SCUOLA DI DOTTORATO
UNIVERSITÀ DEGLI STUDI DI MILANO-BICOCCA

Department of
Informatics, Systems and Communication

PhD program in Computer Science, Cycle XXXV

On the Effectiveness of Automatic Test Case Generation for Safety-Critical Software

Name: ELSON KURIAN

Registration number: 855363

Tutor: Prof. Davide Ciucci

Supervisor: Prof. Giovanni Denaro

Co-Supervisor: Prof. Daniela Briola

Coordinator: Prof. Leonardo Mariani

ACADEMIC YEAR: 2021-2022

Acknowledgments

The PhD journey has been a life-changing experience. It would not have been possible without the support and guidance I received from countless people.

First and foremost, I am incredibly grateful to my supervisor, Prof. Giovanni Denaro, for his invaluable advice and for believing in me during my PhD study. Also, I'd like to thank my co-supervisor, Prof. Daniela Briola, for all the help, feedback, and patience she's given me over the years.

I would like to thank Prof. Pietro Braione for his technical support and for his valuable suggestions. In addition, I would like to acknowledge my colleagues Andrea Riboni, Matteo Modonato, and Luca Guglielmo for the friendly and stimulating environment in the Software Testing and Analysis Laboratory.

I would like to acknowledge Dario D'Avino the industrial resource person from the Rete Ferroviaria Italiana, Italy, for your support.

I would like to express my gratitude and love to my wife Tinu Beth Thomas, my brothers Linson Kurian and Dinson Kurian, and my parents for their support, without which it would not have been possible. I owe it to you.

Finally, I would like to acknowledge all my friends who supported me on the journey.

Abstract

Software systems for automating safety-critical tasks in application domains like, avionics, railways, automotive, industry 4.0, and healthcare must be highly reliable. In this thesis, we focus on safety-critical software written in Scade, a model-based modelling language largely adopted in industry, that we used for developing safety-critical Scade programs for the railway domain. Automated test generation (state-of-the-art) based on symbolic execution and bounded model checking can be beneficial for systematically testing safety-critical software to facilitate test engineers in pursuing the strict testing requirements mandated by the certification standards while controlling the cost of the testing process. At the same time, the development of safety-critical software is often constrained by programming languages or coding conventions that ban linguistic features believed to downgrade the safety of programs, e.g., they do not allow dynamic memory allocation and variable-length arrays, limit the way in which loops are used, forbid recursion, and limit the complexity of control conditions. We leverage state-of-the-art test generators based on symbolic execution and bounded model checking in order to define an original toolchain for generating test cases for Scade programs. We evaluate the effectiveness of our toolchain for automatic test generation with a benchmark of 37 Scade programs developed as part of an onboard signalling unit for high-speed railway systems developed in collaboration with an industrial partner.

Sommario

I sistemi software per l'automazione di attività safety-critical in domini applicativi (ad esempio avionica, ferrovie, automotive, industria 4.0 e assistenza sanitaria) devono essere altamente affidabili. In questa tesi, ci concentriamo sul software safety-critical scritto in Scade, un linguaggio di programmazione model-based ampiamente adottato nell'industria, da noi utilizzato per lo sviluppo di programmi Scade safety-critical per il dominio ferroviario. La generazione automatica di test (stato dell'arte) basata sull'esecuzione simbolica e sul bounded model checking può essere utile per testare sistematicamente il software safety-critical al fine di facilitare gli ingegneri nel rispettare i severi requisiti di test imposti dagli standard di certificazione, controllando al contempo il costo del processo di testing. Allo stesso tempo, lo sviluppo di software safety-critical è spesso vincolato da linguaggi di programmazione o convenzioni di codifica che vietano caratteristiche del codice che si ritiene riducano la sicurezza dei programmi, ad esempio, non consentono l'allocazione dinamica della memoria e array di lunghezza variabile, limitano il modo in cui vengono utilizzati i cicli, proibiscono la ricorsione e limitano la complessità delle condizioni di controllo. Abbiamo sfruttato generatori di test allo stato dell'arte basati sull'esecuzione simbolica e il bounded model checking al fine di definire una toolchain originale per la generazione di casi test per programmi Scade. Abbiamo valutato l'efficacia della nostra toolchain nel generare i casi di test automaticamente con un benchmark di 37 programmi Scade sviluppati come parte di un sistema di segnalazione di bordo per sistemi ferroviari ad alta velocità sviluppato in collaborazione con un partner industriale.

Contents

Acknowledgments i

Abstract ii

Abstract (Italian) iii

1 Introduction 1

2 State-of-the-art 7
2.1 Automatic Test Generation 7
2.1.1 Random Testing 7
2.1.2 Search-Based Testing 10
2.1.3 Bounded Model Checking 13
2.1.4 Symbolic Execution 17
2.2 KLEE a Symbolic Execution Engine 22
2.2.1 Overview 22
2.2.2 Key Features and Properties 25
2.3 CBMC ("C Bounded Model Checking") 27
2.3.1 Overview 27
2.3.2 Generating the Formula 28
2.3.3 Converting the Formula to CNF 29
2.4 Related work on Test Generation for Scade 32
2.4.1 Automated model-based testing 33
2.4.2 Formal methods for safety-critical software 34
2.4.3 Automated test generation for Scade models 35

3 Automated Test Generation for Scade programs 37
3.1 Safety-Critical Development with Scade 37
3.2 A Sample Scade Program (Working Example) 40
3.3 The TECS Toolchain 44
3.3.1 KCG 45

3.3.2 Test Driver Synthesis 45
3.3.3 Test Input Generation 48
3.3.4 Test Synthesis 52

4 Experiments and Results **56**
4.1 Experimental Assessment 56
4.2 Subject programs 57
4.3 Research Questions and Metrics 60
4.4 Experiment setting 62
4.5 Results 64
4.5.1 RQ1: Effectiveness of automatic test case generation for safety-critical software in Scade. 65
4.5.2 RQ2: Efficiency of symbolic execution and bounded model checking for generating test cases for safety-critical programs in Scade 74
4.6 Threats to validity 78

5 Conclusion **80**

How to use TECS **82**

List of Publications **84**

List of Abbreviations **85**

List of Figures

2.1 Warm-up C code for random testing 9
2.2 Warm-up C code for search-based testing 12
2.3 Warm-up C code for bounded model checking 14
2.4 Warm-up C code for symbolic execution 18
2.5 C code for example1 19
2.6 Warm-up C code for KLEE 23
2.7 KLEE outputs details 24
2.8 Ktest-tool format for test cases result 26
2.9 Sample program to illustrate the model checking code transformation 29
2.10 Sample program that uses dynamic memory allocation 30
2.11 Warm-up C code for CBMC, example 1 31
2.12 Unwound version of the warm-up C code for CBMC, example 1, with loop unwinding set to 4 31
2.13 Warm-up C code for CBMC, example 2 32
2.14 Code Coverage results generated by CBMC for C program Figure 2.13 32

3.1 A sample Scade model for a car wing mirror controller 41
3.2 Excerpt of the C program that KCG generates for the Scade model in Figure 3.1 43
3.3 Components and workflow of TECS 45
3.4 Algorithm of the analysis driver 48
3.5 KLEE representation for the inputs used in of the C program Figure 3.2 51
3.6 CBMC representation for the inputs used in of the C program Figure 3.2 53
3.7 A test case generated for the sample program of Figure 3.2 . . 54

List of Tables

2.1 Test suite generated by KLEE for C program Figure 2.6 23
2.2 Test suite generated by CBMC for C program Figure 2.13 . . . 31

4.1 Subject programs 58
4.2 Statistics of the subject programs 59
4.3 Number of test cases and O-MC/DC Coverage for TECSK and TECSC 66
4.4 Faults identified in the subject programs considered in our case study 69
4.5 TECS comparison with search-based testing 71
4.6 Comparison between automatically (TECSK [TECSC) and manually derived test suites 73
4.7 TECS: execution time, number of paths and generated test cases 75
4.8 Data on the queries issued to the constraint solver 78

Chapter 1

Introduction

Today, software is an essential and critical component of information technology's all-encompassing impact. In particular, in this thesis, we focus on safety-critical software, that is, software that must ensure highly reliable automation in areas like avionics, railways, automotive, industry 4.0, and patient monitoring, where failures can have disastrous effects. Because of these risks, the development process of safety-critical software typically encompasses several quality-oriented requirements, driven by the goal of satisfying the concerned certification authorities [1, 2]. Software testing is the most extensively used method for hunting software bugs nowadays, a main challenge being to generate sufficient test inputs. Indeed, when dealing with large embedded system software, the activity of generating the test inputs will often take a lot of time, and require an impractical amount of human effort. Because of this, automating the process of generating test inputs has been goal of many research efforts so far [3, 4, 5, 6, 7]. Moreover, in the specific case of safety-critical software, automated test case generation can also play a crucial role in achieving the testing objectives mandated by the standards for safety-critical systems while keeping the associated costs under control.

In this context, my research is focused on the development of an automated test generation tool for an industrially-relevant class of safety-critical software, namely, safety-critical software developed in Scade. Scade is a synchronous language designed for the development of embedded safety-critical software. Using Scade fits a common tenet of many development processes for safety-critical software, that is, to rely on programming languages that, by their design choices and controlled semantics, may both decrease the chances of introducing subtle faults in the programs and mitigate the hard work required to satisfy the certification requirements [8]. Scade is a system modelling language and a model-based development environment for embedded

software largely adopted in industry[1] [9, 10, 11, 12, 13, 14, 15] and certified according to the CENELEC norms [2]. Scade allows specifying models with a formalism based on finite state machines, that forbids constructs like dynamic memory allocation, variable-length arrays, non-statically-in-bound accesses to arrays, pointer arithmetic, recursion, and unbounded loops. Thanks to these restrictions, Scade models can be automatically translated to equivalent C programs that guarantee the certification standards [16, 17] required by ERA, the European Union Agency for Railway[2].

In particular, my research has been conducted in the context of a joint research and development effort of Rete Ferroviaria Italiana (RFI, the public company that manages the railway infrastructure in Italy) and the University of Milano-Bicocca, aimed at developing an on-board signalling unit for high-speed railway systems, compliant with the ERTMS[3] standard specification. This onboard unit is an embedded safety-critical component that shall handle signals from several track-side devices, e.g., transponders deployed along the railway and control units at the stations, and shall notify the driver or even activate the braking devices of the train under some dangerous conditions. With the aim of ensuring the highest degree of software integrity, RFI is relying on the model-based Scade programming language for the development of this safety-critical software.

Challenges of Automated Test Generation The mainstream test generation approaches are random testing, search-based testing, symbolic execution and bounded model checking. Random testing and search-based testing randomly sample the input space of the target program, either in a purely random fashion or guided by heuristics based on the improvement of a *tness* function that represents the extent to which the test cases fulfilled the test objectives [18, 19, 20, 21, 22]. Symbolic execution and bounded model checking systematically explore the execution space of the program under test. Symbolic execution computes the execution conditions of program paths as sets of constraints over symbols that represent the possible program inputs, and then solves these constraints to concrete test data with off-the-shelf constraints solvers [23, 24, 25, 26, 27, 28, 29, 30]. Bounded model checking

[1] Ansys, the company that commercializes Scade and the supporting SCADE Suite model-based design environment, reports uses of Scade at Subaru for automotive applications, and for many other safety-critical, embedded applications, including, avionics and ight control, autonomous vehicles and gas turbines. [www.ansys.com/products/embedded-software/ansys-scade-suite]

[2] www.era.europa.eu

[3] https://www.era.europa.eu/domains/infrastructure/european-rail-traffic-management-system-ertms_en

encodes the semantics of all execution paths (up to a bounded length) as a boolean formula, and addresses a set of reachability properties of interest by solving the formula in conjunction with the constraints that represent the properties [31]. By encoding coverage objectives as a reachability property, bounded model checking can be exploited to generate test cases.

In our research, we focus on symbolic execution and bounded model checking. This choice is motivated by the importance of fulfilling the relevant test objectives (e.g., the coverage targets required by the certification standards) while testing safety-critical software. By exploring the program paths systematically, symbolic execution and bounded model checking should be in principle able to generate at least a test case for every reachable test objective, a goal that the random and search-based techniques cannot generally guarantee. In fact, although the heuristics used in search-based testing are mostly based on dynamic program analysis (which generally results in more lightweight approaches than by using static analysis as in symbolic execution and bounded model checking), nonetheless the strategies based on random sampling notoriously have limited effectiveness when pursuing test objectives that may depend on singular or quasi-singular inputs. In general, this *hard-to-randomly-hit* test objectives there exist in the target programs, making developers of safety-critical software often hesitant to accept the limitations of search-based testing tout court.

While focusing on symbolic execution and bounded model checking, we must face the major challenges that result from common limitations of these techniques:

(i) Coping with the so-called *path explosion problem*: Since the number of execution paths of a program grows exponentially with the amount of decision logic in the program and is generally unbounded for programs that include recursive calls and loops governed with arbitrary conditions, symbolic execution and bounded model checking seldom succeed to analyze all execution paths in finite time. On the contrary, the systematic exploration approach often engages symbolic execution and bounded model checking in a very fine-grained analysis of some specific parts of the program execution space, while leaving other parts untested.

(ii) Suitably handling non-numeric inputs, i.e., pointers or references to *dynamically allocated, possibly recursive data structures*: For the analysis to be precise, symbolic execution and bounded model checking shall be able to discriminate the executions in which the references within the input objects in the heap could be either assigned to null-values or

be an alias of each other, or yet correspond to distinct objects, respectively [32]. This further exacerbates the computational requirements for the analysis. The number of objects and object configurations to be discriminated could even be unbounded for inputs defined as recursive data structures.

(iii) Tolerating the limitations of SMT solvers in computing the solutions of *complex path constraints*: Failing to solve the execution conditions of a program path can depend on either the path being indeed *infeasible*, i.e., not executable with any input, or the path constraints being too hard for the current SMT solver to be decided within the allowed time budget, or yet outside of the theories supported by the SMT solver. In the latter cases, the solver is unable to either provide a solution or prove that a solution does not exist. The inability of solving complex path constraints can result in missed test cases, or waste large portions of the test budget in the analysis of execution paths that depend on unsatisfiable conditions that the constraint solver failed to pinpoint.

Our research hypothesis Our research is based on the main research hypothesis that the above issues (hindering the practicality of symbolic execution and bounded model checking) may hold for many general-purpose programs, but they have reduced impact for a significant class of industrial safety-critical systems, where symbolic execution and bounded model checking can therefore work effectively. As we commented above, safety-critical software often relies on programming languages or coding standards that ban some linguistic features, based on the (empirically motivated) ground that those features are common causes of subtle failures. For example, one of the tenets of safety-critical software development is avoiding unbounded consumption of time or space resources at runtime, to cope respectively with divergence or crashes. For this reason, languages for safety-critical software development like SaferC [8] and Scade[4] (used in the avionics and in the railway domains, respectively), or coding standards like Misra[5] (required in the automotive industry) restrict what the programmers are allowed to do. Relevant restrictions include: forbidding programmers from allocating memory dynamically, instead requiring all the memory to be allocated by local or global variables with predictable size; statically bounding the maximum number of iterations of loops; and avoiding recursion. Some consequences of this regime are that in such applications the total number of execution paths

[4] `https://www.ansys.com/it-it/products/embedded-software/ansys-scade-suite`

[5] `https://www.misra.org.uk/Publications/tabid/57/Default.aspx#label-dvg`

is finite, every execution path has a finite depth, and many programming constructs that yield an explosion in the size of the execution state space are not used.

Contributions To study the effectiveness of generating test cases with symbolic execution and bounded model checking for Scade programs, we first introduce a novel test generator for Scade programs. We refer to this test generator as Tecs (Test Engine for Critical software in Scade). Tecs renders a tools chain that (i) uses Kcg to render a Scade program as a C program, (ii) exploits a mature, state-of-the-art symbolic executor for C and a mature, state-of-the-art bounded model checker for C, in order to obtain a suitable set of test inputs for the C program, and (iii) recasts the test inputs as test scripts for the original Scade program under test. Tecs is specifically engineered with APIs that facilitate its integration with existing test generators for the C program. The currently integrated test generators are Klee [33], which is based on symbolic execution, and Cbmc [31], which is based on bounded model checking. Tecs makes several distinctive design choices that explicitly exploit the programming constraints guaranteed for programs in Scade.

Other than the relevance of the Tecs prototype, our main contribution is an industrial-scale case study on the practicality of symbolic execution and bounded model checking, as embodied in our test generator Tecs, to automatically generate test cases for industrial safety-critical programs in Scade. Our case study encompassed 37 Scade programs out of the implementation of the on-board signalling unit for high-speed railway systems developed jointly with RFI. The results of this thesis have been partially published at the International Conference of Software Engineering [34] and in the Journal of System and Software [35]. Our findings indicate that:

- Our approach enables test engineers to automatically synthesize test cases that achieve high model coverage and can assist them to reveal subtle program faults.

- We compared Tecs with search-based testing by using the tool Afl [19], a test generator that is very popular for security vulnerability testing based on random and search-based input selection heuristics, as a possible replacement for the instantiation of our tool. The results clearly indicate the weaknesses of the random selection approach, which missed many test objectives that our approach allows for accomplishing.

- Yet, for some considered programs, we compared the test cases that Tecs produced with the ones that were already manually designed by

the developers. The comparison revealed the usefulness and effectiveness of our test generator for Scade models.

Structure of the Dissertation : The thesis is divided into 5 Chapters, and is organized as follows:

- Chapter 2 provides an overview of mainstream state-of-the-art approaches for automatic test generation and, in detail, two specific techniques, KLEE and CBMC, that we define in this thesis.
- Chapter 3 discusses TECS our original approach for automatic test generation for Scade programs.
- Chapter 4 provides the main contribution of this thesis to evaluate the effectiveness of our toolchain for automatic test generation with a benchmark of 37 Scade programs developed as part of an onboard signalling unit for high-speed railway systems.
- Finally, Chapter 5 presents the Conclusion and future work of my thesis.

