Towards Benchmark Optimization by Automated Equivalence Detection

Daniel Schemmel, René Glebke, Mirko Stoffers and Klaus Wehrle
Communication and Distributed Systems
RWTH Aachen University, Germany
Email: firstname.lastname@comsys.rwth-aachen.de

***Abstract*—Especially in the case of Cyber-Physical Systems (CPSs), testbed validations and benchmarks, while necessary, incur significant setup and operation costs. Optimized benchmark sets reduce the number of tests that need to be performed, which ultimately reduces costs. In this paper, we propose a new methodology to provide automated assistance for optimizing existing benchmarks or for creating new ones from scratch.**

The proposed methodology is based on complete Symbolic Execution of a single control loop iteration, optionally expanded with a Nondeterministic Finite Automaton (NFA) model that represents possible changes in the environment or the system in between control loop iterations. This enables us to compute a *stress number* that represents the computational burden put upon the controller by a respective benchmark. By iteratively searching for benchmarks with high stress numbers and automatically detecting and pruning benchmarks that induce the same path through the controller code, we can ultimately create a minimal set of relevant benchmarks for a CPS.

I. Introduction

Testbeds, while very useful to perform meaningful benchmarks in, are expensive to not only set up but also maintain—for Cyber-Physical Systems (CPSs) especially so. Benchmarks are often based on expensive prototypes and under conditions resulting in much more than the usual wear and tear. Expendable supplies for not (yet) batch-produced tools are major cost drivers. In addition to tools and materials, personnel is required to install, supervise, reset, reconfigure, and reposition the physical components of the system. Concluding, each and every benchmark can cost considerable amounts of money. An optimized benchmark set containing fewer cases therefore directly equates to not only time, but also cost savings.

However, such an optimization must not result in missing important benchmarks. The key challenge in creating an optimized benchmark set therefore is to choose it as small as possible without sacrificing precision. We propose a methodology that will assist the developer by reasoning about *equivalence classes* of benchmarks. Performing just a single benchmark of an equivalence class enables reasoning about *all* other benchmarks in the same class without performing additional benchmarks. For example, choosing an extremum of parameters from such an equivalence class provides an upper or lower bound on the possible responses. Choosing the other extremum in a second benchmark fully confines the response function to a bounded interval. If additional knowledge such as linearity is available, the entire function can be derived for that interval. By additionally estimating the input-dependent complexity of the computations that the controller needs to perform (which we refer to as *stress numbers*), we can identify benchmarks that cause certain computation times.

To derive the equivalence classes and stress numbers reasonably, it is of utmost importance to identify a reliable source of information. In particular, we strictly avoid additional domain knowledge provided by users as an ultimate requirement, which would not only pose additional manual effort but also introduce a potential source of error. Instead, we use the real code that is used to operate the control loops commonly found in systems that interact with the physical world. We utilize *exhaustive Symbolic Execution* – a complete, dynamic code analysis method – to fully capture the behavior of one control loop iteration. Additionally, we retrieve expressions (so called path constraints) for the different paths that the execution of the program may follow, which enables us to derive input equivalence classes. A Nondeterministic Finite Automaton (NFA) model of the behavior of the environment or the controlled system may be additionally provided in order to reduce the search space during symbolic execution. This model need by no means be complete nor accurate; it should rather exclude strictly impossible behavior to lower the number of possible equivalence classes.

Based on the conducted analyses, we can issue the minimal benchmark set necessary to rigorously evaluate a given CPS. Our proposed methodology can be employed in very different scenarios, such as:

- validation and correctness testing of control programs,
- derivation of benchmarks of certain complexity (e.g., best-case or worst-case), both for the controller itself and, using model-based testing, of the whole system,
- derivation of equivalence classes of behavior of the controller or the system,
- optimization of existing benchmark suites by showing up opportunities for reduction of the benchmark set.

