# Towards Language Independent (Dynamic) Symbolic Execution

Stefan Klikovits*†, Manuel Gonzalez-Berges†and Didier Buchs*
*University of Geneva, Software Modeling and Verification Group
Geneva, Switzerland, Email: {firstname.lastname}@unige.ch
†European Organization for Nuclear Research (CERN), Beams Department
Geneva, Switzerland, Email: {stklikov,mgonzale}@cern.ch

***Abstract*—Symbolic execution is well-known for its capability to produce high-coverage test suites for software source code. So far, most tools are created to support a specific language. This paper elaborates on performing language independent symbolic execution and three ways to achieve it. We describe the use of one approach to perform dynamic symbolic execution using translation of a proprietary language and show the results of the tool execution on a real-life codebase.**

## I. Introduction and Background

Software testing is a popular technique amongst software developers. The recent advance of computing power increased the usability of dynamic symbolic execution (DSE) [1] to a point where it is now even included in commercial tools [2]. DSE implementations are mostly designed to support one specific language (e.g. Java, C), or their underlying low-level language (e.g. x86 byte code). While the advantages of having such tools are indisputable, software developers working with proprietary or less popular languages often cannot benefit. Our research focuses on proposing a solution for languages that do not have dedicated (D)SE tools. The drive for our explorations comes from a practical application at CERN, which uses a proprietary scripting language to control and monitor large installations.

### A. Motivation

The European Organization for Nuclear Research (CERN) uses a proprietary software called Simatic WinCC Open Architecture (WinCC OA) to control its particle accelerators and installations (e.g. the electrical power grid). To support the design of such systems the BE-ICS group maintains and develops the *Joint COntrols Project* (JCOP), as set of guidelines and software components to streamline developments. JCOP is based upon the WinCC OA SCADA platform which is scriptable via *Control* (CTRL), a proprietary programming language inspired by ANSI C.

Until very recently CTRL code did not have a dedicated unit test framework. The development of such a testing library filled this need, but after over a decade of development the CTRL code base of JCOP sizes some 500,000 lines of code (LOC). This code has to be manually (re-)tested during the frequent changes of operating system versions, patching or for framework releases. Over the decades-long lifetime of CERN's installations, this testing is repeatedly (often annually) required and involves a major overhead. To overcome this issue, the use of automatic test case generation (ATCG) was decided.

### B. Symbolic Execution

Symbolic execution (SE) is a whitebox ATCG methodology that analyses source code's behaviour. The approach works by constructing the code's execution tree. The program's input parameters are replaced with symbolic variables, their interactions recorded as operations thereon. The conjunction of all symbolic constraints of an execution tree branch, provides a *path constraint*. Finding a solution for it, e.g. using a satisfiability modulo theories (SMT) solver, provides inputs that will lead the program to follow this branch. SE experiences shortcomings when it comes to uninstrumentable libraries, impossible/infeasible constraints (modulo, hashes) or too complex constraints. To overcome these limitations, dynamic symbolic execution has been introduced. DSE switches to concrete value execution in cases where SE reaches its limits. We refer the reader to [1] for an overview of SE and DSE.

This paper is organised as follows: Sect. II introduces language independent SE. Sect. III describes an implementation to bring SE to CTRL. Sect. IV presents results of the implemented solution. Sect. V explores related work and Sect. VI concludes.

## II. Language Independent Symbolic Execution

Most SE and DSE tools operate on low-level (LL) representations (LLVM, x86, .NET IL) of popular high-level languages (Java, C, C#). The reason is that LL representations are simpler, leading to fewer operations types that have to be treated by the symbolic execution engine. This means that only source code which compiles into these LL representations is supported. Other languages miss out on the functionality. Another disadvantage of using a specific low-level representation are implicit assumptions on data types. Only data types supported by the operating language are supported. This voids the possibility to use own or modified data types and languages.

