Analysis of Source Code Using UPPAAL

Mitja Kulczynski
Kiel University, Kiel, Germany
mku@informatik.uni-kiel.de

Axel Legay
Univeristy of Louvain, Louvain-la-Neuve, Belgium
axel.legay@uclouvain.be

Dirk Nowotka
Kiel University, Kiel, Germany
dn@informatik.uni-kiel.de

Danny Bøgsted Poulsen
Aalborg University, Aalborg, Denmark
dannybpoulsen@cs.aau.dk

In recent years there has been a considerable effort in optimising formal methods for application to code. This has been driven by tools such as CPACHECKER, DIVINE, and CBMC. At the same time tools such as UPPAAL have been massively expanding the realm of more traditional model checking technologies to include strategy synthesis algorithms — an aspect becoming more and more needed as software becomes increasingly parallel. Instead of reimplementing the advances made by UPPAAL in this area, we suggest in this paper to develop a bridge between the source code and the engine of UPPAAL. Our approach uses the widespread intermediate language LLVM and makes recent advances of the UPPAAL ecosystem readily available to analysis of source code.

1 Introduction

Over 30 years of research in applying formal methods to program verification has resulted in a plethora of tools [2, 3, 4, 7, 10, 13, 16] each being developed by different groups. We could write an entire paper just about the differences of all these tools, but overall they can (very) roughly be divided into two categories based on their input format: 1. tools that accept real source code, and 2. tools that use their own format based on formal models (e.g. Finite Automata, Timed Automata and Petri Nets). In the former category we find tools such as CPACHECKER [3], CBMC [13], DIVINE [2] and JavaPathfinder [18] focused on locating *programming errors* while in the latter we find UPPAAL [16], TAPAAL [7], ALLOY [11], TLA+ [5] PRISM [14, 15] and SPIN [10] focused on finding errors in the *design* of a system. The tools in each of these categories are successful in their own right, but there is only little flow between the groups. Two notable exception is DIVINE [2] that started its life as a general-purpose model checker and now focused on verifying LLVM and Zaks and Joshi [19] utilising SPIN to verify LLVM programs.

In this paper we present our initial work to bridge the gap from automata-based UPPAAL models to source-code analysis. We do this by interfacing LLVM-programs with UPPAAL using UPPAALs extendibility through dynamic link libraries [12]. In the dynamic link library resides an interpreter communicating with UPPAAL in regards to what happens with the discrete state-space while UPPAAL manages the exploration algorithms and timed aspects of the state space. In this way, we allow reusing the very efficient state exploration algorithms already implemented in UPPAAL but avoid mapping the entire expressivity of LLVM into UPPAAL. Furthermore, since an entire suite of tools is built around the UPPAAL core we hope to leverage these tools in the future. We especially have high hopes for doing schedulability analysis of concurrent programs in the future, and for using the statistical model checking (SMC) engine of UPPAAL to not only speed up the search, but also as a strategy for finding programming errors in a similar way as Chockler et al. [6] used SMC for the *satisfiability problem*. The usage of simulation-based techniques should also alleviate the scalability issue that hampered previous attempts at reusing model checking tools for software verification techniques.

Submitted to:
F-IDE 2021

Another potential advantage of integrating source code analysis into UPPAAL is that environmental behaviours are easily modelled with the stochastic hybrid automata formalism used by UPPAAL. Therefore we can easily change the analysis of a program source code in shifting environments by changing model parameters.

2 Program Model

In our work we are concerned with mapping LLVM code to UPPAAL, but we implement the integration to UPPAAL using our own intermediate representation called UL (Uppaal LLVM). The main reason for this is two-fold: firstly the LLVM language is huge and trying to cover it entirely is beyond the scope of this paper, and secondly by basing our translation on our own intermediate representation makes the translation independent of the input formalism, and we can — in principle — perform analysis for any input format with UPPAAL. From a maintenance point of view it also makes sense to define the analysis on your own internal representation as it makes the analysis independent of the input format: if we used LLVM directly we (potentially) have to modify large parts of our infrastructure for new releases of LLVM whereas we by having our own UL "just" need to modify our LLVM loading mechanism.