A. Structure

Section II details related work in this area and Section III gives some background on Symbolic Execution. Section IV presents our approach in more detail and Section V elaborates on the main challenges that our approach faces. We conclude in Section VI.

II. Related Work

The field of CPSs is extremely diversified, ranging from small, passive sensor networks to large-scale industrial appliances. Yet, common challenges arise based on the fact that the systems directly interact with the physical environment. Input from sensors can constantly change and may not always provide predictable or even useful values (e.g., in cases of electromagnetic disturbances). Likewise, output signals sent towards actuators such as motors or heating elements must not necessarily result in the intended behavior to actually be observable, e.g., in cases of mechanical failures. Another particularity of CPSs is that they do not necessarily always start off with a clean slate; battery-backed subsystems may cause non-default inputs to be supplied to the control programs even when they are restarted [1]. Furthermore, Programmable Logic Controllers (PLCs), used to control assembly line machinery and industrial robots, employ a so-called *cyclic scanning* mode, which means that programs or parts of them need to be repeatedly executed within bound, short periods of time in order to account for changes in the input signals. Failure to properly account for these idiosyncrasies in the control programs may lead to physical damage of equipment or, in the extreme, even loss of human lives. Testing the compliance of the employed programs to the expected behavior or their level of adherence to standards is hence one of the paramount objectives when evaluating CPSs. Due to their particular importance in industrial settings and hence the severeness of consequences in cases of faults, we in the following mainly concentrate on techniques applied to PLCs. However, the ideas presented in the following are generally similarly applicable also to non-PLC systems such as sensor nodes and PCs.

One early approach towards *statically* analyzing compliance in CPSs was presented by Moon in 1994 [2]. Control programs written in a precursor of the IEC 61131-3 languages nowadays used for programming PLCs [3] are rewritten as inputs to a model checker. The desired behavior of the programs can be expressed in the form of temporal logic propositions over the relations between input signals (sensor readings and internal state variables) and outputs (target operation values for controlled equipment and new internal states). The model checker then formally either proves adherence to the specifications or provides *counterexamples*, i.e., variable combinations that contradict the expressed desired behavior and hence point to paths in the program execution that require modification in order to ensure proper operation of the CPS. This form of analysis has since seen further development in the CPS domain, including adoption to both the peculiarities of microcontrollers [4] and the inclusion of multiple IEC 61131-3 languages into a single tool [5]. The inherent soundness and completeness of this approach (a model checker will always provide a correct assessment regarding the desired behavior of the program), however, comes at the price of potentially prohibitively high time and memory requirements when applied to real-world programs [6].

When complete coverage of all possible combinations of variable states is not an option, the search space can also be sampled. In [7], a variation of the meta-heuristic hill-climbing algorithm is used to enhance the stochastic probing in a search-based testing approach applied to parts of a train control software. More systematic approaches can automatically generate test cases, for example from line- or branch coverage metrics [8]. The resulting variable combinations necessary to reach a certain line or branch may then be used in unit or regression tests.

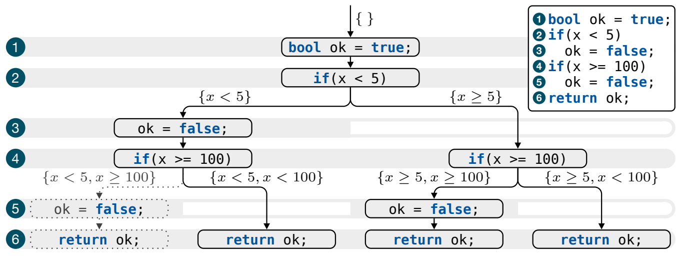
```
bool ok = true;
if(x < 5)
  ok = false;
if(x >= 100)
  ok = false;
return ok;
```

Figure 1. Symbolic Execution of a simple interval test. The variable x is assigned a symbolic value, which causes three paths through the code to be reachable. One of the four depicted cases is not reachable (denoted by the dotted lines), as x cannot be both, smaller than 5 and larger or equal to 100.

