Automated Test Case Generation for the CTRL Programming Language Using Pex: Lessons Learned

Stefan Klikovits[1,2], David PY Lawrence[1], Manuel Gonzalez-Berges[2], and Didier Buchs[1]

[1] Université de Genève, Centre Universitaire d'Informatique, Carouge, Switzerland
`{stefan.klikovits,david.lawrence,didier.buchs}@unige.ch`
[2] CERN, European Organization for Nuclear Research, Geneva, Switzerland
`{stefan.klikovits,manuel.gonzalez}@cern.ch`

Abstract. Over the last decade code-based test case generation techniques such as combinatorial testing or dynamic symbolic execution have seen growing research popularity. Most algorithms and tool implementations are based on finding assignments for input parameter values in order to maximise the execution branch coverage. In this paper we first present ITEC, a tool for automated test case generation in CTRL, as well as initial results of test cases executions on one of CERN's SCADA frameworks. Our tool relies on Microsoft's Pex for its code exploration. For the purpose of using this existing test generation tool, we have to translate the proprietary CTRL code into C#, one of Pex's operating languages. Our main contribution lies in detailing a formal foundation for this step through source code decomposition and anonymization. We then propose a quality measure that is used to determine our confidence into the translation and the generated test cases.

Keywords: automated test case generation · resilience · software testing · translation validation · execution environment resilience

1 Introduction

At the Large Hadron Collider (LHC), its experiments and several other installations at CERN physicists and engineers employ a Supervisory Control And Data Acquisition (SCADA) system to mediate between operators and controllers/frontend computers which connect to the sensors and actuators. As such applications require hundreds of controllers to be configured, CERN has developed two frameworks on top of Siemens' *Simatic WinCC Open Architecture* (WinCC OA) [4] SCADA platform to facilitate their creation.

Due to lack of tool support for the WinCC OA's scripting language Control (CTRL) [5], it was so far not possible to write and execute unit tests in an efficient manner. Recently CERN started the development of such a unit testing framework to fill this need. However, after more than ten years of development, CERN is left with over 500,000 lines of CTRL code for which only a very small set of unit tests exist. Hence, the verification of the source code remains a mainly manual task leading to high testing costs in terms of manpower and slower release times.

This situation is especially tedious during the frequent changes in the WinCC OA execution environment. Before every introduction of a new operating system version, the installation of patches or the release of a new framework version the code base needs to be re-tested. Over the lifetime of the LHC, these environment changes happen repeatedly (often annually) and involve a major testing overhead. To overcome this issue, we decided to look into automatic test case generation. Since no tool exists that natively supports CTRL code we faced the choice between two solutions: 1. Develop an automatic test case generation tool specifically for CTRL; 2. Translate the CTRL code into the operating language of an existing automatic test case generation tool.

In order to reuse the theoretical knowledge gained over the years by established tools, we chose to translate CTRL code to C# in order to use Microsoft Research's Pex tool [11], a program that performs test case generation through dynamic symbolic execution [3]. To support this solution, we developed a tool called *Iterative TEst Case* system (ITEC). 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 use.

After finishing the first version of the tool, we applied it on a large part of the CTRL code base at CERN and had initial execution results. Based on this, we are now able to cast a critical eye over the quality of our approach.

The cornerstone of the chosen solution is the translation from CTRL to C# in order to use Pex. Evidently, if this translation is erroneous, the generated test cases would not be trustworthy. To ensure the quality of our approach, we must be able to validate the translation.

In this paper, we will shortly introduce how ITEC works and show first execution results. Based on these results and our experience, we will mainly focus on the validation of the translation from CTRL code to C#. Finally, we will discuss the translation and test quality metrics that take our translation validity study into consideration.

This paper is structured as follows: Section 2 discusses of the related work in terms of language translations and their verification, Section 3 presents an overview of ITEC and its individual components, Section 4 shows our execution results on one of the frameworks used in production, Section 5 presents our methodology to verify our language translation, addressing first general concepts before diving into the applied translation from CTRL to C#, Section 6 discusses a metric to express the overall quality of the tests based on coverage metrics, finally Section 7 gives an outlook on future work and concludes.

2 Related Work

As the main contribution of this paper lies in the verification of translation validity, we address in this section related works on this subject. Due to spacial constraints and the large amount of research in this field, we will however not be able to give an exhaustive description of all related works, but instead select the ones that are most closely related to our works.

The translation of CTRL code falls into the domain of source-to-source compilers or transpilers. The field of compiler verification has been extensively studied for compilations from high-level to low-level languages, including optimizing compilers using formal correctness proofs for the compiler software. An alternative approach introduced by [13] and extended by [12, 15] is called translation validation. These works are based on the idea to check the translation output for equivalence to the input, rather than the compiler itself.