Types In UL we have four different integer types (**i8**,**i16**,**i32** and **i64**). Like in LLVM integer types are finite width bit vectors with no interpretation in regards to signedness. Instead each instruction of UL decides whether it interprets the bit pattern as being 2s-complement encoded signed number or an unsigned binary number. In addition to integer types UL has a Boolean type (**Bool**) and an address type (**Addr**) that are pointers to places in memory. We let $\mathbb{T}$ be the set of all types in UL.

Instructions Given a finite set of variables $\mathtt{R}$ we denote all instruction sequences of UL by $\mathscr{L}(\mathtt{Sequence})$. All possible instruction sequences of UL are generated by the EBNF in Figure 1. $\mathscr{L}(\mathtt{Sequence})$ refers to the language generated by the production rule $\langle\mathtt{Sequence}\rangle$. A typical instruction sequence are, for example, an arithmetic expression, a Boolean comparison operation, or a memory operation. Our instructions do not have associated types. A Type in UL is instead associated directly to registers through a map $\Gamma : \mathtt{R} \to \mathbb{T}$. Given this map we can straightforwardly create a type system, which we omit in this paper.

Memory UL uses a zero-indexed byte-oriented memory layout, so formally the memory is just a function $\delta : \mathbb{N} \to \mathbb{B}^8$. We refer to the set of all possible memory states by Δ. We can update the n^{th} byte in memory δ by simply modifying the image of n in δ. Thus $\delta[n \mapsto \mathtt{b}]$ sets the n^{th} byte to $\mathtt{b} \in \mathbb{B}^8$.

Timing Information A classical Control Flow Automaton (CFA) is a tuple $(\mathtt{R}, \Gamma, \mathtt{L}, \mathtt{l}, \mathtt{E})$ where $\mathtt{R}$ is a set of registers, $\Gamma : \mathtt{R} \to \mathbb{T}$ maps registers to types, $\mathtt{L}$ is a set of control locations, $\mathtt{l} \in \mathtt{L}$ is the initial location and $\mathtt{E} \subseteq \mathtt{L} \times \mathscr{L}(\mathtt{Sequence}) \times \mathtt{L}$ is the set of flow edges annotated with instruction sequence to be executed while moving along that edge. Such a model is sufficient if we are only concerned with the "flow" of a program, but we know that timing is an just as important aspect of a program: an airbag has to deploy at the right time, and not just at some point after a crash. In a security context it is also fairly well-known that measuring timing of a program can sometimes reveal confidential information about it. Even using very low capacity covert timing channels (2 bits per minute) leaking secret data e.g. a credit card number can be done in less than 30 minutes [1]. For these reasons we want to extend our model with timing information. To this end assume there exists a function $\Omega : \mathscr{L}(\mathtt{Sequence}) \to \mathscr{I}$ — where $\mathscr{I}$ is the set of intervals in $\mathbb{R}$ — that assigns upper and lower bound on the execution time of an instruction sequence. Notice that for defining this function it suffices to define the intervals for individual instructions as the

$\langle\texttt{Sequence}\rangle$ ::= $\langle\textit{Internal}\rangle$ $\langle\textit{InstrSeq}\rangle$ | $\langle\textit{InstrSeq}\rangle$

$\langle\texttt{InstrSeq}\rangle$ ::= $\langle\textit{InstrSeq}\rangle$ $\langle\textit{Instr}\rangle$ | $\langle\textit{Instr}\rangle$ | $\langle\textit{Assigns}\rangle$

$\langle\texttt{Instr}\rangle$ ::= $\langle\textit{Arith}\rangle$ | $\langle\textit{Cast}\rangle$ | $\langle\textit{Cmp}\rangle$ | $\langle\textit{Memory}\rangle$

$\langle\texttt{Internal}\rangle$ ::= **Assume r** | **NegAssume r** | **Assert**

$\langle\texttt{Assigns}\rangle$ ::= **r** | **r** $\leftarrow$ **NonDet** | **r** $\leftarrow$**Copy** Op

$\langle\texttt{Arith}\rangle$ ::= **r** $\leftarrow$ **Add** $\langle Op\rangle, \langle Op\rangle$ | **r** $\leftarrow$ **Sub** $\langle Op\rangle, \langle Op\rangle$ | **r** $\leftarrow$ **Div** $\langle Op\rangle, \langle Op\rangle$ | **r** $\leftarrow$ **SDiv** $\langle Op\rangle, \langle Op\rangle$ | **r** $\leftarrow$ **Mult** $\langle Op\rangle, \langle Op\rangle$ | **r** $\leftarrow$ **LShl** $\langle Op\rangle, \langle Op\rangle$ | **r** $\leftarrow$ **AShr** $\langle Op\rangle, \langle Op\rangle$ | **r** $\leftarrow$ **LShr** $\langle Op\rangle, \langle Op\rangle$