Chapter 2

State-of-the-art

*This chapter surveys the mainstream approaches to automatic test generation, i.e., as random testing, search-based testing, symbolic execution, and bounded model checking. We survey the most relevant concepts and works related to the contributions of this thesis. Finally, we go over the state-of-the-art related to two speci c techniques, one for symbolic execution (*KLEE*) and the other for bounded model checking (*CBMC*), which are the ones used for automatic test generation in the approach that we de ne in this thesis.*

2.1 Automatic Test Generation

The idea behind automatic test generation is to generate software test cases automatically and to validate the system under test with a set of test cases that thoroughly exercise the functionality of the system being tested. It can be used to quickly generate a large number of test cases, saving time and effort compared to manually creating the same tests.

The first test generation technique was developed in the 1970s by researchers at IBM's Thomas J. Watson Research Center. They recognized the need for a faster, more efficient way to generate software tests, and so they searched for a way to automate the process. Over time, researchers have expanded on the concept and improved upon it, making it more useful.

2.1.1 Random Testing

Random testing is a dynamic black-box software testing technique where the inputs of the tests are generated randomly depending on the requirements, specifications, or some test adequacy criteria [36]. The results of the executed tests are compared with the software specifications to determine whether

the test passes or fails. In the absence of specifications, exceptions of the programming language are employed to identify failures, that is, an exception that occurs during the execution of a test indicates a bug of the program. Notice however that this assumption can lead to false alarms, since there can be many examples of useful scenarios in which an exception that occurs during the execution of a test does not indicate bugs in the program.

Data generation is the most important phase of random testing and is aimed to produce the test data set based on the input domain, without regard to any test adequacy requirements. Uniform distribution and operational profile are the two most often employed strategies. With a uniform distribution, the probability of selection for each input is equal. This ensures that the test suite is unbiased and represents all possible scenarios. An operational profile is a set of conditions used to define how inputs and outputs should interact during testing. It involves defining the number of times an operation should be executed and what the expected output of the operation should be.

A generic random testing process can be summarized as a four-step procedure:

(i) Identification of the input domain.

(ii) Selection of test inputs based on the input domain.

(iii) The SUT is executed on these test inputs.

(iv) The results are compared to the system specifications. If any input leads to incorrect results, the test fails; otherwise, it passes.

Step (ii) is the point where different testing techniques usually differ: random testing allows us to generate a large number of test suites automatically in a random fashion. It may be performed according to an operational profile, where there are conditions typically involving boundary values, data ranges, and functional specifications guiding the generation of random test inputs. This approach may save some time and effort compared to more deliberate input selection approaches.

Techniques for generating random test data are applicable to all projects. If the input domain is evenly structured, random automated test generation can be used to execute many more test cases than with a manual generation, and we can obtain unexpectedly good results by employing the proper approach. However, if a fault is only revealed by a small portion of the program's input, random testing will generally fail to hit it. Considering the code in Figure 2.1, it is clear that the probability of the first statement (line

```
1  typedef enum {false, true} boolean;
2  boolean test(int x, int y) {
3    if (x == y)
4      return true;
5    else
6      return false;
7  }
8  int main(){
9    boolean result;
10   int a = rand(); //some random value
11   int b = rand(); //some random value
12   result = test(a,b)
13   return 0;
14 }
```

Figure 2.1: Warm-up C code for random testing

4) being executed is substantially smaller than that of the second statement (line 6). As the complexity of the structure increases, the chance of execution decreases proportionally. Some well-known approaches (and tools) at the state-of-the-art adopting a random testing approach are QuickCheck [37],Randoop[1] [20] and Yeti-TEST[2] [38].

Literature survey Random testing was first mentioned in the literature in 1970 by Hanford, who described a tool that generates random data for testing PL/I compilers [39]. Later in 1979, Glenford J. Myers mentioned random testing in his first edition of "The Art of Software Testing" [40]. Bird and Munoz presented a technique for randomly generating self-checking test cases in 1983 [41]. In [42] Richard mentioned that using a random number generator the test data is generated randomly from the specified input domain.

Recently, the effectiveness of random testing has been often debated. For example, random testing was mentioned by Myers as one of the least effective testing procedures [43]. However, Ciupa et al. [44] stated that Myers's statement is not based on any experimental evidence and, later on, other experiments [45, 42] stated that random testing is an effective testing

[1] https://randoop.github.io/randoop/
[2] https://github.com/yui/yeti

technique. It is reported [46] that, if a large number of test cases randomly generated are exploited, it may be possible to find subtle faults in a SUT.

Another criticism is that random testing generates test data without knowledge of which program states get executed, and thus it may end up with generating numerous test cases with identical program states. It is also argued that random testing may often happen to generate many test inputs that violate the provided SUT's preconditions, thereby reducing its effectiveness [47].

With respect to other techniques, it can be observed that the simplicity and relatively low cost of random testing make it more practical to generate a large number of test cases than systematic testing techniques, which may take significant time and resources to generate test cases [48]. However, empirical comparisons show that partition testing and random testing are not necessarily equally effective, because random testing may easily miss test points in some subdomains, resulting in low coverage [49]. Nonetheless, the analysis that Ntafos conducted on several comparisons found that random testing is a good complementary strategy to use, especially in the latest testing phases whereas partition testing may face cost-effectiveness issues [50].

Random testing has also been used for vulnerability testing. For instance, Miller et al. used a random testing approach to generate random ASCII character streams to check abnormal terminations and non-terminating behaviours in Unix utilities, showing that it helped find security holes in the Unix operating system [51]. Subsequently, their technique was extended, for example, to generate random keystrokes, mouse movements and mouse button clicks, to discover errors in the applications running on X Windows, Windows NT and Mac OS X [52, 53].

2.1.2 Search-Based Testing

Search-based software testing (SBST) grounds on heuristics and optimization techniques (e.g., Genetic Algorithms) to build effective tests. Technically, search-based testing generates test data by pursuing the optimization of a *tness* function that represents the satisfaction of a set of test objectives that comprise a test adequacy criterion, in order to maximise code coverage of the software under test with respect to that adequacy criterion [54]. Thus, the purpose of the fitness function is to identify the test objectives that, when met, contribute to the intended test adequacy criterion. The search algorithm finds test inputs that optimize the achievement of this test objective, by using the fitness function as a guide. Different fitness functions can be defined to capture various test objectives, allowing the same basic search-based

optimization strategy to be applied to a wide variety of test data generation scenarios.

Once the fitness function is defined, it can be exploited with different types of optimization algorithms, such as the Hill Climbing method or Genetic Algorithms.

Hill Climbing begins its search at a random location. The present point's neighbours in the search space are examined for fitness. If a better candidate solution is discovered, Hill Climbing relocates to the new location and examines the neighbourhood surrounding the candidate solution. This procedure is repeated until there are no better candidate solutions in the neighbourhood of the present location in the search space, a so-called "local optimum". If the local optimum is not the global optimum, it may be advantageous to "restart" the search and execute a climb from a different initial position in the *tness landscape*.

Genetic Algorithms are inspired to the Darwinian evolution concept of "survival of the fittest". Each candidate solution within the search space under consideration is referred to as an "individual". The current group of individuals under consideration is referred to collectively as the "population". The initial population is formed at random, and each individual's fitness is assessed. A process that favours the best individuals determines which individuals will serve as crossover parents. During crossover, components of each individual are recombined to produce two offspring that share characteristics with their parents. In the "reinsertion" phase, the next population generation is selected, and the new individuals' fitness is assessed. This cycle repeats until the Genetic Algorithm finds a solution (according to the fitness function at hand) or the search resources are exhausted (e.g., a time restriction or a particular number of fitness evaluations).

Summarizing, the main two requirements that need to be fulfilled in order to apply a search-based optimization technique are the Representation and Fitness functions. Representation means that the candidate solutions for the problem at hand should be capable of being captured so they can be handled by the search algorithm. The fitness function is then responsible for evaluating the quality of a given solution, in order to assess whether a current solution is an optimal solution according to certain criteria, or how far it is from being optimal. By evaluating candidate solutions, the fitness function steers the search to promising areas of the search space. Problem-specific fitness functions must be defined for each new problem.

For instance, consider in Figure 2.2 a C code with a function called *SUT* with two integer parameters. In a search-based test data generation technique using Hill climbing, we can start with a random candidate solution represented as random values for the inputs, e.g., *x= 1, y= 10*, and then

```
1 typedef enum { false, true } boolean;
2 boolean SUT(int x, int y) {
3   if (x == 2 □ y)
4     return true;
5   else
6     return false;
7 }
```

Figure 2.2: Warm-up C code for search-based testing

explore the neighbourhood candidates, i.e.,

$$\mathrm{x} = x+1, x+2, x+3, x \quad 3, x \quad 2, x \quad 1...$$

$$\mathrm{y} = y+1, y+2, y+3, y \quad 3, y \quad 2, y \quad 1...$$

by evaluating their fitness. A possible fitness function could address branch testing, by defining the objectives of covering the branch at line 4 (when the values of x and y are such that $x == 2 \quad y$, i.e., when they are solutions of the equation $x \quad 2 \quad y == 0$) and the branch at line 6 (when the values of x and y are the solution of the inequality $x \quad 2 \quad y$ 6= 0). For example, the fitness function based on the formula of the first branch ($x \quad 2 \quad y == 0$) would evaluate that the initial solution *x= 1, y= 10* is sub-optimal ($x \quad 2 \quad y = \quad 19$, where $\quad 19$ can rate the amount of sub-optimality, i.e., it is sub-optimal with score $\quad 19$) and would help choose the best neighbour, until reaching an optimum value *x= 5, y= 10*.

Literature survey Search-Based Software Testing was introduced by Webb Miller and David Spooner [55] in 1976. Their approach was a simple technique for generating test data consisting of floating-point inputs and was a completely different approach to the test data generation techniques being developed at the time, which were based on symbolic execution and constraint solving. In Miller and Spooner's approach, test data were sought by executing a version of the software under test, with these executions being guided toward the required test data through the use of a 'cost function' (what we referred to as a fitness function), coupled with a simple optimization process. Inputs that were 'closer' to executing a desired path through the program were rewarded with lower cost values, whilst inputs with higher cost values were discarded. Miller and Spooner did not continue their work in test data generation, and it was not until 1990 that their research directions were continued by Korel [56, 57].

In 1992, Xanthakis applied Genetic Algorithms to the testing problem. Since then there has been an explosion of work, applying meta-heuristics more widely than just test data generation. Search-based optimization has been used as an enabler to a plethora of testing problems, including functional testing [58, 59], temporal testing [60, 61], integration testing [62], regression testing [63], structural testing [64], stress testing [65], mutation testing [66], test prioritisation [67], state machine testing [68] and exception testing [69]. Search-based approaches have also been applied to problems in the wider area of software engineering, leading Harman and Jones to coin the phrase 'Search-Based Software Engineering' [70] in 2001. The term 'Search-Based Software Testing' began to be used to refer to a software testing approach that used a metaheuristic algorithm, with the amount of work in Search-Based Test Data Generation alone reaching a level that led to a survey of the field by McMinn in 2004 [71].

2.1.3 Bounded Model Checking

Bounded Model Checking (BMC) is an efficient technique for program analysis. It reduces the program analysis problem to a boolean satisfiability (SAT) or satisfiability Modulo Theory (SMT) problem. It is designed for the formal verification of finite state transition systems. The program states and the transition function defined by the statements in the program, as well as the properties to be verified for the program, represented with Boolean formulas, and conjunctive joined to represent the verification problem at hand: the (joined) Boolean formula is true, if and only if the underlying state transition system can make a finite sequence of state transitions that leads to certain interesting states. The transformation of the program to a Boolean formula operates on the source code directly, or on an Intermediate Representation(IR) produced with a compiler, and is usually very accurate, taking into account the semantics of the programming language, the memory manipulations, how machines do arithmetic, and so forth.

The basic idea is to turn the verification problem into a propositional formula that is satisfiable if and only if there exists some counterexample of the property being verified, usually considering counterexamples of a maximum length *k*, hence the name *bound* model checking [72].

Consider Figure 2.3 showing a C code to illustrate the use of bounded model checking. Here we will use a model-checking tool called the C Bounded Model Checker (CBMC) to verify the C code safety properties within fixed execution depth and generate test cases for the code. For the verification, the entry point resides in the main function on the C code. The *__CPOVER_input()* function is used to provide the symbolic values for the

```
1  int main(int m, int n) {
2    x = n;
3    __CPROVER_input("n",n);
4    __CPROVER_input("m",m);
5    __CPROVER_assume(n >= 1 && n <= 100);
6    __CPROVER_assume(m >= 1 && m <= 100);
7    while (0 <= m && m < n){
8      x = x - 1;
9      m = m + 1;
10   }
11   __CPROVER_assert (0 <= x, "err");
12 }
```

Figure 2.3: Warm-up C code for bounded model checking

variables. The *__CPROVER_assume()* function holds the Boolean condition argument, and calls to this function that do not satisfy the condition are discarded from the analysis. The *__CPROVER_assert()* function is a user-defined specification supported by CBMC for code verification. When the model checker is invoked, *__CPROVER_assert()* will attempt to verify that the condition given as the first argument holds on a path up to a specified bound. The second argument is a diagnostic string that will be reported in the results if it finds a counter-example for the assertion.

In this example, we use the command *cbmc bmc_code.c unwind 3* on the command line, telling CBMC to unroll the loop at most three times. The result says that CBMC instantiated the SAT problem as a formula over 1214 boolean variables and 4372 clauses, and found it to be unsatisfiable This resulted in the message "verification successful" for the assertion at line 12, which is the property we wish to check. If there would be a counterexample for this property, we could pass the command line argument *trace* to see an execution trace that leads the program to violate the property.

The tool can be used to generate test cases according to a test criterion, by using the non-reachability of the criterion's test objectives as the verification properties. Thus, if there is a counterexample, it is a test case that executes the test objectives. For example, by adding the command *cover mcdc*, the tool is able to generate test cases that satisfy the MC/DC test criterion for the C code in Figure 2.3. As a result, CBMC generated two test cases for the Figure 2.3, namely, T1 (n = 3, m = 8) which skips the while, and T2 (n = 4 and m = 3) which enters the while.

Literature survey Model Checking was introduced by Clarke and Emerson in 1981 [73]. It can be used when the design to be verified is modelled as a finite state machine, and the specification is formalised by writing down temporal logic properties. The design's reachable states are then traversed to verify the properties, and if a property fails, a counterexample in the form of a sequence of states that lead to violating the property is generated [74]. The initial model-checking algorithm directly enumerated the reachable states in the system, in order to validate the given properties [75]. However, this method can be burdensome, since the number of states can increase exponentially with the number of variables in the model. In order to overcome this situation, Burch et al. introduced the symbolic model checking technique in 1992 [76, 77].

In symbolic model checking, sets of states are implicitly represented by Boolean functions and reduce the complexity. With Reduced Ordered Binary Decision Diagrams [78] (ROBDD, or BDD for short), it was possible to manipulate the boolean formulas effectively and pushed the barrier to systems with 1020 states and more [76]. The memory requirement for storing and processing BDDs was the limiting factor for these solutions due to a large number of the required boolean function representations. Then, Biere et al. introduced the next version called Bounded Model Checking (BMC) in 1999 [72] to handle this.

The main advantages of BMC are: (i) due to the depth-first nature of the SAT search procedure is very fast to find counterexamples; (ii) it is able to find counterexamples of minimal length, which leads the user to understand counterexamples easily; (iii) it required less space than a BDD (Binary Decision Diagrams) based approach; and finally, (iv) it does not need a manually selected variable order or time-consuming dynamic reordering due to default splitting heuristics [72].

BMC was originally based on a SAT solver, which can handle propositional satisfiability problems with many variables [74]. Nowadays, there are many formal verification researches and developments happening in the area due to the high relevance of modern model-checking approaches. Some major tools from academia are SMACK[3] and ESBMC[4], from industrial research there are Corral[5] and F- SOFT [79], and for the industry are CBMC[6] and QPR [80].

SMACK [81] is a modular software verification toolchain as well as a verifier used to verify the assertions in the programs. Assertions are tested up

[3] https://smackers.github.io/

[4] http://www.esbmc.org/

[5] https://github.com/boogie-org/corral

[6] https://www.cprover.org/cbmc/

to a given bound on loop iterations and recursion depth in its default mode; it also has experimental support for unbounded verification. SMACK is a translator from the popular intermediate representation (IR) of the LLVM [82] compiler into the Boogie intermediate verification language (IVL) [83]. Sourcing LLVM IR exploits an increasing number of compiler front-ends, optimizations, and analyses. Targeting Boogie takes use of a standard platform that facilitates the construction of algorithms for verification, model checking, and abstract interpretation due to Boogie's minimal syntax and its easily translation into the SMT format of automated theorem provers [84]. Due to the use of modelling dynamically-allocated memory for generating quantified invariants over unbounded maps, SMACK is suitable for fully automatic *unbounded veri cation* methods (e.g., based on computing fixed points) and may require a powerful reasoning engine; whether such applications are feasible remains to be seen [81].

ESBMC is a context-bounded model checker based on SMT [84] for C/C++ programs. It can provide APIs for C and Python languages to access internal data structures and allows inspection and verification processes. It can verify user-defined assertions and program safety properties. The main features of ESBMC are (i) the *clang* front-end, (ii) the floating-point backend and (iii) the *k*-induction proof rule [85].