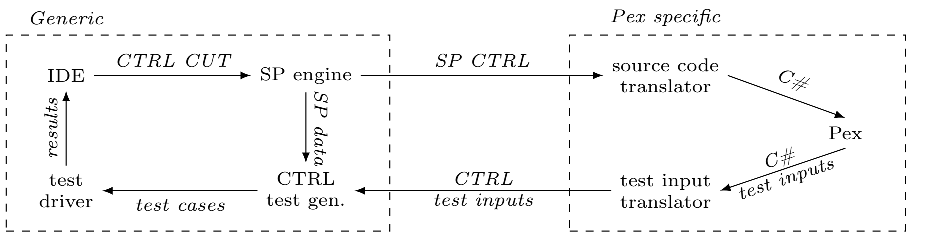
Since the translation is based on the abstract syntax tree (AST) that we obtain from parsing CTRL, the translation can also be seen as a model transformation. While many approaches to validation of model transformations have been proposed, we found the white box approach to validation of model transformation given in [9] of interest for our cases, as the authors argue for the use of large sets of generated test cases to perform validation. [6] introduces a test adequacy criterion for model transformations of the model driven architecture. They discuss partitioning and the choice of representative values as a means to gain trust in the model transformation program. Although their example is based on UML models, we see their approach as a general enough for our code translation purposes.

Another approach treats the translation as specific generation of code from an AST. The design, creation and also validation of code generators has been extensively discussed in [7], where a detailed overview on the topic of testing of code generators is given.

3 Tool Description

There exist different approaches to automatic test case generation and a lot of tools have been implemented for various programming languages. Unfortunately, none of these tools natively supports CTRL code with all its particularities, such as implicit castings, special data types and reference parameters. When facing the choice whether it was preferable to implement our own automated test case generation (ATCG) tool for CTRL or to adapt our code to be able to use an existing tool, we chose the latter solution in order to build upon the experience and the knowledge others earned through the years. CTRL's language specificities led us in choosing Microsoft's Pex tool and the underlying C# language. Amongst the other options, this one seems to be the most flexible in terms of language constructs and supports. Furthermore, C# and CTRL are similar in many points, reducing the effort required to translate the code.

In the following subsections we will give a brief overview of the ITEC workflow, depicted in Figure 1.



IDE . . . Integrated Development Environment SP . . . Semi-purification/semi-purified CUT . . . code under test

Fig. 1. ITEC workflow
The figure depicts ITEC's components (nodes) and the information passed between them (arrows). The dashed boxes separate the components logically into two types: generic and Pex-specific components.

3.1 Semi-purification and CUT Isolation

The first step is the isolation of code under test (CUT). Code dependencies such as function calls other CTRL routines, accesses to global variables or data stored in a database complicate the testing process.

In order to effectively generate unit test cases, we have to remove these dependencies and replace them with predefined values. For this purpose we use semi-purification [8], an approach where dependencies are replaced by additional input parameters to the CUT. The values generated for the new parameters will then be used to specify test doubles for the test execution.

3.2 Translation

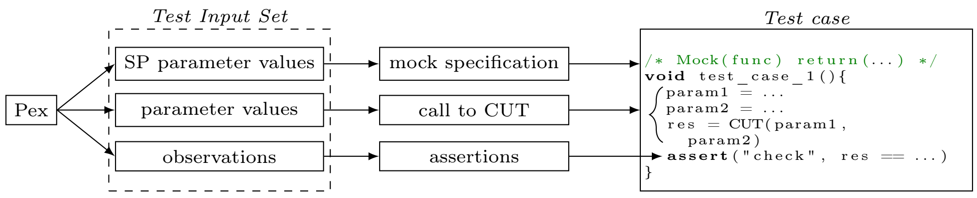
The semi-purified code is then translated to the ATCG tool's operating language – in our case this is C#. The translation process is non-trivial, as statements and constructs need to be mapped from one language to another.

To mention two examples for which the translation required adaptations: 1. C# does not allow indexers (index references to list elements) to be passed as reference parameters; 2. CTRL variables are automatically initialized with a default value, while C# requires explicit initialization. Additionally we manually implemented most of the built-in CTRL standard library, including missing data types in C#. More details on the translation validity will be given in Section 5.

3.3 ATCG Execution

Following the translation, the C# code is combined with the manually translated artefacts and the parameterized unit test (PUT). The PUT can be thought of as a parameterized routine that executes the CUT and subsequently performs assertions. PUTs are not Pex-specific but can be found in many modern unit testing frameworks such as JUnit, NUnit and others.

