Symbolic simulation for debugging and analysis of REKO models using KLEE

Marta Vicente Romero
2013

Master of Science (120 credits)
Computer Science and Engineering

Luleå University of Technology
Department of Computer science, Electrical and Space engineering
Acknowledgment

The development of this Master’s Thesis has been done during an Erasmus Program in Luleå University of Technology (LTU) and it has been an important experience in my life as it has given me the opportunity to be part of an international technical environment.

First and foremost, I would like to thank my supervisor Per Lindgren and Johan Erikson for their guidance and the effort they have invested in the definition and development of this project. I would also like to show my gratitude for the support that I have received from LTU’s department of Computer Science, Electrical and Space Engineering and from the students involved in related projects such as Martin Rajniak and Alberto Duran.

I would like to specially thank everyone who has contributed in some way to this project with their advice and corrections: Anna Diez, Josep Pigem, Xandru Fernandez, Xavier Borrás, Ernesto Ubieto and Roger Fadurdo.

Last, but definitely not least, I would like to express my gratitude to Luleå exchange community, my University colleagues from Spain and my family for their encouragement and support along the way.
# Table of Contents

Acknowledgment ................................................................................................................................. i  
ABSTRACT ............................................................................................................................................... 3

1. Introduction ........................................................................................................................................ 4
   1.1 Objectives ....................................................................................................................................... 5
   1.2 Methodology ................................................................................................................................... 5
   1.3 Tools and technologies ................................................................................................................... 5
   1.4 Report outline ............................................................................................................................... 6

2. Worst Case Execution Time Analysis ............................................................................................... 6

3. Symbolic execution and KLEE engine ............................................................................................... 7
   3.1 Symbolic Execution ....................................................................................................................... 7
   3.2 KLEE a symbolic execution engine .............................................................................................. 8
      3.2.1 Introduction ............................................................................................................................ 8
      3.2.2 Low-level Virtual Machine (LLVM) ....................................................................................... 10
      3.2.3 Coverage measurement .......................................................................................................... 11
      3.2.4 Testing real software .............................................................................................................. 12
      3.2.5 Error reports .......................................................................................................................... 12
      3.2.6 Other approaches .................................................................................................................... 14
      3.2.7 Other uses of KLEE ............................................................................................................... 15
      3.2.8 Limitations of KLEE .............................................................................................................. 17

4. REKO IDE ........................................................................................................................................... 18
   4.1 Concurrent Reactive Components and Concurrent Reactive Objects ........................................... 18
      4.1.1 Time-Constrained Reactions ............................................................................................... 18
      4.1.2 Concurrent Reactive Objects .............................................................................................. 18
      4.1.3 Concurrent Reactive Components ....................................................................................... 19
      4.1.4 Classes and Instances ............................................................................................................ 19
      4.1.5 Kernel support and interaction with the environment ............................................................. 20
      4.2 REKO IDE utilities ..................................................................................................................... 20
         4.2.1 Graphic representation .......................................................................................................... 21
         4.2.2 Example of a model ............................................................................................................. 22
         4.2.3 XML representation ............................................................................................................. 23
         4.2.4 Code synthesis framework ................................................................................................. 25

5. REKO IDE extension: “Test Cases by Klee” .................................................................................... 28
   5.1 Integration into REKO IDE ........................................................................................................... 29
   5.2 Design and Implementation: “Test Cases by KLEE”. .................................................................. 30
      5.2.1 First stage: KLEE Test Cases Generation ........................................................................... 30
      5.2.2 Second stage: REKO Test Cases Generation ...................................................................... 32

6. Results ................................................................................................................................................. 34
   6.1 Symbolic execution applied to WCET analysis ............................................................................ 37

7. Limitations ........................................................................................................................................... 38

8. Conclusions and Future Work ......................................................................................................... 40

9. REFERENCES ..................................................................................................................................... 41
ABSTRACT

The verification of embedded software plays an important role in the design process of dependable and/or real-time systems. Testing remains an important means to validate desired system operation and offers accurate results (as we are examining the actual system and not a model thereof). However, coming up with test cases that stress possible errors is a difficult and time-consuming task. To this end, automatic test case generation offers an attractive solution.

This thesis is based on the expansion of the REKO models validation, being REKO a component-based software which supports building system models for resource-constrained embedded systems based on Concurrent Reactive Objects and Components. The use of KLEE has been studied in order to generate test cases revealing both common errors (arithmetic and pointer dereferencing) and giving high code coverage (exciting feasible execution paths) in the existing REKO models.

Furthermore, framework has been developed which automatically produce test that can be used to perform execution time measurements based on the KLEE tests.
1. Introduction

Embedded systems (ES) are purpose built computers which form part of larger systems [38] and are usually subject to physical constraints. When it comes to a real-time embedded system, they should react in real time to the changes of the environment. The integrity and correctness of the system does not only depend on functional requirements but also on non functional requirements such as timing and resource constraints [40].

Their increasing popularity results in their proliferation in daily life, even though people are not aware of their dependence on them. When it comes to safety-critical systems the effect of an error is usually extremely significant. A failure can cause damage and/or injuries, e.g., in the automotive industry where it can be found ABS (Anti-lock Braking System) and ESP (Electronic Stability Program) systems as well as in other areas such as aerospace, railway, military or medical devices. Therefore, the correctness of embedded systems operations is crucial.

The development of new and more complex features in today’s embedded systems brings challenges to design and verification. Extensive research has been done in order to obtain consistent design methodologies, especially with regard to hardware and software models [39].

One recent approach to component based design of embedded real-time software is the REKO framework developed at EISLAB Luleå University of Technology (LTU). The REKO tool suit is further described in the PhD thesis [42]: “A Reactive Approach to Component-Based Design of Resource-Constrained Embedded Systems” by Jimmie Wiklander.

The Thesis [42] is based on recent research at EISLAB in the area of component-based modelling and design for software running on resource constrained embedded systems. The REKO tool features an Integrated Development Environment (IDE), providing a graphical representation and construction of system models. Furthermore, REKO features a compiler utility that generates C-code from the model. The resulting C-code can be compiled together with a target specific run-time system into an executable (either bare metal or hosted dependent on the target). In addition, the tool provides built in static analysis of REKO models. In particular analysis for Stack Resource Policy (SRP) based scheduling is provided, resulting in a static priority assignment allowing pre-emptive execution on a shared stack (allowing for simple, hardware supported and memory efficient scheduling). Furthermore, models passing SRP analysis are ensured to be deadlock free during run-time.

Model testing and verification is an expensive, time-consuming task, especially on large systems. That is why investing in the testing methodologies and the automation of the process is so important. Software fault localisation in early phases of development reduces the final costs of the product [10].

This project’s main objective is the automatic analysis of the REKO models in terms of the validation of the C-code generated. To this end, a new utility has been developed and integrated into REKO tool. The new framework is based on the use of symbolic execution tools to automate the discovery of errors in the REKO models. In order to understand the symbolic execution possibilities, research on different tools based on symbolic execution and a more thorough review of KLEE, a symbolic execution engine, has been carried out.
KLEE has been integrated into the REKO IDE to automatically provide test cases and feed the results back into the model. In the context of REKO, the output of KLEE is a set of concrete state variable assignments and method parameter settings that trigger different feasible paths, errors and assert violations. Hence, the information derived from KLEE can be put to different type of usage, e.g., the feasible paths as input for code coverage, and execution time estimation, the error assignments can be used to track bugs in the program and the assert violations can be used to track violations to user defined criteria (such as pre and post conditions, etc.).

### 1.1 Objectives

The overall goal of this Master’s Thesis is the integration of symbolic execution to analyse REKO IDE generated models.

In order to achieve this, a new framework for REKO has been developed and it enables the use of KLEE to analyse its models. The specific objectives are outlined below:

- Automatic test case generation with KLEE of the REKO models (states and methods parameters).
- Automatic generation of new test ready to be executed and containing the KLEE test cases values.

### 1.2 Methodology

The work process has been separated into two main parts:

The first one is a researching phase in which KLEE and REKO were analysed. The review of the state-of-the-art of symbolic execution engines, the study of the applications of KLEE and the possible contributions to REKO model analysis were the topics in which the research has been focused. In most of the cases, information was obtained from scientific articles and PhD thesis, which are referenced at the end of the document.