$\langle\texttt{Cast}\rangle$ ::= **r** $\leftarrow$ **SExt** $\langle Op\rangle$ | **r** $\leftarrow$ **ZExt** $\langle Op\rangle$ | **r** $\leftarrow$ **Trunc** $\langle Op\rangle$ | **r** $\leftarrow$ **BoolSExt** $\langle Op\rangle$ | **r** $\leftarrow$ **BoolZExt** $\langle Op\rangle$

$\langle\texttt{Cmp}\rangle$ ::= **r** $\leftarrow$ **LEq** $\langle Op\rangle, \langle Op\rangle$ | **r** $\leftarrow$ **SLEq** $\langle Op\rangle, \langle Op\rangle$ | **r** $\leftarrow$ **NEq** $\langle Op\rangle, \langle Op\rangle$ | **r** $\leftarrow$ **Eq** $\langle Op\rangle, \langle Op\rangle$ | **r** $\leftarrow$ **GEq** $\langle Op\rangle, \langle Op\rangle$ | **r** $\leftarrow$ **SGEq** $\langle Op\rangle, \langle Op\rangle$

$\langle\texttt{Memory}\rangle$ ::= **r** $\leftarrow$ **Load** $\langle Op\rangle$ | **Store** $\langle Op\rangle, \langle Op\rangle$

$\langle\texttt{Op}\rangle$ ::= **r** | $[n]^{\mathbf{ib}}{}_2$ | $[n]^{\mathbf{ib}}$

Figure 1: EBNF for generating instruction sequences of UL. Let **r** $\in \mathtt{R}$, $n \in \mathbb{N}$ and $z \in \mathbb{Z}$. The notation ($[n]^{\mathbf{ib}}{}_2$) $[n]^{\mathbf{ib}}$ is our notation encoding a number n into (2s-complement) bit-vector.

execution time of a sequence is the sum of the individual instructions. We use the Ω function to enrich a classical CFA coping with time.

Remark 1. Expert-knowledge of the execution platform is needed to properly asses the execution time of individual instructions. We are not concerned with assessing that for now, but do acknowledge this as non-trivial and an aspect worth investigating in the future.

Transition Semantics The first thing to define is the domain of the types in UL. Figure 2 shows the mappings in UL. The state that we execute an instruction sequence in consists of the sequence of instructions and an environment giving values to the registers $\mathtt{Env} : \mathtt{R} \to \cup_{t\in\mathbb{T}}\mathtt{dom}(t)$. A special state $\dagger$ signifies an error happened during execution. Externally the memory $\delta \in \Delta$ and the types of register Γ are passed on to the transitions. The result is a modified environment $\mathtt{Env}'$ and an updated memory $\delta' \in \Delta$. Thus the transition rules take the form $\delta, \Gamma \vdash \langle \mathtt{Inst}, \mathtt{Env}\rangle \to \mathtt{Env}', \delta'$.

Type	Domain
ib	$\mathbb{B}^b$
Bool	$\{\mathtt{true}, \mathtt{false}\}$
Addr	$\mathbb{B}^{64}$

Figure 2: Semantic domains of types in UL

The transition rules are obmitted do to space constraints and will not be shown here. They essentially look up values of their operands, perform the associated operations in the bit vector logic and assigns the left hand side to the result. The instructions **SExt**, **ZExt** resp. **BoolSExt**, **BoolZExt** and **Trunc** sign-extend or zero-extend or truncates the right hand operand to the type of the left hand side.

Remark 2. UL does not allow function calls. This is because we assume all function calls has been inlined. Inlining functions for verification purposes is a common practice [9, 13] and drastically simplifies the interpreter code.

