Statistical Model Checking of LLVM Code*

Axel Legay[1], Dirk Nowotka[2], Danny Bøgsted Poulsen[2], and Louis-Marie Tranouez[1]

[1] INRIA, Rennes, France
[2] Kiel University, Kiel, Germany

Abstract. We present the new tool Lodin for statistical model checking of LLVM-bitcode. Lodin implements a simulation engine for LLVM-bitcode and implements classic statistical model checking algorithms on top of it. The simulation engine implements only the core of LLVM but supports extending this core through a plugin-architecture. Besides the statistical model checking algorithms Lodin also provides an interactive simulation front-end. The simulator front-end was integral for our second contribution - an integration of Lodin into Plasma-Lab. The integration with Plasma-Lab is integral to allow reasoning about rare properties of programs.

1 Introduction

Statistical Model Checking (SMC) [17] is an approximate verification technique that has attained a high interest from the formal methods community in recent years - evidenced by statistical model checking tools being developed [1, 2, 13, 15, 16] and by classical model checking tools implementing statistical methods [6, 11]. The reason for this interest is two-fold: firstly SMC is simulation-based and can therefore be applied to models for which the model checking problem [7] is undecidable, secondly SMC scales better with increased state spaces. Another interest of the formal verification community is applying formal methods to the analysis of real-life code [3, 4, 18]. These works are mainly focused on applying an exhaustive state space exploration of the source language. In this paper we present a tool, Lodin, that permits applying SMC-based techniques to programs. Lodin relies on a pre-compilation of the program with `clang` to produce a LLVM-bitcode [12] file used as the input model of Lodin. Functions defined externally of the program itself (e.g. system calls) are given semantics in Lodin through platform plugins. In this way Lodin is configurable to analyse embedded programs for various execution environments. Simulation-based techniques have the major downfall of rare properties requiring an infeasible number of samples to locate one with the property. To manage this, we seamlessly integrate Lodin with Plasma-Lab and get access to their implementation of importance splitting. Importance splitting is an efficient rare event simulation technique where a property is decomposed into several sub-properties that must be satisfied before the main property is satisfied.

* This work has been partially supported by the BMBF through the ARAMiS2 (01IS160253) project.

2 LODIN

LODIN[3] is a fairly new software analysis tool with the goal of analysing programs without modelling the program in an analysis-specific modelling language. LODIN achieves this ability by using LLVM bitcode [12] as its model language - thereby making LODIN available to any source language translatable to LLVM. The analysis techniques available in LODIN is currently explicit-state model checking and statistical model checking [17]. In this paper we focus on the latter.

Architecture LODIN consists of a user interface, algorithms or a simulator, state generators and a system model (Figure 1). The system model is a state and transition representation of the program under analysis. The system exposes a successor generation interface for higher architectural levels. During the generation of successor states, the system calls an interpreter module responsible for implementing the semantics of LLVM instructions. In between the algorithms and system level is a state generator level. This is mostly relevant for the explicit-state model checking part of LODIN. It allows for selecting various techniques to reduce the searched state space. For statistical model checking there is only a probabilistic state generator that selects what transition to perform according to probabilities obtained from the system. Real-life programs are developed to run under some execution environment providing core functionality to the program. To make LODIN as oblivious to the specific execution environment as possible, the core interpreter of LODIN has no built-in semantics for these. Instead it is possible to extend LODIN with platform plugins providing support for an execution environment. A platform plugin registers all the functions implemented by it when loaded by LODIN and the interpreter lets the plugin handle calls to one of these functions. In addition to implementing these external functions, platform plugins also have an interface to do their own transitions. This is useful for mimicking an interrupt system / signalling system.