The second one is the implementation of the framework that makes the integration of KLEE into REKO possible and produces tests with the values generated by KLEE.

To this aim fortnightly meetings have been held to follow up the ongoing work and to define the way to follow. Additionally, the information about the previous work carried out in this area at LTU was provided in different meetings and presentations.

### 1.3 Tools and technologies

The following frameworks and tools are used for the development and research of the present Master’s Thesis.
• REKO

The IDE developed at LTU, REKO is a tool for developing component-based software which is able to build system models for resource-constrained embedded systems graphically. It is based on Concurrent Reactive Components and Concurrent Reactive Objects. C-code synthesis is an included utility in the IDE that provides the opportunity to obtain executables that can be run on a target platform. REKO has been implemented in Java using the software development environment Eclipse. The same technologies are used in the development of the new REKO utilities.

• KLEE and LLVM

The use of KLEE needs to compile the program into LLVM bitcode. The LLVM version used is: 2.8 with llvm-gcc4.2.

The current project uses KLEE release: 165988. The symbolic engine has been used as an auxiliary application. Therefore, it preserves its command line and its basic behaviour. It would be compatible with the developed framework developed during the present project while KLEE preserves its original output format.

1.4 Report outline

Current research is outlined in this paper starting with the description of Worst Case Execution Time Analysis and its classification. The following section reviews the concepts of symbolic execution and KLEE. The details of the REKO IDE are explained in Section 4. Furthermore, Section 5 describes in detail the new functionality: “Test cases by KLEE”. At the end the results are considered and the work is evaluated in the subsequent conclusion.

2. Worst Case Execution Time Analysis

The development of safety-critical real-time systems should introduce tools to improve the verification in the software development process, as was explained in the introduction. The verification of the execution time is an important analysis to meet time requirements associated to resource-constrained systems.

The Worst Case Execution Time (WCET) is the longest time that a program, task or small code section spends in its execution on a specific platform. Obtaining precise measures is difficult due to several factors such as not only the processor complexity or timing anomalies but also the difficulty of obtaining the necessary accuracy [WCET2]. The level of accuracy is flexible depending on the needed level of safety. The time requirements of each system critical part should be complied as the only way to obtain a correct behaviour of the full system [49].

Due to the difficulties of obtaining precise WCET estimates, WCET analysis has been a large research field which has traditionally been divided into static program analysis and dynamic analysis [52, 1, 50].
• **Static Analysis:** This method is based on free-of-execution procedures where a formal verification is done using different techniques. Therefore, a control-flow path analysis can be combined with a hardware model which has usually a high complexity to be designed, among others techniques that can also be used to improve the results.

• **Dynamic Analysis:** It is a measurement-based analysis. A set of inputs is executed on a specific target platform or a simulator. The WCET estimate is the longest time recorded during the experiments. The inputs cannot usually cover all the possibilities in the execution, thus, it can only represent an estimation of the real WCET. The measures are not considered safe bounds because demonstrating that the worst case has been found among the inputs used for the analysis is hard without making any other analysis.

One way to enrich the measurement-based results is to improve the set of test cases. The more paths on the code the inputs cover, the more accurate the results will be. For this reason the use of symbolic execution is explained in terms of improving the accuracy of the results through the quality of the test suites.

### 3. Symbolic execution and KLEE engine

#### 3.1 Symbolic Execution

Symbolic execution is a program analysis technique whose development began in the 70's. At the beginning, there were no precise or fast enough solvers that could provide effectiveness to this technique. However, in recent years, an important effort has been made to develop and research symbolic execution tools.

Software testing is still an important part of the design process of embedded software and, therefore, significant efforts have been devoted to its improvement and development [10]. Working with hand-written test suites is a time-consuming task and in general terms cannot obtain a high coverage. Besides, they do not trigger potential errors. This is why automating testing processes and increasing coverage are often the goals pursued in many investigations [5].

The key idea of the symbolic execution [9] is to perform a program with symbolic inputs instead of concrete values. The system will create a path condition including a set of constraints that will be modified whenever it finds a new branch point. The result is that every possible execution path of the program will be explored. The execution could be represented as a tree in which each node is a branch in the program. In the case of KLEE, the specific testing tool explored in the current project, a constraint solver is invoked to generate concrete values when a path terminates or a dangerous operation is reached. These concrete inputs are used as the generated test cases and can be used to reproduce the execution path. One of the most important applications is to be able to reproduce bugs and it can be used to build test suites.
3.2 KLEE a symbolic execution engine

3.2.1 Introduction

KLEE is a symbolic execution based tool which mainly aims to generate automatic tests. These tests achieve high coverage on complex programs with few test cases. KLEE belongs to a group of tools considered Dynamic Symbolic Execution (DSE) tools that combine concrete and symbolic execution [5] (They are also considered as concolic tools).

KLEE uses a symbolic input which can have any value instead of random or manual values. This symbolic input is built as a constraint, which is added when a new branch condition is reached. In addition, whenever a dangerous operation is found, KLEE introduces additional checks to trigger possible errors. It uses two kinds of memory locations: symbolic and concrete locations.

In order to provide symbolic locations, the code must be changed to introduce the call: `klee_make_symbolic(&a, size(a), "a")` which notifies to KLEE the address and size of the memory positions which will be later analysed as symbolic. The symbolic locations are related to the expressions that save the conditions of the path. These conditions will be later analysed with the constraint solver that in this case is STP [20]. Regarding the high cost involved in these operations, the tool tries to avoid calls to the STP by saving in cache the results previously obtained. The concrete locations contain concrete values to be used as a conventional execution in any program. Thus, in case that no variable is marked as symbolic, the program is executed in a conventional way.

From this perspective, one of KLEE limitations could be that it can only interpret about branches in the executed code. It does not explore dead code or unreachable code and it cannot produce test cases of these parts of the code. This is a common characteristic in a wide number of symbolic execution tools.

EXE [7] was an earlier tool that was redesigned with the KLEE development. The use of EXE required more resources than the new KLEE design. As a result, KLEE is able to handle a larger number of concurrent states due to the representation of the states (symbolic process). This is one of the main changes in the architecture of the tool, but more optimizations have been introduce in comparison with EXE, such as query optimization through the simplification of the queries that will be sent to the solver among others [1].

The user has a C program that is compiled to LLVM bitcode. First of all the symbolic inputs are marked in the code. Then, the file is executed by KLEE. Symbolic files and symbolic arguments can be introduced from the environment and different options can be used to control some aspects of the execution as well as the different outputs. KLEE generates the following output files as default:

- info: General information about the executions contained in this file such as total completed paths, total number of instructions explored, number of generated test or the execution line among others.

- testN.ktest: This file contains the data of the symbolic variables, including the arguments used in the execution of KLEE, number of objects that have been made symbolic, and for each one the name associated to the symbolic memory, the size and the data obtained by KLEE.
testN.type.ktest: A file related to the ktest ordinary file is produced to show the kind of error that has been found. The type is specified depending on the kind of error.

messages.txt: This file provides information about execution issues, such as the notification of external calls.

warnings.txt: Notification of errors and other warnings are shown in this file.

run.istats: Statistics about the execution of each line are included in this file and can be visualise via KCachegrind.

run.stast: Statistics about the execution are saved in this file. The information can be visualised with the tool: klee-stats. The information that will be displayed is the time spent in producing the test suite, the branch coverage and the percentage of LLVM instructions that were covered in the execution.

The following tools that allow carrying out a complete analysis are integrated in KLEE:

ktest-tool: This tool is used to visualise the test resulting from the execution of KLEE. It will visualise the files with the extension “.ktest” which contain the generated values. This tool allows visualising the 4-bytes data as integer representation but it cannot to differentiate the components of a structure displaying the data as a buffer.

ktest-replay: This tool is used to reproduce test cases generated by KLEE.

klee-stats: This utility is used to display the statistical information that KLEE is able to provide, as it was mentioned before. The most significant information is: number of executed instructions, total wall time (s), total user time, instruction coverage in the LLVM bitcode (%), branch coverage in the LLVM bitcode (%), time spent in the constraint solver among others. All the options and details of the information can be consulted by the help option in the command-line.

