YASSi: Yet Another Symbolic Simulator Large (Tool Demo)

Sebastian Pointner[1(✉)], Pablo Gonzalez-de-Aledo[1], and Robert Wille[1,2]

[1] Johannes Kepler University Linz, Linz, Austria
{sebastian.pointner,robert.wille}@jku.at, pablo.aledo@gmail.com
[2] Software Competence Center Hagenberg GmbH (SCCH), Hagenberg, Austria

Abstract. Safety critical systems have finally made their way into our daily life. While recent industrial and academic research could already improve the design cycle for such systems, ensuring the functionality of such systems still remains an open question. Such systems which are composed of hardware as well as software components have to be checked since any wrong behavior of the system could end up in harming human life. To this end, program analysis techniques can be applied in order to ensure that the program works as intended and that no unwanted behavior is executed. However, approaches like static or dynamic program analysis which are widely applied for this purpose still lead a large number of fault positive results. To overcome such limitations an alternative approach called symbolic execution has been proposed. In this work, we present a tool called YASSi which implements this approach. Applying YASSi allows to symbolically execute programs written in the C/C++ language. By this, YASSi can be applied for several applications needed for the checking program for safety critical properties like (1) assertion checking, (2) reachability analysis, or (3) stimuli generation for digital circuits.

Keywords: Symbolic simulation · Assertion checking · Stimuli generation

1 Introduction

The technical progress achieved by academia as well as industry within the last decades led to more and more complex systems. These systems, which are composed of software and hardware components, have made their way into our daily life and are especially very important in terms of functional safety applications. To this end, it is of utter most importance to ensure that applications like the trigger unit of an airbag or a breathing apparatus for medial emergencies work as intended.

In order to ensure that such systems are getting realized correctly, the hardware as well as the software part of the system has to be checked for correctness.

G. Kotsis et al. (Eds.): DEXA 2020 Workshops, CCIS 1285, pp. 25–31, 2020.
https://doi.org/10.1007/978-3-030-59028-4_3

Since the hardware and the software part of the system are getting designed with abstract programming languages, program analysis techniques can be applied for this purpose. To this end, static program analysis like Control-Flow-Analysis [1] as well as dynamic program analysis like Dynamic-Program-Slicing [2] have emerged in the past. However, static as well as dynamic program analysis techniques both have major limitations (e.g. a significant number of false positives). Hence, in contrast to these program analysis techniques, the approach of symbolic execution emerged in the past [3]. Symbolic execution has been investigated heavily in the past and already led to a significant number of tools. These tools—including KLEE [4], DIVINE [5], Forest [6], or CBMC [7]—allow to symbolically execute programming languages like C/C++. However, symbolic execution is not limited to C/C++ only. Approaches like [8] also show that those methods can even be utilized for the execution of abstract hardware descriptions such as provided in SystemC.

In this work, we present the tool YASSi (*Y*et *A*nother *S*ymbolic *Si*mulator) as our state-of-the-art approach for the realization of symbolic execution for academic research. Our approach is capable to symbolically execute program code written in the C/C++ language. To this end, YASSi is based on the *Low Level Virtual Machine* (LLVM) compiler infrastructure [9] and utilizes the power of modern reasoning engines like Z3 [10] or Boolector [11] for decision finding. Applying YASSi allows the user to symbolically explore his/her design and, therefore, to check certain design properties or to perform reachability analysis. Compared to other state-of-the-art symbolic execution tools, we intentionally have designed YASSi in a fashion that it is easily extendable for academic research purposes.

The remainder of this paper is structured as follows. The next section briefly reviews the principle of symbolic execution for program analysis. Based on this, we are introducing YASSi as our approach for an academic tool for symbolic execution in Sect. 3. Afterwards, we discuss the applicability of YASSi for specific applications in Sect. 4, before we conclude the paper in Sect. 5.

2 Background

This section briefly reviews the approach of symbolic execution as an advanced program analysis technique [3]. In general, symbolic execution rests on the idea of rather than using concrete values for so-called *free variables* (e.g. the inputs of a function), the value of such variables is treated symbolically.