After the compilation of these resources, the ATCG tool (Pex) is triggered.



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

Fig. 2. Test case generation from Pex output

3.4 Test Case Creation

In this phase, the values from the generated test input sets are re-translated into CTRL. The next step is to use the semi-purification knowledge to separate the Pex-produced values into CUT parameters, parameters added by semi-purification and other observations. Depending on the category, the values are used for different parts of the test case. Figure 2 shows the three categories of values that Pex produces and their transformation to code. It is to be said, that since CTRL does not support reflection natively, mock specifications have to be transformed into mock functions at execution time.

3.5 TC Execution, Mock Generation

At test run time, the mock specifications (see Figure 2) are used to create function doubles which simulate the dependencies' expected behaviour. The current version of mock specifications are fairly simple, but an extension has been proposed to allow for more complex behaviour (different behaviour based on timing, iterations, etc.). Function doubles replace dependencies during test execution and return predefined values. They can also perform some simple assertions, if specified.

4 Results

To test the effectiveness of ITEC we executed it on 1111 functions found in JCOP [2], one of CERN's two WinCC OA frameworks. The test case generation was performed with eight concurrent threads on a Windows 7 virtual machine with eight CPUs (each 2.4 GHz) and 16 GB RAM. The generation and execution of test cases had a time out of two and one minute, respectively. We ended up with 602 functions that were successfully translated to C# and used for test input generation.

The remaining 509 functions were not translated due to the following reasons (texts in brackets provide references in Figure 3): 166 contained unsupported features or functions (*Unsupported*), 159 could not be translated due to unavailable dependencies (*SP Err*), 184 invalid translations (*Translation Err*) that led to compilation errors.

For the 602 successful translated functions Pex produced 3972 test inputs. During translation of these inputs to CTRL, filtering of invalid and untranslatable input removed 294 errors (*TCGen Err*), leaving 3678 implemented CTRL test cases.

An analysis of the test case executions showed that 2465 test cases had a matching between CTRL execution result and Pex' predictions, whereas 1184 did not (no match between CTRL and Pex execution), 29 test cases crashed during execution.

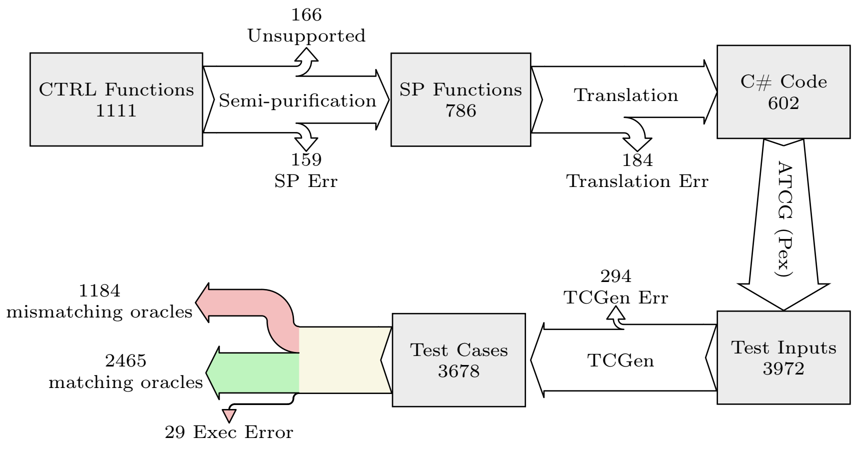


Fig. 3. Sankey diagram, displaying the quantitative analysis of the execution on a subset of the JCOP framework

We further analysed the code coverage of the respective CUTs using WinCC OA's built-in line coverage reporting. Table 1 shows the number of functions grouped by their line coverage. The first column displays the coverage ranges, the second column shows the number of functions with that coverage when executing all test cases and the third column the number of functions when only taking test cases with matching oracles into account.

The first row of the table indicates that over a third of the functions reach 100% code coverage, both with all test cases and when only taking test cases with matching oracles into account. In general the coverage of relatively few functions drops due to restricting the coverage calculation to only test cases with matching oracles. However, one thing is clearly noticeable. In total, there are only 15 functions with 0% coverage, meaning that there exist no tests at all for these functions. The reason for this could be that either no test inputs were generated, due to too complex constraints, or that only invalid test inputs were created which could not be translated to CTRL. When only looking at matching oracle test cases this number grows to 51, indicating that there are 36 functions that only have test cases which mismatch the expected results.