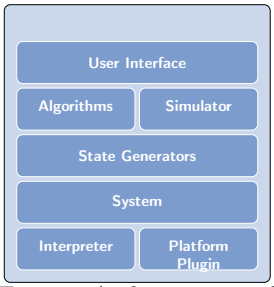

Fig. 1: Architecture of LODIN

Preparing files LODIN requires input in LLVM bitcode. We achieve this by compiling the program with `clang -emit-llvm -S -c file.c -o file.ll`. This is sufficient for programs without external dependencies. Properties are specified as expressions over LLVM registers thus we run `opt -instnamer file.ll -S -o fileN.ll` to generate the file `fileN.ll` in which registers have been given names. When a program has external dependencies and verification therefore requires the use of a platform plugin, the program must be compiled with headers specific for that plugin. If the header files are located in `/path/to/includes` then programs should be compiled with the command

[3] available at `https://spark.informatik.uni-kiel.de/data/lodin/FM18/Lodin-FM.zip`

```
clang -nodefaultlibs -ffreestanding -fno-builtin -emit-llvm -S -c -I/path/
    to/includes file.c -o file.ll
```

ensuring `clang` compiles the program without using any of its built-in libraries and only rely on the header files included on the command line.

How to use LODIN is a command line tool and is invoked by

```
./Lodin [options] file.ll query.q
```

where `[options]` includes options for selecting a platform plugin, setting a random seed and so on. The `query.q` file contains a one line query. The possible statistical model checking-based queries are generated by the below EBNF:

⟨*Query*⟩ ::= 'Pr' '[' '<=' ⟨*integer*⟩ ']' '(' '<>' ⟨*bool*⟩ ')'
| 'Estimate' '[' '<=' ⟨*integer*⟩ ',' ⟨*integer*⟩ ']' '{' 'max' ⟨*arith*⟩ '}'
| 'EnumStatesSMC' '<=' ⟨*integer*⟩ ⟨*integer*⟩

⟨*bool*⟩ :== 'DataRace'
| '[' ⟨*processid*⟩'.'⟨*string*⟩ ']'
| ⟨*arith*⟩ ⟨*comp*⟩ ⟨*arith*⟩
| '('⟨*bool*⟩ '&&' ⟨*bool*⟩ '&&' ... '&&' ⟨*bool*⟩ ')'
| '(' ⟨*bool*⟩ '||' ⟨*bool*⟩ ... '||' ⟨*bool*⟩ ')'
| 'Exists' '('⟨*char*⟩ ')' '(⟨*bool*⟩')'
| 'Forall' '('⟨*char*⟩ ')' '(⟨*bool*⟩')'
|

⟨*comp*⟩ ::= '<' | '<=' | '==' | '>=' | '>' | '!='

where `<arith>` is an arithmetic expression over LLVM registers. LODIN also has limited support for using source variables in arithmetic expressions - this is however dependent on the debugging symbols contained in the input file. An expression `[0.func]` is true if the zeroth process can call the function `func`. The expression `Exists (p)(<bool>)` is true if for some process the Boolean expression is true. Any occurrence of `p` is replaced by an actual process during the evaluation. On the query side, `Pr[<=500] (<> <bool>)` estimates the probability of the Boolean expression being true within 500 steps. The number of samples needed is automatically adjusted using the Clopper-Pearson interval[8]. A query `Estimate [<=500,5000] {max < arith>}` generates 5000 runs each of 500 steps and estimates the expected maximal value of the expression. Finally, `EnumStatesSMC <=500 5000` generates 5000 runs each of 500 steps and counts the number of different states encountered during those simulations.

Listing 1.1: Calculating the Fibonacci Numbers. A main function initialising t1 and t2 is omitted.

```
#include <pthread.h>

int i=1, j=1;

#define NUM 16
#define NULL 0


void *
t1(void* arg)
{
  int k = 0;

  for (k = 0; k < NUM; k++)
    i+=j;

  pthread_exit(NULL);
}

void *
t2(void* arg)
{
  int k = 0;

  for (k = 0; k < NUM; k++)
    j+=i;

  pthread_exit(NULL);
}
```