For this reason we propose the use of language independent symbolic execution. Apart from the development of a dedicated (D)SE engine for a programming language, there exist three possibilities for language independent symbolic execution.

First, a SE tool that operates on a generic abstract syntax tree (AST). SE already performs operations on symbolic variables, which are either subtypes of or proxies for real data types. Hence, a tool operating on a model of the source code, i.e. an AST, could perform truly language independent SE. This approach comes with two difficulties: a) specifying an AST generic enough for all targeted languages and their particular features and differences, b) deterministically translating a language parser's representation into the generic AST.

The second approach is a symbolic execution engine based on callback functions. The source code under test is parsed using a modified version of its native parser. The parser triggers actions in the symbolic execution tool when it comes across the variable interactions. The advantage of this approach is that existing parsers for the source language can be adapted to issue the required calls, leading to an easily implemented SE framework. The caveat is a large number of messages being issued by the parser, potentially leading to low performance.

Lastly, a translation into the operating language of an existing tool. This solution, while similar to the first, is distinct in that usually the target language and the tool cannot be modified. The target language has to offer similar concepts as the source, otherwise only a subset of the sources can be supported without major overhead. It is also necessary to re-implement any standard-library or built-in functionality that the source language relies on. A difficulty arises when the target language has different semantics or data types, requiring trade-offs and leading to unsupported features.

## A. *Semantics*

The semantics of a language play an important role. In all three approaches, it is necessary to precisely capture the meaning of each statement. Failure to do so would lead to divergences and hence wrongly produced constraints and input data. As an example one can look at differences of zero- and one-based list/array indices. This minor change in the language semantics can lead to severe errors such as *out of bounds*-errors, changed loop behaviour, and similar.

It is also important that the source language's data types are supported by the symbolic execution tool. Some existing (D)SE tools such as Microsoft Pex [3] support the use of own data types (*classes*). However, these class data types are treated differently from native types (they are `nullable`) and sometimes replaced by stubs. Additionally, existing tools do not allow for the modification of their native data types, leading to unsupported features. An example would be that some languages (e.g. JavaScript) support native implicit casting between `int` and `string` values. This behaviour cannot be reproduced in C# and is hence not supported by Pex.

A solution would be to define all necessary *sorts* (data types) for the underlying SMT solvers (e.g. Z3, CVC4). Current tools adjust the SMT solver configuration based on their operation language. An SE framework that supports the specification and use of user-defined sorts can provide a solution to these problems. Alternatively, it is possible to abandon SMT solvers and explore other ways of solving constraints. Term rewriting systems are well known for their capability to express semantics and offer constraint solving capabilities. This solution, while offering less performance, provides more flexibility and support for native data types.

Using term rewriting systems, it would also be possible to perform SE/DSE on generic models, opening the door to many different kinds of analyses.

## III. Implementation

Since no tool exists that natively supports CTRL code, CERN is faced with the choice between three solutions: 1) develop an ATCG tool specifically for CTRL; 2) develop a language independent ATCG tool; 3) translate the CTRL code into the operating language of an existing ATCG tool.

Given CERN's practical need, the last option was chosen. A short evaluation led to Microsoft Research's Pex tool [3], a program that performs test case generation through DSE. In order to use Pex for CTRL code, we developed a tool called *Iterative TEst Case* system (ITEC). ITEC translates CTRL to C#, in order to execute Pex and obtain input values for automatically generated CTRL test cases. This tool helped to build up regression tests that can then be reused on the evolving system to ensure its quality. ITEC works on the assumption that the current system reached a stable state after 13 years of continuous development and use.

## A. *Architecture*

CERN's focus mainly lies in the test case creation for CTRL code. Hence, in a first step, the re-use of existing software is preferred over the creation a generic ATCG framework. ITEC relies heavily on an existing CTRL parser, which has been implemented in Xtext [4] during a previous project at CERN. ITEC's workflow is separated into six consecutive steps: 1) code under test (CUT) selection[1] 2) semi-purification 3) C# translation 4) test input generation 5) test case creation 6) test case execution.

ck] (start) – (one);

In the initial task, the CUT selection, the user or the tool automatically (in bulk execution mode) chooses which code is to be tested. ITEC analyses the sources for dependencies (global variables, database values, subroutines) which are to be replaced, if necessary.

