Considering Execution Environment Resilience: A White-Box Approach

Stefan Klikovits[1,2(✉)], David P.Y. Lawrence[1,3], 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
[3] Honeywell International Sarl., Rolle, Switzerland

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. Only few of them consider dependencies from outside the Code Under Test's scope such as global variables, database values and subroutine calls as influences to the execution path. In order to fully test all possible scenarios these dependencies have to be taken into account for the test input generation. This paper introduces ITEC, a tool for automated test case generation to support execution environment resilience in large-scaled, complex systems. One of ITEC's corner stones is a technique called semi-purification, a source code transformation technique to overcome limitations of existing tools and to set up the required system state for software testing.

Keywords: Resilience · Automated test case generation · Software testing · Semi-purification · 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/front-end computers which connect to the sensors and actuators. As such applications require the configuration of hundreds of controllers, CERN has developed two frameworks on top of Siemens' *Simatic WinCC Open Architecture* (WinCC OA) [7] SCADA platform to facilitate their creation.

Due to lack of tool support for the WinCC OA's scripting language *Control* (CTRL) [8], it was so far not possible to write and execute unit tests in an efficient manner. Recently, the Industrial Controls Engineering group at 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

© Springer International Publishing Switzerland 2015
A. Fantechi and P. Patrizio (Eds.): SERENE 2015, LNCS 9274, pp. 46–61, 2015.
DOI: 10.1007/978-3-319-23129-7_4

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.

This paper introduces the *Iterative TEst Case* system (ITEC), a system for the automatic generation of test cases for the frameworks. ITEC relies on existing *automatic test case generation* (ATCG) methodologies such as dynamic symbolic execution [1,2] or combinatorial testing [3,4] to generate test input. Currently, many ATCG methodologies do not support the modification of source code dependencies, such as global variables, external resources (e.g. databases) and function/subroutine calls. While during manual unit test creation set-up routines and test doubles [11] (mocks, stubs, etc.) are frequently used to prepare and simulate a desired system state, techniques such as random testing and combinatorial testing and their respective tool implementations often do not support generation of test doubles and set-up routines. To overcome this caveat we introduce a technique called *semi-purification* which creates a bridge between existing testing tools and the mentioned requirements. Semi-purification is a process that replaces dependencies with additional function parameters, so that ATCG tools can generate test input for all code execution paths.

The rest of this paper is structured as follows: Sect. 2 describes the development style and applications at CERN and illustrates the resulting problems. Section 3 gives an overview of ITEC and describes the general work- and information flow. Section 4 explains semi-purification, in detail, followed by the introduction of some non-trivial semi-purification challenges in Sect. 5. Section 6 concludes.

2 Problem Description

The EN-ICE group at CERN has been developing and maintaining two frameworks for the development of SCADA applications for accelerators, experiments and infrastructure. WinCC OA, the underlying platform, provides amongst many other features the functionality to obtain and display data from sensors and send commands to actuators. Figure 1 depicts the typical three layers schema (field objects/sensors - frontend controllers - SCADA) of control systems at CERN and their connection to the operator work stations.

Currently about 600 individual WinCC OA systems are in action for the LHC and its experiments alone[1] – all of them relying on the functionality of the frameworks. Additionally many more applications use the frameworks, such

[1] Gonzalez-Berges, M. – Presentation at ETM Day, (CERN, 2013).

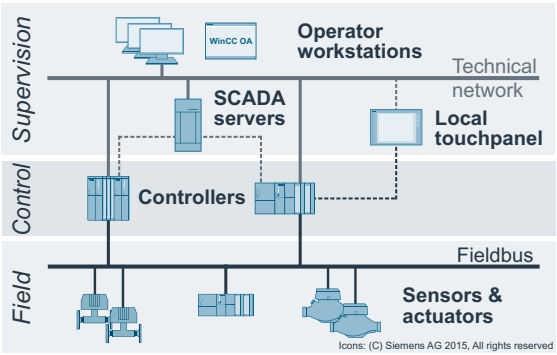

Fig. 1. Layer model dedepicting the connection of field objects, frontend controllers (e.g. PLCs) and Operator Work Stations (OWS) through SCADA applications

as the electrical network supervision with its tens of thousands devices and hundreds of thousands measurement and control points.