Static analysis methods may also be employed to estimate Worst-Case Execution Times (WCETs) for control programs and their respective cycles (e.g., [9]–[11]), although the general problem is undecidable for arbitrary control flows (including loops) [12].

In contrast to the approaches presented so far, *dynamic* analysis techniques do not analyze the code of programs to be evaluated but directly execute the programs themselves and reason from their run-time behavior. For example, in [13], Biere et al. use Symbolic Execution to increase the precision of a WCET analysis.

While these approaches have only recently gained attention also in the CPS domain, few promising application fields have already been shown. Kormann and Vogel-Heuser for example propose coupling PLC programs with simulated physical systems, which then allows the simulation of hardware faults and the generation of software test cases for such situations without having to manipulate the programs or an actual physical system [14].

Building on their previous developments in [5], the authors of [6] propose a methodology that is perhaps most similar to our approach. Using a mixture of Symbolic Execution (cf. Section III) and a concretization of selected variables during run time, their *concolic* execution engine allows the automatic generation of test cases for PLC programs during execution. The concretization of variables yields under-approximations which may increase coverage at the cost of completeness. Thereby, they showed an increase in branch coverage for a vendor-specific implementation of a library of standardized safety functions, while at the same time reducing test case generation times significantly.

III. Background on Symbolic Execution

Symbolic Execution has become a popular dynamic analysis technique whose primary domain is automated test case generation and bug detection [15]–[21]. The primary intent behind Symbolic Execution is to improve upon exhaustive testing by symbolically constraining inputs instead of iterating over all possible values.

Figure 1 illustrates the principle of Symbolic Execution. It shows the execution tree that results from executing a C snippet testing `x` for inclusion in the range $[5, 100)$, where `x` is set to an unconstrained symbolic value, i.e., `x` can take any value. The first constraint on `x` is added in line two: the execution forks into two different paths as `x < 5` could evaluate to `true` as well as `false`. A similar situation follows in line three, except that `x >= 100` cannot become `true` if `x < 5` was previously `true`. If no model exists satisfying all constraints along the path, that branch is not explored as it cannot occur during (ordinary) execution. This step is decided with a *Satisfiability Modulo Theories (SMT)* solver that answers queries in the theories of (quantifier free) bit-vectors, floating point numbers[1], and arrays thereof. Thus, Symbolic Execution can derive this execution tree efficiently, i.e., without brute-forcing every possible `x`.

The major limitation of Symbolic Execution is scalability due to two different root causes:

1) SMT solving is $\mathcal{NP}$-complete for the required theories[2]. While solvers exist that can answer most queries occurring in the wild in reasonable time, counter-examples must exist unless $\mathcal{P} = \mathcal{NP}$. Practically speaking, this occurs most heavily in cryptographic computations and is usually circumvented by aborting queries exceeding a preset amount of computation time, thereby causing the analysis to become incomplete.
2) As execution needs to fork every time it might possibly take different paths, the number of paths that the Symbolic Execution engine needs to consider becomes very large for many real-world programs. This is referred to as the *path explosion* problem.

Both problems are significantly mitigated by the target domain: single iterations of control loops usually neither construct complex queries, nor is their control flow of significant complexity.

IV. Benchmark Optimization

In this section we propose our methodology for deriving an optimized benchmark set for CPSs. By executing only a single iteration of a control loop we can already reason about properties of the implementation of the control loop like correctness or computational costs of that iteration. To capture the full behavior, however, multiple iterations of the control loop need to be executed.

[1]Floating point support for Symbolic Execution is still evolving and currently limited in practice [22].

[2]Bitblasting [23] provides a polynomial-time reduction to the Boolean Satisfiability (SAT) problem for the used theories.