In order to understand how KLEE is able to analyse the code (Figure 1), Figure 2 shows an example which explains the methodology used by the software in order to easily follow the paths from the code.

```c
#include <stdio.h>
int example(int a, int b){
    if(a>0){
        return a/b;
    }
    else
        return a;
}
int main(){
    int a, b;
    klee_make_symbolic(&a, sizeof(a), "a");
    klee_make_symbolic(&b, sizeof(b), "b");
    return example(a,b);
}
```

*Figure 1: Simple C-code example using KLEE*
Figure 2: Path condition tree of the example in Figure 1

This example shows how the path is followed to generate test cases. Firstly, the code is modified to mark the symbolic variables as aforementioned. In this case, the modification is applied on variables “a” and “b”.

Once KLEE is run, it provides 3 test cases that are shown in Figure 2. Following the path, the values which the function inputs will get can be seen. Furthermore, an illegal operation is reached since it involves a possible division by zero. This error appears clearly in this case. However, these errors can be hidden in other complex programs where KLEE is also able to find them, as well as other types of errors.

There are several operations that KLEE can consider dangerous such as instructions that generate division by zero, buffer overflow and null pointer dereferences which will be discussed later on (see Section 3.2.5).

However, the use of KLEE has more applications than to detect bugs. The generated test cases can be used as part of test suites. Additionally, even if the tool cannot analyse unreachable code, the produced test suite helps to find it. The general procedure is explained in Section 3.2.3, and it is applicable to the test cases explained through the report.

3.2.2 Low-level Virtual Machine (LLVM)

LLVM is a framework used to develop compilers. LLVM is a set of compiler and toolchain technologies designed to be modular and reusable features [Error! Reference source not found.]. It can be used to optimise compilation time for any, conventional or research, programming language that the user wants to define. The successful results obtained have generated a variety of front-ends, including C and C++.

Nowadays, several projects using LLVM can be found on their home page. Among all projects, KLEE comprises a symbolic virtual machine built on top of LLVM compiler for GNU C. A C-code program is compiled into LLVM bitcode before is interpreted by KLEE. Specifically, it is used the llvm-gcc which corresponds to the LLVM front-end [12].
3.2.3 Coverage measurement

An important issue in software testing is the coverage. Coverage can be considered as an effective measure of test suites. It provides information about the reached branches by the test suite or the percentage of executed code among others. Therefore, this information is used to evaluate test suites completeness.

KLEE produces its own information regarding covered code such as other factors. It is displayed as a result of the execution of the klee-stats tool (see Section 3.2.1). What is more, auxiliary information is located in the file run.istast and can be visualised via KCachegrind. KCachegrind [34] is a complex visualization tool for profile data which provides number of executed instructions and information about the run code. A call graph of a program is also generated where it shows how the functions call each other.

However, in order to maintain autonomy between the tool and the coverage results KLEE developers use gcov [14] to determine the coverage on their tests and to avoid the effect of potential bugs on KLEE [8]. Gcov is the provided test coverage tool of GCC and is used to determine how the code is exercised. Consequently, the analysis of a test suite via gcov helps to discover which areas remain untested, where unreachable code is located and thus which parts of the program are potentially improvable.

This tool generates coverage information in terms of basic block coverage. It is a criteria which is similar to statement coverage (or line coverage), where a basic block is considered a set of sequential statements without any branch [36]. The tool analyses the statements that have been executed, and it is also able to show the coverage per branch.

The first step is to indicate to the compiler the necessary options to produce the instrumentation of the compiled code. Two files are generated for each source file with executable code. One is named with the extension “.gcno” and contains information regarding the program’s flow arcs. The other one is named with the extension “.gcna”. This file is created when the application is run and it contains information about the execution paths. This file records the information for each “execution of the program with the different inputs from the different test cases.

Finally, an annotated source file is generated containing the number of times that the lines have been executed and which lines have never been executed. At the end of the session, this file shows how the code has been explored as it accumulates the data from the first to the last execution.

The use of lcov, a front-end of gcov, is one way to visualise data from different executions. Lcov collects coverage data from gcov and needs to use an html file to the visualisation. This html file is provided by using genhtml. Code parts that are never reached by the test suite are shown by colours. Therefore, a direct identification of the areas is done.

While gcov needs the execution of each test case to obtain the information, KCachegrind uses the information that KLEE directly generates. However, KLEE developers preferred gcov that is a reliable and known tool that does not generate any controversy in the results since does not depend on the KLEE statics results.
The kind of coverage measure used in the experiments is line coverage (also known as statement coverage). The advantage of using gcov is that it ensures the independence of the tool for obtaining reliable coverage results.

However, this criteria underestimates the coverage of the test produced by KLEE because path coverage would obtain better results than line coverage. Path coverage measures all different paths (as a unique sequence of branches) with all possible values enclosed in the code while line coverage is a weaker measurement than path coverage and failures can go undetected even with 100% coverage [41]. Moreover, the coverage is lower when external library code is analysed as the general code in the library contains dead code.

3.2.4 Testing real software

KLEE has become a helpful instrument to detect new bugs in real software. GNU COREUTILS tools, BUSYBOX, and the HISTART kernel are examples of the analysed software. Therefore, the efficiency of the tool has been intensely tested and the author of the OSDI’08 article [8] claims that KLEE is a reliable tool. It was run on more than 89 COREUTILS utilities obtaining rates of coverage over 90% and higher in other applications.

It has to be pointed out that although COREUTILS is a really tested software by the developers team and it has been installed on millions of computers, KLEE was able to find 10 new bugs. The most common errors found by KLEE in the previously mentioned software were generally memory error crashes.

3.2.5 Error reports

This section provides a detailed description of the above mentioned errors that can be found when using KLEE. For a better understanding of these errors, some examples have been added. Whenever a bug is reached, a test is generated triggering the program crash. The kind of error produced is specified in the file warnings.txt and also in the name of a generated file. For each detected error, the most important output files are: one ordinary test file called: testN.ktest and a file that specifies the kind of error with the format: testN.type.err (with “type” depending on the type of error).

- Divide by zero:

It has been previously shown the way in which KLEE is able to reach this kind of bugs. KLEE generates the combination of values that forces the program to execute the division by zero when it depends on symbolic variables (see Figure 1). The generated file will be named: testN.div.err.

- Pointer errors:

Memory errors are possible to detect and they are considered by KLEE as out of bounds. These errors can be null pointer dereferences, memory allocation mistakes or buffer overflows.
Figure 3: Example of memory shows a small but clear example in which KLEE reaches and generates a test case that reproduces the error. The test might produce a segmentation fault when the program is executed with these values. In the example (Figure 3: Example of memory), two test cases are generated, but only one of them will be a crash. When “a” takes the value 0 and the program tries to assign a value to the pointer, it is detected that this pointer has not been initialised correctly.

It is interesting to remark that even if symbolic variables are not used, KLEE will detect this kind of bugs. The generated file is called as: testN.ptr.err.

```
int main(){
    int a;
    klee_make_symbolic(&a, sizeof(a), "a");
    int* p = NULL;
    if(a==0)
        *p=a;
    else
        a--;
}
```

Figure 3: Example of memory error

- Improper calls to free():

A no proper free call is other error that can be reported. Errors can be detected in different cases such as double free(...) calls or mistaken memory positions liberation such as to deallocate space in memory for: a primitive data type variable, a pointer without any assigned space or static memory.

The example in Figure 4: Example of inappropriate use of free() shows an incorrect use of free() where static memory is forced to deallocate. In this case, one error is reported as: “free of global”. The generated file will be named: testN.free.err.

```
int array[]={1,2,3};
int main(){
    klee_make_symbolic(&a, sizeof(a), "a");
    if(a==0)
        free (array);
}
```

Figure 4: Example of inappropriate use of free()

- Assertions

It is possible to introduce KLEE assertions by including the library “klee.h”. The tool produces an error file for the cases that do not fulfil the assertion.