The frameworks' libraries and UI controls are primarily written in WinCC OA's proprietary scripting language *Control* (CTRL). CTRL's syntax is based on ANSI C, but provides some specific extensions such as for accessing the WinCC OA database. Other language modifications include the removal of features such as *pointers*, *structs* and *typedefs* and introduction of *reference parameters*, *dynamic arrays* and implicit type casting.

The syntax modifications are significant enough to exclude the use of existing unit testing frameworks. Hence, to date, mainly manual testing has been employed for verification. It has been considered to translate CTRL and use existing unit test case frameworks, but the backlog of not unit test covered code would still remain significant.

At the time of writing, CERN supports the execution of WinCC OA on two *operating systems* (OS) and typically up to two versions each (currently *Windows 7*, *Windows Server 2008 R2* and *Scientific Linux CERN 6*). As CERN's security protocol requires the switch to current operating system versions, regular upgrades are essential. The operators also introduce patches and updates in frequent intervals. Additionally, new WinCC OA releases are published every one to two years. Due to Siemens' support policy it is necessary to upgrade to new versions, which often involves updating the software because of changed features and/or compatibility issues.

This all results in an ever changing execution environment over the entire runtime of the LHC (over 30 years in total). Before every change, the software has to be tested and adapted if necessary. Due to the frameworks' sizes and the lack of automated CTRL test cases, this task takes a long time to complete.

The current release testing process requires developers to thoroughly verify the functionality of all components and fix any defects discovered. After several iterations of the test-fix cycle, the code base reaches a point that is deemed ready for shipment.

Although the manual verification has been complemented by a few automated user interface tests, testing remains a repetitive manual process, consuming time that developers could spend more productively on other tasks.

3 Iterative Test Case Generation System

To overcome the lack of test cases and code coverage, we propose the *Iterative TEst Case* (ITEC) generation system. ITEC generates and executes test input for pieces of source code, referred to as *Code Under Test* (CUT). The size of the CUT can vary in granularity from individual procedures to multiple functions connected by subroutine calls.

As displayed in Fig. 2, ITEC's workflow is split into four subtasks, each individually essential for the quality of the overall result.

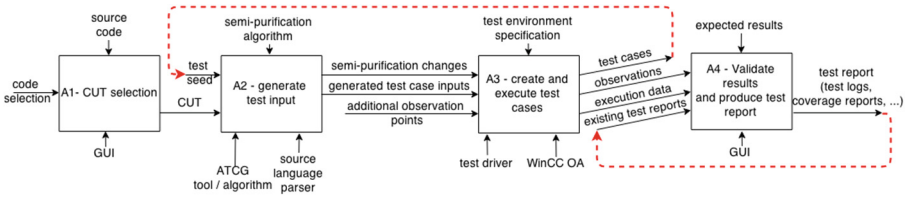

Fig. 2. Overview of the system's standard workflow on a high level with all in- and outputs of the system

A1. In the first step, the user chooses the *Code Under Test* (CUT). This can be a single function, a library or an entire component.

A2. Before the test case generation, verified test cases ("test seeds") can optionally be fed into the system to gather information from previous generations and existing (possibly manually implemented) test cases.

The test input generation itself is based on modified versions of the CUT. These modifications are made by a process we call *semi-purification*, which assures that external dependencies such as global variables, database values and other libraries are taken into account for the generation.

After the test data generation the resulting test input has to be cleansed from unviable values. The cleaning is necessary, as due to the transformation of the CUT it is possible that third party ATCG methodologies produce values that are impossible for the original code base.

A3. Based on the generated test input, the next task is to create and execute test cases. The required test setup procedures will be derived from the analysis of the semi-purification changes in *A2*. Subsequently, the test cases are executed and the changes to the system recorded. These observations will be added as assertions to the test cases.

A4. As the correctness of the assertions generated in *A3* has not been verified yet, ITEC will take advantage of existing test reports from previous test runs. These reports are used to assist in the decision to accept/refute the test observations (results). However, as there might not be enough test results available, or the existing information could be misleading, the final decision lies with the test engineer who serves as the test oracle.

After the judgment, the user has the choice to select a subset of the generated tests cases to be stored as future regression tests. Additionally a test report containing information about the generated test cases and the test execution logs is produced.