It seems though that Pex works well to achieve high code coverage (56% of the functions have $\geq 75\%$ line coverage).

It should be noted that even though the number of test cases with mismatching results between CTRL and C# is high, the effort put into their generation is not wasted. One can easily see that automatically updating the expected outcome of the test cases could produce a powerful regression test suite. Clearly, our translation from CTRL to C# is not entirely correct, as we would not have test cases with mismatching oracles otherwise.

Line coverage	**# Functions (all TCs)**	**# Functions (with matching oracle)**
100 %	230 (38 %)	215 (36 %)
75 % - 99	129 (21 %)	120 (20 %)
50 % - 74 %	110 (18 %)	111 (18 %)
1 % - 49 %	118 (20 %)	105 (17 %)
0 %	15 (2 %)	51 (8 %)

Table 1. Code (line) coverage: all test cases and test cases with matching results only and total

We did however also identify several caveats. Firstly, producing "sensible" input data seems to be a difficult task for the generation tool (Pex). In our case it would have been beneficial to have Pex produce a list of strings with certain format. Even though theoretically Pex is capable of doing this, the processing time increases dramatically as a result and in many cases the added constraints on the produced data lead to a very low number of test cases and coverage. The execution cost in terms of time and memory grows exponentially with the number of constraints.

Another problem is that Pex's working principle is based on block coverage. This means that Pex will try to procude the smallest set of inputs to cover the CUT. However, this also includes that boundary values or mutation considerations are not taken into account. This results in testsuites that verify the corresponding outcome (same input produces same output) but not the negative cases (changed CUT leads to failing tests).

5 Translation validation

At the time of writing, the translation from CTRL to C# has not yet been formally verified. In fact, the translation to the ATCG tool's (Pex') operating language represents an essential step in the workflow of our test case generation system. It seems self-evident that an erroneous translation from CTRL to C# would not only lead to a misguided exploration but also invalidate the produced test cases and results. Hence, a validation is required. Unfortunately Siemens does not provide a clear semantic for the CTRL language. For that reason, proofs cannot be utilized to show the validity of our translation.

In this section we will show how to verify a translation from one language to another, using testing and decomposition.

5.1 Syntactical Translation

To start, we need to clearly define the meaning of code translation from a source language to a target language.

Definition 1 (Syntactical translation of a source code in a language to another language). *A syntactical translation is a partial function* $st : L_{src} \to L_{dst}$ *that takes a piece of code* c_1 *written in a source language* L_{src}*,* $c_1 \in L_{src}$*,* *and translates it to a piece of code c_2 written in a destination language L_{dst}, $c_2 \in L_{dst}$:*

$$c_2 = st(c_1), c_1 \in L_{src} \textit{ and } c_2 \in L_{dst} \tag{1}$$

Note that this translation is purely syntactic. Although the translation aims to preserve the code's semantics, the definition above does not take this equivalence into account.

Our next goal is therefore to show this syntactical translation validity. Conceptually, we would like to show an equivalence between the source code and the destination code. However, we cannot show this equivalence in the syntactical domain.

5.2 Semantic Equivalence

We must therefore observe the semantic, denoted with the symbol $[\![.]\!]$ in the following definitions, of each piece of code in their respective language.

Definition 2 (Semantic of source and destination code). *Given a code c_1 written in the language L_{src}, its execution with a given parameter interpretation σ is denoted as:*

$$[\![c_1]\!]^{\sigma}_{L_{src}} = f_{src} \tag{2}$$

$$f_{src} : dom_{L_{src}} \times ... \times dom_{L_{src}} \rightarrow dom_{L_{src}} \tag{3}$$

Similarly, for a code c_2 written in the language L_{dst}, its execution with a given parameter interpretation σ' is denoted as:

$$[\![c_2]\!]^{\sigma}_{L_{dst}} = f_{dst} \tag{4}$$

$$f_{dst} : dom_{L_{dst}} \times ... \times dom_{L_{dst}} \rightarrow dom_{L_{dst}} \tag{5}$$

To show the semantic equivalence of two codes c_1 and c_2 in their respective languages L_{src} and L_{dst}, we must also be able to map the data types defined in both languages. This mapping defines a relationship between values of equivalent domains in both languages.