Figure 5 shows a simple example which produces two test cases. The first one is a negative value that makes the assertion fail. An error file is generated in this case. The other file contains a value that makes the assumption of the assertion true.
```c
#include <klee/klee.h>

int main()
{
    int a;
    int * ptr ;
    klee_make_symbolic(&a, sizeof(a), "a");
    klee_assert(a>=0);
    return 0;
}
```

Figure 5: Example of using assertions

### 3.2.6 Other approaches

#### 3.2.6.1 DART: Directed Automated Random Testing [15, 48]

This tool, which combines random inputs with symbolic execution, is enclosed in the Dynamic Symbolic Execution (DSE) tools group. While the risk of using only random inputs is the difficulty to test all the branches of the code, the approach introduced by DART is to provide a combination of random inputs and directed search by using concrete and symbolic execution.

At the beginning, a random value is generated in order to initialise each external variable. A random value is returned for each call to an external function. A difficult task for random inputs is to obtain values which reach every branch in a program. Therefore, random testing has not a high coverage. However, DART combines this random values with symbolic execution. After the initialisations with random values, the variables are symbolic. A path constraint is built by collecting constraints when a new branch is found. Then the next path is obtained by negating one constraint to follow other branch. If a path cannot be solved because of the constraint solver the constraint will be simplified by assigning concrete values.

However, compared to KLEE, DART has some limitations. For example, it cannot deal with dereference pointers or explore multiplications involving more than one non-constant value. Nevertheless, part of the relevancy of this tool is that it is the precursor to CUTE and JCUTE.

#### 3.2.6.2 CUTE and JCute: Concolic unit testing and explicit path model-checking tools [6, 16, 17]

CUTE is defined as a Concolic tool. Concolic tools refer to the combination of both symbolic and concrete execution. DART is considered the closest work to CUTE by its authors. CUTE generates test cases for sequential C-code programs. CUTE allows the user to determine which inputs will be symbolic by specifying them in the code. This option can be used to replace any external user input [16]. The code is instrumented to enable CUTE to explore the execution paths. The main improvement of CUTE over DART is that CUTE can handle pointer operations and data structures.
On the other hand, the implementation of JCUTE, a tool used to generate test cases for JAVA code, allows dealing with concurrent programs. It is able to generate concrete inputs and different schedules which specify the order or thread executions by exploring partial orders of concurrent events. The research comes up with two algorithms: race-detection and flipping algorithm (further explained in [44] and [45]), which allow JCUTE to produce different schedules. These schedules make the detection of errors (such as data races) and deadlocks possible. However, the tool may miss the detection of some potential errors although it avoids the problem of false warnings (a further discussion can be found in Thesis [45]).

### 3.2.6.3 BitBlaze: A New Approach to Computer Security via Binary Analysis [18, 19]

BitBlaze Analysis Platform project is constituted by different components with different goals trying to build a system that can be extended. The components are called: Vine, TEMU and Rudder.

Vine provides static analysis that supplies the translation into a formal specified intermediate language (IL), and different utilities for static analysis. As a component for dynamic analysis, TEMU is developed based on QEMU as a system emulator. It is able to extract semantic information such as process, threads and symbol information of the OS-level. The last component is working as a plugin of TEMU and is the part responsible of combining concrete and symbolic execution. This last component is called Rudder and it is able symbolically to run programs with symbolic variables, using STP as a query solver.

The aim of the platform is to offer support to solve different security problems with new approaches such as “vulnerability detection, diagnosis and defence”. The new technique: Loop-extended symbolic execution (LESE) [19] is an example of how the infrastructure offered by BitBlaze is used to develop new approaches. In this case, LESE is a symbolic execution tool which increases the coverage in programs with loops, allowing the scanning of the loops with different number of iterations by symbolic variables.

### 3.2.6.4 Other tools developed for industrial use [02, 21, 22, 23, 24]

There are other tools that should be taken into account such as some symbolic execution applications developed in industrial environments. For instance, certain applications developed by Microsoft like SAGE (Scalable, Automated, Guided Execution) [21], PREFIX [22] or PEX [23], which are used to generate test suit cases automatically for .NET, or various tools developed by Fujitsu, such as Symbolic PathFinder [24].

### 3.2.7 Other uses of KLEE

Tools based on KLEE can be found as a result of later developments that based their construction or their techniques on KLEE properties, or new extension of this software. The most representative tools are explained in this section.
3.2.7.1 EDS: Execution Synthesis [25]

A prototype of the Execution Synthesis technique was developed to debug C programs automatically. Its goal is to help fixing bugs by providing an explanation which contains enough information to find the cause of the error. It is focused on solving concurrent program bugs. A program and a bug report are the inputs required for the tool. The user will obtain an execution of the program that reaches the reported bug. This technique consists of two parts. The first one is used to find a feasible path to explain the error. The second searches schedules symbolically in case of multi-threaded programs. It uses heuristic techniques which are selected according to the error reported, deadlocks or data races. The execution information provided by EDS can be analysed with a debugger (such as gdb) to fix the problem more easily.

EDS extends KLEE in different ways. However, the most relevant improvement is that it supports multithreaded symbolic execution. Its use is recommended “for debugging long-running, performance-sensitive software, like Web servers, database systems, e-mail servers, application servers, game servers”, as explained by Cristian Zamfir and George Candea [25], because the programmer does not need to instrument the user code.

3.2.7.2 KLEENET [26, 27]

This approach has extended KLEE providing a debugging environment to generate distributed execution paths. Bugs are reached through unmodified sensor network applications, which are executed with symbolic inputs. The different inputs obtained (such as loss, duplication and corruption packets, or node failures) are automatically generated by driving the sensor to corner cases.

The main aim is finding vulnerabilities in distributed systems before the deployment and to facilitate the replication, which is not a trivial task in these systems. The effectiveness of the tool has been shown by finding bugs in n Contiki’s μIP protocol stack.

3.2.7.3 Cloud9 [28, 46, 47]

Cloud9 parallelises symbolic execution on large shared-nothing clusters. A program is symbolically executed in parallel on a cluster. This tool inherits KLEE capabilities such as recognising memory errors and failed assertions. Furthermore, the parallel symbolic execution prevents CPU and memory resources limitations.

In addition, this symbolic execution engine builds a symbolic model extending the symbolic file system model (already implemented in KLEE) to allow the tool to operate with multi-threaded software and features such as threads or sockets. What is more, Cloud9 provide a symbolic scheduler which can be used to explore and test exhaustively all the possible scheduling decisions [28].

The tool has been designed as a Web Service with the objective of providing an automated testing platform where real systems can be check against errors.
3.2.7.4 KLOVER [29]

Klover is a KLEE-extension tool developed to automatically generate test cases in industrial C++ programs. In this case, it extends KLEE to support the C++ language, thus supporting new features such as templates, exception handling and class inheritance. Therefore, the new tool is built on top of KLEE optimising uClib++ library to obtain the results for C++. The most important optimisations have been: minimising the number of generated paths, translating expressions to reduce the cost and accelerating the library's decision procedures.

3.2.7.5 Selective Symbolic Execution [30]

The aim of this approach is to apply symbolic execution to small portions of a system (considered as a set of applications, libraries, kernel, etc.) allowing users to specify the scope of the tests. A prototype was developed based on the technique presented by Selective Symbolic Execution using QEMU [31] and KLEE engines.

While KLEE can only interact with the environment through the symbolic file system, as it was exposed before, the new approach simulates that complex programs run in a full symbolic environment. A system is usually divided into small modules to be tested in separate stages. Therefore, the tool allows the user to specify which part of the system will be tested. This tool combines the symbolic execution of the identified part of the system and the concrete execution of the part which is out of the scope. QEMU is modified in order to use the symbolic states of KLEE and the responsible for the execution of the part that will run in a concrete way. This approach allows analysing big systems by overcoming previous tools' limitations related to complex environment interactions.

3.2.8 Limitations of KLEE

There are limitations to the application of KLEE, even if it is used for academic or industrial uses. Some of them are inherent to symbolic execution.

The path explosion problem is a common characteristic in symbolic execution based tools. The number of paths grows exponentially even in small programs and become unmanageable. KLEE has different approaches to decrease the impact of the path size. KLEE allows not only the use of different heuristics but also the research in different approaches of prunes redundant paths [11]. Nevertheless, this issue would not represent a drawback in our purpose, since it is focused into simple systems.