4 Generating Test Cases

The generation of test cases has experienced growing research interest. Although already introduced in the 1970s and 1980s the recent gain in computing power made the usage of methodologies such as adaptive random testing [6], combinatorial test case generation [3,4] and dynamic symbolic execution [1,2] more feasible. Leaning onto the insights of existing quality assurance research in the areas of specification based testing [12], test selection [13] and also quality assurance for resilient systems [14], we aim to build on these *Automatic Test Case Generation* (ATCG) processes for our purposes. In most cases ATCG methods produce a set of different test inputs that can be mapped on the input parameters of the function under test. The aim is to produce test inputs that execute the *Code Under Test* (CUT) along as many different execution paths as possible.

For this purpose we have to first define the notion of a function, for which we create test cases.

Definition 1 (Pure function). *We define a function* $f = \langle \pi, impl \rangle$ *to be a program function, that depends on its input parameter names* $\pi \in \Pi$ *and its implementation* $impl \in Impl$*, and consists of a sequence of statements describing its behaviour.*

Definition 2 (Input parameter value assignment). *Before the execution of a function* f*, it is necessary to assign values to its input parameters. This assignment is defined as follows:*

$$assign : \Pi \rightarrow V \tag{1}$$

Definition 3 (Function execution). *We define exec as the execution of a function* f *with a given inputs assignment in a system with a certain state. The result of this function execution is a deterministic return value.*

We define a unit test case for such a function in the following manner:

Definition 4 (Test case). *A test case* $t = \langle i, e \rangle$ *consists of* $i = assign(\pi)$*, an assignment of input parameter values, and* e*, the expected value for the return value.*

However, in many cases the CUT is not a pure function [5], meaning that next to the parameters Π, f has a set of additional dependencies D from outside its function scope, such as global variables or access to external resources through the file system or databases.

Definition 5 (Function dependencies). *The dependencies d of a non-pure function f is a set of variables and resources, defined outside f's scope part of the system state. ($d \subseteq state$). For our purposes we assume we can access the variable/resource value through $val(d_i); d_i \in d$.*

Further, the function can change the system state by modifying global variables and resources. These so-called observable side effects are defined by the set S, where S is the set of changes to the system produced by the execution of a function.

Definition 6 (Non-pure functions and side effects). *Let f be a non-pure function defined with parameters $\pi \in \Pi$, dependencies $d \in D$ and observable side effects $s \in S$. Then the execution of f with an input parameters assignment on a system with a given state, is the transition of the system to a new state and the production of a return value.*

$$exec(f, assign, state) \rightarrow r, state' \tag{2}$$

s is the set of new values that have been changed by the transition outside the function's scope.

$$s = state' \backslash state \tag{3}$$

For our purpose we will denote a non-pure functions as $f = \langle \pi, d, impl \rangle$.

We therefore extend our previous definitions to define test cases for non-pure functions.

Definition 7 (Test case for non-pure functions). *Given a non-pure function under test f, a test case $t = \langle p, i, e \rangle$ for f consists of the preparatory procedure p, the input value assignment i and the expected values e for some or all of the observations of the return value and the side-effects $r \cup s$.*

In order to keep the test cases of such a function deterministic, the system has to be brought into a known state before execution. That means that a preparatory procedure must be executed in order to set each dependency to the required value.

Definition 8 (Preparatory procedure). *The execution of a preparatory procedure $p \in Impl$ of a test case for a non-pure function with dependencies d modifies the state of the system so that each $d_i \in d$ returns the desired value.*

Many ATCG methodologies (e.g. random testing, combinatorial testing) perform their generations in a black-box fashion based on a function's signature but ignore that external dependencies in the function body influence its behaviour too. Therefore, to effectively cover all necessary combinations of parameters

and dependencies we have to modify the CUT using a technique called *semi-purification*. This ensures that the generation will take these dependencies into consideration. The generated values for the additional parameters can then be used to replace the dependencies with test doubles.

4.1 Removing Dependencies Through Semi-purification

Leaning onto the notion of pure functions, we refer as *semi-pure* to those procedures whose outcome only depends on its input values, but may have a certain number of side effects. Accordingly, semi-purification is the process of converting a not semi-pure function into a semi-pure one. During the semi-purification process dependencies from outside the CUT's scope are discovered and replaced by new input parameters. Since the resulting functions only depend on their input parameters, we can then employ standard ATCG methodologies to create test input.