Example 1. *Figure 1a shows a code snippet composed of using two free variables and three conditional statements. Depending on the value held by the variables variable_a and variable_b, the execution of the code snippet returns a different value back to the host system.*

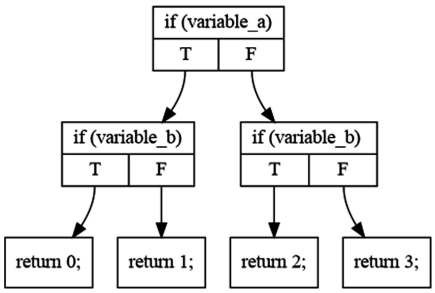
In order to perform a symbolic execution, the system has to keep track of all possible execution possibilities. To this end, symbolic execution has to evaluate all possible outcomes of branches within the source code. Moreover, the execution

engine keeps track of the positive branches (i.e. the taken branch) as well as the negative branches (i.e. the non-taken branch).

Example 2. *Consider again Fig. 1b which shows the* Control Flow Graph *(CFG) of the code snippet. Every time the execution reaches a branching condition, it is checked whether there is a possible valid assignment for the branching variable. In case of the branch as shown on top of Fig. 1b, the symbolic execution engine has to ensure that there is a valid assignment possible for variable_a which ends up in taking the branch as well as not taking the branch.*

```
1  int main ()
   {
3    int variable_a;
     int variable_b;
5
     if (variable_a){
7      if (variable_b){
         return 0;
9      } else {
         return 1;
11     }
     } else {
13     if (variable_b){
         return 2;
15     } else {
         return 3;
17     }
     }
19 }
```

(a)



(b)

Fig. 1. Source code and according CFG showing nested branches.

The symbolic execution engine has to decide whether a branch can be taken or not. To this end, modern symbolic execution engines are invoking the power of modern reasoning engines, e.g. for *Satisfiability Modulo Theories* (SMT), for this purpose. Moreover, the so-called *branching conditions* can be formulated using the so-called SMT2 constraint language [12]. The reasoning engine decides for every branching condition if there is an assignment possible for every *free variable* in order to take or not take the branch. The solutions generated by the reasoning engine are then directly applied by the symbolic execution engine to perform the execution of the target code.

3 The YASSi Tool

This section now introduces the YASSi symbolic simulation tool[1]. To this end, the section first introduces the modular architecture of YASSi in general before the major components of YASSi (i.e. the front-end and the back-ends) are getting discussed.

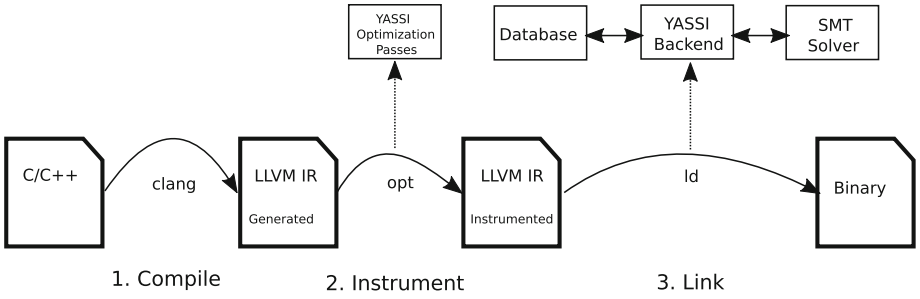


Fig. 2. Basic architecture of YASSi.

The basic architecture of YASSi is illustrated in Fig. 2. As can be seen in the figure, YASSi performs several steps before the symbolic execution can be performed by running a special binary. To this end, the generation of this binary is controlled YASSi's front-end. This binary is linked to particular YASSi back-end which keep track of the symbolic execution and commands over a reasoning engine for decisions making. In the following, we are now describing the YASSi front-end as well as the YASSi symbolic execution core back-end.

3.1 YASSi's Frontend

In the architecture of YASSi, we are using the front-end in order to prepare the program we want to execute symbolically. As can be seen in Fig. 2, the first step is the compilation of C/C++ sources into LLVM's *Intermediate Representation* (IR) format [13]. LLVM's IR format is based on a load and store architecture and breaks down the complexity of C/C++ codes into a basic instruction set. In the second step, the generated IR code is getting instrumented using LLVM's optimization tool. To this end, we are going to alternate LLVM's IR code and add special function calls for each particular instruction which is needed for the eventual symbolic execution. Once the code instrumentation has been done, we are linking the instrumented code to our back end which resolves the inserted function calls. The result of YASSi's front-end is a binary which is ready for symbolic execution. After the execution of the binary has terminated, the front-end can access the database in order to analyze the results generated by the symbolic execution run.