On the one hand, the common bottle neck in this kind of software is the constraint solver. It is not able to handle very complex constraints. Specifically, the main efforts in the development of KLEE lie in the simplification of the constraint queries and the minimization of the calls to the STP solver.

On the other hand, the code requires modification in order to be analysed symbolically. As it was explained in Section 3.2.1, it is necessary to mark which variables will be symbolic to KLEE in order to run the program symbolically. However, the directives that should be introduced are not a big issue in comparison with the annotation of other analysis tools such as the annotations of static analysis engines in WCET.

According to specific code, KLEE is not able to run loops with symbolic conditions, and in a similar way, it is not possible for it to analyse objects with symbolic sizes.
4. REKO IDE

REKO is a component-based software which is able to build system models for resource-constrained embedded systems graphically. It is based on Concurrent Reactive Components and Concurrent Reactive Objects. C code synthesis is an included utility in the IDE that provides the opportunity to obtain executables that can be run from a target platform.

In this section, all know-how concepts, proceedings and tools included in REKO are thoroughly described with the aim of providing the reader with a good understanding of the IDE.

4.1 Concurrent Reactive Components and Concurrent Reactive Objects

This section highlight the key aspects of time-constrained reactions, reactive objects, their abstraction to hierarchical concurrent reactive components, and discuss modeling interaction between components/objects as well as a system's interaction with its environment.

4.1.1 Time-Constrained Reactions

Time-constrained reactions lie at the heart of the model. Interaction between the system and its environment, as well as between components of the system is modeled as discrete events occurring at specific times. Following a reactive approach, functionality is specified in terms of time-constrained reactions to such events. Timing requirements on system operation can be specified by defining the earliest and the latest reaction time (baseline and deadline) relative to the time of the event triggering the reaction. The time window between the reaction baseline and its deadline is called a permissible execution window for the reaction (Figure 6).

![Figure 6: A permissible execution window for a reaction to an event. Here $t_{after}$ is the baseline offset, $t_{before}$ is the time between the baseline and the deadline. (Figure from [41])](image)

4.1.2 Concurrent Reactive Objects

The fundamental modeling construct of the framework is a concurrent reactive object (CRO). Each object has a state and one or several methods, and it is reactive in the sense that it reacts to an incoming event by executing one of its methods. Thus a CRO is either idle (maintaining its state) or executes a method.
Methods execute run-to-end, that is, once a method has started execution, no other method of the same object may preempt it. However, any two methods of different objects may execute concurrently. A method’s code can perform computations on local variables, read/mutate the object’s state, and invoke a method of the same or another object by sending a message to it.

Methods execution can be of two kinds (which corresponds to two kinds of messages sent between objects): asynchronous, which are executed concurrently with the caller and can be delayed by a certain amount of time, and synchronous, with the caller blocked until the invoked method completes execution, optionally returning a value (such methods cannot be delayed). The permissible execution window of an asynchronous message can be either inherited or explicitly specified in the code relatively to the caller’s baseline; in either case, it is viewed in the model as a separate reaction. A synchronous message, on the other hand, always inherits the caller’s time constraints and is viewed as a part of the original time-constrained reaction. Both asynchronous and synchronous messages can carry data (the values of arguments of the invoked methods).

Each CRO has a provided interface (the methods, or input ports, of the object that can be invoked by other objects) and a required interface (the methods, or output ports, in the object’s environment that it may invoke). In the model, an output port in the required interface of an object can be linked to an input port in the provided interface of another object, creating a communication path between two different CROs. Multiple output ports can be connected to a single input port.

4.1.3 Concurrent Reactive Components

Concurrent reactive objects create a flat, non-hierarchical structure of a system. To support efficient development of complex software system, REKO supports a component based design methodology based on the notion of concurrent reactive components (CRC), which contain no own state or methods but instead encapsulate a number of objects and other components, creating a simple hierarchical structure. Like objects, components have provided and required interfaces, but the ports in the interfaces are connected not to methods, but to ports in the interfaces of inner objects or components. Thus communication across component boundary is in fact communication between two objects belonging to two different components, which allows for an efficient implementation where a component hierarchy has no overhead at run-time.

It is worth noting that while a CRO can only execute one method at a time with concurrency existing between different objects, a CRC can be concurrent in itself as it may encapsulate multiple objects.

4.1.4 Classes and Instances

Every instance of a CRO/CRC belongs to a class, which can be defined at the top level in a module or locally within another class; in the latter case, it can only be used inside that class. A CRO definition defines the object’s state variables, methods (these are written manually in C and the tool verifies that the C-code complies with the model’s requirements as outlined below), and the provided and required interfaces of the object.
In the C-code of a method we may:

1. Define local variables

2. Read/update the object’s state

3. Perform calculations on the object’s state, method argument, and local variables

4. Invoke an asynchronous method of this object or of another object using the name of an output port.

   \texttt{ASYNC(PortName, BaselineOffset, RelativeDeadline, Argument)}

5. Invoke a synchronous method of another object using the name of an output port:

   \texttt{SYNC(PortName, Argument)}

6. Return a value (only if the method is synchronous). A CRC definition defines the provided and required interfaces and specifies instances of CROs and CRCs encapsulated in the component.

### 4.1.5 Kernel support and interaction with the environment

A system model (i.e. the system’s structure in terms of CROs and communication paths between them, and C-code written for each method) can be used to generate C-code for the application. Execution on a hardware platform also requires infrastructure supporting the model in the form of a lightweight kernel supporting scheduling of method execution and message passing between objects.

Embedded systems interact with their environment and this must be reflected in the model. This is modeled using a special construct, an environment interface component that defines an interface to the hardware. An instance of such interface component enables reading from hardware registers and writing to them and is also used to connect specific hardware interrupts to asynchronous methods of particular objects.

### 4.2 REKO IDE utilities

REKO provides a graphical user interface (GUI) where CRC/CRO models can be designed and represented graphically. Actions such as creating, browsing and editing models can be performed by the IDE. The XML representation and the Code Synthesis framework form part of the IDE utilities. A good understanding of the code's specific features is needed to follow the new implementation tool description. This is why they are also described in this section.

An example is provided to illustrate a model design and explaining how the tool is actually used. The model design has the minimum number of components to introduce all of the possible elements that it can contain.
4.2.1 Graphic representation

REKO has a graphic user interface that makes the creation of reusable components easier. All components should follow the restrictions described in Section 4.1.

Different colours are used to distinguish each kind of component. Orange is used to define entry points, blue is the colour assigned to CRC, and CRO are marked in yellow (see Figure 7). Components are represented by boxes and contain the names at the top. The provided and required interfaces are located on the right and left sides of the box, in a separated list of boxes.

![Figure 7: Different kinds of elements in the model](image)

REKO's main screen is divided in three different sections (Figure 8):

On the left section, the hierarchy of the created components is displayed.

![Figure 8: Different sections of the graphic interface](image)

On the right section, the components which correspond to the one selected in the left section can be seen. In this section, it is possible to create and delete components as well as to create new methods or access the details of the defined elements.

A secondary view shows the details of the components (see Figure 9). In this view, several options can be defined depending on the nature of the component. Both state and methods can be defined for CROs and only methods can be defined for entry points. However, this view is not available for CRCs as they are not made up by any of these elements.
Finally, in the middle section (see Figure 10), instances of the different components can be created. Furthermore, the relation between them can also be defined (following the requirements specified in Section 4.1). The relation between the instances is created by connecting output ports with input ports which represent the required and provided interface.

4.2.2 Example of a model

In this section, a detailed example is provided to explain the XML and C code generated by the tool.

First, the module is defined and it contains all the remaining components that comprise the model. In the following example, it is called: “mod”. Figure 7 contains all the components defined for this example. The hierarchy of the model is built depending on the kind of components defined.
A CRC called “container” encloses the instances of the components displayed in Figure 11: Components belonging to the CRC called “container” Figure 11.

The component called “highest_comp” includes two CROs as is shown in Figure 12. In this case, in order to make the example easier to understand, the ports have been called similarly to the methods to which the ports are related.