To start with, we define the semi-purification process for external resources and global variables. These efforts are derived from the concept of *localization*. Localization is a refactoring technique that completely replaces the access to global variables and has been first picked up programmatically by Sward and Chamillard [9] for Ada programs. The topic was revisited for C programs by Sankaranarayanan and Kulkarni [10]. Leaning onto this technique, we not only replace global variables, but also database and other resource accesses with new input parameters.

Definition 9 (Semi-purification). *Semi-purification is the process of converting a function's dependencies d into additional parameters π.*

$$SP : D \rightarrow \Pi \tag{4}$$

For simplification, we can imagine that when SP is applied onto an entire function $f = \langle \pi, d, impl \rangle$, it converts some (or all) of the dependencies.

$$SP : \Pi \times D \times Impl \rightarrow \Pi \times D \times Impl \tag{5}$$

$$SP(\pi, d, impl) = \langle \pi \cup \Delta\pi, d', impl' \rangle \tag{6}$$

where $d' \subseteq d$ and $\Delta\pi = \{SP(d_i) | d_i \in d\}$.

In our case, we want to remove all global variables and external resources, hence after the semi-purification $d' = \emptyset$. We require this process to be neutral, meaning that the return values and side effects are the same when executing the functions.

Definition 10 (Semi-purification neutrality). *SP has to be implemented so that it is neutral for any given f. Neutrality is given, iff*

$$\forall assign, \forall \Delta assign, \forall state : exec(f, assign, state) = exec(SP(f), \{assign, \Delta assign\}, state') \tag{7}$$

where $state' \subseteq state$ and $\forall \Delta\pi_i \in \Delta\pi : val(d_i) = \Delta assign(\Delta\pi_i); \Delta\pi_i = SP(d_i); d_i \in d$.

It is to be said that it is possible to check this neutrality but it is hard to prove it. We rely that the experiences and assertions of our predecessors [9,10] are valid. At a future stage we will aim to investigate this matter deeper. Our tool will use an algorithm that will work according to the properties described in this paper, however, at the time of writing the implementation phase is not completed.

Example of Semi-purification. The following subsection shows a small example of the process of semi-purification. Listing 1 shows a non-pure function that relies not only on its input values, but also on the values that are obtained from a database (`dbGet(x)`) and a global variable (`GLOBAL_VAR`). To deterministically test `f(x)` it is necessary to execute the preparatory procedure before executing the function call to the CUT. In our example test case (Listing 2) we do this by explicitly setting the values of the global variable and the data point.

Listing 1. A non-pure function

```
1  f(x){
2    if GLOBAL_VAR:
3      return dbGet(x)
4    else:
5      return -1
6  }
```

Listing 2. A test case for a `f(x)`

```
1  test_f(){
2    dbSet("test",5) // prepare
3    GLOBAL_VAR = True
4    x = f("test") // act
5    assert(x == 5) // assert
6  }
```

To convert `f(x)` into a semi-purified function we introduce additional input parameters (`a` and `b`) for its dependencies. The references to `GLOBAL_VAR` and the database access (`dbGet(x)`) are replaced by `a` and `b`, respectively. The semi-purified version of the function from the previous example can be seen in Listing 3. The according test case (Listing 4) does not require a preparatory procedure.

Listing 3. Semi-purified `f(x)`

```
1  f_sp(x,a,b){
2    if a: //GLOBAL_VAR:
3      return b // dbGet(x)
4    else:
5      return -1
6  }
```

Listing 4. Test case for `f_sp(x,a,b)`

```
1  test_f_sp(){
2    x = f("test",True,5) // act
3    assert(x == 5) // assert
4  }
```

Recursive Application of Semi-purification. Oftentimes the CUT is not an individual procedure, but includes subroutine calls to perform its operation. In these cases the semi-purification has to be applied recursively and the changes made to a subroutine (*callee*) need to be propagated to the *caller* in order to keep the CUT valid.