Definition 3 (L_{src} and L_{dst} domains mapping). *There exists a partial function h such that it maps a variable value defined in the language L_{src} to a variable value defined in the language L_{dst}:*

$$h : dom_{L_{src}} \rightarrow dom_{L_{dst}} \tag{6}$$

To ease the definition of the partial function H that applies the mapping for all parameters of a given function, we first define the parameters themselves in both domains:

$$dom_{L_{src}} \times ... \times dom_{L_{src}} \in P_{L_{src}} \tag{7}$$

$$dom_{L_{dst}} \times ... \times dom_{L_{dst}} \in P_{L_{dst}} \tag{8}$$

Now we can define the partial function H over these parameters:

$$H : dom_{L_{src}} \times ... \times dom_{L_{src}} \rightarrow dom_{L_{dst}} \times ... \times dom_{L_{dst}} \tag{9}$$

Given the parameters p_{src}, the semantic of H is the following:

$$p_{src} \in dom_{L_{src}} \times ... \times dom_{L_{src}} \tag{10}$$

$$H(p_{src}) = \langle h(p_i) | p_i \in p_{src}, 0 \leq i \leq |p_{src}| \rangle \tag{11}$$

To introduce the translation validity, let's first consider the picture depicted on Figure 4. Three main parts can be distinguished on this picture.

– The top arrow leading from c_1 to c_2 depicts the translation st of the code c_1 written with the language L_{src} to the code c_2 written in L_{dst};
– The second arrow shows the domains mapping H between L_{src} and L_{dst}. This mapping is used to adapt values chosen for the interpretation σ in dom_{src} to the interpretation made in dom_{dst};
– Finally, the bottom arrow shows the wanted equality between the execution of the code c_1 and c_2.

$$\begin{tikzcd}
\boxed{c_1} \arrow[r, "c_2 = st(c_1)"] \arrow[d, dash, dashed] & \boxed{c_2} \arrow[d, dash, dashed] \\
\sigma \arrow[r, "H"] \arrow[d, dashed] & H(\sigma) \arrow[d, dashed] \\
\boxed{[\![\cdot]\!]^{\sigma}_{L_{src}}} \arrow[r, "\equiv"] & \boxed{[\![\cdot]\!]^{H(\sigma)}_{L_{dst}}}
\end{tikzcd}$$

Fig. 4. High level picture of the translation validity

Formally, we could define this semantic equivalence with the definition 4.

Definition 4 (Semantic equality).

$$h([\![c_1]\!]^{\sigma}_{L_{src}}) = [\![c_2]\!]^{\sigma'}_{L_{dst}}, \forall \sigma, \forall c_1 \in L_{src}, c_2 = st(c_1) \in L_{dst}, \sigma' = H(\sigma) \tag{12}$$

Take note that in order to define the translation correctness, one must consider all possible codes c_1 written in the language L_{src} and all possible parameter interpretations σ.

Although it is theoretically correct, this is impractical. In fact, constructs or data types of the source language might not be translatable to the destination language. If we take a step back to Definition 1, the syntactical translation is defined as a partial function for this reason. In our translation, we did not find any construct that cannot be translated from CTRL to C#. However, in some cases, the translation can be really arduous and could easily lead to translation errors.

Furthermore, the domains mapping functions h/H are also partial functions (Definition 3). In fact, some data types of the language L_{src} might not be mappable to types in L_{dst}. In our translation, we came across the data type *shape* that is virtually impossible to translate to C#, as illustrated in Example 1.

Example 1 (Domains not existing or not translatable). CTRL provides the data type `shape`. Shapes are pointers to graphics elements, that are used to display information in user interface panels. Which element is pointed to is identified by the graphical object's name. As these names can be set and modified at runtime, it is impossible to know the shape's type, state and attributes.

Based on the previous remarks, proving the validity is close to impossible, especially since a full mapping between the source and the destination languages might not exist. Dániel Várro et al. discuss on that matter when considering model transformation verification in [14]. He mentioned two main concepts as requirements to verify model transformation:

1. *Syntactic completeness*: the source language covers the destination language in terms of constructs;
2. *Syntactic correctness*: the translation leads to a syntactical correct model.

In our case, both requirements can be violated. In fact, one can violate syntactic completeness if the source language does not cover the destination language, as it is the case with the data type *shape* for example. We can however argue that this requirement can be satisfied since both languages are Turing-complete, yet it would be very arduous to do so. As for syntactic correctness, this can be violated due to an erroneous translation that leads to a code that is syntactical incorrect. Furthermore, combinatorial explosion threatens to quickly become a problem if we assume that we need to exercise all possible codes with all input combinations to verify our translation.

5.3 Testing to Increase Confidence

However, we can still increase our confidence in the translation using both testing and the execution of the source code as an oracle.