Corral is an SMT-based verifier inside the Microsoft Static Driver Verifier (SDV) [86], accepting programs in Boogie. It can provide information when it hit a user-supplied bound on the number of loop iterations along with 'bug found' and 'verified' [87].

F-SOFT is a verification tool for the C programs analysis which is a combination of SAT-based verification, static analysis and predicate abstraction. It translates the C program into the Boolean model represented by the control flow graph (CFG) of the program and is to be analyzed by the DiVer verification engine [88], which includes BDD-based model checking and SAT-based model checking [79].

CBMC is a bit-precise bounded model-checking implementation for C programs into a formula and if satisfied (satisfiability decided using MiniSat 2.2.0 [89]), it executes the programs under a specified loop unwinding bound, and checks the absence of violated claims and memory safety properties. Its ability to detect counterexamples for relevant program faults has been empirically demonstrated on many programs and fault of different types [31].

QPR Verify is an extensive version of the bounded model checking approach implemented in the tool LLBMC[7]. It is intended to verify industrial embedded software written in C/C++ language and emphasise runtime er-

[7] https://llbmc.org/

rors. Main features support both functional and usability checks, such as providing code traces for each runtime error, improved efficiency and scalability and GUI customization [80].

2.1.4 Symbolic Execution

Symbolic Execution [23, 24, 25, 26, 27, 28, 29, 30] is a program analyzing technique introduced in the mid-70s to test and analyze software code with certain properties that might be violated. In a concrete execution, a program is executed with a concrete input, and a single control flow path is explored. As a result, in most cases, concrete executions can only be analysed through limited paths. However, the ultimate goal of code analysis is to inspect each space of program inputs and understand how they behave. In contrast, symbolic execution can simultaneously explore multiple paths that the program could take under different inputs. As a result, this analysis can yield strong guarantees on the checked property.

The core idea behind symbolic execution is to execute the program with symbolic inputs (symbols that indicate arbitrary values) rather than concrete inputs. Whenever the system finds a new branch point, it will generate a new path condition that satisfies a set of constraints. As a result, every possible path is analyzed and explored. During the analysis, symbolic execution explores the execution space in the form of a tree, with each node representing a branch in the program.

A symbolic execution engine is responsible for carrying out the operations. It keeps two elements for each explored path: (i) a first-order boolean formula that describes the execute-ability condition of the branches taken along that path, and (ii) a symbolic memory store that associates variables with symbolic expressions or values. Each branch execution updates the formulas and symbolic store. The constraint is typically based on a Satisfiability Modulo Theories (SMT) solver, which is used to determine whether any property is violated in each explored path.

Consider the C code in Figure 2.4 where x, and y are user inputs, and we would like to test whether the fault represented as the *assert false* statement at line 8 could happen. A test entails concretely executing a program with two specific inputs and verifying the results. Symbolic execution considers how the program executes abstractly on a set of related inputs.

The state is fully characterized by three variables (x, y and z). If we run this C code with user inputs (consider as concrete input) x, y with 2, 2 respectively, the branch condition $x > y$ (line 3) becomes false and $z = 2$. This causes the branch condition $z < x$ (line 8) to fail, and thus the assertion is not executed. Let's re-consider: $x = 2$ and $y = 1$. Now, branch condition

```
1  void foo(int x, int y){
2    int z = 0;
3    if(x > y){
4      z = x;
5    }else{
6      z = y;
7    }
8    if(z < x){
9      assert false;
10   }
11 }
```

Figure 2.4: Warm-up C code for symbolic execution

x >y (line 3) is true, and z = 2. Still, the branch condition z < x (line 8) fails, leading to not executing the assertion. And so forth for other concrete inputs.

Here, instead of executing the C code with concrete input values (like x = 2, y = 1), symbolic execution can evaluate with symbolic input values (like x = α, y = β), and track execution paths with these symbolic inputs. If branch conditions are dependent on unknown symbolic values, the symbolic execution engine selects one of the paths to drive and records the condition on the symbolic values that would lead to that particular path. After completing that execution path, the symbolic execution engine will go back to the branch condition and explore other possible paths throughout the program.

To get a better idea of how symbolic analysis works, consider the abstract execution path in the previous C code (Figure 2.4). Initially the variables values are x = α, y = β and z = 0. After considering the first branch condition, i.e., x > y (line 3), the symbolic value of z is ($\alpha > \beta$) ? α or (α β) ? β , according to the two possible path conditions, which are $\alpha >$ β and α β. In next branch condition z < x (line 8), which checks z< α. The previous condition states that the value of z either be equal to α (and the check $\alpha < \alpha$ will always fail) or β (but the check $\beta < \alpha$ depends on the path condition α β and cannot thus be satisfied for any input). These two constraints demonstrate that the program will never reach the *assert false* statement.

Generalized Symbolic Execution (GSE)[90] enhances conventional symbolic execution by allowing it to handle programs with pointers and recursive data structures as inputs. GSE uses lazy initialization to handle inputs with a recursive data structure. GSE begins the execution method with uninitial-

```
1 void example1(int x, int y){
2   if (x == hash(y)) abort(); //error()!
3 }
```

Figure 2.5: C code for example1

ized variables, then non-deterministically initializes fields when they are first accessed during the method's symbolic execution.

Dynamic Symbolic Execution (DSE) or Dynamic test generation [91] for automation test generation consists of instrumenting and running a program while collecting execution path constraints on inputs from predicates encountered in branch instructions executed in the current execution path. Then, it derives new inputs using an SMT solver to steer subsequent executions towards new program execution paths. Because of its two inherent qualities, DSE has become an interesting technique in software engineering research. Firstly provided sufficient runtime information, DSE does not suffer from substantial false positives. DSE tools generate alarms only when software exceptions occur during runtime. Secondly, each new input generated by the SMT solver may result in a new execution path.

Before DSE, static technique approaches and black-box fuzzing (i.e., random testing) were conventionally used in many security tools, e.g., coverity [92], appScan [93], webInspect [94]. Now, DSE is a popular research area. The rationale for this is that DSE is far more accurate than static analysis and has higher code coverage than black-box fuzzing.

The example shown in Figure 2.5 is a function *hash* invoking an external function that might be a complex arithmetic function. As a result, it is often impossible to create input values to trigger the error statement in line 2. This is a very common example due to complex statements in the program code like arithmetic computations, pointer operations and calls to library functions or operating systems. The runtime information of the program P can be used to solve the problems brought by external calls and complex arithmetic functions. With the reference of the example in Figure 2.5, typically we cannot generate the inputs values for x and y to trigger the statement in line 2 in the first run. Dynamic execution collects the calculation of *hash(y)* and sets x the same as *hash(y)*, to reach the statement line 2 and trigger the *abort()* error.

Literature survey Many research efforts investigated test case generation using dynamic symbolic execution like, Directed Automated Random Test-

ing (DART) [91], Execution Generated Executions(EGT/EXE) [95, 96] or Concolic Testing (CUTE) [97]. They showed that this technique of implementation helps automatic test generation in various ways. In particular, it improves scalability with respect to the exponential number of path conditions that must be dealt with in the programs. Different approaches were studied to further overcome this problem by employing heuristics to path exploration [96, 26], interweaving symbolic execution with random testing [98], caching function summaries for later use by higher-level functions [99], or eliminating redundant paths by analyzing the values read and written by the program [100].

We survey the most notable test generation approaches (and tools) based on dynamic symbolic execution below.

CUTE (Concolic Unit Testing Engine) [97] was developed at the University of Illinois at Urbana-Champaign for C programs, extending DART to support multi-threaded application programs that use pointer operations to manage dynamic data structures. CUTE avoids pointer analysis imprecision by representing and solving pointer constraints roughly. Note that concolic execution is a synonym for dynamic symbolic execution.

CREST [101] is an open-source tool for C program concolic testing. CREST is an extendable framework for developing and experimenting with heuristics for selecting paths to test the programs which have many execution paths.[8] CREST has been used by numerous research organizations since it was published as an open-source program in 2008. For example, CREST has been used for building tools for augmenting existing test suites to test the updated code [102] and for identifying SQL injection vulnerabilities [103]. Also, CREST was modified to run on a cluster for testing a flash storage platform[104] and used to experiment with more sophisticated concolic testing heuristics[105].

Pex [28] implements Dynamic Symbolic Execution to generate test inputs for .NET code and also supports C#, VisualBasic, and F#. Pex computes models or generates test inputs for a satisfiable constraint system using SMT solver Z3 [106]. It uses approximations for theories for which Z3 has no precise decision procedures, e.g. for string [107] and floating point arithmetic [108]. Pex combines many searching strategies which select the order in which different execution paths are attempted, to achieve high code coverage quickly [109]. In addition to the test case generation capabilities, it comes with a mock and stub framework, which makes it easy to write and reuse models for .NET libraries [110]. Pex enables Parameterized Unit Testing [111], an extension of traditional unit testing. This tool is available as a

[8] Available at `http://code.google.com/p/crest`

Visual Studio 2010 Power Tool[9].

EXE [96]is a symbolic execution tool for C developed comprehensively for testing complex software, with a focus on systems code. EXE models memory with bit-level accuracy to handle the complexity of system code. The integration of low-level optimizations in the STP constraint solver [96, 112] leads EXE to quickly solve many constraints. As a result, EXE was able to generate high-coverage test suites automatically, and find out security vulnerabilities and deep bugs in complex code, including file systems, packet filters, device drivers and network servers and tools [96, 100, 95, 113].

KLEE [25] is a redesigned version of EXE, built on top of the LLVM [82] compiler framework. Similar to EXE, it executes codes in a mixed concrete/symbolic manner, models memory with bit-level accuracy, various constraint-solving optimizations, and uses search heuristics to achieve high code coverage. By utilizing state sharing at the object-level, rather than at the page-level as in EXE, KLEE is able to store a significantly higher number of concurrent states than EXE. Also, KLEE is able to handle external interaction — e.g., offering models designed to explore all possible legal interactions with the outside environment. KLEE has been open-sourced in June 2009.[10] Since then, numerous users from the academic community and the industry have used it. These users have built upon KLEE in a variety of areas, ranging from wireless sensor networks [114], to automated debugging [115], reverse engineering and testing of binary device drivers [116, 117], exploit generation [118], online gaming [119], and schedule memorization in multi-threaded code [120].

Automated Whitebox Fuzzing (SAGE)[26] is used for security testing and expands the wide scope of systematical dynamic test generation from unit testing to whole-application testing. Whitebox fuzzing was first implemented in SAGE [26] and then has been adopted in tools like CatchConv[11] and Fuzzgrind[12] Whitebox fuzzers have found many new security vulnerabilities (buffer overflows) in Windows [26] and Linux applications [121]. Notably, SAGE found many bugs during the development of Microsoft's Windows 7, which saved millions of money by avoiding extra expenses for security patches. Since 2008, SAGE has been continuously *fuzzing* hundreds of applications in their security labs.

[9] http://msdn.microsoft.com/en-us/vstudio/bb980963.aspx

[10] KLEE is available at http://klee.llvm.org.

[11] http://www.sf.net/projects/catchconv

[12] https://github.com/dpc-grindland/Fuzzgrind

2.2 KLEE a Symbolic Execution Engine

In this section, we introduce the symbolic execution tool KLEE in further detail, as it is a tool that we used for the research work presented in this thesis. In fact, as already motivated in the Chapter 1, our research on the suitability of automated test generation for safety-critical programs in Scade focused on symbolic execution and bounded model checking. In particular, while elaborating our proposal, we exploited the fact that the Scade compiler translates the programs to C[13], and thus we could reuse leading-edge test generators for C, as the technological ground to build our prototype. As for symbolic execution, we relied on KLEE. As we commented above, KLEE is one of the most relevant symbolic execution tools at the state-of-the-art, and it addresses programs in C. KLEE belongs to the category of Dynamic Symbolic Execution (DSE) tools, which combine concrete and symbolic executions.

2.2.1 Overview

KLEE is a symbolic execution tool built on top of the LLVM platform [82]. It executes LLVM bit-code and primarily targets programs written in C complied with the CLANG compiler.

Executing KLEE on programs requires marking which inputs will be dealt with symbolically. Consider an Integer variable a: to provide symbolic value for the memory location, the program code must include a function call: $klee\_make\_symbolic(\&a, sizeof(a),$"a"$)$ which notifies to KLEE the size of a and its memory position, which will be later analysed symbolically.

KLEE solves the path conditions with the STP constraint solver[14]. With respect to its predecessor EXE [96], KLEE optimizes the queries to the constraint solver more aggressively by logic simplifications [122]. Also, to mitigate the cost of constraint solving, the tool attempts to prevent calls to the STP, by reusing previously cached results.

Warm-up example for KLEE

Consider C program in Figure 2.6 that aims to find out if the given input is 10 or more than 10. To execute the program with KLEE, first, the symbolic inputs are marked in the code and the program is compiled into LLVM bitcode. Next, the file is executed by KLEE. Symbolic files and

[13] As the most systems to build embedded programs, Scade exploits on cross-compilation: It rst translates the programs to C, and then relies on a C compiler to generate the binary code for the target platform.

[14] https://stp.github.io/

```
1  #include <klee/klee.h>
2  int isTenOrMore(int x){
3    if(x == 10)
4      return 1;
5    if (x > 10)
6      return 0;
7    else
8      return -1;
9  }
10 int  main(){
11   int a;
12   klee_make_symbolic(&a, sizeof(a), "a");
13   return isTenOrMore(a);
14 }
```

Figure 2.6: Warm-up C code for KLEE

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.

Table 2.1: Test suite generated by KLEE for C program Figure 2.6

test case	variable	value
1	*a*	10
2	*a*	0
3	*a*	16777216

Figure 2.7 shows the details of KLEE execution for the C program. KLEE executed 31 instructions and generated 3 test cases, which satisfied all the path conditions in the C program. Table 2.2.1 shows the tabular representation of test data generated by KLEE for the warm-up C code Figure 2.6.

KLEE output files

KLEE generates the following files as default:

(i) *info*: General information about the execution such as total completed paths, the total number of instructions explored, the number of generated tests or the execution line among others.

```
1    KLEE: output directory is "/home/klee-out-0"
2    KLEE: done: total instructions = 31
3    KLEE: done: completed paths = 3
4    KLEE: done: partially completed paths = 0
5    KLEE: done: generated tests = 3
```

Figure 2.7: KLEE outputs details

(ii) *testN.ktest*: This file contains the data of the symbolic variables, including the arguments used in the execution of KLEE, the number of objects that have been made symbolic, and for each one the name associated with the symbolic memory, the size and that data obtained by KLEE.

(iii) *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.

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

(v) *warnings.txt*: Notification of errors and other warnings are shown in this file.

(vi) *run.istats*: Statistics about the execution of each line are included in this file and can be visualised via KCachegrind.

(vii) *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 into KLEE:

(i) *Ktest tool*: This tool is used to visualize the test resulting from the execution of KLEE. It will visualize the files with the extension ".ktest" which contains the generated values. Figure 2.8 shows the visualization for the 3 test cases generated by KLEE for the C program in Figure 2.6. This tool allows visualising the 4-byte data as integer representation but it cannot differentiate the components of a structure displaying the data as a buffer.

(ii) *ktest replay*: This tool is used to reproduce test cases generated by KLEE.

(iii) *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: the 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.

2.2.2 Key Features and Properties

(i) *Bit-level correctness*: KLEE uses bit-vectors to represent program variables which model integer operations with bit-level accuracy (overflow, bitwise operations)

(ii) *Compact state representation*: Execution states are cloned with per-object shared ownership when possible. Concrete objects are stored with minimal memory overhead.

(iii) *Execution-Generated Testing*: KLEE implements the Execution Generation Testing technique, which makes it possible to call external functions that are outside of the executor's control.

(iv) *Solver optimization*: The constraint solver is wrapped in multiple layers that perform caching, simplifications and other optimizations of the formulas.

Expression Language

KLEE is able to represent SMT formulas with its own language called KQuery and it logs solver queries . KLEE also has a standalone tool called KLEAVER that will evaluate the KQuery queries. The in-memory representation is a tree structure defined in the EXPR module. The Expression language is closely related to the SMT-LIB format and can be translated into it.

Low-Level Virtual Machine (LLVM)

LLVM [82] is a compiler development framework. LLVM is a collection of compilers and toolchain technologies that are designed to be modular and

```
1  $ ktest-tool klee-last/test000001.ktest
2  ktest file : 'klee-last/test000001.ktest'
3  args       : ['isTenOrMore.bc']
4  num objects: 1
5  object 0: name: 'a'
6  object 0: size: 4
7  object 0: data: b'\x00\x00\x00\x00'
8  object 0: hex : 0x0a000000
9  object 0: int : 10
10 object 0: uint: 10
11 object 0: text: ....
12 $ ktest-tool klee-last/test000002.ktest
13 ktest file : 'klee-last/test000002.ktest'
14 args       : ['isTenOrMore.bc']
15 num objects: 1
16 object 0: name: 'a'
17 object 0: size: 4
18 object 0: data: b'\x00\x00\x00\x00'
19 object 0: hex : 0x00000000
20 object 0: int : 0
21 object 0: uint: 0
22 object 0: text: ....
23 $ ktest-tool klee-last/test000003.ktest
24 ktest file : 'klee-last/test000003.ktest'
25 args       : ['isTenOrMore.bc']
26 num objects: 1
27 object 0: name: 'a'
28 object 0: size: 4
29 object 0: data: b'\x00\x00\x00\x01'
30 object 0: hex : 0x00000001
31 object 0: int : 16777216
32 object 0: uint: 16777216
33 object 0: text: ....
```

Figure 2.8: Ktest-tool format for test cases result

reusable features. It can be used to optimise compilation time for any programming language that the user wants to define. The successful results obtained have generated a variety of front-ends, including C and C++. Nowadays, many projects build-on LLVM. Among all projects, KLEE comprises

a symbolic virtual machine built on top of the 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 /llvm-clang that corresponds to the LLVM front-end [82].