Listing 5. Recursive Semi-purification

```
1  functionA(x){
2    a = functionB(x)
3    return a
4  }
5
6  functionB(x){
7    b =  GLOBAL_VAR
8    b++
9    return b
10 }
```

Listing 6. Semi-purified CUT

```
1  functionA(x,y){
2    a = functionB(x,y)
3    return a
4  }
5
6  functionB(x,y){
7    b = y
8    b++
9    return b
10 }
```

An example of this behaviour can be seen in Listing 5 which displays the CUT (`functionA`). The semi-purification of the subroutine (removing the dependency to `GLOBAL_VAR`) modifies the function signature of `functionB`. This change has to be incorporated into the function call in `functionA` and also added to `functionA`'s signature. The resulting code can be observed in Listing 6.

4.2 Creating Test Inputs and Preparatory Routine

Following the semi-purification process we can execute ATCG tools on the modified CUT. The result is a set of test case inputs that could be used to verify the semi-purified version of the CUT. However, we aim to generate regression tests for the original CUT and hence need to re-convert the output.

The result of applying ATCG methodologies onto semi-purified functions, is a power set of assignments of values to the input parameters $assign' : (\pi \cup \Delta\pi) \rightarrow V$. For simplicity, instead of one assignment $assign'$, we will imagine the elements of the output as two separate mappings $assign$ and $\Delta assign$, one for the original parameters and one for the additional semi-purified parameters.

Definition 11 (Test input generation for semi-purified functions). *We define the test input generation TIG for a semi-purified function f' that produces a power set of parameter value assignments, $assign : \Pi \rightarrow V$ and $\Delta assign : \Delta\Pi \rightarrow V$.*

$$TIG : f \rightarrow \mathcal{P}(assign) \tag{8}$$

$$TIG(\pi \cup \Delta\pi, \emptyset, impl) = \bigcup_{i \in 0,\ldots,n} \{assign_i, \Delta assign_i\} \tag{9}$$

However, as the target is to produce test cases for the original CUT, the generated test inputs need to be re-transformed accordingly. This means that we have to take the value assignments for the newly introduced parameters in f' and convert them into a preparatory procedure.

Definition 12 (Inverse Semi-purification). *Given a function f and its corresponding semi-purified version f' we define SP^{-1} as the operation that converts $\Delta assign$ into a preparatory procedure P.*

$$SP^{-1} : \Delta assign \rightarrow P \tag{10}$$

where the execution of a $p \in P$ asserts that each $d_i \in d$ returns the value defined in $\Delta assign(SP(d_i))$.

The preparatory procedure includes for example the setting of global variables, data point values or the preparation of other external resources such as the file system state. Figure 3 shows the schematic representation of the information flow for the test input generation.

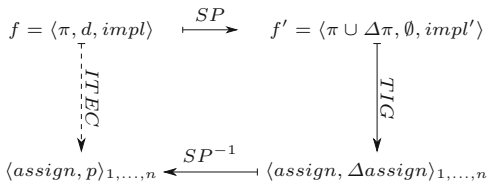

Fig. 3. Using ATCG on non-pure functions by removing dependencies on global variables and external resources.

4.3 Semi-purification of Subroutine Calls

Using the above described method, it is possible to create functions that only depend on their input parameters. Recursive application of semi-purification on subroutines permits the generation of test input for function interactions. However, when testing programs with a deep control flow graph, the semi-purification process might easily result in a procedure with dozens of input parameters and many subroutines that have to be rewritten. Whereas the approach we introduced so far permits the generation of integration tests, this approach comes with caveats. The generation of test inputs for CUTs of this complexity is computationally expensive, as for most ATCG methodologies the complexity rises with the number of input parameters. Additionally, in many cases the function calls connect multiple libraries, voiding the initial goal of creating unit tests.

To overcome this limitation, we introduce a semi-purification process that removes a function's *subroutine calls* (SRC). The two processes can be seen as complementary actions, each removing different dependencies. For the rest of this paper we continue with the following notation: Global variable and resource dependencies remain D, subroutine dependencies are identified by D_{SRC}. SP_{SRC} is the process of replacing some of these subroutine calls with additional input parameters $\Delta\Pi'$. The set of remaining subroutines D'_{SRC} contains SRCs that should not be replaced (e.g. built-in string operations or hashing procedures). For certain CUTs and situations this set is empty.