Program	Runs	Satisfying	CI		Time (s)
fib/fib_4.ll	19 242	2789	[0.14 ,	0.15]	3.80
fib/fib_8.ll	7453	370	[0.04 ,	0.05]	2.26
fib/fib_16.ll	299	0	[0.00 ,	0.01]	0.16
fib/fib_32.ll	299	0	[0.00 ,	0.01]	0.28
ptrace/ptrace.ll	33 249	10 412	[0.31 ,	0.32]	22.82
gossip/gossip_2.ll	34 470	11 575	[0.33 ,	0.34]	219.68
gossip/gossip_3.ll	13 187	1229	[0.09 ,	0.10]	94.41
gossip/gossip_4.ll	8450	481	[0.05 ,	0.06]	66.30
petersons/petersonsBug.ll	10 870	816	[0.07 ,	0.08]	1.64
petersons/petersons.ll	299	0	[0.00 ,	0.01]	0.05
robot/robot.ll	2507	38	[0.01 ,	0.02]	109.65
stack/stack.ll	299	0	[0.00 ,	0.01]	7.16

Table 1: LODIN results. The Runs columns is total number of generated runs, Satisfying is the number of satisfying runs while the CI column is a 95% confidence interval.

Example 1. Consider the program in Listing 1.1 where two threads cooperatively attempt to calculate the 32^{nd} Fibonacci number. With LODIN we estimate the expected number of i at termination of the program with `Estimate [<=5000,5000] {max @0.main.%tmp11}` , where `main.%tmp11` is a register in the compiled LLVM containing the value of the `i` variable. The result of this query is 438037. In addition to estimating the value, the query also outputs the values of the runs to a file. That file can the be be used to generate a histogram.

Example 2. The Fibonacci program considered in Example 1 is only correct if it at termination has found the 32^{nd} Fibonacci number (2178309). Using LODIN we estimate the probability of having either $i = 2178309$ or $j = 2178309$ using `Pr[<=5000] (<> [0.VERIFIERError])` which asks for the probability that a state is reached where the 0^{th} process can call `VERIFIERError` - a call the main function does if either $i = 2178309$ or $j = 2178309$. Verifying the query with LODIN results in the probability being in the range $[0, 0.01]$ with confidence 0.95 and no satisfying traces found.

In Table 1 we show results for a range of programs we have applied LODIN to. For space limitations we omit descriptions of the programs and instead refer the reader to [14] which contains both the source code and descriptions of the programs. We will note though, that the verification queries are all of the form `Pr[<=N] (<> ... )`.

3 PLASMA-LAB

In Example 2 we saw that simulation-based techniques may fail to find traces satisfying rare events - in the particular example the event is rare because it requires a very specific interleaving of the two threads. In the following we integrate LODIN with PLASMA-LAB and see how importance splitting can help guiding the simulation to one of these rare interleavings.

PLASMA-LAB [5] is a modular platform for statistical model-checking[4]. The tool offers a series of SMC algorithms, including advanced techniques for rare

[4] Available for download at `https://project.inria.fr/plasma-lab/`

event simulation, distributed SMC, non-determinism, and optimization. They are used with several modeling formalisms and simulators. The main difference between Plasma-Lab and other SMC tools is that Plasma-Lab proposes an API abstraction of the concepts of stochastic model simulator, property checker (monitoring) and SMC algorithm. In other words, the tool has been designed to use external simulators, input languages, or SMC algorithms. This also allows us to create direct plug-in interfaces with external specification tools, without using extra compilers.

Plasma-Lab architecture is illustrated in Figure 2. The core of Plasma-Lab is a light-weight controller managing the experiments and the distribution mechanism. It implements an API that allows controlling the experiments either through user interfaces or through external tools. It loads three types of plugins: 1. algorithms, 2. checkers, and 3. simulators. These plugins communicate with each other and with the controller through the API.

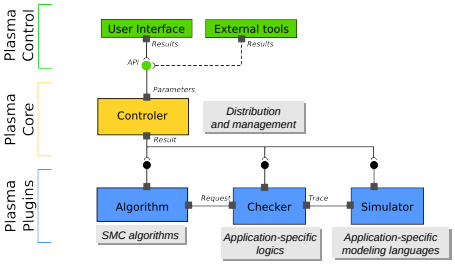

Fig. 2: Plasma-Lab architecture.

In Plasma-Lab rare properties are decomposed into intermediate properties using a notion of score function over the model-property product automaton. Intuitively, a score function discriminates good paths from bad, assigning higher scores to paths that are "closer" to satisfy the overall property. The model-property product automaton is usually hidden in the implementation of the checker plugin. Therefore Plasma Lab includes a specific checker plugin for importance splitting that facilitates the construction of score functions. The plugin allows writing small observer automata checking properties over traces and compute the score function. These observers implement a subset of the Bounded Linear Temporal Logic presented in [10].

