of $<=$.

	correct version	
# int.	time (sec.)	memory (KB)
10	138	884
20	940	976
30	5076	1340

	buggy version	
# int.	time (sec.)	memory (KB)
10	0.01	884
20	0.03	976
30	0.08	1340

Again, the experimental results show that the memory needed to complete the verification is very small and that the computation time remains reasonable.

5 Discussion

Traditionally, model checking is a highly expensive computational activity. The main drawback of the method is the memory needed to finalize the verification of large systems. On the contrary, our method has the advantage of storing only one execution path of bounded length at a time. This is why the experimental results show that the method is extremely efficient in both computation time and memory comsuption. This is also due to the fact that we work directly on the C source code of the program. Indeed, we don't have to go through a translation phase from a specific description language to an internal representation suitable for the model checker.

However, our implementation is still too naive to be generalized into the design of a complete verification tool.

Firstly, since we use a SIGUSR1 system signal in order to stop the path generation, we cannot verify programs that use this kind of signal. There are really a lot of such programs (automount, apache, snort and most of the system daemons). Moreover, since the program is not entirely under an interpreter control, the program might deadlock, and therefore the tracker might never be able to send a SIGUSR1 interrupt. As a consequence, taking a sample might never terminate. In order to avoid this, we added a timer that breaks the program whenever a fixed timeout expires.

Another problem of our implementation is that we encapsulate the original program into a C++ program. It means that we can only deal with languages compatible with C++. To enlarge the spectrum of the target languages, the use of external tools could maybe lead to some improvements. Some research groups started to work with GCC in order to deal with more languages [11], but they deal with one of the intermediate representation of the compiler, and this has an over-cost in term of computation time/memory consumption.

To overcome these problem, we started recently the design of a tool based on the same mechanism as GDB (the Gnu Debugger). Using GDB, we have access to more information than the ones given by our prototype. For example using GDB we can consider non determinism in a more clever way, due to the presence of breakpoints. Since we don't need all the functionalities of GDB, we use the ptrace() function. Indeed, this function is at the heart of the breakpoint mechanism of GDB.

6 Conclusion and Future Work

In this paper, we presented a prototype implementation of an approximation-based model checking method for the verification of programs written in C and C++. The main characteristics of this method are the following:

- The method handles C and C++ source code directly, without any modelling phase (that otherwise must be done by a highly qualified engineer). It gives the method two advantages: it saves human time and cost, and avoid modelling errors.
- Since we proceed by a sampling of program execution, we never construct the whole transition system underlying the program. So the prototype never stores more than one execution trace in memory, thus avoiding the state space explosion phenomenon.

From the efficiency point of view, we showed in section 4 that the computation times are very good and that the memory comsuption is low.

In future work, we plan to develop a complete program verification platform for more languages using the ideas we presented in section 5. We also plan to use the natural massive distribution scheme usable for verification method based on sampling [12] in order to verify efficiently very large programs.

Acknowledgments. Many thanks goes to Radu Grosu for a lot of comments and advice about the verification of programs and to the anonymous referees for their comments.

References

[1] T. Ball, R. Majumdar, T. D. Millstein and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. *Conference on Programming Language Design and Implementation*, pages 203–213,2001.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDD's. *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1999. Lecture Notes in Computer Science, 1573:193–207, Springer-Verlag.

[3] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13:103–179, 1992.

[4] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[5] E.M. Clarke, O. Grumberg, and D.A. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[6] P. Dagum, R. Karp, M. Luby and S. Ross. An optimal algorithm for Monte-Carlo estimation. *SIAM journal of computing*, 29(5):1484–1496, 2000.

[7] L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of concurrent probabilistic processes using MTBDDs and the Kronecker representation. *6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2000. Lecture Notes in Computer Science, vol. 1785, Springer-Verlag.

[8] D. C. DuVarney and S. P. Iyer. C Wolf — a toolset for extracting models from C programs. In International Conference on Formal Techniques for Networked and Distributed Systems (FORTE), 2002. Lecture Notes in Computer Science, vol. 2529, Springer-Verlag.

[9] P. Godefroid. VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software. Proceedings of CAV'97 (9th Conference on Computer Aided Verification), 1997. Lecture Notes in Computer Science, vol. 1254, pages 476-479, Springer-Verlag.

[10] R. Grosu and S. Smolka. Monte-Carlo Model Checking. *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005. Lecture Notes in Computer Science, vol. 3440, pages 271–286, Springer-Verlag.

[11] R. Grosu, X. Huang, S. Jain and S.A. Smolka. Open Source Model Checking. *In Proc. of SoftMC'05, the 3rd Workshop on Software Model Checking*, 2005, Edinburgh, UK.

[12] G. Guirado, T. Hrault, R. Lassaigne and S. Peyronnet. Distribution, approximation and probabilistic model checking. *4th Parallel and Distributed Methods in Verification (PDMC 05)*. Electronic Notes in Theor. Comp. Sci., 2005. To appear.

[13] K. Havelund and W. Visser. Program Model Checking as a New Trend. *journal on Software Tools for Technology Transfer (STTT)*. 4(1):8–20, 2002.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with Blast Proceedings of the 10th SPIN Workshop on Model Checking Software (SPIN), Lecture Notes in Computer Science 2648, Springer-Verlag, pages 235-239, 2003.

[15] T. Herault, R. Lassaigne, F. Magniette and S. Peyronnet. Approximate Probabilistic Model Checking. *Proc. of the 5th Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*, Venice, Italy, Lecture Notes in Computer Science, 2937:73-84. 2004.

[16] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:13-30, 1963.

[17] R. Lassaigne and S. Peyronnet. Approximate Verification of Probabilistic Systems. In *Proc. of the second joint PAPM-PROBMIV*, Lecture Notes in Computer Science, 2399, 2002.

[18] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[19] D. Monniaux. An Abstract Monte-Carlo Method for the Analysis of Probabilistic Programs. In *28th Symposium on Principles of Programming Languages (POPL '01)*.

[20] M. Musuvathi, D. Park, A. Chou, D. Engler and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. Proceedings of the Fifth Symposium on Operating System Design and Implementation, December, 2002.

[21] A. Pnueli and L. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, pages 1:53–72, 1986.

[22] K. Sen, M. Vishanathan, and G. Agha. Statistical model checking of black-box probabilistic systems . In *Proc. of the 16th Int. Conf. Computer Aided Verification*, Lecture Notes in Computer Science, 3114:202–215, 2004.

[23] M.Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. *Proc. 26th Foundation of Computer Science (FOCS)*, pp. 327–338, 1985.

[24] W. Visser, K. Havelund, G. P. Brat, S. Park and F. Lerda. Model Checking Programs. Autom. Softw. Eng. 10(2): 203-232 (2003).

[25] H. L. S. Younes and R. G. Simmons. Probabilistic Verication of Discrete Event Systems using Acceptance Sampling. *In Proc. of the 14th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, 2404:223–235. 2002.