[1] YASSi is available at http://github.com/gledr/YASSi.

3.2 YASSi's Backends

YASSi commands over multiple back-ends. However, in this work we are only introducing YASSi's symbolic execution core back-end. As introduced above, YASSi's front-end performs code instrumentation by inserting callback function calls which are getting resolved by the back-end. Instead of executing the LLVM instructions, we are calling YASSi's back-end which processes the information internally. To this end, YASSi is controlling the program execution based on Z3 which is getting used as reasoning engine in the back. Moreover, every-time the program branches, YASSi considers both execution paths and tries to find a valid solution for both paths. Therefore, YASSi creates clauses for each path-condition and forwards them to the reasoning engine. If the reasoning engine determines a valid solution, the branch is getting considered successfully and the execution continues. Same as for branches is getting used for assertions as well as other exceptions.

4 Application of YASSi

This section finally discusses some of the applications YASSi can be used for. Since the principle of symbolic execution allows it to target a wide variety of applications, we focus on those which we successfully applied with YASSi thus far.

- Assertion Checking:
  YASSi is capable to check certain properties during execution. To this end, YASSi is capable to check assertion and to work with non-deterministic variables. Therefore, YASSi tries to violate the assertion by invoking the SMT solver. YASSi is not only capable to check assertions, it is also capable to check for traps like division by zero or out of boundary index accessing of data-structures like arrays.
- Reachability Analysis:
  Another application case for YASSi which has been applied successfully is reachability analysis. To this end, we have applied YASSi in order to check, that certain parts of the code are unreachable which we could use to exclude certain functional safety issues.
- Stimuli Generation:
  Next to exception checking and reachability analysis, we were able to use YASSi for stimuli generation. To this end, we were able to generate stimuli with a coverage for abstract descriptions of digital circuits. Our approach allowed it to extract these stimuli into a database and directly to apply them for checking the model. As already mentioned above, YASSi commands over multiple back-ends. Moreover, YASSi commands over a so-called replay back-end which directly can be applied to check the reached branch and line coverage [14] for a particular set of generated stimuli.

5 Conclusion

In this work, we considered symbolic simulation as program analysis technique for C/C++ codes. This is motivated by the ever growing complexity of modern systems build up using hardware as well as software components designed using programming languages like C/C++. In order to address this, we introduced the tool YASSi as our approach for a state-of-the-art symbolic simulator for academic research. We have build YASSi in a modular fashion, which allows it directly to extend the tool for eventual later applications. YASSi has been build on top of the LLVM toolkit for compiler construction together with modern reasoning engines like Z3. YASSi is further under heavy development, and we keep adding more applications. The next milestone for the tool will be the support of floating-point variable based on the SMT2 bitvector floating-point type. YASSi is available at http://github.com/gledr/YASSi.

Acknowledgments. This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria as well as by BMK, BMDW, and the State of Upper Austria in the frame of the COMET Programme managed by FFG.

References

1. Midtgaard, J.: Control-flow analysis of functional programs. ACM Comput. Surv. CSUR **44**, 1–33 (2012)
2. Agrawal, H., Horgan, J.R.: Dynamic program slicing. SIGPLAN Not. **25**, 246–256 (1990)
3. King, J.C.: Symbolic execution and program testing. Commun. ACM (1976)
4. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the Conference on Operating Systems Design and Implementation, San Diego, USA (2008)
5. Baranová, Z., et al.: Model checking of C and C++ with DIVINE 4. In: Automated Technology for Verification and Analysis, Pune, India (2017)
6. Gonzalez-de-Aledo, P., Sanchez, P.: FramewORk for embedded system verification. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 429–431. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_36
7. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
8. Herdt, V., Le, H.M., Große, D., Drechsler, R.: Verifying SystemC using intermediate verification language and stateful symbolic simulation. IEEE Trans. CAD **38**, 1359–1372 (2019)
9. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization, San Jose, USA (2004)
10. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

11. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0 system description. J. Satisfiability Boolean Model. Comput. **9**, 53–58 (2015)
12. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard - Version 2.0. Technical report, New York University (2010)
13. The LLVM Team: clang: a C language family frontend for LLVM. Accessed 23 Mar 2020
14. Martin, G., Bailey, B., Piziali, A.: ESL Design and Verification: A Prescription for Electronic System Level Methodology. Morgan Kaufmann Publishers Inc., San Francisco (2007)