An actual transition is performed on a network of CFAs, that is a structure $\mathscr{C}_1 \parallel \ldots \parallel \mathscr{C}_n$ where each $\mathscr{C}_i = (\mathtt{R}_i, \Gamma_i, \mathtt{L}, \mathtt{l}^i, \mathtt{E}_i)$ is a CFA. Again we assume the existence of a function Ω. The state of such a network is a tuple $(s_1, s_2, \ldots, s_n, \delta)$ where each $s_i = (l_i, \mathtt{Env}_i, x_i)$, $l_i \in \mathtt{L}$, $\mathtt{Env}_i : \mathtt{R}_i \to \cup_{t\in\mathbb{T}}\mathtt{dom}(t)$, $x_i \in \mathbb{R}$ and

$\delta \in \Delta$ is a memory state. Enriching the CFA with a timed behaviour results in specifying the following two transitions: a state $S = (s_1, s_2, \ldots, s_i, \ldots, s_n, \delta)$

1. can transit to state $S' = (s'_1, s'_2, \ldots, s'_i, s'_n, \delta')$ (written $S \to S'$) where $s_i = (l_i, \texttt{Env}, x_i)$ and $s'_i = (l'_i, \texttt{Env}', 0)$ if there exists an edge $e = (l_i, \mathit{Inst}, l'_i) \in \texttt{E}_i$, $\delta, \Gamma_i \vdash \langle \mathit{Inst}, \texttt{Env} \rangle \to \texttt{Env}', \delta'$ and $x_i \in \Omega(\mathit{Inst})$.

2. can delay d time units to state $S' = (s'_1, s'_2, \ldots, s'_i, \ldots, s'_n, \delta)$ (written $S \xrightarrow{d} S'$) if, for all i, $s_i = (l_i, \texttt{Env}, x_i)$ such that $s'_i = (l_i, \texttt{Env}, x_i + d)$ and there exists an edge $e = (l_i, \mathit{Inst}, l'_i)$ such that $x_i + d \in \Omega(\mathit{Inst})$.

Remark 3. The semantics for networks of CFAs is the traditional interleaving semantics. Our semantics, however, considers the execution of each edge atomically and does not properly reflect all possible interleavings of a real program — unless each edge has exactly one instruction as in the semantics of LLVM by Legay et al. [17]. This is not a problem as long we ensure all interleavings of memory accesses are possible. We can guarantee this by splitting edges so that an edge has exactly one memory access which is guaranteed to be the last instruction.

3 Integration with UPPAAL

In this section we will describe the general translation of a network of CFAs $\mathscr{C}_1 \parallel \ldots \parallel \mathscr{C}_n$ into a UPPAAL Timed Automaton [8]. For starters each CFA $\mathscr{C}_i$ is represented by a single Timed Automaton $\mathscr{A}^i$ with the same graph structure and stores register values in an integer array `lState`. It also has one clock `x`. Memory is similarly represented using an integer array `memory`. In regards to memory there is one last component we need, and that is an integer stored in the `memory` locating the next unused byte. For simplification our semantics did not include global registers, but we are nevertheless including them in our actual tool. These are stored in a UPPAAL integer array `glob`.

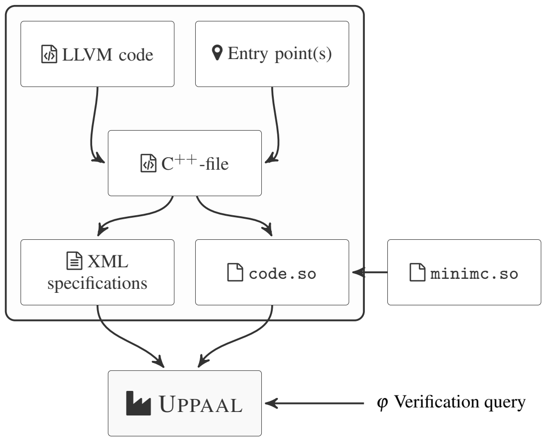

Figure 3: tool chain

Remark 4. On the surface this translation framework might seem a bit unnecessarily complicated but it does provide advantages. Firstly, since the state of CFAs are kept inside UPPAAL we can take advantage of UPPAALs capabilities: namely model checking and statistical model checking. Secondly, since UPPAAL knows about the graph structure of the CFA, we can use those locations as part of verification queries.