Definition 5 (Defining tests to increase translation's confidence). *Assuming a function sel that selects a relevant set of interpretations σ for a given source code written with the language L_{src}:*

$$sel : L_{src} \rightarrow \mathcal{P}(\sigma) \tag{13}$$

To increase our confidence in the translation, we need to show that for all chosen interpretations $sel(c_1)$ the execution of both source and destination code are equal, given the domains mapping h/H.

$$h([\![c_1]\!]^{\sigma}_{L_{src}}) = [\![st(c_1)]\!]^{H(\sigma)}_{L_{dst}}, \forall \sigma \in sel(c_1), c_1 \in L_{src} \tag{14}$$

Note that the selection methodology implemented by the function *sel* is crucial to increase our confidence in the translation, yet we will not address this matter in this paper. We could mention test input generation techniques such anti-random [10] for example or selection hypothesis such as [1] that address the selection of a relevant set of test inputs. For the sake of the argument, we will assume that the methodology chosen is of the greatest quality.

According to our current definition, we must generate tests every time we translate a new source code. Even if this technique works, one can understand that this is a labour intensive activity. Hence, we are now able to increase our confidence in the translation for a source code that satisfies the two requirements previously mentioned, even though it is still cumbersome.

One can make a reasonable assumption saying that the overall semantic of a function is given by the composition of the semantic of its basic block. For that purpose, we assume that no compiler optimizations are applied in order to keep the current structure of the code and satisfy our previous assumption. This assumption eases the overall verification of the translation. In fact, we don't have to check every piece of source code to ensure the quality of our translation, but only to check basic blocks.

Let us now formally define how to verify basic blocks for the translation.

Definition 6 (Structure of a piece of code). *In our case, a piece of code c_1 written in the language L_{src} is a function. This function can be decomposed in a signature sig and a block of statements body:*

$$c_1 = \langle sig, body \rangle \text{ with } c_1 \in L_{src} \tag{15}$$

The body itself is composed of either control blocks, i.e. loop and conditional blocks, or statements:

$$\begin{aligned} &body \subseteq \mathcal{P}(block) \\ &stmt : statement \rightarrow block \\ &ift : statement \times block \rightarrow block \\ &while : statement \times block \rightarrow block \end{aligned} \tag{16}$$

Note that block is part of the source language:

$$block \in L_{src} \tag{17}$$

From this code structure, we can therefore decompose a source code into its basic blocks.

Definition 7 (Decomposing a source code in basic blocks). *To be able to decompose a source code, we need first to define how to decompose basic blocks.*

$$dec : block \rightarrow \mathcal{P}(statement) \tag{18}$$

$$\begin{aligned} &dec(stmt(stmt1)) = \{stmt1\} \\ &dec(ift(stmt1, block1)) = stmt1 \cup dec(block1) \\ &dec(while(stmt1, block1)) = stmt1 \cup dec(block1) \end{aligned} \tag{19}$$

The overall decomposition of a source code is therefore given by:

$$\begin{aligned} &decomposition : L_{src} \rightarrow \mathcal{P}(statement) \\ &decomposition(\langle sig, body \rangle) = \bigcup_{\forall block \in body} dec(block) \end{aligned} \tag{20}$$

Furthermore, we generalize the later decomposition by anonymizing variables and constants in statements to only preserve data types. The following example should clarify the decomposition with anonymization:

Example 2 (Decomposition of a source code c_1). Considering the following code c_1:

Listing 1. Example code for decomposition

```
void func(int abc) {
  abc = abc + 1;
  if (abc == 2) {
    abc = 10;
  }
}
```

The decomposition with anonymization of the example code into basic blocks leads to the following result:

$$
\begin{aligned}
anonymize(decomposition(c_1)) =& anonymize(\\
& dec(stmt(abc = abc + 1)) \\
& \cup\ dec(ift(stmt(abc == 2), stmt(abc = 10))) \\
anonymize(decomposition(c_1)) =& \{int = int + int, int == int, int = int\}
\end{aligned} \tag{21}
$$

Based on this source code decomposition, we are able to know the anonymized statements that are executed and we can therefore generate tests for each of these statements independently.

Definition 8 (Verifying translation semantic by testing it on basic blocks). *As we were previously addressing tests selection over a whole function, one can therefore define a new selection function addressing only anonymized statements:*

$$sel_{stmt} : stmt \to \mathcal{P}(\sigma) \tag{22}$$

The set of statements of a given source code c_1 that must be verified is:

$$stmts = anonymize(decomposition(c_1)), c_1 \in L_{src} \tag{23}$$

One can now redefine the semantic equivalence required for the translation over all statements from the source code c_1 with domains mapping h/H:

$$h([\![stmt]\!]^{\sigma}_{L_{src}}) = [\![st(stmt)]\!]^{H(\sigma)}_{L_{dst}}, \forall stmt \in stmts, \forall \sigma \in sel_{stmt}(stmt) \tag{24}$$

As we discussed before, this verification can be partial if we have syntactical incompleteness between our languages. However, we can ensure the quality of the translation up to a certain level. We will discuss of that matter in the next section.