2.3 CBMC ("C Bounded Model Checking")

As for the bounded model checking of the C versions of the Scade programs, the research work presented in this thesis relied on CBMC. Thus, in this section, we introduce the bounded model checking tool CBMC in further detail.

CBMC [31] is a very well-known and widely adopted formal verification tool for C and C++ programs exploiting Bounded Model Checking. It is compatible with C89[15], C99[16], most of C11[17] and most of the compiler extensions provided by GCC[18] and Visual Studio[19]. It is also compatible with SystemC[20] with Scoot. It supports array bounds verification (buffer overflows), user-specified assertions, exceptions and pointer safety. Furthermore, it can also verify C and C++ for consistency with other languages like Verilog[21]. The verification is achieved by unwinding the loops in the program and passing the resulting equation to a decision procedure.

2.3.1 Overview

CBMC is able to reason at low-level on ANSI-C programs. In CBMC, the transition relation and specification of the complex state machine are jointly unwound to provide a Boolean formula that is satisfied if an error trace exists. A SAT procedure is then used to validate the Boolean formula and if satisfied, a counterexample is derived from SAT procedure output. It also checks that adequate unwinding is done to assure that no counterexample may exist by using *unwinding assertions*. It comes with a graphical user interface (GUI) with minimal information for the user and hides the implementation details. If a counterexample is found, the GUI allows tracing the error. The following sections explain the CBMC formula generation and conversion to CNF.

[15] https://pubs.opengroup.org/onlinepubs/7908799/xcu/c89.html
[16] https://en.cppreference.com/w/c/99
[17] https://en.cppreference.com/w/c/11
[18] https://gcc.gnu.org/
[19] https://github.com/microsoft/vscode
[20] https://systemc.org/
[21] https://www.verilog.com/

2.3.2 Generating the Formula

The Model Checking process performed by CBMC to determine the validity of a bit vector equation has five steps:

- We assume that the ANSI-C program is already preprocessed, e.g., all the #*de ne* directives are expanded. We then replace side effects by equivalent assignments using auxiliary variables, *break* and *continue* by equivalent *goto* statements, and *for* and *do while* loops by equivalent *while* loops.

- The loop constructs are unwound. Loop constructs can be expressed using while statements, (recursive) function calls, and goto statements. Functions calls can complicate the analysis since they can bring more variables into the loop that need to be bounded. To precisely analyze loops with function calls, CBMC needs to identify all-recursive call sites, gather all relevant variables, and estimate their values. With those values, CBMC will perform the loop bounding process and execute the loop for the specified number of iterations. The while loops are unwound by duplicating the loop body n times. Each copy is guarded using an if statement that uses the same condition as the loop statement. The if statement is added for the case that the loop requires less than n iterations. After the last copy, an assertion is added that assures that the program never requires more iterations. The assertion uses the negated loop condition. We call this assertion an unwinding assertion. These unwinding assertions are crucial for our approach: they assert that the unwinding bound is actually large enough. If the unwinding assertion of a loop fails for any possible execution, then we increase the bound n for that particular loop until the bound is large enough.

- Backward goto statements are unwound in a manner similar to while loops.

- Function calls are expanded. Recursive function calls are handled in a manner similar to while loops: the recursion is unwound up to a bound. It is then asserted that the recursion never goes deeper. The return statement is replaced by an assignment (if the function returns a value) and a goto statement at the end of the function.

- The program resulting from the preceding steps only consists of (possibly nested) if instructions, assignments, assertions, labels, and goto instructions with branch targets that are defined after the goto instruction (forward jumps). This program is then transformed into a static

```
1 a = a + b;
2 if (a != 1)
3   a = 2;
4 else
5   a++;
6 assert (a <= 3);
```

Figure 2.9: Sample program to illustrate the model checking code transformation

single assignment (SSA) form, which requires a pointer analysis. We omit the full details of this process.

Consider the Figure 2.9, from this sample code, the above procedure can produce two bit-vector equations: C (for the constraints) and P (for the property). The results are, C :- $a_1 = a_0 + b_0$ ^ $a_2 = 2$ ^ $a_3 = a_1 + 1$ ^ $a_4 = ite(a_1 \not= 1, a_2, a_3)$ and P :- $a_4 \quad 3$, where ite is the if-then-else operator. To check the property, need to convert C ^ : P into CNF by adding the intermediate variable and passing it to an SAT solver.

2.3.3 Converting the Formula to CNF

Most operators are easily converted to CNF, and the process is similar to that of generating suitable arithmetic circuits. For the advantage of circuit-level SAT solvers, CBMC may also output the bit-vector equation before it is flattened down to CNF. It supports programs that use dynamic memory allocation, such as dynamically sized arrays or data structures like lists or graphs. As an example, the fragment in Figure 2.10 uses *malloc* to create a variable number of integers, then inserts one value into the last array element before deallocates the array: while the integer n remains bounded, its maximum value requires the reservation of far too many literals in order to construct a CNF for the shown in the above fragment. As a result, dynamically created arrays are not converted to CNF by creating literals for each possible array element. Instead, arrays with variable size are implemented using uninterpreted functions.

Cbmc assertions and assumptions

Apart from automatically checking the properties of the program, CBMC also provides helpers to specify assertions and assumptions in the programs. They can be used to aid CBMC with more information about the program.

```
1 void foo(unsigned int n) f
2   int *p;
3   p = malloc(sizeof(int)*n);
4   p[n-1] = 0;
5   free(p);
6  g
```

Figure 2.10: Sample program that uses dynamic memory allocation

These keywords can be used for program instrumentation. Program instrumentation is a procedure that changes or adds part of codes to verify some properties of the program.

- `__CPROVER_assert(expr)` can be used to assert a condition. It takes a Boolean expression `expr` as an argument. When CBMC encounters one of these assert statements, it tries to generate a formula to check assertion failure. The generated formula is verified using SAT solvers. If the formula is satisfied then the assertion fails and CBMC generates an error and produces a counter-example showing the possible trace of the error.
- `__CPROVER_assume(expr)` keyword reduces the number of considered program traces and allows assume-guarantee reasoning. It takes a Boolean expression `expr` as the argument.

Warm-up example for CBMC

The basic idea of CBMC is to model a program's execution up to a bounded number of steps. Technically, this is achieved by a process that essentially amounts to "unwinding loops". Loop unwinding, also called loop unrolling, is the process of converting loops into sequential statements. Consider the C program in Figure 2.11: CBMC can execute up to a bounded number for unwinding loops, say 4 times for this example. A BMC instance that will find bugs with up to four iterations of the loop would contain four copies of the loop body and essentially corresponds to checking the loop-free program mentioned in Figure 2.12 (generated if running CBMC with the following command line argument *cbmc BMC_Test.c unwind 4 bounds-check*). If we run CBMC for the Figure 2.11 without mentioning the flag –unwind, CBMC does not stop on its own. The built-in simplifier is not able to determine a run time bound for the loop.

```
1  void BMC_Test(int argc, char □□argv) f
2    while(cond) f
3      Body Code
4    g
5  g
```

Figure 2.11: Warm-up C code for CBMC, example 1

```
1  void BMC_Test(int argc, char □□argv) f
2    if(cond) f
3      Body Code Copy 1
4      if(cond) f
5        Body Code Copy 2
6        if(cond) f
7          Body Code Copy 3
8          if(cond) f
9            Body Code Copy 4
10         g
11       g
12     g
13   g
14 g
```

Figure 2.12: Unwound version of the warm-up C code for CBMC, example 1, with loop unwinding set to 4

CBMC generates test cases which cover the MC/DC coverage. Consider the C program in Figure 2.13 that aims to find out if the given input is 10 or more than 10. Figure 2.14 shows the code coverage results from the CBMC execution for the C program. CBMC is able to find 4 coverage goals and generate 3 test suites to satisfy the goals (see Table 2.2).

Table 2.2: Test suite generated by CBMC for C program Figure 2.13

test case	**variable**	**value**
1	*a*	10
2	*a*	9
3	*a*	262153

The CBMC supports all ANSI-C operators and pointer constructs allowed by the ANSI-C standard, including dynamic memory allocation, pointer

```
1  int isTenOrMore(int x){
2    if(x == 10)
3      return 1;
4    if (x > 10)
5      return 0;
6    else
7      return -1;
8  }
9  int  main(){
10   int a;
11   __CPROVER_input("a",a);
12   return isTenOrMore(a);
13 }
```

Figure 2.13: Warm-up C code for CBMC, example 2

```
1  □□ coverage results:
2  [isTenOrMore.coverage.1] file isTenOrMore_cbmc.c
3  line 3 function isTenOrMore decision/condition 'x==10' false: SATISFIED
4  [isTenOrMore.coverage.2] file isTenOrMore_cbmc.c
5  line 3 function isTenOrMore decision/condition 'x==10' true: SATISFIED
6  [isTenOrMore.coverage.3] file isTenOrMore_cbmc.c
7  line 5 function isTenOrMore decision/condition 'x>10' false: SATISFIED
8  [isTenOrMore.coverage.4] file isTenOrMore_cbmc.c
9  line 5 function isTenOrMore decision/condition 'x>10' true: SATISFIED
10 □□
```

Figure 2.14: Code Coverage results generated by CBMC for C program Figure 2.13

arithmetic, and pointer type casts. The user interface is meant to appeal to system designers, software engineers, programmers and hardware designers, offering an interface that resembles the interface of tools that the users are familiar with.

2.4 Related work on Test Generation for Scade

This thesis focuses on the capabilities of symbolic execution and bounded model checking to generate test cases for Scade programs. In this section, we survey the existing approaches that investigate test generation for Scade.

2.4.1 Automated model-based testing

Our approach can be seen as related to *model-based testing*, which derives test cases by analyzing program specifications or program behaviours expressed in suitable modelling languages, e.g., UML class diagrams, state machines or sequence diagrams [123, 124]. Model-based testing consists of deriving test data by analyzing either program specifications or program behaviours expressed as models, e.g., with class diagrams, state machines or sequence diagrams [123, 124]. Indeed Scade is a model-based programming language that exploits state machines and data flow models for defining software behaviors [14, 15]. Model-based testing has been successfully applied to complement the verification of formal specifications expressed in languages such as B, Z or VDM [125]. For a comprehensive survey of model-based testing, we refer to the work of Utting et al. [126] and Dias Neto et al. [127].

Formica et al. propose ATheNA [128], a novel search-based software testing framework that combines fitness functions to guide the search exploration towards software failures that are automatically generated from the requirements specification and manually defined by engineers. They implement an ATheNA-S instance of ATheNA that targets Simulink®. models. AtheNA-S could generate failure-revealing test cases when applied to a large case study from the automotive domain. The main difference from our approach is that AtheNA generates test cases if and only if failure-revealing test cases are found.

In particular, the approach that we investigate in this thesis shares similarities with the ones of Polyglot [129, 130] and SAUML [131], which exploit symbolic execution to generate test cases for systems modelled with statecharts and UML-RT state machines, respectively[129, 130, 131]. Polyglot translates statecharts to programs (specific programs in Java) and then exploits symbolic execution (by means of the symbolic executor SPF [132] that addresses Java), to generate test cases that achieve path coverage up to some specified depth. SAUML extends symbolic execution to directly analyze the UML-RT models (i.e., it works without converting the models to programs) to check properties like reachability and invariants, and to generate test cases.

As we will explain in detail in the next chapter, the work reported in this thesis differs from both these approaches in the way we distinctively use symbolic execution and bounded model checking within an analysis algorithm tailored to the characteristics of the Scade models, which foster programs with finite path spaces and input data structures comprised of finite sets of distinct fields. The structural testing approach of Tecs is naturally complementary and could be profitably integrated with test cases generated by

exploiting functional model-based testing.

2.4.2 Formal methods for safety-critical software

Safety-critical systems need to strictly comply with their requirements as they were elicited in the earliest phases of the development process. *Formal* methods [133] define one or more languages with mathematically precise semantics that can be used to describe the requirements, the domain constraints and the designs, and to prove or disprove relevant properties thereby, e.g., absence of deadlock or unreachability of unsafe states. Most formal methods define mathematically rigorous procedures to ensure that the artefacts produced at every step of a development process *re ne* the artefacts produced at earlier steps, thus preserving all their relevant properties. The downside of these approaches is the degree of mathematical sophistication that they demand from software engineers and designers, who should be able to model a system with a formal specification, prove (or disprove) its properties, refine an abstract (not directly computable) specification progressively to a concrete (computable) one, and translate a concrete specification to an executable program in a given programming language. To this end, formal methods are often accompanied by tools that assist in performing their tasks, with various degrees of automation, which anyway hardly balance the aforementioned complexity.

Formal methods differ in the breadth of their scope. At one end of the spectrum, methods like B or its successor Event-B [134] aim at producing a complete, correct-by-construction approach, encompassing all the phases of the development lifecycle. These methods usually refrain from testing the final implementation, in the assumption that having proved both a sufficient set of correctness properties on the abstract designs, and their preservation through the refinement steps may suffice to ensure that the final program is correct *by-construction*. Other formal approaches do not have the generality of a full correct-by-construction method and focus only on assisting a well-defined part of the software development process. This is the case of Alloy [135], a language and a tool for modelling systems that are suited to assist the specification and abstract design activities. Similarly, Z [136] is customarily used as a system modelling language, although there also exists a well-established theory of refinement for Z [137].

Formal approaches that do not have the generality of correct-by-construction methods can benefit from software testing to provide some degree of assurance that the derived implementations comply with the corresponding requirement specifications. Anyway, even correct-by-construction approaches might require testing, to cope with the *weak* (i.e., unproved)

points of the refinement and translation chain, or simply to comply with certification requirements [125].

2.4.3 Automated test generation for Scade models

Scade can be regarded as a formal modelling approach focused on the detailed design and implementation phases of the software lifecycle. The Scade language is derived from the synchronous dataflow programming languages Lustre [138], with some programming constructs derived from the programming language Esterel [139] and from the graphical, state-machine-based language SyncCharts [140]. Scade has formally defined semantics. All its constructs are computable, and therefore it is suited to express concrete designs rather than requirements and high-level system models. The Scade Suite development environment provides a model-based test coverage measurement tool that, from a Scade model and a test suite, calculates the coverage of different categories of elements in the model (states, transitions, conditions in transition guards, MC/DC coverage).

Lakehal and Parissis proposed research works that address automated test generation for Scade or Lustre [141]. This work introduces a set of coverage criteria for Lustre and Scade programs, defined over the graph of operators in the programs, and an automated tool that builds test suites that maximize these coverage criteria. The performance of the test generator is assessed by measuring the mutation analysis [142]. Lutess [143] generates test cases at random based on a description of the environment. Lurette [144] and Gatel [145] focused on generating test cases for invariants or safety properties described in Lustre. These approaches could be extended for programs in Scade, but none of them deals with automatically generating test cases for achieving high structural coverage as we investigate in this thesis. We aim to systematically analyze the C code that corresponds to the Scade programs, while the approach of [141] does not consider the generated C code. The authors of [141] propose dedicated coverage measures, specific for synchronous dataflow programming languages, while we aim at covering all execution paths in the programs.

Wakankar et al. demonstrate an automated test generation for Scade models based on model checking. Scade models are manually translated to SAL [146] is a specification language that provides the way to represent the specification of the system to transition system, and the SAL-ATG [147] tool automatically generated test scripts for that SAL models and then translated back into Scade simulation input file format[148]. This approach is different from our approach due to manual translation from Scade to SAL model, and SAL-ATG takes system requirements and the property as the input

and generates counter examples, and the test goals to be specified as trap variables are also defined manually [148].

Toennemann et al. introduce the code-to-model transformation concept which manually creates the Scade model from equivalent C code and transfers the existing test cases for the C Code to SCADE test format automatically [149]. The motivation of this work is to enable original equipment manufacturers (OEMs) to further use and maintain legacy code in new development environments. This approach differs from our work because, the core idea behind this work is to transform of code-to-model concept and with the generated test cases, test the model automatically. But the main two parts such as code to model conversion and test case extension are manual work [149].

An interesting tool is RT-Tester [150, 151], which is used in industry to perform V&V activities for avionic, automotive and railway systems: it starts from a concrete test model describing the expected behaviour of the system under test, renders the models into a set of expressions in propositional logic and then solves the formulas with an SMT solver to generate test cases. It works similarly to bounded model checking, representing the execution semantics with propositional logic, and solving propositional formulas that capture test cases built according to a given testing strategy. We found very limited experimental data on the effectiveness of this approach in the available papers.

Chapter 3

Automated Test Generation for Scade programs

This chapter discusses Tecs*, our original approach to automatically generating test cases in a systematic fashion, based on symbolic execution and model checking, for safety-critical programs in Scade. We first introduce the distinctive characteristics of safety-critical programs developed in Scade, which, as the research hypothesis of our work, should enable the effectiveness of generating test cases with symbolic execution and bounded model checking, and we present a simple Scade program that we use as a working example while presenting our approach. We then introduce the approach* Tecs*, embodied in our original test generator (itself called* Tecs*).* Tecs *built on the symbolic executor* Klee *and the bounded model checker* Cbmc*, but it also makes several distinctive design choices that explicitly exploit the programming constraints guaranteed for programs in Scade.*

3.1 Safety-Critical Development with Scade

Scade is a system modelling language that allows the design, implementation and verification of reliable embedded software systems. Ansys Inc. develops the language and commercializes the Scade Suite development environment, which allows the design of embedded cyber-physical systems based on the Scade language, simulates their behaviour, and generates quantifiable/certifiable code from the models. Scade is customarily used to develop high-assurance and safety-critical embedded systems in a wide range of application domains such as, e.g., avionics, automotive and railway. The key safety objectives are *Synchronous*, they are fully deterministic models, simple and steady, only safe constructs, modular, typed, good matching between

language and graphical diagrams (semantics and intuition), and runs in finite memory.