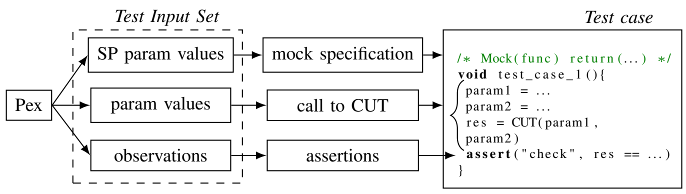
Listing 1. SP doubles: Before

```
get15OrMore(x){
  a = dependency(x)
  return a > 15 ? a : 15
}
dependency(x){
  r = randomValue() + 5
  return r * x
}
```

Listing 2. SP doubles: After

```
get15OrMore(x, b){
  a = dependency(x, b)
  return a > 15 ? a : 15
}
dependency(x, b){
  observe(x)
  return b
}
```

[1]we use to *code* under test instead of *system* under test, as we focus on individual functions or code segments rather than systems



```
/* Mock(func) return(...) */
void test_case_1(){
 param1 = ...
 param2 = ...
 res = CUT(param1,
 param2)
 assert("check", res == ...)
}
```

Fig. 1. Test case generation from Pex output

Following the CUT identification, a process called *semi-purification* (SP) [5] is applied. Semi-purification replaces a CUT's dependencies with additional input parameters. SP is required as CTRL, similar to most other procedural languages, does not support mocking. The result of SP is a modified version of the CUT, where the function is only dependent on its input parameters. The benefit is that it is possible to use any form of ATCG (random testing, combinatorial approaches), independent of them being white- or black-box techniques. Replacement of subroutines with *semi-purification doubles* (an adapted form of test stubs [6]) allows for the specification of additional observation points. The observations can later be used for the definition of assertions in test cases. Listing 1 shows a short program that returns a random value $\geq 15$. During the SP process the dependency is replaced with a SP double of same name. The resulting code is displayed in Listing 2. The additional parameter `b` was added to replace the return value in the SP double and simulate the CUT behaviour. Additionally an observation point was inserted[2].

ITEC uses Microsoft's Pex tool to generate test input. As Pex operates on the .NET Intermediate Language, the semi-purified CTRL code has to be translated to a .NET language. C# was chosen since it is syntactically similar to CTRL.

After the translation, the CUT is added to other artefacts and compiled into a `.dll` file. The required resources are:

- **PUT:** Parameterized unit tests (PUTs) are entry points for Pex' exploration. They also specify observation points and expectations towards parameter values.
- **Pex factories:** Factories are manually created, annotated methods that serve as blueprints for data types. They help Pex generate input values. ITEC uses factories to teach Pex how to generate certain CTRL data types.
- **Data types:** Many CTRL data types are not natively present in C# , e.g. `time`, `anytype` or CTRL's dynamic list-types. They were re-implemented in C#.
- **Built-in functions:** CTRL's standard library provides functions for various actions (e.g. `string` operations). They had to be re-implemented to ensure the source code's compatibility.
- **Other:** Some additional libraries were developed to support the generation. One example is a *Serializer* for generated values.

Following the compilation, Pex is triggered on the resulting executable.

CTRL test cases are created from the results of Pex's exploration. Each set of values generated by Pex represents a test case for the semi-purified CUT. These values can be classified into three different categories:

1) **Parameter values:** The parameter values for the original CUT are used as arguments for the test's function call to the CUT.
2) **SP parameter values:** Additional parameters introduced by the SP process are used to specify test doubles for the test case execution. The values for semi-purified global variables are assigned before the call to the CUT.
3) **Observations:** Observations are transformed into assertion points. Additionally to `return` and reference parameter values there is the possibility to assert database writes and similar commands.