6 Quality Metric

Based on the process described in section 5, we base our confidence of a CUT on the validity of its individual statements' translations. We must therefore take them into consideration when addressing tests generation results. In fact, we cannot pragmatically consider that a code coverage of 100 % is trustable if none of the individual statement translations have been verified.

For that purpose, we define a quality metric ϕ that represents our confidence in the translation and its correctness.

Definition 9 (Correctness confidence measure for code). *Given a piece of code c written in a given language L, the function ϕ represents the correctness confidence as a boolean value defining that the code for the translation is tested (1) or not (0).*

$$\phi : L \to \mathcal{B} \tag{25}$$

This means that for individual statements ϕ denotes whether the basic blocks have been tested or not, as defined in the previous section.

Based on a confidence of a (CUT's) basic block's individual confidence measures, we then define our confidence into the composition of blocks (such as a function) as follows:

Definition 10 (Correctness confidence of a composition). *Let c_1 be a code under test. Let further stmts be the body of the function, consisting of individual statements $stmt_1, \ldots, stmt_n$. We define that our confidence in the entire code as the mean average of the anonymized statements' correctness confidence.*

$$\phi(c_1) = \frac{\sum_{i=1}^{n} \phi(anon_i)}{n}, anon_i \in anons \tag{26}$$

where anons is $\{anonymize(stmt_1), \ldots, anonymize(stmt_n)\}$, the set of anonymized statements occurring in c_1.

In a case where multiple statements have the same anonymous representation (e.g. multiple integer additions), only one of them is chosen for the calculation. This is to avoid repeated statements or loops to influence the measure.

We then further define our confidence into the correctness of an individual testcase execution.

Definition 11 (Confidence in test case results). *Let tc be a test case using input interpretation σ that executes the code under test c_1. Let stmts be the statements $stmt_1, \ldots, stmt_n$ of c_1. We define our confidence in the correctness of a result obtained by executing tc as the product of the confidence values of the executed statements denoted as $stmts|_\sigma$.*

$$\phi_{tests} : L_{src} \times dom_{src} \times ... \times dom_{src} \to \mathcal{B} \tag{27}$$

$$\phi_{tests}(c_1, \sigma) = \prod_{\forall stmt_i \in stmts|_\sigma} \phi(anonymize(stmt_i)) \tag{28}$$

Note that for test case executions, the confidence calculation of every statement is taken into account, even if multiple statements have the same anonymization. For repeated execution of the same statement, such as in a loop, every execution is taken into account. This is possible, since the usage of coverage measures provides the execution count for each individual statement.

6.1 Example Calculation of Correctness Confidence

Listing 2 displays a function for which we want to calculate the correctness confidence and Listing 3 shows this function with anonymized statements. For this example we will assume that the anonymized functions $int++$, $int + int, int = int + int$, have been fully tested ($\phi = 1$), while $int > int$ and $int\%int$ have not been validated ($\phi = 0$).

First we will calculate the correctness confidence for the entire translated function. According to definition 10 we will calculate the mean of the individual types of anonymized statements. For our example, we therefore will end up with the following calculation (subscript texts indicate the line numbers).

$$\phi(func) = \frac{1_{L2} + 1_{L4} + 0_{L5} + 0_{L6}}{4} = 0.5 \tag{29}$$

Assuming the existence of a passing test case tc that would assert that the result of $func(x, y)$ with $\sigma = \langle 3, 5 \rangle$ is 8, we would calculate its correctness confidence as follows:

$$\phi(func, \sigma) = 1_{L2} * 1_{L3} * 1_{L4} * 0_{L5} * 1_{L8} = 0 \tag{30}$$

Listing 2. Example function

```
1   int func(int a, int b) {
2     a++
3     a++
4     b = b+2
5     if(a > b){
6       return a % b
7     } else {
8       return a + b
9     }
10  }
```

Listing 3. Anonymized `function`

```
1   int func(int, int){
2     int++
3     int++
4     int = int + int
5     if(int > int) {
6       return int % int
7     } else {
8       return int + int
9     }
10  }
```

7 Conclusion and Future Work

This paper presents the lessons learned during the creation and the execution of our tool, ITEC.