The Scade modelling language belongs to the family of synchronous languages, such as LUSTRE [138] and ESTEREL [139]. Synchronous languages assume that all the communications and computations in the systems that their models represent are performed instantaneously. A Scade model is reactive and structured as a collection of communicating finite-state machines, procedures and functions. States are the fundamental memory component of state machines, and each state machine must start in a unique state and can have one or more states. State transitions are the elements that connect one state to another. When there is a strong transition, it deactivates the source state and activates the target state, allowing it to begin its internal activity. In cases of weak transition, it permits the source state to complete its activity while delaying the start of the action of the destination state until the following cycle. Each state may have a hierarchical structure, similar in spirit to, but with richer semantics than, the Statecharts [152] or UML state machine languages [124]. The computation of a Scade model is performed as a sequence of discrete steps referred to as *execution cycles*. At each execution cycle, the outputs and the next state of the model are calculated from the inputs and the current state. At the end of a cycle, the execution of the model performs an instantaneous transition to the next state as it enters the next cycle. A valid Scade model must enjoy the property of running each execution cycle in bounded space and time, and Scade rejects models that are not deterministic or not deadlock-free. Scade has both a textual and an equivalent graphical syntax, and the SCADE Suite development environment allows editing a model in either format.

Integrated into the SCADE Suite development environment, the automatic code generator KCG translates the Scade models to semantically equivalent programs in either the Ada or the C programming language[1]. The programs generated by KCG are provably equivalent to the Scade models of which they are a translation. By virtue of the aforementioned properties of the Scade models, KCG is able to translate them to C programs that also are deterministic, deadlock-free, and run in bounded space and time. Moreover, in compliance with the Scade language, KCG aims to ensure that the generated programs are both *embeddable*, i.e., deployable in embedded, resource-constrained environments, and *compliant* with the most demanding safety levels of certification standards as, e.g., DO-178C [1], IEC 61508 [153], EN 50128 [2], and ISO 26262 [154]. To this end, KCG translates a Scade model

[1]SCADE Suite invokes a third-party Ada or C cross-compiler to generate binary code for the target platform of choice from the KCG translation of a model.

to a program expressed in a suitable subset of the C programming language that does not contain programming constructs that are deemed *intrinsically unsafe* or unfriendly with resource-constrained environments. A more precise characterization of the C language subset that KCG uses as a target for the translation of Scade models follows: A Scade program is structured as a collection of communicating components, each designed as a state machine or as a pure dataflow component, where the outputs depend on the inputs and an internal state, or stateless functions, where the outputs depend directly on the inputs. The computation of a Scade program proceeds as a sequence of discrete steps referred to as execution cycles. At each execution cycle the outputs and the next state of each component are calculated based on the current inputs and the current state, and at the end of a cycle, all components perform an instantaneous transition to the next state as they enter the next cycle. Furthermore:

- Its semantics is unambiguous and precise (e.g., no undefined behaviours);
- It is ISO C18 compliant;
- It conforms to the MISRA C 2012 coding standard rules;
- All the memory objects have either static or automatic storage duration, i.e., there is no use of dynamic or thread-local memory; Moreover, variable length array types are not used;
- It has no recursive function calls;
- All loops are statically bounded: Their number of iterations is determined by constant values known at code generation time;
- It uses as statements only selections (`if`), iterations (`for`, `while`, `do ...while`), function calls, non-compound assignments, `return`s, and blocks; Moreover, the controlling or optional expressions in the selection and iteration statements, the expressions denoting the called function and the arguments in function calls, the left and right operands in assignments, and the operand of `return` statements have no side effects;
- Array elements are always accessed by the declaration name of an array-typed variable or field, via the array subscript operator with a numeric index; There is no use of the array subscript operator with pointers that are not explicitly declared as arrays;

- Except in the case of accessing array elements via the array subscript operator with a numeric index, there is no dynamic address calculation ("pointer arithmetic" expressions) and no casting of memory addresses to/from other types; Pointer types are only used in the declarations of formal parameters of functions, to implement "by pointer" parameter passing, and enforcing that the formal and the actual parameters are exactly of the same type for any calls;

- The indices of all array accesses vary in intervals whose left and right bounds are constants known at code generation time, and always within the range of the definition of the corresponding array; As a consequence, all the array accesses are statically guaranteed to be in-bound w.r.t. the corresponding array.

The restrictions over the C language adopted by KCG are motivated by the required compliance with the highest safety levels of the certification standards that the generated code must address. These standards discourage, or utterly forbid, the use of dynamic memory, unrestricted aliasing, unbounded iteration and recursion, to ensure that the program always runs in bounded space and time. Furthermore, KCG does not ever produce recursive data structures when translating Scade programs in C: indeed, the main purpose of recursive data structures is implementing unbounded containers, but since a well-formed Scade model always runs in bounded space there is no real need for its C translation to use unbounded containers. Therefore, the nature of Scade models— they are being deterministic, deadlock-free, and bounded in space and in time—is precisely what allows such a limited fragment of the C language to adequately express the full semantics of the Scade language.

In the target environment, the embedded software must interact with the sensors and the actuators of the hardware platform. In order to link the Scade programs to the hardware developers must implement suitable *glue code*, i.e., peripheral drivers, interfacing the KCG code generated from a Scade model and the external environment.

3.2 A Sample Scade Program (Working Example)

We will use a simple Scade model to introduce the main concepts and terminology about Scade, and to show how a Scade model is converted into C code: this will help for better understanding how our approach described in Section 3.3 works.

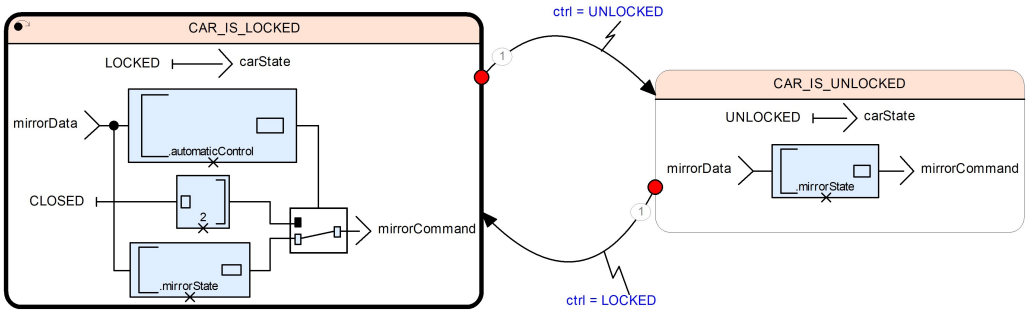

Figure 3.1: A sample Scade model for a car wing mirror controller

Figure 3.1 shows a Scade model that describes a simple controller for the wing mirrors of a car, for which it is possible to activate the behaviour of closing the wing mirrors automatically when the car gets locked. The state machine has two states (the boxes in the left and right part of the figure, respectively) that represent whether the car is either locked or unlocked, respectively. The input signal *ctrl* governs the possible transitions between these two states. The program starts in the state CAR_IS_LOCKED (the state on the left of the figure) and then if *ctrl* gets set to $UNLOCKED$ the program changes state to $CAR\_IS\_UNLOCKED$ (the state on the right of the figure). Conversely, if *ctrl* gets set to $LOCKED$ the program returns to CAR_IS_LOCKED. The signal *ctrl* can be thought of as the input that the car receives from a remote controller.

The program has three further inputs and three outputs. The three inputs are aggregated in the data structure *mirrorData*, which is referred in both states of the Scade models in Figure 3.1. The data structure *mirrorData* consists of two fields. Field *mirrorData.automaticControl* (dereferenced with the Scade operator represented as a rectangle in the top part of state CAR_IS_LOCKED) controls whether or not the automatic-closing behaviour is currently active. Field *mirrorData.mirrorState* (dereferenced in both program states) is an array of two items, each defining the latest state (either $OPEN$ or $CLOSED$) that the driver has set for either wing mirror. The three outputs are *carState*, which records the current state of the car, and the two items of the array *mirrorCommand*, which indicate the commands (either $OPEN$ or $CLOSED$) sent to the wing mirrors. The *carState* output is simply assigned as $LOCKED$ or $UNLOCKED$ in the two states of the program, respectively. Scade represents the assignment with an arrow that connects a value to the receiving variable, e.g., $LOCKED$ *!* *carState* represents the assignment of the output *carState* in the program state CAR_IS_LOCKED.

The main behaviour of the program is to define the commands sent to the wing mirrors when the control system is in each of the two program states, respectively. If the automatic closing behaviour is active, the wing mirrors shall close automatically upon locking the car. Otherwise, they shall just remain as they are. Upon unlocking the car, the wing mirrors shall always return as they were when the car got locked. The program encodes this behaviour as follows. When the car gets locked (state CAR_IS_LOCKED) the outputs *mirrorCommand* are assigned with the if-then-else block represented as the white rectangle in the bottom-right part of state CAR_IS_LOCKED in Figure 3.2. The if-then-else block takes *mirrorData.automaticControl* as a condition (entering from the top of the block): If the automatic control is active, the outputs *mirrorCommand* are both assigned as the constant $CLOSED$ (entering at the top-left corner of the block). Otherwise, if the automatic control is not active, they are assigned to the values in the array *mirrorData.mirrorState* (entering at the bottom-left corner of the block). When the car gets unlocked (state $CAR\_IS\_UNLOCKED$) the outputs *mirrorCommand* are always assigned the values of *mirrorData.mirrorState*.

Compiling the Scade program of Figure 3.1 with KCG yields the C program excerpted in Figure 3.2. The program defines the entry function `WingMirrorControl_CarControl` (excerpted at the bottom of the figure) that encodes the behaviour of the system. This function will be continuously executed at each execution cycle on the target board. As parameters, the function takes pointers to two data structures `inC` and `outC` of type `inC_WingMirrorControl_CarControl` and `outC_WingMirrorControl_CarControl`, respectively: `inC` wraps the inputs that the state machine receives at the beginning of each execution cycle, and `outC` wraps the outputs of the state machine, along with a special field (`WingMirrorFSM_state_nxt`) that KCG generates to encode the next state of the state machine after each execution cycle. The top part of the code lists the type definitions for both `inC` and `outC` data structures, and their nested types.

The body of the entry function consists of two switch statements executed in sequence. The first switch statement calculates the next state, and stores it in the temporary variable `WingMirrorFSM_state_act`. The second switch statement calculates the outputs and assigns the fields of `outC`. For example, when the first switch statement computes the next state `SSM_st_CAR_IS_UNLOCKED_WingMirrorFSM`, corresponding to the model state $CAR\_IS\_UNLOCKED$, the second switch statement assigns the outputs `outC->mirrorCommand` to the values of the inputs `inC->wingMirrorData.mirrorState`, the output `outC->carState`

```
1  typedef struct f
2    Lock ctrl;
3    MirrorData mirrorData;
4  g inC_WingMirrorControl_CarControl;
5  typedef struct f
6    MirrorStateArray mirrorCommand;
7    Lock carState;
8    SSM_ST_WingMirrorFSM WingMirrorFSM_state_nxt;
9  g outC_WingMirrorControl_CarControl;
10 typedef struct f
11   kcg_bool automaticControl;
12   MirrorStateArray mirrorState;
13 g MirrorData;
14
15 typedef MirrorState MirrorStateArray[2];
16 typedef enum fUNLOCKED, LOCKEDg Lock;
17 typedef enum fOPEN, CLOSEDg MirrorState;
18
19 void WingMirrorControl_CarControl(
20 inC_WingMirrorControl_CarControl *inC,
21 outC_WingMirrorControl_CarControl *outC) f
22   SSM_ST_WingMirrorFSM WingMirrorFSM_state_act;
23   kcg_size idx;
24   switch (outC->WingMirrorFSM_state_nxt) f
25     case SSM_st_CAR_IS_UNLOCKED_WingMirrorFSM:
26     if (inC->ctrl == LOCKED) f
27       WingMirrorFSM_state_act = SSM_st_CAR_IS_LOCKED_WingMirrorFSM;
28     g
29     else f
30       WingMirrorFSM_state_act = SSM_st_CAR_IS_UNLOCKED_WingMirrorFSM;
31     g
32     break;
33     case ...
34   g
35   switch (WingMirrorFSM_state_act) f
36     case SSM_st_CAR_IS_UNLOCKED_WingMirrorFSM:
37     kcg_copy_WingMirrorArray(outC->mirrorCommand, inC->mirrorData.mirrorState);
38     outC->carState = UNLOCKED;
39     outC->WingMirrorFSM_state_nxt = SSM_st_CAR_IS_UNLOCKED_WingMirrorFSM;
40     break;
41     case ...
42   g
43 g
```

Figure 3.2: Excerpt of the C program that KCG generates for the Scade model in Figure 3.1

to `UNLOCKED`, and the output `outC->WingMirrorFSM_state_nxt` to `SSM_st_CAR_IS_UNLOCKED_WingMirrorFSM`.

3.3 The TECS Toolchain

Our toolchain Tecs, *Test Engine for Critical software in Scade*, aimed at generating test cases for Scade programs. Figure 3.3 illustrates the components and the workflow of Tecs: the input is a Scade program developed with the Scade Suite development environment (top left part of the figure), and the output is a test suite that can be executed with Scade Test, the test execution environment of Scade Suite (bottom left part of the figure). Since we target unit-level testing, here on in this report we use the term *Scade program* to generally refer to the Scade component under test, which can be itself part of a larger Scade program.

Tecs relies on the Kcg compiler to convert the Scade program under test into an equivalent C program. Then, Tecs includes a *Test driver synthesis component* that augments the obtained C program with an analysis driver written itself in C. The analysis driver embodies the actual analysis algorithm that Tecs uses to explore the state space of the program under test: It assigns the program inputs with symbolic values and bounded values, and then calls the original program multiple times, aiming to trigger the possible transitions of the state machine model that the Scade program represents. Thus, by executing the analysis driver, Tecs steers multiple analysis passes of the execution paths in the program, with each new pass depending on the results of the previous pass. As we explain in detail in Section 3.3.2, the Test driver synthesizer tailors the analysis algorithm to the specific signature of the program under test.

To accomplish symbolic execution and bounded model checking according to the analysis algorithm provided with the analysis driver, Tecs relies on Klee and Cbmc, a well-known state-of-the-art symbolic executor for programs in C [25] and state-of-the-art bounded model checking for programs in C [31], respectively.

As the final step, Tecs constructs a Scade test case (Figure 3.3, *Test synthesis*) for each of the selected C tests. It thus obtains a test suite in Scade format, which can be executed within the Scade test environment. The Test synthesis is rather an engineering effort, though important to finalize the generated test suites.

Below we describe the components that comprise Tecs in detail and its workflow.

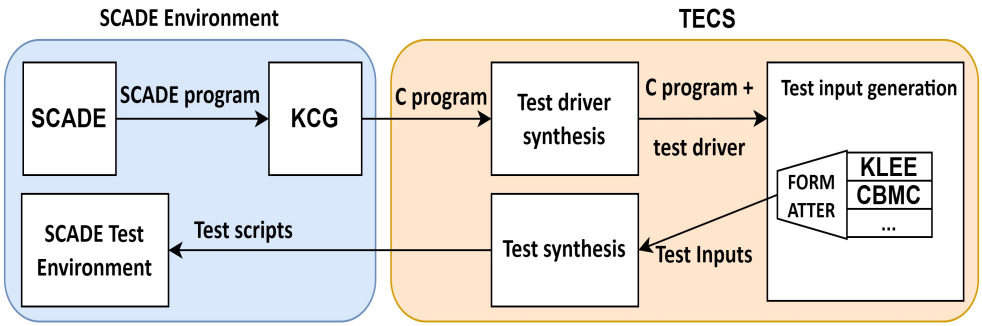

Figure 3.3: Components and workflow of TECS

3.3.1 KCG

TECS relies on the KCG compiler, a cross-compilation utility that is part of SCADE Suite, to convert the Scade program under test into a semantically equivalent program in the C programming language (Figure 3.3, *C program*). The C program encodes the execution cycle semantics of the corresponding Scade program: Given current values for the inputs, the outputs and the state, it computes the new values of the outputs and the next state at the end of the execution cycle.

The C programs generated by KCG are provably equivalent to the Scade programs of which they are a translation. In particular, to comply with the semantics of Scade, KCG produces C programs that are deterministic, deadlock-free, and run in bounded space and time[2]. To this end, KCG generates C code that dismisses some constructs out of the expressive power of the C language, as we already explained in Section 3.1.

3.3.2 Test Driver Synthesis

The C programs generated with KCG cannot be exploited *as-they-are* for the sake of generating proper test cases for the corresponding Scade programs. In fact, a KCG-generated C program implements a single execution cycle of the Scade program from which it was obtained. To exercise the relevant behaviours of a Scade component, we aim at generating test cases that run suitable sequences of execution cycles of the component. A proper

[2]This also ensures compliance with the most demanding safety levels of certi cation standards (as, e.g., DO-178C [1], IEC 61508 [153], EN 50128 [2], and ISO 26262 [154]), which require that the program always runs in bounded space and time.

SCADE test case shall start from an initial state in which all outputs and the state of the Scade component under test are set to default values, and then progress by running multiple execution cycles of the component, setting suitable inputs at each cycle.

To steer the execution of test cases against the C programs generated with KCG, TECS enriches each target C program with a test driver. The test driver represents the execution of a test case that, as we described above, first initializes all output and state variables to valid default values, and then executes a sequence of calls of the target C program, by allowing test generators to pass suitable input values at each call.