Figure 1 visualises the split of this information and shows how the values are used in the test cases.

The last step is the test case execution. To run the tests it is necessary to wrap them inside a construct of functions that will permit the observation of success or failure. Note that in our case, success means that the observations during the CTRL execution match the observations made by Pex. Additionally, test doubles are generated from their specifications and the code is modified to call the stubs instead of the original dependencies.

### B. *Challenges & Lessons Learned*

There are several challenges we faced during the creation of ITEC. One challenge is the translation of CTRL code to C#. Small changes in semantics have big impacts on the generated values. For example, list indices in C# are zero-based, while in CTRL they start at one. This means, that these lists had to be re-implemented, adding to constraint complexity, as Pex is optimised to native types. Additionally, C# is incapable of dealing with index or casting-expressions as reference parameters. These statements had to be extracted and placed before the function call, the resulting values written back after. There are numerous similar differences, leading to re-implementation of data types and functionality, while increasing the complexity of path constraints.

The validation of the translation is an important challenge. In [7] we give one proposal to solve this problem. The full list of challenges has been described in more detail in [8].

The lesson learned during implementation of the translator is that re-implementation of data types can be time-consuming and difficult. Often an increase in applicability and translation validity comes with a drop in performance. Finding a solution to these issues is one of the big challenges of our approach.

Despite the effort of implementing a translator, SP engine and TC generator, we believe that the implementation of a DSE tool for CTRL would be more costly.

## IV. Current Results

We executed the tool on 1521 functions in the CTRL framework. For 52.0 % (791) of the functions ITEC was able to execute Pex. The other 48.0 % failed due to one of the following reasons:

[2]The observation is not required here, but added to show the functionality

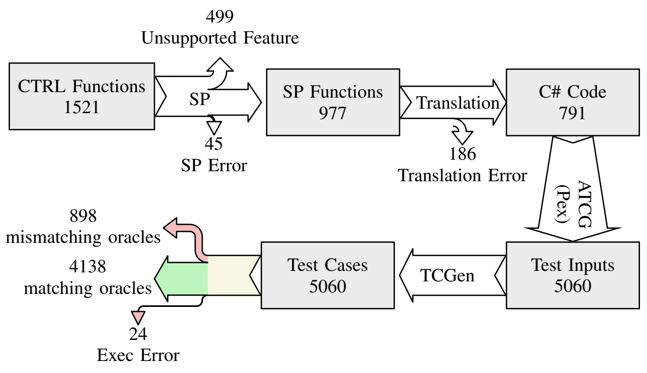


Fig. 2. Sankey diagram displaying the test generation and execution for the JCOP framework

– use of unsupported features, data types and functionality (32.8%): We explicitly excluded some functionality due to their complexity. Example: user interface interactions.
– errors due to unresolvable resource links (3.0%): The CERN-developed CTRL IDE occasionally has problems linking function invocations to the correct definitions.
– compilation errors of the translated C# (12.2%): The translator actively does not account for some language differences as they can be seen as "bad coding practice". ITEC serves as a motivation to avoid/alter these parts of the code base. Other concepts, such as the casting of native C# data types as explained above, cannot be translated.

For the 791 former functions, ITEC generated between $0$ and $104$ test cases (mean: $6.37$; median: $4$; first and third quartile: 2 and 7). Figure 2 displays the process and numbers.

The execution of these test cases lets us look at the line coverage data, as produced by WinCC OA's CTRL interpreter. The coverage shows a distribution as follows: $76\%$ of the functions are fully covered, $9.9\%$ have a coverage higher than $75\%$, for $7.2\%$ of the functions the coverage is above $50\%$, the rest has either a coverage under $50\%$ or no recorded data due to errors during the execution.