The most simple example that our methodology can be applied to is a control system consisting of a sensor, a controller, and an actuator. When we execute the control loop once we reason about any possible sensor input. For multiple iterations we need to assume some interaction between the actuator and the sensor, e.g., an actuator that turns on a heater such that a temperature sensor eventually provides higher values. If no or only very rudimentary models of the environment or the controlled system are available, possible inputs by the sensor and reactions by the actuator to control signals (e.g., actual state *vs.* target state) may be nearly unrestricted from the Symbolic Execution engine's point of view, i.e., the inputs to the control loop are considered wholly uncorrelated. Rudimentary models hence allow us to provide a correct and complete set of benchmarks. More precise models can, in turn, be used to narrow down the set of possible input/output values in the system, e.g., by specifying the range of values sent by the temperature sensor.

In the following, we discuss in more detail how our approach executes a single control loop iteration and how it operates with multiple iterations. After that, we sketch how the results are deduced.

A. Single-Iteration Mode

Deriving equivalence classes of benchmarks can be easily achieved by virtue of the origins of Symbolic Execution. Errors are detected owing to automatically performed checks, such as for division by zero, and any additional assertions, which allows for semantic testing. For small and simple programs, such as single iterations of a control loop, it is rather likely that the Symbolic Execution will complete without pruning any paths. In that case, it is guaranteed that all bugs that could be detected have been detected.

During the Symbolic Execution, we trace the instructions along each path. For the kind of hardware platforms used in CPSs, this often enables a fairly precise measure of the execution time with little additional post-processing, by just adding up the costs of the individual instructions. Using a more complicated analysis, it is also possible to provide accurate timing predictions for much more sophisticated platforms [24].

To optimize a benchmark suite, or to generate an optimized benchmark suite from scratch, the resulting test cases are grouped by their execution time to discover how to put the most (or the least) stress on the control loop. If no further information exists, inputs from the most computation intensive category are preferred. This is, however, less than satisfactory, as the full behavior of a CPS is hardly captured by a single iteration of its control loop.

B. Multiple-Iterations Mode

To escape this dilemma, we need to identify possible variations in the input after each state. To this end, we utilize a model that describes possible transitions in the environment. As detailed above such a model can be of arbitrary degree of abstraction from completely unrestricted to precisely specified. Multiple models might also be useful to represent different operating environments, hardware faults (e.g., a faulty temperature sensor might read 7 K as the current room temperature) or equipment configurations.

Both the model and the Symbolic Execution Tree can now be described as an NFA. The model describing the environment could, for example, encode temperature change from one state to another by having multiple transitions to new possible temperatures. This is even simpler for the Symbolic Execution Tree, which already is a graph with points from which multiple different decisions can be made. By joining these two NFAs at their decision points, we can generate a combined NFA[3]

On the combined NFA, we perform an iterative search for a sequence of loop iterations that have a high chance of requiring significant computational effort. The expected computational effort is then represented as a *stress number*. Therefore, the search effectively attempts to find benchmarks with a maximum stress number.

As the control loop will usually not terminate, but rather continue running, the analysis must stop at some point without considering the entire (infinite) sequence of iterations. While it is of course beneficial to analyze as long a sequence as possible, the time required for the analysis must also be taken into account. For a given analysis time, sequence length and variety of sequences are in conflict, and must be balanced for an optimal result.

C. *Deducing Results*

Our approach provides two major benefits over naïve Symbolic Execution:

1) By categorizing the leaves of the Symbolic Execution tree by the costs of the instructions consumed (i.e., their *stress*), we can effectively perform *state-merging* to derive temporally equivalent benchmarks, i.e., those of similar computational complexity. State-merging is commonly used in Symbolic Execution to reduce the path explosion.
2) As the individual loop iteration is analyzed as exactly as possible before the iterative search begins, the danger of choosing sub-optimal branches for exploration is significantly reduced. Any distinctive paths are found before the search, and can be prioritized during the search.