The CRO “access_start” has one method called “start”, whose code is shown in Figure 13. This example contains a synchronous call that shows how the connections between the components are made. On the other hand, the component “access_data” has a defined state and two methods which are used throughout the output ports “data_set” and “data_get” (See Figure 12).

```c
{ 
    SYNC(data_set,10); 
    int result=SYNC(data_get,0); 
}
```

**Figure 13: Method enclosed in “access_start”**

### 4.2.3 XML representation

XML is the format chosen for the model storage. The model created by the framework is recorded as an XML file. In the next paragraphs, the possible tags are explained throughout the example. The tool can be used to open, edit and create new files with the schema specified bellow.

The file header contains the root name and the working directory, which are considered the configuration features.

Then, a name is given to the module that contains the rest of the model which is being explained herein.
The CRO and CRC definitions are constituted by classes which can contain: states, methods, or instances of other classes.

The definition of the class is made up by the following attributes:

- **name**: name of the class.
- **type**: it can take on the following values: *obj*, *envint*, or *comp*.
  - "*obj*" indicates that the class is a CRO
  - "*envint*" indicates the required interface ports corresponding to the environment.
  - "*comp*" indicates that the class is a CRC.
- **arg**: it specifies the required interface as a list of output ports.

The tag `<result>` contains the attribute "*con*":

- **con**: it specifies the provided interface as a list of input ports.

Depending on whether the class represents a CRO or a CRC, it will contain different tags or attributes according to the restrictions explained in Section 4.1.

A class definition for a CRO can contain: one state and one or several methods.

The state is included as C-code definitions of variables. However it is not compulsory to have it.

Methods enclose the functionality of the system and enforce the restrictions explained by the CRO model (see Section 4.1).
The attributes for the tag “method” are:
- **arg**: list of arguments of the method.
- **ret**: type of return of the method.
- **code**: this section contains the functionality as C-code. The method should call other methods throughout synchronous and asynchronous calls.

Both CROs and CRCs can contain instances of other classes:

```xml
<inst name=" "
    class=" "
    classarg=" ">
</inst>
```

- **name**: name of the instance.
- **class**: instantiated class name.
- **classarg**: this argument is a list of the output ports that can be used for the class and its instance.

### 4.2.4 Code synthesis framework

Code Synthesis is one of the most important functionalities integrated into REKO. The translation of the XML model into C-code is generated through an intermediate compiler by fulfilling the restrictions of the CRO/CRC model. This code should only be edited by means of the tool and a direct manipulation should never be performed. Code synthesis is the last step before using gcc to generate a binary (see Figure 19).
The analysis of this C-code is necessary to understand this project. The new approach uses the generated C-code to introduce KLEE directives and to analyse the methods of the model. Therefore, understanding the code structure followed in the generation is important as it is useful to understand the new development. The example used in the explanation of the graphic tool is also used in this section (see Section 4.2.1).

4.2.4.1 General issues

A Virtual Runtime System (VRS), which is commonly called wrapper.c, is needed to correctly execute the generated code. The VRS contains the definition of the synchronous and asynchronous functions and some auxiliary structures such as METHOD or OBJECT. The main function is also located in the VRS.

The nomenclature of the elements follows the hierarchy imposed by the model in order to create unique identifiers in the final result. Therefore, if there are two states with the same name belonging to different classes, they will be named differently in the C-code to avoid ambiguity.

The example of the code synthesis will be simplified for a better understanding of the code structure.

4.2.4.2 Structures and definitions of the methods

First, the definitions of the classes and their methods, states or instances in the file are translated into structures (C-typedefs) and function prototypes. Next, figures show the definition of the class “access_data” that corresponds to a CRO.

The state is defined by a C-struct which only has two fields in this example (see Figure 20). The class is defined by other struct in which a pointer to the state can be found (see Figure 21). Finally, the function prototypes are defined. In addition to the arguments of the function, a pointer to the class which the method belongs is also defined. This pointer allows the method to access the state (see Figure 22).

```
typedef struct {
    int a;
    int b;
} access_data_state;
```

*Figure 20: Simplification of the definition of the state*

```
typedef struct {
    int pl;
    access_data_state *state;
} access_data_class;
```

*Figure 21: Simplification of the definition of a class*

```
void set (access_data_class* self, int value);
```

*Figure 22: Simplification of the definition of function prototypes*
The struct which defines the class contains a field for each child class or instance. Figure 23 shows an example which illustrates how the hierarchy is specified. All the elements of the class which are used by its methods are specified too, such as the state (see Figure 21) or the output ports definition (see Figure 24).

```c
typedef struct {
    env_class env_inst;
    highest_comp_class comp_inst;
} container_class;
```

Figure 23: Simplification of the class definition example

```c
typedef struct {
    int pl;
    access_start_state *state;
    METHOD* data_get;
    OBJECT* data_get_obj;
    METHOD* data_set;
    OBJECT* data_set_obj;
} access_start_class;
```

Figure 24: Simplification of the definition of the output ports in a class

### 4.2.4.3 Static object structure

This C-struct contains the initialization of all the instances that are present in the model. The relation between the fields of the classes and the references to them are defined in this part of the code.

```c
obj_type obj={...
    .container_comp_inst = { ...
        .start_inst = {
            .pl = __unimplemented_pl,
            .state = &start_inst_state,
            .data_get = &get,
            .data_get_obj = &data_inst,
            .data_set = &set,
            .data_set_obj = &data_inst,
        }, ...
    }, ...
};
```

Figure 25: Simplification of the fragment of the static object structure

It is noteworthy that the state is defined as a global variable that is assigned to the attribute state in this point. On the other hand, methods are assigned to the output (see Figure 25). These assignments enable the correct access to the elements and fulfil the requirements to maintain the hierarchy.

### 4.2.4.4 Method definitions

The methods are defined by C-functions that have been previously defined by their prototypes. They contain the functionality of the system with the restrictions explained in Section 4.1.
In order to force the code to fulfil the requirements, local definitions are generated to prevent the misuse of the global memory (states) or methods. However, other possible incorrect behaviours can be contained in the body of the methods that the user introduces without any control from the tool.

```c
#define data_get &self->data_get
...

void start (start_class_t* self) {
    SYNC(data_set,10);
    int result=SYNC(data_get,0);
}
```

Figure 26: Simplification of local definitions and function body

Figure 26: Simplification of local definitions and function body contains an example of synchronous calls. The call “SYNC(data_set,10)” has “data_set” as output port. In the generated code this port corresponds to the method assigned in the static object struct (see Section 4.2.4.3), which in this case is correlated with “set”.

However, the best way to see the dependences and relations is by means of the graphic representation, where the required interface of the instance of the class “access_start” is related to the method set of “access_data” class (see Figure 12).

### 4.2.4.5 Test Functions Definitions

In the last section of the code, functions are defined to include measurement data. There is one test function for each method defined in the model.

### 5. REKO IDE extension: “Test Cases by Klee”

In this project, a new utility for the IDE has been developed. The result is a new extension: “Test Cases by KLEE” which generates a new C-code version based on the code generated by the original REKO. The newly generated code is able to debug the model through tests that trigger programming faults. Besides, the added functionality also allows the user to execute the cases automatically and use the values which KLEE generates as inputs. After this execution it is possible to detect unreachable code.

As a result of executing the newly generated with KLEE, it is obtained the KLEE output which contains values which reach all possible paths, detecting errors as part of the process.
The output is fed back to the REKO tool, using these results as inputs of the original code\(^1\) to obtain executable files. At the end of the process (see Figure 28), the code is ready to be compiled with a C-compiler and to be executed including the values generated by KLEE.

### 5.1 Integration into REKO IDE

The different phases of the process are hidden to the user. The programmer has an initial source to analyse that is made up by the model design and the Virtual Runtime System (VRS), as explained in Section 4.2.4.1.

As a result the files will be ready to be executed and analysed with the KLEE values included. Therefore, the user will not need to understand the middle states.

It is also possible to change the execution configuration of KLEE in order to introduce different options among those offered by KLEE. However, the understanding of KLEE is not required to use the new utility, since the final product is independent of the use of the symbolic engine.

The new utility “Test Cases by KLEE” has been developed by extending the intermediate compiler created to enable the “Code Synthesis”.

The intermediate compiler, which originally generated the code explained in Section 4.2.4, is modified to introduce KLEE directives in a first stage. Consequently, the new code can be interpreted by KLEE. In a second stage, it is extended to support the generation of code with the values obtained by KLEE as it will be explained in the next sections.