With reference to Figure 3.3, the task of generating the code of the test driver is carried out in the *Test driver synthesis* step of the toolchain. This step results in a C program inclusive of the test driver, which can be exploited with test generators to explore the possible sequences of execution cycles of the program under test.

TECS generates the test driver for a given program under test by customizing the template code shown in Figure 3.4. Specifically, it will customize the lines marked with the comment "*Adapt wrt* KCG *code*" in the figure, by replacing the type and function names showed as italic text, with the corresponding type and function names defined in the C program generated with KCG, as follows. Lines 2 and 3 declare program variables that instantiate the inputs and the outputs of the program under test, respectively, where the type names *InputType* and *OutputType* shall be replaced with the specific types of the input and output data structures defined in the C program. Line 4 calls the function *init* that sets the initial values of the outputs: *init* shall be replaced with the specific init function that KCG defined as part of the C program. Yet, line 10 calls the function *program* that represents the execution cycle semantics of the component under test: *program* shall be replaced with the specific name of the component, as defined in the C program generated with KCG.

When executed, the test driver proceeds as follows. It relies on the init function generated by KCG (Figure 3.4, line 4) to initialize the values in the output data structure. This structure includes a field for each program output as well the field *out.state* that represents the current program state. Then, the test driver iterates through the loop at lines 7– 12, where it executes the function *program* multiple times (line 10). Each execution of *program*, that is, each execution cycle of the Scade program under test, receives dedicated input values that the test driver sets by calling function *takeInputs* (line 9). The loop iterates as long as the value of field *out.state* corresponds to a program state not yet visited at a previous iteration. This allows for exercising sequences of execution cycles in the scope of the single-state-path-

coverage (SSPC) testing criterion, i.e., execution sequences that traverse at most once the states of the state machine that comprises the Scade program under test.

The call to function *takeInputs* (Figure 3.4, line 9) encapsulates the logic for the test driver to receive new input values at each execution cycle. As explained in the next section, it also allows generators to exploit the test driver for controlling the test generation process. Figure 3.4 further specifies the logic of function *takeInputs* at lines 14–17: It first enumerates all fields at any nesting level of the input data structure (function *enumerateFields*, line 15), allocating fresh memory to all pointer-typed fields and returning the references to all leaf, non-pointer fields, and then assigns a new value to each leaf field separately (function *provideInputs*, line 16).

To enumerate the fields of the input data structure, Tecs (via the test driver) exploits the knowledge that, based on the semantics of Scade and the guarantees from Kcg, all data structures are statically allocated and not recursive, the size of all arrays is statically specified, and there is no pointer aliasing. This implies that the input data structures are always made of a finite set of statically identifiable fields, including the elements of the array-typed fields. In this way, Tecs induces a specialized, efficient input-provision mechanism, which is specific for testing Scade programs and has the advantage of not having to cope with null pointers or pointer aliasing.

Technically, the Tecs generates the code of function *enumerateFields* (Figure 3.4, line 15) by relying on ANTLR4 [155] to parse the type definitions of all fields of the input data structure, as given in the C program generated by Kcg. Each leaf field is then represented as a structure (denoted as *LeafField* at line 15) that includes an identifier label for the field, and a pointer to the memory allocated for containing the value of the field. By naming convention, the identifier label of the leaf fields includes i) the name of the field, ii) the primitive type of the field and iii) the number of the execution cycle in which they will be used. For example, the identifier `in::a::b_int_1` would represent the *int*-typed field b of the sub-structure a within the input structure in, as assigned at the first execution cycle.

Function *provideInputs* takes the responsibility to fill input values into the fields of the input data structure. This function represents the API that we must implement for integrating any given test generator in the toolchain, in order to delegate the test generator to control the program inputs while accomplishing the test generation tasks.

```
1: function TestDriver
2:     InputType in;                                  ▷ Adapt wrt Kcg code
3:     OutputType out;                                ▷ Adapt wrt Kcg code
4:     init(&out);                                    ▷ Adapt wrt Kcg code
5:     int cycle = 1;
6:     Set visited = empty_set();
7:     while (!contains(visited, out.state)) do
8:         add(visited, out.state);
9:         takeInputs(&in, cycle);
10:        program(&in, &out);                        ▷ Adapt wrt Kcg code
11:        cycle = cycle + 1;
12:    end while
13: end function

14: function takeInputs(InputType *in, int cycle)
15:    LeafField[] leafFields = enumerateFields(in, cycle);
16:    provideInputs(leafFields);                     ▷ Input provider API
17: end function
```

Figure 3.4: Algorithm of the analysis driver

3.3.3 Test Input Generation

Our toolchain Tecs leverages state-of-the-art test generators for C programs (Figure 3.3, step *Test input generation*), in order to produce test inputs for exercising the Scade programs under test. This step consists in executing the given test generator on the C program that contains the test driver, as follows:

(i) we provide a test-generator-specific implementation of the API *provideInputs* called by the test driver (Figure 3.4, line 16),

(ii) we compile the C program along with the test driver, the provided implementation of *provideInputs* and a main function that calls the test driver,

(iii) we execute the test generator on the program, and let the test generator generate test data for the program.

(iv) we post-process the test data (that each given test generator produces in its specific output format) to render them in a common format (Figure 3.3, *formatter*).

Below we explain the implementations of *provideInputs* that allow TECS to work with the test generators KLEE and CBMC.

Integrating TECS with KLEE

KLEE generates test cases based on symbolic execution. Working as a symbolic executor, it models the input values as unconstrained symbols, interprets the statements in the program in function of the input symbols, and computes the execution conditions of the program paths as logic constraints over the input symbols. Finally, KLEE solves the execution conditions of the analyzed program paths with an SMT solver (e.g., STP [112] or Z3 [106]) to obtain concrete inputs that make those program paths execute.

In our setting, KLEE executes the intermediate binary code of the program compiled with LLVM.

To make TECS work with KLEE, we link the program to an implementation of the API *provideInputs* that assigns the relevant inputs with symbolic values by using the primitive *klee_make_symbolic* provided from KLEE. Specifically, the implementation of *provideInputs* calls *klee_make_symbolic* for all primitive inputs enumerated in test driver at each execution cycle (Figure 3.4, line 15) as follows:

klee_make_symbolic(*elds[i].r*, sizeof(* *elds[i].r*), *elds[i].l*);

where *elds[i]* represents the i^{th} input field received as input of *provideInputs*, and *elds[i].r* and *elds[i].l* represent the memory address and the identifier label of that field, respectively.

Let us consider, for instance, the working example program that we introduced in Figure 3.1. Concerning the corresponding C program of Figure 3.2, the analysis driver synthesis that the data structure of the type `inC_WingMirrorControl_CarControl`, which represents the program inputs, includes a field `ctrl` and a field `WingMirrorData`, respectively. The former field is defined as an enumeration type, i.e., a primitive type, and the latter field is an array, i.e., a non-primitive type. Thus the Driver synthesis component (Figure 3.4, line 16) inspects the definition of the array, revealing that it consists of two items of primitive types (again an enumeration). For this program, the Driver synthesis component allows deriving the C code for 2 cycles. The generated main function ultimately consists of C code that initializes a new instance of the data structure in memory for each state cycle and which relies on *Input provider API* to initialize the primitive field `inC_step_0__ctrl` with a new fresh symbol (Figure 3.5, line 2), initializes the non-primitive field `inC_step_0__wingMirrorData` as a

new array instance with two items and initializes the two items in the array with further fresh symbols (line 8). The operation *klee_make_symbolic* for initializing the inputs with fresh symbols takes three parameters: one is the input to be initialized passed by reference, the second is the size of the variable and the other one is a name (a string of characters) to be associated to that symbolic values. For the enumeration type, the driver function finds the range of the enum list and provides a *klee_assume(), line 5* function to make fresh values within the range. Upon generating test inputs as possible concrete values of the symbols, KLEE will use the provided name `"step:0,dataType:kcg_bool,parameter:inC_step_0__wingMirrorData__automaticControl"` (line 10) to indicate the input data to which those values refer. This kind of name conversion is foreseen to support the engineering work of the Test Synthesis component. This process will repeat for all the inputs up to *n* cycles for the target function.

In this way, running KLEE at step *Test input generation*, we obtain test inputs for the program paths that traverse the program under test through the test driver, i.e., the program paths visited when calling *program* (Figure 3.4, line 10) multiple times in the while loop of the test driver.

Integrating TECS with CBMC

CBMC generates test cases according to bounded model checking. It encodes the semantics of the statement in the target C program as a boolean formula, expresses a reachability problem for each branch in the program as a constraint to be evaluated in conjunction with the program formula, and then computes test inputs for each branch by solving the reachability problems with a constraint solver.

In our setting, CBMC works directly on the source code of the program, targeting the main function that in turn calls the test driver.

Bounded model checking is a radically different type of static analysis with respect to symbolic execution, however, CBMC and KLEE are similar in the requirement of having to mark the inputs to be handled in their constraint-solving problems. In CBMC the relevant inputs must be marked as nondeterministic values by means of a group of API functions that begin with the prefix *nondet_*. Thus, for CBMC, we provide an implementation of the API *provideInputs* that suitably calls the *nondet_* functions for each primitive field of the input data structure of the program under test, at each execution cycle.

For instance, the working example program (Figure 3.1) concerning the corresponding C program of Figure 3.2, the analysis driver synthesis integrated with CBMC have the same engineering methods but few differences

```
1  inC_MirrorChecker_StateMachines inC_step_0;
2  lock inC_step_0__ctrl;
3  klee_make_symbolic(&inC_step_0__ctrl, sizeof inC_step_0__ctrl,
4  "fstep:0,dataType:enum kcg_tag_lock,parameter:inC_step_0__ctrlg");
5  klee_assume(inC_step_0__ctrl >= UNLOCKED & inC_step_0__ctrl <= LOCKED);
6  inC_step_0.ctrl=inC_step_0__ctrl;
7
8  WingMirrors inC_step_0__wingMirrorData;
9  kcg_bool inC_step_0__wingMirrorData__automaticControl;
10 klee_make_symbolic(&inC_step_0__wingMirrorData__automaticControl,
11 sizeof inC_step_0__wingMirrorData__automaticControl,
12 "fstep:0,dataType:kcg_bool,
13 parameter:inC_step_0__wingMirrorData__automaticControlg");
14 inC_step_0__wingMirrorData.automaticControl=
15 inC_step_0__wingMirrorData__automaticControl;
16 WingMirrorState inC_step_0__wingMirrorData__mirrorState[2];
17 WingMirrorState inC_step_0__wingMirrorData__mirrorState____0;
18 klee_make_symbolic(&inC_step_0__wingMirrorData__mirrorState____0,
19 sizeof inC_step_0__wingMirrorData__mirrorState____0,
20 "fstep:0,dataType:enumkcg_tag_WingMirrorState,
21 parameter:inC_step_0__wingMirrorData__mirrorState____[0]g");
22 klee_assume(inC_step_0__wingMirrorData__mirrorState____0 >= OPEN &
23 inC_step_0__wingMirrorData__mirrorState____0 <= CLOSED);
24 inC_step_0__wingMirrorData.mirrorState[0] =
25 inC_step_0__wingMirrorData__mirrorState____0;
26 WingMirrorState inC_step_0__wingMirrorData__mirrorState____1;
27 klee_make_symbolic(&inC_step_0__wingMirrorData__mirrorState____1,
28 sizeof inC_step_0__wingMirrorData__mirrorState____1,
29 "fstep:0,dataType:enum kcg_tag_WingMirrorState,
30 parameter:inC_step_0__wingMirrorData__mirrorState____[1]g");
31 klee_assume(inC_step_0__wingMirrorData__mirrorState____1 >=
32 OPEN & inC_step_0__wingMirrorData__mirrorState____1 <= CLOSED);
33 inC_step_0__wingMirrorData.mirrorState[1] =
34 inC_step_0__wingMirrorData__mirrorState____1;
35 inC_step_0.wingMirrorData=inC_step_0__wingMirrorData;
```

Figure 3.5: KLEE representation for the inputs used in of the C program Figure 3.2

with KLEE. Figure 3.6 shows the generated main function for TECS integrated with CBMC. The two differences are *__CPROVER_input()* (Line 3), which has two parameters: one is the name provided to the input data to which those values refer, second is for the provide the fresh boundary values for the variable and then,*__CPROVER_assume()* (Line 5) to control fresh boundary values within the range.

CBMC requires some special care in the way we can associate the nondeterministic inputs with corresponding identifier labels, which is needed for being able to interpret the results from the test generator. As we already explained, this task is carried out in the test driver in the code of function *enumerateFields* (Figure 3.4, line 15) by producing a string that concatenates the data about the field name, its type, and the number of the current execution cycle. However, since CBMC does not interpret the string operators in its formulas, to work with CBMC, we cannot use string concatenation in the code of the test driver. Conversely, we must produce the code of function *enumerateFields* such that it looks up the identifier labels from a statically unfolded list. This results in the additional requirement of statically specifying the maximum number of execution cycles to be handled. This is arguably a limitation we incur with CBMC, but not with the other approaches considered in this thesis.

3.3.4 Test Synthesis

The last step of the toolchain (Figure 3.3, *Test synthesis*) renders each generated test input as a test script for SCADE Test. This step exploits the identifier labels that the test driver associated with the test inputs, in order to map each input value with specific input fields, correct types and proper execution cycles in the test scripts.

To synthesize the test cases in SCADE Test format, the TECS Test synthesis renders the test inputs that TECS yielded for a given execution path in the form of suitable *SSM::set* test statements and renders the regression oracles that TECS yielded for that path in the form of suitable *SSM::check* test statements. For the execution paths that TECS explored by issuing multiple calls of the program under test, the corresponding test cases shall include a separate test step (*SSM::cycle*) for each program call, and the Test synthesis shall consistently map the test inputs that correspond to each program call with the inputs of each step within the SCADE test cases.

The Test synthesis relies on a set of naming conventions that the analysis driver enforces when defining the names for the fresh values. In detail, the analysis driver makes sure that the name of each fresh symbol specifies (i) the name of the input field initialized with the fresh symbol, (ii) the type of

```
1  inC_MirrorChecker_StateMachines inC_step_0;
2  lock inC_step_0__ctrl;
3  __CPROVER_input("fstep:0,dataType:enum kcg_tag_lock,
4  parameter:inC_step_0__ctrlg",inC_step_0__ctrl);
5  __CPROVER_assume(inC_step_0__ctrl >= UNLOCKED &
6  inC_step_0__ctrl <= LOCKED);
7  inC_step_0.ctrl=inC_step_0__ctrl;
8
9  WingMirrors inC_step_0__wingMirrorData;
10 kcg_bool inC_step_0__wingMirrorData__automaticControl;
11 __CPROVER_input("fstep:0,dataType:kcg_bool,
12 parameter:inC_step_0__wingMirrorData__automaticControlg",
13 inC_step_0__wingMirrorData__automaticControl);
14 inC_step_0__wingMirrorData.automaticControl=
15 inC_step_0__wingMirrorData__automaticControl;
16 WingMirrorState inC_step_0__wingMirrorData__mirrorState[2];
17 WingMirrorState inC_step_0__wingMirrorData__mirrorState____0;
18 __CPROVER_input("fstep:0,dataType:enum kcg_tag_WingMirrorState,
19 parameter:inC_step_0__wingMirrorData__mirrorState____0g",
20 inC_step_0__wingMirrorData__mirrorState____0);
21 __CPROVER_assume(inC_step_0__wingMirrorData__mirrorState____0 >= OPEN &
22 inC_step_0__wingMirrorData__mirrorState____0 <= CLOSED);
23 inC_step_0__wingMirrorData.mirrorState[0] =
24 inC_step_0__wingMirrorData__mirrorState____0;
25 WingMirrorState inC_step_0__wingMirrorData__mirrorState____1;
26 __CPROVER_input("fstep:0,dataType:enum kcg_tag_WingMirrorState,
27 parameter:inC_step_0__wingMirrorData__mirrorState____1g",
28 inC_step_0__wingMirrorData__mirrorState____1);
29 __CPROVER_assume(inC_step_0__wingMirrorData__mirrorState____1 >=
30 OPEN & inC_step_0__wingMirrorData__mirrorState____1 <= CLOSED);
31 inC_step_0__wingMirrorData.mirrorState[1] =
32 inC_step_0__wingMirrorData__mirrorState____1;
33 inC_step_0.wingMirrorData=inC_step_0__wingMirrorData;
```

Figure 3.6: CBMC representation for the inputs used in of the C program Figure 3.2

Name of the fresh symbol (field, type, sequence) field	type	seq	Test input	enum value
inC.ctrl	enum Lock	1	0	UNLOCKED
inC.wingMirrorData.automaticControl	boolean	1	false	-
inC.wingMirrorData.mirrorState[0]	enum MirrorState	1	0	OPEN
inC.wingMirrorData.mirrorState[1]	enum MirrorState	1	0	OPEN
outC.carState	enum Lock	1	0	UNLOCKED
outC.mirrorCommand[0]	enum MirrorState	1	0	OPEN
outC.mirrorCommand[1]	enum MirrorState	1	0	OPEN
inC.ctrl	enum Lock	2	1	LOCKED
inC.wingMirrorData.automaticControl	boolean	2	true	-
inC.wingMirrorData.mirrorState[0]	enum MirrorState	2	0	OPEN
inC.wingMirrorData.mirrorState[1]	enum MirrorState	2	0	OPEN
outC.carState	enum Lock	2	1	LOCKED
outC.mirrorCommand[0]	enum MirrorState	2	1	CLOSED
outC.mirrorCommand[1]	enum MirrorState	2	1	CLOSED

(a) The test inputs that TECS generated for an execution path (through the analysis driver) for the sample Scade program of Figure 3.2

```
#####################################################
## WingMirrorControl_WingMirrorFSM, Test case: 00002
#####################################################

#Test step 1
SSM::set ctrl UNLOCKED
SSM::set wingMirrorData.automaticControl false
SSM::set wingMirrorData.mirrorState {(OPEN,OPEN)}
SSM::check carState UNLOCKED
SSM::check mirrorCommand {(OPEN, OPEN)}
SSM::cycle

#Test step 2
SSM::set ctrl LOCKED
SSM::set wingMirrorData.automaticControl true
SSM::set wingMirrorData.mirrorState {(OPEN, OPEN)}
SSM::check carState LOCKED
SSM::check mirrorCommand {(CLOSED, CLOSED)}
SSM::cycle
```

(b) The SCADE test case synthesized out of the test inputs from TECS

Figure 3.7: A test case generated for the sample program of Figure 3.2

the input field, and (iii) the sequence number of the program call for which the analysis driver instantiated the fresh symbol.