Definition 13 (Semi-purification of Subroutine Calls). *Semi-purification SP_{SRC} of subroutine calls is the process of converting a function under test (with subroutine calls) $f = \langle \pi, d \cup d_{SRC}, impl \rangle$ into a function $f'' = \langle \pi \cup \Delta\pi', d \cup d'_{SRC}, impl' \rangle$ where $\Delta\pi'$ is the set of newly introduced parameters to replace some (or all) of the subroutine dependencies d_{SRC}.*

$$SP_{SRC} : \Pi \times D \times Impl \rightarrow \Pi \times D \times Impl \tag{11}$$

$$SP_{SRC}(\pi, d \cup d_{SRC}, impl) = \langle \pi \cup \Delta\pi', d \cup d'_{SRC}, impl' \rangle \tag{12}$$

where $d'_{SRC} \subseteq d_{SRC}$ and $\Delta\pi' = \{SP(d_i) | d_i \in d_{SRC}\}$.

ATCG methodologies can then generate the input for the function with fewer dependencies (or none at all). The resulting input assignments

$(assign, \Delta assign')$ have to be re-transformed into test input for the original function. The additional test input for replaced subroutine calls ($\Delta assign'$) needs to be converted into a preparatory procedure P' that create test doubles for the masked out SRCs.

The entire workflow combining the two semi-purification routines (SP and SP_{SRC}) is displayed in Fig. 4.

Definition 14 (SP_{SRC} neutrality). *SP has to be implemented so that it is neutral for any given f. Neutrality is given, iff*

$$\forall assign, \forall \Delta assign, \forall state : exec(f, assign, state) = exec(SP(f), \{assign, \Delta assign'\}, state') \tag{13}$$

where $state' \subseteq state$ and

$$\forall \Delta\pi'_i \in \Delta\pi' : val(d_i) = \Delta assign'(\Delta\pi'_i); \Delta\pi'_i = SP(d_i); d_i \in d \tag{14}$$

Definition 15 (SP^{-1}_{SRC}). *The inverse semi-purification of SRCs is defined as the generation of a preparatory procedure $p' \in Impl$ for Δd_{SRC}.*

$$SP^{-1}_{SRC} : \Delta assign' \rightarrow P' \tag{15}$$

so that the execution of a $p \in P$ asserts that each subroutine $d_i \in d_{SRC}$ returns the value defined in $\Delta assign'(SP(d_i))$.

For subroutines the creation of a preparatory procedure includes the specification of test doubles for these functions that provide the specified values.

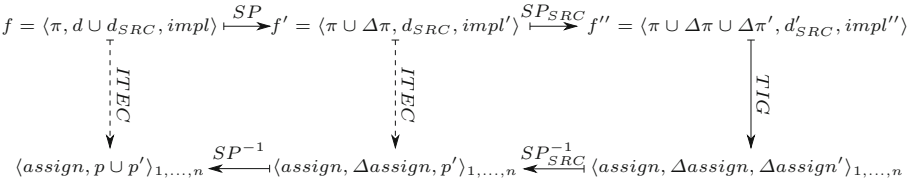

Fig. 4. The full semi-purification workflow, including subroutine replacement

5 Problematic Areas of Semi-purification

The application of semi-purification reaches limits in certain areas. The following section will introduce some of these areas and show our plans to overcome these problems.

5.1 Loops

One such situation is the presence of subroutine calls inside iterations (loops and recursions).

Looking at the semi-purified function `sleepUntilReady` in Listing 7 where the parameter `a` has been newly introduced to replace the dependency on `dbGet(notReadyDP)`.

Listing 7. Loop example

```
1  sleepUntilReady(a){
2    while a : // replaces dbGet(notReadyDP)
3      sleep(5) // sleep for 5 seconds
4  }
```

The algorithm that we introduced so far is not sufficient to fully test the functionality of this function. ATCG methodologies would create two test inputs (`True` and `False`) for `a`. However, the case where the function is executed with `a = True`, the test case would result in an endless loop.

In order to generate a test case for the `a = True` we have to change the parameter `a` to be a list of values and the loop to access a new element of this list as shown on the Listing 8. Using these modifications it is possible to generate the test input to achieve the desired behaviour.