---

\(^1\) Original code refers to the one produced in the original REKO code generation.
5.2 Design and Implementation: “Test Cases by KLEE”

The process followed by the new utility can be summarised in two single stages. The first one involves “KLEE Test Cases Generation”, which includes: “Code Generation for KLEE” and “KLEE execution”. The second one is “REKO Test Cases Generation”, which includes a parser and the “New Code Generation”. Figure 28 shows the way to obtain the results.

![Figure 28: Test Cases generation process](image)

5.2.1 First stage: KLEE Test Cases Generation

The aim of this stage is to automatically obtain the test cases which KLEE produces from the model. To make the use of KLEE possible, its directives are introduced in the generated code. It is also necessary to determine which elements should be made symbolic, as well as where they should be defined as symbolic and where to introduce KLEE directives.

On the one hand, the methods which will be included in the analysis are the methods which are reached from an entry point along synchronous and asynchronous calls. Although methods which are not used can be defined in the model, they will never be executed and, thus, will be ignored.

On the other hand, all the variables involved in the execution of these methods are made symbolic. Arguments of the methods and states which belong to the reactive objects are the elements which need to be symbolic. Therefore, all the paths which depend on these variables will be checked against failures with KLEE.

At the end of this stage, the C-code that can be executed by KLEE and which generates the test cases is produced.

5.2.1.1 Code Generation for KLEE

This is a first step required to adapt the code to the use of symbolic execution. It is necessary to introduce different changes such as KLEE directives inserted in the C-code to allow the symbolic execution.

As it was mentioned in previous sections, the test cases generated by KLEE contain values that reach all possible paths that states may follow. However, only the method parameters of entry points or asynchronous calls are analysed.

For the synchronous calls, they will be executed as part of the interrupt drive execution, Thus, they do not need to be analysed individually. The value of its parameters cannot be changed by any other method while the execution is being carried out, consequently
they are not made symbolic. A function that contains the call to the entry points and the methods which appear in the asynchronous calls is defined at the end of the code.

The execution of entry points allows running all the interrupt drive execution of the model. As asynchronous calls do not invoke the actual method during the execution of other tasks, they are called individually to perform their own execution. Therefore, all the code (involved in an interrupt drive execution) is executed at least once.

In the next example, a scheme that shows how the function that contains the calls to the entry points and the methods which appear in the asynchronous calls is structured, as it has already been explained throughout this section (see Figure 29).

```
...  int selection;
     switch(selection) {
       case 1:
         TASK1; //Example entry point
         break;
       case 2:
         TASK2; //Example entry point
         break;

       case 3:
         TASK3; // Example method from an ASNC(...);
         break;
     }
...
```

*Figure 29: Scheme of the function that contains the entry points and the methods which appear in the asynchronous calls.*

Test cases are generated separately for each entry point or each method which appear in the asynchronous calls. This is why there are only related variables with one task in a specific test. To this aim an auxiliary variable called "selection" is made symbolic. The execution of "case 1:" comprises an entry point. Its execution will analyse the code in the synchronous calls and ignore the code located in the methods from asynchronous calls. However, this last code will be analysed in subsequent cases.

As for method arguments, they are made symbolic before the method is called. For this purpose, a global variable is defined for each method argument. The global variable is made symbolic and then the method is called with this global variable as argument.

Figure 30 shows how the argument from the method "start" is made symbolic. Then a unique short name is assigned.

```
klee_make_symbolic(&a, sizeof(int), "arg_a_3");
start(&start_inst, a);
```

*Figure 30: Code simplification to make the arguments of the methods symbolic*

If a method is reached in the execution, the state belonging to the class where the method is defined is made symbolic. Consequently, the same state can be made symbolic several times.
Figure 31 shows which data is necessary to make the states symbolic. The first parameter is a reference to the variable state. At the end, a short name is assigned to it.

```
    klee_make_symbolic(&state, sizeof(state_t),"state_6");
```

*Figure 31: Code simplification to make the state symbolic*

### 5.2.1.2 KLEE execution: Test Cases

The result of this step is a file which can be compiled with llvm-gcc to LLVM bytecode. The test cases which reach every path are produced after executing the code under KLEE with the arguments displayed in Figure 32.

```
# llvm-gcc -std=c99 -g -v --emit-llvm -c wrapper.c
# klee --allow-external-sym-calls wrapper.o
```

*Figure 32: Arguments used in the execution of KLEE*

Each file contains only values for one method. However, one method may be associated with more than one KLEE test case. Each test includes a variable which determines the method from where it comes (the variable selection was explained in the previous section).

All the outputs are collected in a file called “ktest_files.txt”, which includes all the test cases generated.

### 5.2.2 Second stage: REKO Test Cases Generation

The second stage of the project consists in introducing the data generated in the previous phase into the original C-code. This section explains the process responsible for obtaining the data and introducing it into the C-code in order to obtain a test that can be executed.

#### 5.2.2.1 Parser: Obtaining the data from ktest_file.txt

The tool includes a parser which looks for the input values in the "ktest_file.txt". By means of an auxiliary variable the method from where each test case comes is determined. The information obtained from the test for each variable that was made symbolic is: its symbolic name, its size and the value generated. The symbolic name which was associated to them in the first stage is unique to avoid ambiguity between states from different classes or parameters from different methods.
The values are stored in an array in the same order in which they are produced, since the same variable can be made symbolic more than once in the same execution. This could happen if there were a call for several methods with the same state through a synchronous call. At the end of this stage the values of the symbolic variables are ready to be introduced in the C-code.

5.2.2.2 New Code Generation

One source file which includes the C-code associated to the model is generated for each task. Each file contains the original code including the global definition of the functions parameters and the necessary storage structures.

In addition, the same task is executed with a different sets of values as input so several initialisations are created for each task in the file. Consequently, in the new code there will be as many task calls as different sets of values generated.

The number of sets of values generated depends directly on the parameters and states according to the paths and conditions found in the code for each method. Since other methods are called through synchronous calls within the task, the complete line of execution will be analysed.

Before a task is executed, the structures that store the inputs are initialised. Every time a new method is reached through a synchronous call in the execution, its state is updated with the value that it should have in that precise moment. This value is obtained from the input values structure.

An example of the initialisations is shown in Figure 33. The example is based on the same design used in Section 4.2.2. The method “start_env”, which is considered an entry point, is called through “test_env_inst_start_env”. Figure 33: shows how a set of values is assigned to the storage structures.

```c
... ptr_states=array_states;
pos=0;
memcpy(array_states + pos,"\x00\x00\x00\x05\x00\x00\x00",8);
pos+=8/sizeof(int);
value_arguments=0;
memcpy(&var_sym_argumento_n_start_env,&value_arguments,4);
test_env_inst_start_env();
...
```

**Figure 33: Simplification of the initialisation**

In the last example, there is only one value assigned to the structure, although it is possible to have more values, due to the occurrence of more synchronous calls, for instance.

A pointer has saved the address of the next piece of information that will be assigned to the state in each moment. Therefore, the state is initialised with the current value when a synchronous call is reached during the execution. An example of this implementation is shown below in order to see the initialisations more clearly.
int get(data_class_t *self, int value) {
    memcpy((self->state), ptr_states, sizeof(*self->state));
    ptr_states += sizeof(*self->state) / sizeof(int);
    ...
}

Figure 34: Simplification of the process of assigning a value to the state

As it was explained in Section 4.2.4.1 a Virtual Runtime System (VRS) is needed for the execution of the model. The necessary changes are introduced in the VRS to enable it to run automatically with the new generated files.

6. Results

The new prototype enables REKO to use KLEE to discover bugs and check dangerous operations, as well as to detect unreachable code. REKO models can take great advantage of the new analysis. Both its scope and the different possibilities it offers will be shown below through examples. This section contains portions of code produced by the developed tool.

The example used along Section 4 has been extended to perform a complete view. The differences with the original model are listed below:

1) A new CRO called “functionality” is created in the same level than the class “access_data”.