Figure 3.7 shows the test inputs (Figure 3.7.a) that TECS generates for an execution path through the analysis driver for the sample Scade program of Figure 3.2, and the SCADE test case that TECS synthesizes correspondingly (Figure 3.7.b). The figure indicates the test inputs in tabular form to improve readability. Each row of the table corresponds to a test input provided by TECS for each of the fields. The first three columns represent the name that the analysis driver associated with the fresh symbol. As we described above, each symbol name is comprised of a field-, type- and sequence-specifier. The

fourth column indicates the specific test input value that TECS returned. The fifth column shows the matching enumeration value for test inputs of enumeration types.

As the table indicates, TECS generated 14 inputs for the considered execution path. These 14 inputs refer to two subsequent calls of the program under test that occur within the execution path, as the value of the sequence-specifier, either 1 or 2, indicates that the first 7 test inputs map to the first program call, and the following 7 test inputs map to the second program call, respectively.

Thus, TECS synthesizes a SCADE test case consisting of two test steps (Figure 3.7.b). The first test step sets (*SSM::set*) `ctrl` to `UNLOCKED`, `automaticControl` to false and `mirrorState` to `OPEN` for both wing mirrors. This results in unlocking the car and opening the wing mirrors, and in fact, the test case defines the regression oracles (*SSM::check*) stating that this test step shall lead to a state in which the `carState` is equal to `UNLOCKED` and the outputs `mirrorCommand` are both set to `OPEN`. When the test case executes the statement *SSM::cycle*, SCADE executes the test step and checks the values of the outputs accordingly. The second test step switches `ctrl` to `LOCKED`, and `automaticControl` to true, then expected in the assertions that the `carState` becomes `LOCKED` while issuing `CLOSED` for both `mirrorCommand` outputs.

We added an Appendix: How to use TECS 5 which guides in building the tool from our repository for future studies and engineering works.

Chapter 4

Experiments and Results

This chapter provides the main contribution of this thesis, reporting and analysing the experience in creating test suites with Tecs *on a real industrial project for the development of on onboard signalling unit based on the ERTMS/ETCS standard for high-speed railway systems, developed with RFI. As a railway control and protection system, the ERTMS/ETCS system is safety-critical and must be certi ed before being deployed: The EN 50128 standard provides the reference framework for the development of this category of software systems. With the aim of ensuring the highest degree of software integrity, RFI is relying on the model-based Scade programming language for the development of this safety-critical software. A common tenet is to rely on Scade programming language that, by its design choices and controlled semantics, may both decrease the chances of introducing subtle faults in the programs, and mitigate the hard work for satisfying the certi cation requirements. Out of many Scade programs, we selected 37 Scade programs for the automated test generation approach. Here we are presenting a set of experiment results obtained by exploiting the* Tecs *with the selected programs.*

4.1 Experimental Assessment

We performed our experimental assessment on a benchmark of 37 Scade programs that were already available for testing while doing these experiments, belonging to the onboard signalling unit for high-speed railway systems developed at RFI. This system must comply with the ERTMS/ETCS standard, the European standard aimed at harmonizing the management, control and safety of the European high-speed railway traffic, prescribing how trains, track-side devices (e.g., transponders and radio units) and control stations

must interoperate to ensure safety objectives like train separation, speed control and automatic protection upon adverse events. The standard defines functional safety in terms of a set of procedures that suitable ensembles of ERTMS/ETCS subsystems must perform in reaction to specific events and conditions that can be signalled to them during railway operations.

We executed these 37 Scade programs with our tool Tecs and calculated the metrics such as execution time, number of test cases etc. We used the test cases generated by Tecs in the Scade test suite environment. From the Scade test suite results we found out the program faults and calculated the model coverage for the Scade programs. This information is used to answer the research questions in section 4.3.

4.2 Subject programs

We considered the 37 Scade programs listed in Table 4.1. The table defines an identifier (first column) that we use to refer to each subject program used in the report, and provides a short description (second column) of the task that each program executes.

These programs are part of the onboard signalling unit for high-speed rail that our industrial partner is currently developing. For example, the first program, `shunting`, implements the Shunting procedure. In railway terminology, shunting is the process of sorting railway vehicles into complete trains. When a train is in *shunting mode*, the on-board unit is responsible for the supervision of the speed limit that is allowed during the shunting operations, and to stop the train when it passes the defined border of the shunting area. The shunting procedure that we consider as a subject program shall handle the messages that the train receives from both the driver and the ground signalling equipment, to make decisions on when activating or deactivating the shunting mode.

The other implement several control tasks, such as checking and verifying the consistency of the data that the on-board unit receives from the ground components, computations of information for monitoring and controlling the train, rendering appropriate messages to the driver, and sending commands to the actuators. Table 4.2 summarizes the main statistics on the internal structure of the subject programs, i.e., the number of the states (column *#States*) and state transitions (columns *#Transitions*) of the state machine that corresponds to each Scade program, the number of inputs (column *#Inputs*) and outputs (column *#Outputs*) of each Scade program, and the number of lines of C code that correspond to each program after exporting it with Kcg. For the state transitions, the table reports separately the

Table 4.1: Subject programs

Subject	**#Description**
shunting	Sorts railway vehicles into a complete train
dc_1, dc_2, ..., dc_14	Check data consistency of received messages
radiohole	Deactivates radio connection supervision when the train is in a radio hole area
crossnonlx	Monitors a level crossing area that is not protected by external authorities
baliseinfo	Render's messages from on-railway transporters to the driver
emergency_1	Updates on-board data when receiving an emergency message
emergency_2	Acknowledges radio control centre when receiving an emergency message
mema	Rejects movement authorities if there are emergency messages
trackside	Receives and stores values from trackside equipments
vbc	Updates the list of known transponders
coordfromrbc	Updates the coordinate system as specified by the ground control
adfactordmi_1	Warns the driver if the railway adhesion factor is slippery
adfactordmi_2	Renders the railway adhesion factor in the GUI
driveridins	Updates the driver ID as indicated through the GUI
eirene	Stores the EIRENE number as indicated through the GUI
ertmslevel	Updates the operating level as indicated through the GUI
natvalues	Verifies the national values of the currently traversed region
networkidins	Updates the identifier of the radio network
rbcidins	Stores the ID of the radio control centre ID as indicated through the GUI
trainDataUpdate	Updates the train data stored on board
trainDataInsertion	Inserts new train data among the ones stored on board
message129	Notifies changes of train data to the radio control centre
runnumber_1	Updates the train ID on board
runnumber_2	Notifies changes of the train ID to the radio control centre

Table 4.2: Statistics of the subject programs

Subject	Scade model					C code
	#States	#Transitions		#Inputs	#Outputs	LOC( )
		weak	strong			
shunting	5	2	8	12	14	646
dc_1	1	1	-	13	7	175
dc_2	1	1	-	1	2	43
dc_3	1	1	-	5	3	95
dc_4	1	1	-	3	4	62
dc_5	1	-	1	3	1	32
dc_6	1	1	-	3	4	67
dc_7	1	-	1	3	1	32
dc_8	1	-	1	2	1	30
dc_9	1	1	-	5	15	464
dc_10	1	1	-	3	9	239
dc_11	1	1	-	1	3	69
dc_12	1	1	-	14	17	96
dc_13	1	1	-	3	7	67
dc_14	1	-	1	1	1	35
radiohole	3	2	1	2	2	361
crossnonlx	3	2	1	6	4	556
baliseinfo	1	1	0	1	2	147
emergency_1	1	1	0	9	4	865
emergency_2	1	1	0	9	6	711
mema	1	1	0	4	1	798
trackside	1	1	0	3	0	225
vbc	1	1	0	7	1	1,011
coordfromrbc	1	1	0	1	1	366
adfactordmi_1	1	1	0	3	1	125
adfactordmi_2	1	0	1	1	1	54
driveridins	1	1	0	1	1	262
eirene	1	0	1	3	1	124
ertmslevel	1	0	1	1	1	109
natvalues	1	0	1	1	1	265
networkidins	1	0	1	1	1	109
rbcidins	1	1	0	1	1	189
trainDataUpdate	1	1	0	2	19	136
trainDataInsertion	1	0	1	2	1	291
message129	1	1	0	5	1	353
runnumber_1	1	1	0	1	1	154
runnumber_2	1	1	0	4	1	116

( ) C code LOC values refer to the lines of code in the C functions specific to each Scade program, but each program includes more than 8,000 additional lines of code of data-type declarations, which define the data structures that comprise the inputs and the outputs of the programs.

number of weak and strong (non-weak) transitions, since the weak transitions count double in the sequences of transitions that TECS analyzes, as we explained in Section 3.3.2 (Algorithm 3.4). The lines of C code refer to the code within the C functions that specifically correspond to each Scade program, without counting the lines of code of the data-type definitions in those programs. In fact, each program includes more than 8,000 further lines of code that define the data types used in the C functions, and which TECS parses with ANTLR4 to instantiate the hook functions of the analysis driver.

For instance, the Scade implementation of `shunting` is a state machine with 5 states, 2 weak transitions and 8 strong transitions, in which the states and the transitions are based on computations and conditions that involve 12 input and 14 output variables, respectively, including the variables that represent the messages received and sent from onboard unit. Many subjects (all but `shunting`, `radiohole` and `crossnonlx`) implement computations that the on-board unit shall keep on repeating at each execution cycle, and thus they consist of a single state transition which represents the execution of the computation, and which keeps the program always in the same state. For instance, the `dc_1..14` programs implement data consistency checks that the onboard unit shall perform at each execution cycle. These programs define either a weak or a strong transition according to whether or not, respectively, the check that they implement depends on feedback loops with their own outputs.

At the level of the C code, the considered programs range between 30 and 1,011 lines of code (plus the code defining the data types, i.e., as said, more than 8,000 additional lines of code) being program `dc_8` and program `vbc` the smallest and the largest program, respectively.

4.3 Research Questions and Metrics

Our experimental assessment was driven by two main research questions:

- RQ1: *E ectiveness of automatic test case generation for safety-critical software in Scade.* Our work is based on the research hypothesis that i) generating test cases that achieve high structural coverage of safety-critical software requires systematic exploration of the execution space of the program under test and that ii) in the specific case of safety-critical software written in Scade, we can successfully generate test cases that achieve high structural coverage by relying on symbolic execution and bounded model checking for systematically exploring the program execution space.

Thus, the (core) research question is whether and to what extent our systematic test generation strategy, designed on top of symbolic execution and bounded model checking, is indeed effective for the considered class of industrial programs.

We addressed this research question by experimentally quantifying the following indicators:

a. Structural coverage. We aim at studying to what extent the test cases automatically generated with our approach allow for achieving high structural coverage of the programs under test.

b. Number of generated test cases. To be practically effective, our approach should result in test suites of manageable size, which can be practically handled and maintained by test engineers.

c. Fault detection. We aim at collecting empirical evidence that the generated test cases are useful to detect faults in the programs under test.

d. Comparison with search-based testing. We aim at quantifying the relative strength of the test cases that we generate with our approach, with respect to the test cases that could be generated with a search-based approach. This allows us to validate the hypothesis that a systematic exploration of the program execution space is a necessary condition to generate effective test cases.

e. Comparison with manual testing. We aim at quantifying the relative contribution of automatically generating the test cases with our approach, with respect to manually designing the test suites.

- RQ2: *E ciency of symbolic execution and bounded model checking for generating test cases in a systematic fashion for the class of safety-critical programs that we consider, i.e., safety-critical programs in Scade.* A technical research hypothesis of our work is that the programming rules fostered by Scade enable the efficiency of symbolic execution and bounded model checking, by mitigating the common issues that these techniques suffer against arbitrary programs, that is, (as we discussed in the introduction) execution space explosion, hardness of handling pointer aliases, and complexity of the constraints to be solved. In particular, while the mitigation of the issues with pointer aliases is implicit since Scade guarantees the absence of pointer aliases, the mitigation of the issues with both the size of the execution space and the complexity of the constraints shall be investigated.

We addressed this research question by experimentally quantifying the following indicators:

a. Execution time. We aim at quantifying the efficiency of our technique in terms of the time needed to complete the test generation process.

b. Size of the execution space. We aim at quantifying the size of the execution space that must be explored to accomplish the test generation process. In particular, we quantify the size of the execution space with indicators that differ in the case of generating test cases with either symbolic execution or bounded model checking, respectively:
   - symbolic execution: we quantify the size of the execution space as the number of execution paths that must be symbolically analyzed.
   - bounded model checking: we quantify the size of the execution space as the loop unfolding bound that must be set for allowing bounded model checking to complete the analysis of the execution space.

c. Complexity of the constraints to be solved. We quantify the time taken by the SMT constraint solver to provide the solutions for the constraint-solving queries generated during the analysis.

4.4 Experiment setting

Our case study consisted of a set of experiments, one for each of the subject programs listed in Table 4.2, in which we ran Tecs to generate test cases for the subject programs, executed the test cases in the Scade Suite and collected model coverage data.

We ran Tecs on a cloud facility hosted at our university, using a virtual machine equipped with Linux Ubuntu, 48 CPUs, and 150 GB of ram memory, which allowed for running multiple instances of Tecs in parallel. We integrated Tecs with version 2.3-pre of Klee and version 5.6 of Cbmc: Hereon in this section, we refer to the configurations of Tecs integrated with the backend test generators Klee or Cbmc as TecsK and TecsC, respectively.

We handled the Scade programs with Scade Suite Version 2020 R2, which includes the corresponding version of the Kcg compiler that we use to obtain the C version of the subjects programs. We executed and measured

the O-MC/DC model coverage of the test cases with the tool SCADE Test Version 2020 R2.

During the experiments, for each subject program, we tracked the number of paths that TECS identified during the symbolic execution phase as well as bounded model checking phase, measured the time that it took to complete the test generation process, counted the number of test cases that it generated, and computed the model coverage that the test cases achieve against the Scade programs.

For measuring the model coverage of the test cases we relied on the SCADE Test tool, which automatically computes the model coverage while executing the test cases. The coverage computed with SCADE Test refers cumulatively to the portion of executed states, and the modified condition/decision coverage of the transition guards.

We configured both test generators to address the code coverage of the C programs according to the criterion that, among the ones that they support, can work best to address the O-MC/DC model coverage of the Scade programs under test. TECSK runs KLEE with the option `--only-output-states-covering-new` that makes KLEE output only the test inputs covering new statements. TECSC runs CBMC with the option `--cover mcdc` that makes CBMC address MC/DC coverage on the target C programs. We purposely aimed at the strongest coverage criterion addressable with each tool, though acknowledging that both backend generators rely on criteria of different strengths.

CBMC works by statically unrolling the loops in the programs up to a maximum number of iterations. Yet, it is anyway possible to run CBMC without explicitly specifying this maximum loop unrolling depth, letting it unroll the loops up to the number of iterations defined in the code. We indeed followed this approach on the basis that Scade enforces statically known iteration bounds for the loops in the programs. Nonetheless, on six of our subjects (i.e., with reference to the first column of Table 4.7, the subjects `shunting`, `radiohole`, `crossnonlx`, `trackside`, `vbc` and `natvalues`) executing CBMC without specifying the maximum loop unrolling depth resulted in exhausting the available memory, thus terminating with an error. We, therefore, analyzed these subjects by setting the maximum loop unrolling depth of CBMC to 1000, using the command line parameter `--unwind`.

Also, In the case of programs `shunting`, `radiohole` and `crossnonlx` we were able to compare the test cases generated with TECS with manually selected test suites that were already available for those programs at the time of our experiment. We compared the manual and the automatic test suites with respect to their difference in model coverage, focusing on the items that either test suite covers and the other one does not.

For all other subject programs, the engineers at our industrial partners decided to rely directly on our tool (as Tecs in fact became available while these programs were being implemented), aiming to optimize their effort for designing and implementing the test cases for these programs.

To this end, they augmented the test cases generated with Tecs with (manually defined) assertion-style test oracles, aiming to obtain test suites that could be readily used for component-level testing of the considered programs (other than for future regression testing of those programs). This resulted in a semi-automatic approach to component-level testing empowered by our tool Tecs, and allowed us to further validate the quality of the test suites generated with Tecs in terms of usefulness for detecting component-level failures in the context of our industrial project.

We remark that, on one hand, this choice of our partner affected our ability to extensively crosscheck the differences in effectiveness of automatically and manually generated test suites, respectively, since no manual test suite existed to compare with for any subject program but `shunting`, `radiohole` and `crossnonlx`. On the other hand, we believe that the choice of dismissing fully manual testing in favour of working with semi-automatic test cases (obtained by enriching with assertions the ones generated with Tecs) supports the positive perception of our industrial partner on the effectiveness of our approach.