Listing 8. Modified loop

```
1  sleepUntilReady(a){
2    i = 0
3    while a[i] :
4      sleep(5)
5      i++
6  }
```

Please note, that the example above can lead to an *IndexOutOfBounds* error, that we have to account for. In general it is not possible to create (single thread) test cases for unbounded loops, so we have to transform the loop to be bounded. ATCG tools work around the limitation of unbound loops by having certain timeout constraints on the time or number of execution paths analysed. We aim to investigate this issue in the future by instrumenting the loops and drawing conclusions from this execution information.

5.2 Dependencies Between Subroutines

Another problematic scenario is in the case of dependencies between subroutines. Listing 9 shows such a CUT, containing two read and one write access to a global variable (`SPEED_VAR`). The semi-purification algorithm we introduced above would create the CUT presented in Listing 10.

The problem with the naïve approach of semi-purification is that each access to a dependency is replaced with a new input parameter. Here `a` and `b` access the same variable which's value cannot change in standard (single thread) executions. For this reason the semi-purification process should replace them with the same parameter. The write access remains unmodified as it is a side effect and there are no further reads from this dependency.

Listing 9. CUT with dependencies

```
1  adjustSpeed(){
2    x = getTheSpeed()
3    if x < 10 :
4      doubleTheSpeed()
5  }
6
7  getTheSpeed(){
8    return SPEED_VAR
9  }
10
11 doubleTheSpeed(){
12   speed = SPEED_VAR
13   SPEED_VAR = speed*2
14 }
```

Listing 10. Naïvely semi-purified CUT

```
1  adjustSpeed(a,b){
2    x = getTheSpeed(a)
3    if x < 10 :
4      doubleTheSpeed(b)
5  }
6
7  getTheSpeed(a){
8    return a // SPEED_VAR
9  }
10
11 doubleTheSpeed(b){
12   speed = b  // SPEED_VAR
13   SPEED_VAR = speed*2
14 }
```

Additionally, it is in general necessary to discover sequential write-read scenarios, where an external value is first set, then read. In this cases it is necessary to re-use the value from the write for the read as well, as otherwise the behaviour of the modified CUT is changed.

5.3 Concurrency

The example from the previous subsection shows the generation of impossible scenarios for single-thread execution. However, WinCC OA uses so-called "managers" to perform tasks. Each manager has its own context and works as an individual process.

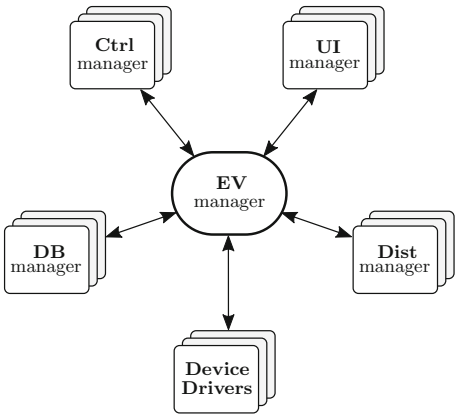

Fig. 5. WinCC OA's manager concept

Figure 5 displays some of the WinCC OA managers such as the *event manager* (EV), the *database manager* (DB) and the *Control manager* (Ctrl). The EV serves as message router and keeps an in-memory image of the current database values, handles alarms and executes functions on the data points.

Each manager is directly connected to the EV and hence it is possible that two processes modify the same resource without noticing the other one's updates.

Figure 6 schematically displays such a behaviour where `Process1` sets a data point value while `Process2` modified the state meanwhile. This scenario becomes inherently more complex, considering that CERN's control systems consist of hundreds of subsystems, which each resemble the one in Fig. 6 and can access each other's data points via communication through the *Dist managers*. To prevent this, it is possible to ask for a lock on a data point and stop other managers from modifying it.

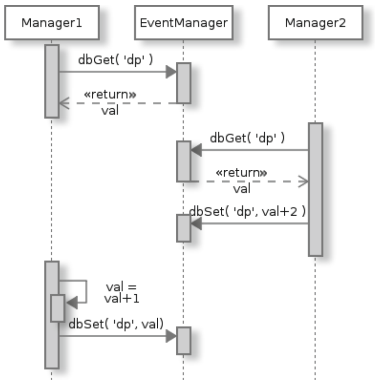

Fig. 6. Process1 overwrites the data point without noticing the changes

For the semi-purification it is essential to detect these situations and avoid race conditions and dirty reads and writes, but also identify when locks are being used.