Plasma-Lab implements two rare event algorithms based on the importance splitting technique, a fixed level algorithm and an adaptive level algorithm [9]. The fixed level algorithm requires the user to define a monotonically increasing sequence of score values whose last value corresponds to satisfying the property. The adaptive algorithm finds optimal levels automatically and requires only the maximum score to be specified. Both algorithms estimate the probability of passing from one level to the next by the proportion of a constant number of simulations reaching the upper level from the lower. New simulations to replace those that failed to reach the upper level are started from states chosen uniformly from the terminal states of successful simulations. The overall estimate is the product of the estimates of going from one level to the next.

LLVM Simulator plugin We have developed a simulator plugin for Plasma-Lab that interfaces with Lodin. This plugin is a pure wrapper around the simulator interface of Lodin. It communicates with the Lodin simulator via standard input and standard output. Lodin exposes the registers of all functions of the program to Plasma-Lab, and exposes Boolean variables corresponding

to the `[0.func]` style propositions of LODIN. The registers are named in the style `Pn_funcname_registerName` where Pn designates a variable belonging to the n^{th} process. If the program has been compiled with debug symbols and without optimisations, LODIN also exposes the original C-source primitive type variables to PLASMA-LAB. For supporting the importance splitting algorithm of PLASMA-LAB, LODIN provides a *State-Tag* that PLASMA-LAB uses to restart a simulation from that given state. In Table 2 we have applied the LODIN PLASMA-LAB integration to the models for which LODIN previously failed in finding a satisfying trace for.

Program	Levels	Probability	Time (s)
fib/fib_16.ll	7	1.5e−3	18.20
fib/fib_32.ll	14	4.0e−6	51.66
stack/stack.ll	13	3.86e−15	530.58

Table 2: PLASMA-LAB Importance Splitting Results. The algorithm was run with a budget of 1000 runs per level.

Example 3. Consider again Example 2 and recall we want to reach a state where the 0^{th} process can call `VERIFIERError`. In order to reach a state where the 32^{nd} Fibonacci number is found, all previous Fibonacci numbers must be found first. In Listing 1.2 we show an excerpt of the observer we use. First the `score` variable is defined as required by PLASMA-LAB. PLASMA-LAB also requires a `decided` variable. The observer should set this to true if it is no longer possible to satisfy a trace. An auxilliary variable, `steps`, is used by the observer to count the steps in the trace. After these variable declarations follows a series of update transitions in the style of reactive modules. Basically these transitions state that, if the sum of `i` and `j` is equal to a given Fibonacci number, and `t1` and `t2` are in the same iteration of their loop then update the score variable to a given value (`t1_tmp9`, `t1_tmp9` and `ti_tmp4` correspond to `i`,`j` and `k` respectively). The last two rules update the `steps` variable and terminate the trace when exceeding 5000 steps.

Listing 1.2: The observer used by PLASMA-LAB for Fibonacci example.

```
observer rare0bserver
      score : int init 0;
      decided : bool init false;
      steps : int init 0;
      [] (P1_t1_tmp9 + P2_t2_tmp9 = 5)& (P1_t1_tmp4=P2_t2_tmp4)-> (score
           '= 1);
      [] P1_t1_tmp9 + P2_t2_tmp9 = 13 & (P1_t1_tmp4=P2_t2_tmp4) -> (
          score '= 2);
      ...
      [] P0_Call_VERIFIERError=1 -> (score'=14);
      [] steps<5000 ->(steps'=steps+1);
      [] steps>=5000 ->(decided'=true);
endobserver
```

4 Conclusion

In this paper we presented LODIN a tool implementing SMC of LLVM code. The tool provides a plugin-architecture making it extendable to many execution environments. The tool also includes a simulation-component that is used to connect LODIN to PLASMA-LAB and thereby provide the first importance splitting implementation for LLVM.