Figure 35: Added functionality
2) This new CRO is made up by the state displayed in Figure 36 and the methods that will be subsequently explained. These methods are related to the output ports from “start_inst”.

```c
typedef struct {
    int n;
    int array[3];
} mod_highest_comp_functionality_state_t;
```

*Figure 36: New state in the example.*

The interaction between the different objects of the example model is represented by a sequence diagram ( ), where message flow is described in order to have a better understanding of the model. The methods invoked in the diagram are specified in Figure 37 and Figure 38.

```c
void start_env(mod_env_class_t* self, int argumento_n) {
    SYNC(start_env_int,0);
}
```

*Figure 37: Entry point*

```c
void start (start_class* self, int a) {
    if(self->state->b<0)
        ASYNC(data_set,a);
    else
        self->state->b = SYNC(maximum,0);
}
```

*Figure 38: Method related with the port start_env_int*
This example generates two test files. The first one corresponds to the task specified in the asynchronous call. The task, “data_set”, does not contain conditional statements, so it will only generate one input that will cover all the code.

The other test file corresponds to the entry point task “start_env”. It will include the inputs to run all the paths from the entry point and it will also run the code of the synchronous calls. As a result, it will produce cases for the method related to the port “maximum” displayed in Figure 40.

```c
int function_maximum(functionality_class_t* self, int x) {
    int maximum=0;
    if(self->state->array[0]>=self->state->array[1]){
        if( self->state->array[0]== self->state->array[2])
            maximum=self->state->array[0];
        else
            maximum=self->state->array[2];
    }
    else if(self->state->array[0]<self->state->array[1]){
            maximum=self->state->array[1];
        else
            maximum=self->state->array[2];
    }
    else
        printf("UNREACHABLE CODE \
");
    return maximum;
}
```

Figure 40: Method related to the port called maximum
This second test file will contain five calls to the task with different initialisations for each call. Four out of five initialisations in the second test file come from the paths contained in the “maximum” method. KLEE decisions are shown in the following picture:

![Figure 41: Path followed by the symbolic execution](image)

In Section 3.2.5, it was explained how KLEE reaches different bugs. The new development enables REKO to generate the new tests for a model. The new tests contain values which reach programming faults, if any exist. Execution will be aborted after executing the test with the error values. Consequently, a debugging phase should start in this case. The obtained values will be useful to troubleshoot the problem and the KLEE generated test cases will provide information about the location of the error.

In the example, there is a branch of the code that is never inspected. The existence of unreachable code is obvious in this example. However, it can be noticed with coverage or profiling analysers in less evident occurrences.

Unreachable portions of the program can be detected using gcov (as it was explained in Section 3.2.3). Therefore, the detection of code which is not executed in any case is possible by combining the results with this kind of tools.

A complete code analysis can be done after the execution of the new tests. The set of new information provided by the KLEE generated cases and the new test code can be analysed in terms of coverage. In addition, if the coverage of the new tests is checked, the unreachable code will be detected. The use of profiling and coverage measurement tools allows pinpointing the code areas which should be optimised.

### 6.1 Symbolic execution applied to WCET analysis

The contribution of symbolic execution to the WCET analysis is a complete test suite including all feasible paths and high coverage. The test suite allows avoiding a handwritten test process. Therefore, the set of test cases generated is small compared to all other combinations of input values and KLEE contributes to it with relevant cases.
There are different types of code which are relevant to this analysis: if/else code, code with simple or nested loops, input or not input dependent and much more characteristics that can be described. If it is considered that the code lacks symbolic conditions (see Figure 42), a finite number of values will be obtained. For this reason, if a specific instance for a problem is given (for instance, the number of elements in a list such as in the example below), KLEE will obtain a limited number of tests.

```c
#define SIZE 4
int a[SIZE];

unsigned int main(){
    unsigned int i, j, temp;
    klee_make_symbolic(&a, sizeof(unsigned int)*SIZE, "a");
    i = 1;
    while(i <= SIZE-1){
        j = i;
        while ((j>0) && (a[j] < a[j-1])){
            temp = a[j];
            a[j] = a[j-1];
            a[j-1] = temp;
            j--;
        }
        i++;
    }
}
```

*Figure 42: Insertsort example. Based on Mälardalen WCET Benchmarks [50]*

In the example above (see Figure 42) the number of elements in the list is fixed to 4, so KLEE generates 24 cases that will exhaustively go through representative paths in the program. Actually, for a list of 4 elements, 4! tests are obtained, which is the number of permutations corresponding to 4 elements. It should be worthy noticed that the number of test cases is lower than the possibilities for 4 integers of 32 bits. In this case the property is extended to other values of the list size (n equal to 2, 3, 5, 8...) and it is satisfied that n! is the number of test cases. Moreover, the example is certainly illustrative obtaining different paths and different combinations of data for each case.

### 7. Limitations

The most important limitations in the prototype implementation are related to the analysis of loops, the endianess of the system and the representation of size in different systems. The last two problems are deeply linked as they are both related to the architecture.

KLEE cannot originally reason about loops with symbolic conditions. In case this kind of loop is found by KLEE, the tool cannot manage properly the analysis of the code due to a large amount of paths generated. However, some options can be considered to prevent this behaviour and obtain transcendent data.

The following figure displays a method that contains a loop with a symbolic variable as part of the condition. More branch conditions have been avoided in the example to keep the loop analysis clear and simple. There are different options that can be considered when a loop with symbolic conditions is included in the code.
int function_loop (functionality_class_t* self, int data) {
    int i, result;
    for (i=0; i<self->state->n; i++)
        result+= data;
    return result;
}

Figure 43: Method related to the port function

The first one is to indicate a limit of time for the execution. It can be specified in the
cmd prompt via: "--max-time=<seconds>". As a result of using this option, a big
amount of tests will be generated with random values from the new tool perspective.
More options can be considered to be introduced as cmd line, such as, for
instance, "--max-depth=<level>", which prunes the search in the indicated level (see
Figure 44).

klee --allow-external-sym-calls --max-depth=5 wrapper.o

Figure 44: Command line in case code with loops and symbolic variables in the
conditions

Other problem is referred to the endianess of the system as it is a known problem that
different systems have different data representations. Linux on x86 or x86-64 is a little-
endian system, where the last byte, positioned with the highest address, will be the
most significant one and the first will be the least significant. The implementation of the
new test generation does not allow to use a system with other representation, such as
big-endian. This is due to the way that KLEE results are produced and the data is
introduced in the new test generated by the new framework. KLEE cannot recognise
the different fields of a C-struct and thus the information is shown as a buffer of data.
The buffer is introduced into the code produced by REKO to feed back the information
of the KLEE tests, so the new tests generated by the new framework are affected by
the endianess of the system where it is executed.

Another important problem is the representation of type size. C standard [Ref 32]
defines only a minimum and a maximum size for the types. This ambiguity may cause
different compilers of different architectures to assign different type sizes. Besides, gcc
describes the size of the types in accordance with the size of the word, thus making
them directly dependent on the architecture [33]. As the data obtained from KLEE uses
Linux x86 variables representation, and therefore its corresponding sizes, another
solution should be found in order to use the new tool in other systems.

To differentiate the kind of architecture and to adapt the tests to its data format could
be one way to solve the problem about the endianess and the representation size. It
can be done by obtaining the data in the format of the system in which the test will be
done, by obtaining the KLEE tests for this system format. However, this is not proved in
the developed framework.
8. Conclusions and Future Work

KLEE has been integrated into the REKO IDE in order to generate test suites for REKO models automatically. The test suites reach all feasible paths with high coverage discovering programming bugs. Consequently, the complexity of the testing process, which is a time consuming task, decreases dramatically.

The new REKO features will be able to obtain the test cases that together with the coverage tool can find unreachable code.

Although the results explained above are successful, the tool has its own limitations. The limitations inherited from KLEE and the solvers such as the treatment of the loops, which is an important issue taken into account in the models that have been analysed.

Furthermore, the present design of the new REKO features allows working in run-time measurements using the files resulting from the new development. In addition, the generated test is designed to facilitate the search of WCETs in future work.

The development of an engine based on KLEE which includes features to perform WCET analysis could be used to improve the accuracy of measurement-based analyses. Therefore, it could be an interesting line of work to continue the current research of EISLAB on REKO models and WCET analysis in general.
9. REFERENCES


42. J. Wiklander, A reactive approach to component-based design of resource-constrained embedded systems. Have to do this reference