Using the equivalence classes provided by the Symbolic Execution, we are now able to suggest optimized benchmark suites by including all those benchmarks into the suite whose behavior is not equivalent to other benchmarks in our model. Depending on the use case we can focus on behavioral or temporal equivalence. For the temporal equivalence we can employ the stress numbers. Note that behaviorally equivalent benchmarks also have the same stress number by construction.

If the optimization of an existing benchmark suite is requested instead of the creation of a wholly new suite, we can guide the search for sequences of loop iterations by means of the benchmarks provided by the user. We determine behavioral and temporal equivalence of the input benchmarks. By constraining the inputs to the user provided data we can increase the performance of the Symbolic Execution as many unreachable paths can be avoided. We issue the benchmarks we determined as equivalent and suggest removing all but one of any equivalence class. We shall note that, especially for hand-written benchmarks, there may however be intent inherent in the old benchmark suite that is not captured by the provided model, or otherwise invisible to our analysis (e.g., repeated movements of actuators over mechanical failure points). For this reason, the developer should carefully choose the benchmarks to exclude.

[3]The combined NFA may be very large, but does not need to be explicitly represented in memory.

V. Challenges

We acknowledge that complete Symbolic Execution is utterly infeasible for any significant program. With this work we attempt to provide an alternative to premature concretization by completely and rigorously analyzing a smaller piece of code, and then creatively stitching it together with a model of how the physical portion of the CPS acts. While this already tackles the problem from multiple angles by enabling selective state space analysis and implicit state merging, the computational complexity of the analysis warrants further research and consideration.

A related challenge is how the model can be built in a manner that is at the same time useful, automated and easy to analyze. A base model which allows nearly unrestricted transitions, for example, is extremely easy to generate automatically, but neither easy to analyze nor very close to the physical reality.

As the state space can easily become excessively large, the search strategy suggested in Section IV-B is of utmost import for the performance of the whole approach. To enhance precision, a tradeoff between width and depth of the search has to be made – neither narrow and deep, nor wide but shallow searches promise to cover the possible state space to a large degree. The strategies that already exist for Symbolic Execution itself might serve as useful building blocks here, but will need to be augmented to consider the combined state space.

VI. Summary

In this paper, we presented work towards a new methodology to optimize and generate benchmark suites for CPSs. We described how Symbolic Execution can be utilized to analyze a single iteration of a control loop with high precision, and how the results of the Symbolic Execution can then be used as input for an iterative search for benchmarks with a high stress number. Finally, the categories established by the Symbolic Execution can be used to detect benchmarks with equivalent behavior (w.r.t. the provided model) and suggest them for pruning.

It is our hope that this approach to benchmark optimization by automated equivalence detection paves the way to more cheaper and more more meaningful benchmarks in the realm of Cyber-Physical Systems.

Acknowledgements

This research is supported by the European Research Council (ERC) under the European Union's Horizon 2020 Research and Innovation Programme (grant agreement №. 647295 (SYMBIOSYS)) and by the German Science Foundation (DFG) within the Priority Programme 1914 ("Cyber-Physical Networking").

References

[1] S. Hauck-Stattelmann, S. Biallas, B. Schlich, S. Kowalewski, and R. Jetley, "Analyzing the Restart Behavior of Industrial Control Applications," in *FM 2015: Formal Methods*, N. Bjørner and F. de Boer, Eds. Cham: Springer International Publishing, 2015, pp. 585–588.

[2] I. Moon, "Modeling Programmable Logic Controllers for Logic Verification," *IEEE Control Systems Magazine*, vol. 14, no. 2, pp. 53–59, April 1994.

[3] *IEC 61131-3:2013 – Programmable Controllers - Part 3: Programming Languages*, International Electrotechnical Commission Std., 02 2013. [Online]. Available: https://webstore.iec.ch/publication/4552

[4] B. Schlich and S. Kowalewski, "Model checking C source code for embedded systems," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 3, pp. 187–202, Jul 2009.