Automated tool chain We developed a tool chain[1] that automates building the UPPAAL model along with connecting it to the CFA interpreter (see Figure 3). The tool accepts an LLVM input-file along with information about the entry-points of the program (in case of a parallel program several entry points can be specified). This information is embedded into a C++-file which is compiled and linked against the library (`minimc`[2]) resulting in a dynamic link library (`code.so`) providing an interface to the interpreter (used in the resulting UPPAAL model as discussed), and also query functions in regards to the structure

[1] https://gitlab.com/dannybpoulsen/uppllvm
[2] https://github.com/dannybpoulsen/minimc

of the CFA. These latter query functions are used by a python script that creates the UPPAAL XML-file. Finally the `code.so` library and XML-file are passed to UPPAAL in which verification questions can be posed. The `minimc` library is needed as this converts LLVM to our UL structure and also performs needed syntactical modifications to the CFA like expanding **NonDet** instructions into several edges (one for each possible value), ensuring each edge only has one memory instruction, inline all LLVM functions, identify assert statements and add a special `AssertViolation` location that is reached only when an assert is violated, and add a `Term` location indicating normal termination of a CFA.

4 The tool chain in action

This section is devoted to demonstrating our tool chain. Next to verifying a password validation form which leaks information over time, we analyse a classical mutual exclusion protocol: Petersons Algorithm. This diverse set of examples demonstrates the capabilities of the presented approach.

4.1 On timing leaks

Timing is an important aspect of many programs and a major reason for integrating software models like LLVM into UPPAAL. In this example we identify potential timing leaks in a wrongly implemented password validation program given in Figure 5. After setting the password to `abcdef` the program enters a loop reading characters by the function `read()`. The characters are immediately compared to the expected password `abcdef`. However, `read()` is an undefined function and therefore our toolchain replaces it by a **NonDet** instruction which forces UPPAAL to search all possible values of k in each iteration of the loop. Adding a clock `G` that is never reset to the UPPAAL model allows us to estimate the run time of all paths through the program by the UPPAAL query

```
E[<=500;1000] (max:  G*(1-main.AssertViolation || main.main_Term)).
```

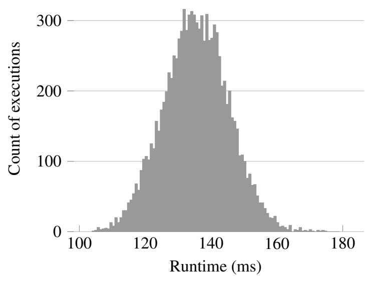

Figure 4: Runtime distribution of Figure 5.

We went up 500ms within 1000 runs. to Figure 4 shows a distribution plot of the run times revealing a variation in the run times. A symbolic analysis (in UPPAAL) reveals that a properly terminating program has a run time in the range $[125, 175]$ ms. Any execution-time outside this range could leak information to an attacker, as it has lower run-time because we broke out of the loop due to mismatching characters.

```
char read ();

#define N 6

int main () {
  char sec[N];
  for (int i = 0; i <N; i++) {
    sec[i] = 'a'+i;
  }
  for (int i = 0; i< N; i++) {
    char k = read();
    if (k != sec[i])
      assert(0);
  }
  return 0;
}
```

Figure 5: Program depending on timing.

4.2 Verifiying a mutual exclusion protocol

As a proof of concept we have implemented a mutual exclusion protocol (Petersons Algorithm) in C — see Figure 6 for an excerpt — and ran it through our toolchain after first compiling it to LLVM using `clang`. Using UPPAAL we are able to verify that the algorithm behaves correctly (i.e. guarantees mutual exclusion). We have also made an implementation of Petersons Algorithm containing an error breaking the mutual exclusion property. The error is initialising the `*opt.mflag` variable wrongly on line 20. Our toolchain does correctly find a path showing the mutual exclusion property is broken in this case. In order to locate the error we had to first find the CFA location (let us call it `Crit` correspodning to line 45 and 26 respectively, and since UPPAAL has knowledge about those we can simply ask UPPAAL `E<> (petersons1.Crit && petersons2.Crit)`.

```
1    int *mflag;
2    int *oflag;
3    int *turn;
4  }Options;
5
6  int turn = 0;
7  int oneflag;
8  int secondflag;
9
10 int crit1 = 0;
11 int crit2 = 0;
12
13 void petersons1 () {
14   Options opt;
15   opt.mflag = &oneflag;
16   opt.oflag = &secondflag;
17   opt.turn = &turn;
18
19   *opt.mflag = 1;
20   *opt.turn = 1;
21
22   while (*opt.oflag
23           && *opt.turn == 1) {
24    /* busy wait */
25   }
26   // critical section
27   crit1 = 1;
28   // end of critical section
29   crit1 = 0;
30   *opt.mflag = 0;
31 }
32
33 void petersons2 () {
34   Options opt;
35   opt.mflag = &secondflag;
36   opt.oflag = &oneflag;
37   opt.turn = &turn;
38
39   *opt.mflag = 1;
40   *opt.turn = 0;
41
42   while (*opt.oflag
43           && *opt.turn == 0) {
44    /* busy wait */
45   }
46   // critical section
47   crit2 = 1;
48   // end of critical section
49   crit2 = 0;
50   *opt.mflag = 0;
51 }
```