References

[1] M. AlTurki and J. Meseguer. PVeStA: A Parallel Statistical Model Checking and Quantitative Analysis Tool. In A. Corradini, B. Klin, and C. Cîrstea, editors, *CALCO*, volume 6859 of *Lecture Notes in Computer Science*, pages 386–392. Springer, 2011. ISBN 978-3-642-22943-5.

[2] P. Ballarini, H. Djafri, M. Duflot, S. Haddad, and N. Pekergin. COSMOS: A Statistical Model Checker for the Hybrid Automata Stochastic Logic. In *QEST*, pages 143–144. IEEE Computer Society, 2011. ISBN 978-1-4577-0973-9. doi:10.1109/QEST.2011.24.

[3] J. Barnat, L. Brim, and P. Rockai. Towards LTL model checking of unmodified thread-based C & C++ programs. In A. Goodloe and S. Person, editors, *NFM*, volume 7226 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2012. ISBN 978-3-642-28890-6. doi:10.1007/978-3-642-28891-3_25.

[4] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011. ISBN 978-3-642-22109-5. doi:10.1007/978-3-642-22110-1_16. URL `https://doi.org/10.1007/978-3-642-22110-1_16`.

[5] B. Boyer, K. Corre, A. Legay, and S. Sedwards. Plasma-lab: A flexible, distributable statistical model checking library. In K. R. Joshi, M. Siegle, M. Stoelinga, and P. R. D'Argenio, editors, *QEST*, volume 8054 of *Lecture Notes in Computer Science*, pages 160–164. Springer, 2013. doi:10.1007/978-3-642-40196-1_12.

[6] P. E. Bulychev, A. David, K. G. Larsen, M. Mikucionis, D. B. Poulsen, A. Legay, and Z. Wang. UPPAAL-SMC: statistical model checking for priced timed automata. In H. Wiklicky and M. Massink, editors, *QAPL*, volume 85 of *EPTCS*, pages 1–16, 2012. doi:10.4204/EPTCS.85.1.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[8] C. J. Clopper and E. S. Pearson. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4):404–413, 1934.

[9] C. Jegourel, A. Legay, and S. Sedwards. An effective heuristic for adaptive importance splitting in statistical model checking. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, pages 143–159. Springer, 2014.

[10] C. Jegourel, A. Legay, S. Sedwards, and L.-M. Traonouez. Distributed verification of rare properties using importance splitting observers. *ECEASST*, 72, 2015.

[11] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011. doi:10.1007/978-3-642-22110-1_47.

[12] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International*

Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar 2004.

[13] A. Legay, S. Sedwards, and L. Traonouez. Plasma lab: A modular statistical model checking platform. In T. Margaria and B. Steffen, editors, *ISoLA*, volume 9952 of *Lecture Notes in Computer Science*, pages 77–93, 2016. doi:10.1007/978-3-319-47166-2_6.

[14] A. Legay, D. Nowotka, L.-M. Tranoues, and D. B. Poulsen. Lodin and Plasma-Lab examples. `https://spark.informatik.uni-kiel.de/data/lodin/FM18/LodinExamples.zip`, 2018. Accessed: 2018-05-08.

[15] K. Sen, M. Viswanathan, and G. A. Agha. VESTA: A Statistical Model-checker and Analyzer for Probabilistic Systems. In *QEST*, pages 251–252. IEEE Computer Society, 2005. ISBN 0-7695-2427-3. doi:10.1109/QEST.2005.42.

[16] H. L. S. Younes. Ymer: A Statistical Model Checker. In K. Etessami and S. K. Rajamani, editors, *CAV*, volume 3576 of *LNCS*, pages 429–433, 2005. ISBN 3-540-27231-3.

[17] H. L. S. Younes, M. Z. Kwiatkowska, G. Norman, and D. Parker. Numerical vs. statistical probabilistic model checking. *STTT*, 8(3):216–228, 2006.

[18] A. Zaks and R. Joshi. Verifying multi-threaded C programs with SPIN. In K. Havelund, R. Majumdar, and J. Palsberg, editors, *SPIN*, volume 5156 of *Lecture Notes in Computer Science*, pages 325–342. Springer, 2008. ISBN 978-3-540-85113-4. doi:10.1007/978-3-540-85114-1_22.