One result we observed during our analysis is that the coverage drops for long functions. While routines with less than 40 LOC are covered to a large extent (over 75 % line coverage), more than half of the functions longer than 40 LOC achieve less coverage. This suggests that it is harder to generate covering test suites for long functions, due to higher complexity in the path constraints. This theory is supported by the fact, that in general longer functions produce fewer test cases and that long routines with smaller test suites have bad coverage metrics.

We refer the reader to the technical document [8] for a more detailed breakdown of the test case generation and a thorough analysis of the test case execution results.

## V. Related Work

Test case generation through symbolic execution has been researched by others before us. Cseppentő et al. compared different SE tools in a benchmark [9], supporting our choice of Pex. [10] shows an approach to isolate units from their dependencies, similar to semi-purification. Bucur et al. [11] perform SE on interpreted languages by adapting the interpreter instead of writing a new engine. Bruni et al. show their concept of lightweight SE for Python, by using Python's operator overloading capabilities in [12]. The testing of database applications via mock objects was presented in [13].

## VI. Summary and Future Work

This paper shows our considerations for automated test case generation for CTRL, a proprietary, ANSI C-like language. We describe different approaches for language independent symbolic execution, before we explain our particular approach. Our solution involves the translation from CTRL to C#, which is supported by Microsoft's Pex dynamic symbolic execution tool. We explain the tool's general workflow and present lessons learned and results from the tool execution on our main codebase.

In future, we aim to extend the tool's applicability by lowering the number of unsupported features and executing it on additional parts of CERN's codebase. We also aim to enhance our analysis by comparing coverage to code complexity measures. Orthogonal to this effort our efforts will evaluate other (D)SE tools and approaches such as the ones described in the first part of this paper.

## References

[1] C. Cadar and K. Sen, "Symbolic Execution for Software Testing: Three Decades Later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.
[2] N. Tillmann, J. de Halleux, and T. Xie, "Transferring an Automated Test Generation Tool to Practice: From Pex to Fakes and Code Digger," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2014, pp. 385–396.
[3] Microsoft Research, "Pex, Automated White box Testing for .NET," http://research.microsoft.com/en-us/projects/pex/.
[4] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.
[5] S. Klikovits, D. P. Y. Lawrence, M. Gonzalez-Berges, and D. Buchs, "Considering Execution Environment Resilience: A White-Box Approach," in *Software Engineering for Resilient Systems*, vol. 9274. Springer LNCS, 2015, pp. 46–61.
[6] G. Meszaros, "Test Doubles," in *XUnit Test Patterns: Refactoring Test Code*. Addison Wesley, 2011.
[7] S. Klikovits, D. P. Y. Lawrence, M. Gonzalez-Berges, and D. Buchs, "Automated Test Case Generation for the CTRL Programming Language Using Pex: Lessons Learned," vol. 9823. Springer LNCS, 2016, pp. 117–132.
[8] S. Klikovits, P. Burkimsher, M. Gonzalez-Berges, and D. Buchs, "Automated Test Case Generation for CTRL," Report EDMS 1743711, 2016. [Online]. Available: https://edms.cern.ch/document/1743711
[9] L. Cseppentő and Z. Micskei, "Evaluating Symbolic Execution-based Test Tools," in *Proceedings of the IEEE Int. Conf. on Software Testing, Verification and Validation (ICST)*. IEEE, 2015.
[10] D. Honfi and Z. Micskei, "Generating unit isolation environment using symbolic execution," in *Proceedings of the 23rd PhD Mini-Symposium*. IEEE, 2016.
[11] S. Bucur, J. Kinder, and G. Candea, "Prototyping symbolic execution engines for interpreted languages," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2014, pp. 239–254.
[12] A. D. Bruni, T. Disney, and C. Flanagan, "A Peer Architecture for Lightweight Symbolic Execution," Tech. Rep., 2011. [Online]. Available: https://hoheinzollern.files.wordpress.com/2008/04/seer1.pdf
[13] K. Taneja, Y. Zhang, and T. Xie, "MODA: Automated test generation for database applications via mock objects," in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 289–292.