Figure 6: Excerpt of Petersons algorithm in C.

5 Conclusion

In this paper we presented our preliminary work towards utilising UPPAAL for model checking of LLVM-code. Our current strategy is to “outsource” LLVM to an external interpreter and allow UPPAAL to act as a simulation controller. Our initial experiments show the integration is functional and warrants further investigations. In the future we will investigate mixing our LLVM-verification model with (stochastic) models of the environment. This would allow analysing the source in different environmental contexts by simply changing parameters of the model. A use-case could be analysing whether a cruise controller for a car responds fast enough when moved to quicker accelerating car which names an important capability of the presented ideas.

References

[1] Agat, J.: Transforming out timing leaks. In: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 40–53 (2000)

[2] Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkai, P., Štill, V.: Model checking of C and C++ with DIVINE 4. In: Automated Technology for Verification and Analysis (ATVA 2017). LNCS, vol. 10482, pp. 201–207. Springer (2017)

[3] Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 184–190. Springer (2011)

[4] Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. pp. 209–224. USENIX Association (2008), http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[5] Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: Verifying safety properties with the tla+ proof system. In: International Joint Conference on Automated Reasoning. pp. 142–148. Springer (2010)

[6] Chockler, H., Ivrii, A., Matsliah, A., Rollini, S.F., Sharygina, N.: Using cross-entropy for satisfiability. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. p. 1196–1203. SAC '13, Association for Computing Machinery, New York, NY, USA (2013), https://doi.org/10.1145/2480362.2480588

[7] David, A., Jacobsen, L., Jacobsen, M., Jørgensen, K.Y., Møller, M.H., Srba, J.: TAPAAL 2.0: Integrated Development Environment for Timed-Arc Petri Nets. In: TACAS. pp. 492–497 (2012)

[8] David, A., Larsen, K.G., Legay, A., Mikucionis, M., Poulsen, D.B.: Uppaal SMC tutorial. STTT 17(4), 397–415 (2015)

[9] Falke, S., Merz, F., Sinz, C.: LLBMC: improved bounded model checking of C programs using LLVM - (competition contribution). In: Piterman, N., Smolka, S.A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7795, pp. 623–626. Springer (2013)

[10] Holzmann, G.J.: The Model Checker Spin. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)

[11] Jackson, D.: Alloy: a language and tool for exploring software designs. Communications of the ACM 62(9), 66–76 (2019)

[12] Jensen, P.G., Larsen, K.G., Legay, A., Nyman, U.: Integrating tools: Co-simulation in UPPAAL using FMI-FMU. In: 22nd International Conference on Engineering of Complex Computer Systems, ICECCS 2017, Fukuoka, Japan, November 5-8, 2017. pp. 11–19. IEEE Computer Society (2017), https://doi.org/10.1109/ICECCS.2017.33

[13] Kroening, D., Tautschnig, M.: CBMC - C bounded model checker - (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis

of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8413, pp. 389–391. Springer (2014), `https://doi.org/10.1007/978-3-642-54862-8_26`

[14] Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 2.0: A Tool for Probabilistic Model Checking. In: QEST. pp. 322–323. IEEE Computer Society (2004)

[15] Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV. Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer (2011)

[16] Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. STTT 1(1-2), 134–152 (1997)

[17] Legay, A., Nowotka, D., Poulsen, D.B.: Automatic verification of llvm code (2020)

[18] Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. 10(2), 203–232 (2003), `https://doi.org/10.1023/A:1022920129859`

[19] Zaks, A., Joshi, R.: Verifying multi-threaded C programs with SPIN. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN. Lecture Notes in Computer Science, vol. 5156, pp. 325–342. Springer (2008)