[5] S. Biallas, J. Brauer, and S. Kowalewski, "Arcade.PLC: A Verification Platform for Programmable Logic Controllers," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*, Sep. 2012, pp. 338–341.

[6] D. Bohlender, H. Simon, N. Friedrich, S. Kowalewski, and S. Hauck-Stattelmann, "Concolic Test Generation for PLC Programs using Coverage Metrics," in *2016 13th International Workshop on Discrete Event Systems (WODES'16)*, May 2016, pp. 432–437.

[7] K. Doganay, M. Bohlin, and O. Sellin, "Search based testing of embedded systems implemented in iec 61131-3: An industrial case study," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, March 2013, pp. 425–432.

[8] H. Simon, N. Friedrich, S. Biallas, S. Hauck-Stattelmann, B. Schlich, and S. Kowalewski, "Automatic Test Case Generation for PLC programs using Coverage Metrics," in *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA'15)*, Sept 2015, pp. 1–4.

[9] K. Koo and W. H. Kwon, "Predicting Execution Time of Relay Ladder Logic for Programmable Logic Controllers," in *Proceedings of the 1996 IEEE Conference on Emerging Technologies and Factory Automation, 1996*, Nov 1996, pp. 670–676.

[10] J. Henry, M. Asavoae, D. Monniaux, and C. Maïza, "How to Compute Worst-case Execution Time by Optimization Modulo Theory and a Clever Encoding of Program Semantics," in *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, Jun 2014, pp. 43–52.

[11] J. Wan, A. Canedo, and M. Al Faruque, "Model-based Design of Time-triggered Real-time Embedded Systems for Digital Manufacturing," in *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, April 2015, pp. 295–296.

[12] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-case Execution-time Problem – Overview of Methods and Survey of Tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[13] A. Biere, J. Knoop, L. Kovács, and J. Zwirchmayr, "The Auspicious Couple: Symbolic Execution and WCET Analysis," in *13th International Workshop on Worst-Case Execution Time Analysis*, ser. OpenAccess Series in Informatics (OASIcs), C. Maiza, Ed., vol. 30. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 53–63. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2013/4122

[14] B. Kormann and B. Vogel-Heuser, "Automated test case generation approach for plc control software exception handling using fault injection," in *IECON 2011 - 37th Annual Conference of the IEEE Industrial Electronics Society*, Nov 2011, pp. 365–372.

[15] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Dec. 2008.

[16] C. Cadar and K. Sen, "Symbolic Execution for Software Testing: Three Decades Later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[17] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'05)*, vol. 40, no. 6, Jun. 2005, pp. 213–223.

[18] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'05)*, vol. 30, no. 5, Jun. 2005, pp. 263–272.

[19] N. Tillmann and J. De Halleux, "Pex–White Box Test Generation for .NET," in *Proceedings of the 2nd International Conference on Tests and Proofs (TAP'08)*, Apr. 2008, pp. 134–153.

[20] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," *ACM Transactions on Information and System Security*, vol. 12, no. 2, p. 10, 2008.

[21] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, Mar. 2011.

[22] D. Liew, D. Schemmel, C. Cadar, A. F. Donaldson, R. Zähl, and K. Wehrle, "Floating-Point Symbolic Execution: A Case Study in N-Version Programming," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*, Oct.-Nov. 2017, pp. 601–612.

[23] V. Ganesh and D. L. Dill, "A Decision Procedure for Bit-Vectors and Arrays," in *Proceedings of the 19th International Conference on Computer-Aided Verification (CAV'07)*, Jul. 2007, pp. 519–531.

[24] F. Rath, J. Krude, J. Rüth, D. Schemmel, O. Hohlfeld, J. Á. Bitsch, and K. Wehrle, "Symperf: Predicting network function performance," in *Proceedings of the SIGCOMM Posters and Demos*. ACM, 2017, pp. 34–36.