ITEC has to be capable of generating test cases for these situations, meaning that while single thread unit tests are essential, ITEC should also allow for generating test cases to prevent concurrency issues and race conditions to happen.

As most of these error-situations can only be observed through exceptions and error codes, it is necessary that the test doubles fulfil those needs too.

6 Conclusion

CERN's recent efforts to create a unit testing framework for the proprietary CTRL language opened the door for a modern quality assurance process. However, after over a decade of development, the backlog of source code without automated test coverage makes changes in the execution environment challenging.

To address this problem and increase the testing process resilience, we outlined ITEC, a testing system that has the purpose to automatically generate test cases at a unit level for existing source code. To that end, ITEC bases its efforts on existing *automated test case generation* techniques such as adaptive random testing, combinatorial testing and dynamic symbolic execution.

Unfortunately, a large majority of the techniques previously mentioned are not well suited for practical tests generation. As a matter of fact, dependencies on global variables, external resources and subroutine calls are usually disregarded. Therefore, to overcome these obstacles often encountered in CERN's source codes, we thoroughly and formally presented a novel technique called *semi-purification*, its goals being to strip the *code under test* (CUT) from dependencies that lie outside the considered scope. For that purpose, we first addressed the semi purification of code in the presence of global variables and external resources. We then quickly extended the semi-purification's scope to also take into account subroutine calls as dependencies.

Finally, we succinctly addressed additional problems that often hinder the use of ATCG techniques. Among these, we discussed the pertinent idea of considering system concurrency with unit tests and illustrated the needs to deal with these kind of situations by examining an existing problem over the considered system.

References

1. Qu, X., Robinson, B.: A case study of concolic testing tools and their limitations. In: 2011 International Symposium on Empirical Software Engineering and Measurement, pp. 117–126. IEEE Computer Society, Los Alamitos (2011)
2. King, J.C.: A new approach to program testing. ACM SIGPLAN Not. **10**(6), 228–233 (1975). ACM, New York
3. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Comput. Surv. **43**(2), 11:1–11:29 (2011). ACM, New York
4. Colbourn, C.J.: Combinatorial aspects of covering arrays. In: Le Matematiche, vol. 58, Catania, Italy (2004)
5. Haskell. Functional programming (2014)
6. Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., Mcminn, P.: An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. **86**(8), 1978–2001 (2013)
7. ETM Professional Control: WinCC OA at a glance. Technical report, Siemens AG (2012)
8. ETM Professional Control: Control script language (2015). http://etm.at/index_e.asp?id=2&sb1=54&sb2=118&sb3=&sname=&sid=&seite_id=118. Accessed 18 Apr 2015
9. Sward, R.E., Chamillard, A.T.: Re-engineering global variables in Ada. In: Proceedings of the 2004 ACM SIGAda International Conference on Ada, pp. 29–34. ACM, New York (2003)
10. Sankaranarayanan, H., Kulkarni, P.: Source-to-source refactoring and elimination of global variables in C programs. J. Softw. Eng. Appl. **6**(5), 264–273 (2013)
11. Meszaros, G.: Test double patterns (Chapter 23). In: XUnit Test Patterns: Refactoring Test Code, pp. 521–590. Prentice Hall PTR, Upper Saddle River (2006)
12. Barbey, S., Buchs, D., Péraire, C.: A theory of specification-based testing for object-oriented software. In: Hlawiczka, A., Simoncini, L., Silva, J.G.S. (eds.) EDCC 1996. LNCS, vol. 1150, pp. 303–320. Springer, Heidelberg (1996)

13. Péraire, C., Barbey, S., Buchs, D.: Test selection for object-oriented software based on formal specifications. In: Gries, D., de Roever, W.-P. (eds.) PROCOMET 1998. LNCS (IFIP), pp. 385–403. Springer, New York (1998)
14. Lawrence, D., Buchs, D., Wellig, A.: Using instrumentation for quality assessment of resilient software in embedded systems. In: Majzik, I., Vieira, M. (eds.) SERENE 2014. LNCS, vol. 8785, pp. 139–153. Springer, Heidelberg (2014)