First, we shortly describe ITEC's workflow and the individual steps that are taken for the test cases generation. Based on this, we present our initial results of the test case generation and execution for one of CERN's CTRL frameworks (more than 1000 individual functions). We realize from these results that we could obtain test cases with mismatching oracles when executing equivalent CTRL and C# codes individually with similar inputs. These results outline possible problems in the translation of CTRL code to C#.

To address this issue, we present formal foundations on the validation of the CTRL to C# translation in order to increase our confidence in the chosen approach. Finally, we discuss of a quality measure that allows us to determine the confidence we put into our translations and hence further into our generated test cases.

Based on the work done so far, we aim to extend this research into several areas: 1. Build an extensive test suite of translation validations for basic blocks, and empirically compare the quality metric's predictions with real data; 2. Improve the translation, both to and from C#, to generate fewer failing test cases; 3. Research into ways to improve test case generation in presence of complex constraints, such as input data matching domain formats; 4. Verify the test cases' effectiveness by systematically executing them on mutated versions of the CUT; 5. Study and introduce a way to make these generated regression tests evolve with time as the code is changing.

References

1. Bernot, G., Gaudel, M., Marre, B.: Software testing based on formal specifications: a theory and a tool. Software Engineering Journal 6(6), 387–405 (1991), `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=120426`

2. CERN: The JCOP Framework. https://j2eeps.cern.ch/wikis/display/EN/JCOP+Framework (August 2014)
3. Csallner, C., Tillmann, N., Smaragdakis, Y.: Dysy: Dynamic symbolic execution for invariant inference. In: Proceedings of the 30th International Conference on Software Engineering. pp. 281–290. ICSE '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1368088.1368127
4. ETM Professional Control: WinCC OA at a glance. Siemens AG (2012)
5. ETM Professional Control: Control script language. http://etm.at/index_e.asp?id=2&sb1=54&sb2=118&sb3=&sname=&sid=&seite_id=118 (2015)
6. Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: testing model transformations. In: Model, Design and Validation, 2004. Proceedings. 2004 First International Workshop on. pp. 29–40 (Nov 2004)
7. Jörges, S.: Construction and Evolution of Code Generators - A Model-Driven and Service-Oriented Approach, Lecture Notes in Computer Science, vol. 7747. Springer (2013)
8. Klikovits, S., Lawrence, D.P.Y., Gonzalez-Berges, M., Buchs, D.: Considering execution environment resilience: A white-box approach. In: Fantechi, A., Pelliccione, P. (eds.) Software Engineering for Resilient Systems - 7th International Workshop, SERENE 2015, Paris, France, September 7-8, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9274, pp. 46–61. Springer (2015), http://dx.doi.org/10.1007/978-3-319-23129-7_4
9. Küster, J.M., Abd-El-Razik, M.: Models in Software Engineering: Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers, chap. Validation of Model Transformations – First Experiences Using a White Box Approach, pp. 193–204. Springer Berlin Heidelberg, Berlin, Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-69489-2_24
10. Malaiya, Y.K.: Antirandom testing: getting the most out of black-box testing. In: Sixth International Symposium on Software Reliability Engineering, ISSRE 1995, Toulouse, France, October 24-27, 1995. pp. 86–95. IEEE (1995), http://dx.doi.org/10.1109/ISSRE.1995.497647
11. Microsoft Research: Pex, Automated White box Testing for .NET. http://research.microsoft.com/en-us/projects/pex/
12. Pnueli, A., Siegel, M., Singerman, E.: Tools and Algorithms for the Construction and Analysis of Systems: 4th International Conference, TACAS'98 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98 Lisbon, Portugal, March 28 – April 4, 1998 Proceedings, chap. Translation validation, pp. 151–166. Springer Berlin Heidelberg, Berlin, Heidelberg (1998), http://dx.doi.org/10.1007/BFb0054170
13. Samet, H.: Automatically proving the correctness of translations involving optimized code. Memo AIM, Stanford University (1975), https://books.google.ch/books?id=1sI-AAAAIAAJ
14. Varró, D., Pataricza, A.: Automated formal verification of model transformations. In: Jürjens, J., Rumpe, B., France, R., Fernandez, E.B. (eds.) CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop. p. 63–78. No. TUM-I0323 in Technical Report, Technische Universität München, Technische Universität München (September 2003), http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2003/csduml2003_vp.pdf
15. Zuck, L.D., Pnueli, A., Goldberg, B.: VOC: A methodology for the translation validation of optimizingcompilers. J. UCS 9(3), 223–247 (2003), http://dx.doi.org/10.3217/jucs-009-03-0223