4.5 Results

In this section, we aim to answer the research questions by reporting the experiment results. RQ1 seeks empirical evidence that we can effectively exploit symbolic execution and bounded model checking to generate test cases for safety-critical software in Scade. As we explained in Chapter 3, Tecs concertizes this hypothesis by tailoring its implementation of symbolic execution and bounded model checking on the restrictions by which Kcg fosters by design safety guarantees in the programs.

Thus, as RQ1 states, we aim to empirically study the output provided by Tecs by measuring the size and the structural thoroughness of the generated test suites. We recall that O-MC/DC is the criterion that Scade advises for software that shall work with a high integrity level. The certification objectives are required by the highest integrity level of the safety standards. The latter metric quantifies the efficiency of Tecs to obtain the corresponding degree of (RQ1.a) structural coverage i.e., the O-MC/DC model coverage that the generated test suites achieve for the Scade programs, within an (RQ1.b) manageable size of test suites which can be handled and maintained

by test engineers. Also, we answer the usefulness of Tecs to (RQ1.c) detect program faults and quantify the relative strength of the test cases that we generate with our approach with an (RQ1.d) search-based testing approach and (RQ1.e) comparing with the manually designed test suites that were already available for three of the considered programs.

For the RQ2, we aim to experimentally quantify the efficiency of symbolic execution and bounded model checking for generating test cases in a systematic fashion for the class of safety-critical programs by answering the computational effort and (RQ2.a) time budget requirements that result from the distinctive design of Tecs and (RQ2.b) size of the execution space explored to accomplish or how long it takes overall to complete the test generation process either with symbolic execution or bounded model checking. Also, we answer the (RQ2.c) complexity of the constraints to solve by evaluating the time taken by the SMT constraint solver.

4.5.1 RQ1: Effectiveness of automatic test case generation for safety-critical software in Scade.

We report the experimental results that show the effectiveness of our tool Tecs for automatic test case generation for Scade programs by evaluating the generated test cases by considering the achieved O-MC/DC coverage, size of the generated test cases, the ability of reveal program faults and comparison with search-based testing and manual testing.

RQ1.a Structural Coverage

We investigate the structural coverage for each subject program, the Table 4.3 reports the O-MC/DC coverage obtained with the test suite generated with TecsK and TecsC, respectively (second and third columns) and the coverage that we obtain by merging the test suites from the two Tecs versions (fourth column). Here, the column titled by TecsK $\lceil$ TecsC reports the coverage of the test suite obtained by merging the test suite that either TecsK or TecsC generated for a program. We consider the coverage result from TecsK $\lceil$ TecsC as the coverage result of Tecs.

Table 4.3 (columns TecsK, TecsC, TecsK $\lceil$ TecsC) highlights in bold typeface the data of the tools that reached the highest coverage for each program, and underlines the cases in which a single tool achieved strictly higher coverage than others. TecsK achieved the highest coverage for 26 programs and TecsC achieved the highest coverage for 24 out of 37 programs. They achieved strictly higher coverage than the other tool 12 times for TecsK and 10 for TecsC. By merging test two suites, TecsK $\lceil$ TecsC achieved

the highest coverage (column TecsK $[$ TecsC) for 15 out of 37 programs, and this confirms our hypothesis that we can obtain a higher model coverage exploiting together both TecsK and TecsC. Achieve a model coverage of 100% for 5 programs, at least 90% for 24 further programs and at least 80% for 7 programs, and 73.50% in the only case of program `dc_12`.

Table 4.3: Number of test cases and O-MC/DC Coverage for TecsK and TecsC

Program	TecsK		TecsC		TecsK $[$ TecsC
	#Tests	%Cov	#Tests	%Cov	%Cov
shunting	21	**86.60%**	56	86.30%	**92.10%**
dc_1	8	90.60%	13	**93.80%**	**95.00%**
dc_2	2	**100.00%**	1	80.00%	100.00%
dc_3	6	**100.00%**	4	**100.00%**	100.00%
dc_4	2	92.40%	4	**95.50%**	95.50%
dc_5	2	89.30%	3	**96.40%**	96.40%
dc_6	2	**89.70%**	5	88.20%	**95.60%**
dc_7	2	80.00%	2	**85.00%**	**90.00%**
dc_8	3	**83.30%**	2	50.00%	83.30%
dc_9	9	**100.00%**	4	**100.00%**	100.00%
dc_10	9	**93.00%**	5	**93.00%**	93.00%
dc_11	2	**100.00%**	2	**100.00%**	100.00%
dc_12	3	72.10%	12	**73.50%**	73.50%
dc_13	4	**98.20%**	4	85.70%	**100.00%**
dc_14	2	81.80%	4	**90.90%**	90.90%
radiohole	6	**94.90%**	21	88.40%	**97.50%**
crossnonlx	13	**84.50%**	54	46.30%	**86.80%**
baliseinfo	2	**95.80%**	10	**95.80%**	95.80%
emergency_1	14	**93.40%**	25	87.40%	**94.50%**
emergency_2	6	82.10%	32	**92.30%**	**92.70%**
mema	7	**88.10%**	24	82.60%	**93.20%**
trackside	3	**98.60%**	10	**98.60%**	98.60%
vbc	12	**93.70%**	18	56.30%	**94.80%**
coordfromrbc	5	**82.60%**	26	80.10%	**86.20%**
adfactordmi_1	3	**84.60%**	2	**84.60%**	84.60%
adfactordmi_2	2	**96.20%**	25	**96.20%**	96.20%
driveridins	10	**89.20%**	9	**89.20%**	89.20%
eirene	3	93.70%	14	**95.80%**	95.80%
ertmslevel	3	**94.40%**	9	**94.40%**	94.40%
natvalues	4	**90.00%**	7	**90.00%**	90.00%
networkidins	3	**94.40%**	4	**94.40%**	94.40%
rbcidins	4	**94.50%**	10	**94.50%**	94.50%
trainDataUpdate	1	89.00%	27	**94.00%**	**95.00%**
trainDataInsertion	3	88.00%	39	**90.90%**	90.90%
message129	10	**83.50%**	7	57.90%	**84.20%**
runnumber_1	3	**93.80%**	9	**93.80%**	93.80%
runnumber_2	3	**84.20%**	1	83.20%	**86.10%**

We inspected the programs with uncovered items in further detail, to investigate the reason why Tecs missed the generation of test cases that

cover those items. TecsK scored more than 60% for all the subject programs. TecsC scored less than 60% coverage for the program `crossnonlx` and `vbc`, mostly due to the limit on the loop unwinding depth that was needed in these cases, and for programs `message129` `dc_8`, for which the inspection of the data revealed that Cbmc was not able to provide suitable solutions for some reachability formulas. Then we tracked the uncovered items and prepare four distinct motivations:

- Items that depend on infeasible program paths: In fact, many subject programs include infeasible paths, the most frequent case being one of the programs structured with some (sub-)procedures, where the procedures define general algorithms, but the program calls them only in specialized contexts (e.g, with constant values, passed for some parameters) and thus inhibits the possibility of executing some branches (e.g., the branches that depend on parameter values different than the used constants).

- Unreported coverage: The Scade Test tool does not report the coverage of the items that, although executed during the test cases, do not map to any observable output of the Scade operators in the programs under test. In the considered programs, this happens for a set of operators defined to update stored data: these operators take an input, and use it to do the update, without producing any explicit output. This leads to the Scade test tool misleadingly classifying some items of our subject programs as uncovered. As we are discussing with our partner, this observation calls for some refactoring of the mentioned operators, to improve the precision of the coverage measurements.

- Functional behaviours out of the scope of the single-state-path-coverage testing criterion that Tecs uses for steering the test generation process: We observed uncovered functional behaviours in program `shunting`. The Scade model of this program includes two model states in which the train expects a message from the ground equipment. These states implement the *degraded behaviour* of assuming that the ground equipment is not responding if the expected message is not received within a specific number of execution cycles. As a matter of fact, these behaviours correspond to execution sequences that iterate in the same state for multiple execution cycles and are thus out of the scope of the single-state-path-coverage testing criterion that Tecs is designed to satisfy.

- Uncovered modified condition/decision targets: As we commented in Section 4.5.1 while discussing the loop unrolling of TecsC, this resulted

in a few uncovered modified condition/decision targets in the current experiments, even if TECS analyzed all execution paths. As said, we aim to overcome this limitation of TECS in future releases.

Out of the above cases, only the last two map to the limitations of our approach: the former limitations suggest, for the programs with missing coverage, the strategy of manually complementing the automatically generated test cases by searching for functional behaviours requiring iterating multiple times through the same state, while the latter could be mitigated by improving the implementation of TECS.

RQ1.b Number of Generated Test Cases

We further investigated the quantity of the tests generated with TECS. The safety-critical system required systematical testing and the number of generated test cases should be manageable in size. Table 4.3 reports the number of test cases automatically generated with TECSK and TECSC, respectively (second and third columns). With reference to Table 4.3, TECSK generated test suites ranging between 1 for the program `trainDataUpdate` to 21 test cases for `shunting`, while TECSC generated test suites ranging between 1 for the program `dc_2, runnumber_2` to 56 test cases for `shunting`, which are manageable in size. From the industrial experience with SCADE, up to 100 test cases are manageable for the test engineers and for the SCADE test tool executing them.

RQ1.c Fault Detection

To further investigate the quality of the test suites generated with TECS, we worked jointly with our industrial partner to exploit those test suites for component-level testing of the considered programs. To this end, the test suites generated with TECS were augmented with assertion-style test oracles defined by test engineers based on the documented requirements, thus resulting in a semi-automatic approach to generating the component-level test cases. Manually adding the assertions took limited effort, a few minutes per test case: It required the test engineers to crosscheck the concrete inputs already provided in the test cases with the expectations defined in the specification documents. This, we remark, is a radically simpler task than the manual effort to design and implement the test cases from scratch, which encompasses a very much larger set of time-consuming activities (such as, identifying a functional partitioning out of the requirements, devising suitable test steps and inputs, and implementing the SCADE test cases from scratch).

Table 4.4 describes the faults that we identified by executing the test suites obtained in this way. Overall, we revealed 7 previously unknown faults in four of the subject programs considered in our experiment. Out of seven faults identified, two faults relate to algorithms that wrongly defined the logic stated in the requirements (faults of type *Wrongly de ned algorithm*), a fault corresponds to an algorithm that missed part of the logic defined in the requirements (fault of type *Missing update of a state variable*), a fault corresponds to a wrongly implemented boundary case (fault of type *Output value out of expected range*), a fault corresponds to wrong data updated to a queue (fault of type *Wrong amount of data written in a queue*). The identified bugs are mainly logic ones (that is, errors in the implementation of an algorithm) or simple ones (for example, $<$ instead of $>$, errors in the initialization of a for a cycle, and so on).

These results support the usefulness of the test suites generated with Tecs for component-level testing.

Table 4.4: Faults identified in the subject programs considered in our case study

Program	**fault**
dc_10	Wrong amount of data written in a queue
	Wrongly defined algorithm
coordfromrbc	Missing update of a state variable
	Array updated with the index starting at second (instead of first) item
emergency_1	Output value out of expected range
	Wrongly defined algorithm
emergency_2	Interrelated variables updated in the wrong sequence
total	7 faults

RQ1.d Comparison with Search-based Testing

We investigated whether our approach could also work by using search-based random testing heuristics in place of symbolic execution and bounded model checking. To this end, we replace the instantiation of our Tecs that used the test generator Afl [19] to produce the test inputs. Afl is a test generator that is very popular for security vulnerability testing: it starts by performing random mutations on a set of (seed) inputs provided by developers and then progresses in a search-based fashion by considering the newly generated inputs that increase code coverage as additional seeds. In our setting, we

executed AFL on the analysis-driver programs generated by TECS, providing initial seeds that included an input value for each program input that TECS handled symbolically when using KLEE: For each subject program, we seeded AFL with the input values from the first test case that we had generated when using KLEE.

The task of AFL was then to discover (by means of its search-based heuristics) further input values, as needed to cover the branches of the program under test. Technically, we exploited the feature of AFL to feed back its own test generation mechanism with the test cases that execute new branches. Upon identifying test inputs that make the program execute new branches, AFL saves those test cases in a queue, aiming to consider them as possible seeds at the next steps. Thus, for each subject program, we proceeded as follows: we executed AFL for 5 hours; We used our tool to translate the test cases in the final queue into test cases in SCADE format; We executed the test cases with SCADE Test to collect the corresponding coverage data. We also repeated each test generation attempt 3 times to control for the random characteristics of AFL.

The Table 4.5 indicates the O-MC/DC coverage for the test cases generated with TECS when equipped with KLEE (columns TECSK) or CBMC (columns TECSC), the coverage obtained when merging the test cases generated with TECSK and TECSC (columns TECSK [TECSC), the coverage obtained with AFL (columns TECSA), the difference between the coverage of TECSK [TECSC and TECSA (column *di _1.*, the coverage obtained when merging the test cases generated with TECSK, TECSC and TECSA (columns TECSK [TECSC [TECSA), and the difference between the coverage of TECSK [TECSC [TECSA and TECSK [TECSC respectively for each subject program (column *Program*).

The data in the table (especially column TECSK,TECSC,TECSA) indicate that the model coverage achieved with TECSA was often significantly lower than the coverage achieved with TECSK and TECSC. The highlights in bold typeface identify the data of the tools that reached the highest coverage for each program, and the underlining identifies the cases in which a single tool achieved strictly higher coverage than others. TECSA achieved the same model coverage as TECSK and TECSC for 7 programs (namely, `dc_2`, `dc_3`, `dc_9`, `dc_10`, `dc_11`, `adfactordmi_2` and `natvalues`), achieved more coverage than TECSK and TECSC only for 1 program (namely, `runnumber_2`), and achieved less coverage than TECSK and TECSC for the remaining 28 programs. In these 28 cases in which TECSA is outperformed by TECSK [TECSC, the difference in coverage ranged between 2.40% and 88.30%, with a median of 21.40%. By inspecting the coverage objectives that were missed with AFL, we tracked most untested code to program branches that depend

Table 4.5: TECS comparison with search-based testing

Program	TecsK	TecsC	TecsK ⟦ TecsC	TecsA	diff_1.	TecsK ⟦ TecsC ⟦ TecsA	diff_2.
shunting	**86.60%**	86.30%	**92.10%**	45.30%	46.80%	92.10%	0.00%
dc_1	90.60%	**93.80%**	**95.00%**	89.40%	5.60%	**99.40%**	**4.40%**
dc_2	**100.00%**	80.00%	100.00%	**100.00%**	0.00%	100.00%	0.00%
dc_3	**100.00%**	**100.00%**	100.00%	**100.00%**	0.00%	100.00%	0.00%
dc_4	92.40%	**95.50%**	95.50%	86.40%	9.10%	**98.50%**	**3.00%**
dc_5	89.30%	**96.40%**	96.40%	89.30%	7.10%	96.40%	0.00%
dc_6	**89.70%**	88.20%	**95.60%**	83.80%	11.80%	95.60%	0.00%
dc_7	80.00%	**85.00%**	**90.00%**	50.00%	40.00%	90.00%	0.00%
dc_8	**83.30%**	50.00%	83.30%	41.70%	41.60%	**91.70%**	**8.40%**
dc_9	**100.00%**	**100.00%**	100.00%	**100.00%**	0.00%	100.00%	0.00%
dc_10	**93.00%**	**93.00%**	93.00%	**93.00%**	0.00%	93.00%	0.00%
dc_11	**100.00%**	**100.00%**	100.00%	**100.00%**	0.00%	100.00%	0.00%
dc_12	72.10%	**73.50%**	73.50%	60.30%	13.20%	**79.40%**	**5.90%**
dc_13	**98.20%**	85.70%	**100.00%**	82.10%	17.90%	100.00%	0.00%
dc_14	81.80%	**90.90%**	90.90%	63.60%	27.30%	90.90%	0.00%
radiohole	**94.90%**	88.40%	**97.50%**	68.20%	29.30%	97.50%	0.00%
crossnonlx	**84.50%**	46.30%	**86.80%**	19.10%	67.70%	**87.00%**	**0.20%**
baliseinfo	**95.80%**	**95.80%**	95.80%	44.80%	51.00%	95.80%	0.00%
emergency_1	**93.40%**	87.40%	**94.50%**	6.20%	88.30%	94.50%	0.00%
emergency_2	82.10%	**92.30%**	**92.70%**	54.50%	38.20%	**92.90%**	**0.20%**
mema	**88.10%**	82.60%	**93.20%**	42.80%	50.40%	93.20%	0.00%
trackside	**98.60%**	**98.60%**	98.60%	19.60%	79.00%	98.60%	0.00%
vbc	**93.70%**	56.30%	**94.80%**	36.90%	57.90%	94.80%	0.00%
coordfromrbc	**82.60%**	80.10%	**86.20%**	56.50%	29.70%	86.20%	0.00%
adfactordmi_1	**84.60%**	**84.60%**	84.60%	71.20%	13.40%	84.60%	0.00%
adfactordmi_2	**96.20%**	**96.20%**	96.20%	**96.20%**	0.00%	96.20%	0.00%
driveridins	**89.20%**	**89.20%**	89.20%	65.10%	24.10%	89.20%	0.00%
eirene	93.70%	**95.80%**	95.80%	66.30%	29.50%	95.80%	0.00%
ertmslevel	**94.40%**	**94.40%**	94.40%	87.50%	6.90%	94.40%	0.00%
natvalues	**90.00%**	**90.00%**	90.00%	**90.00%**	0.00%	90.00%	0.00%
networkidins	**94.40%**	**94.40%**	94.40%	75.00%	19.40%	94.40%	0.00%
rbcidins	**94.50%**	**94.50%**	94.50%	49.10%	45.40%	94.50%	0.00%
trainDataUpdate	89.00%	**94.00%**	**95.00%**	60.00%	35.00%	95.00%	0.00%
trainDataInsertion	88.00%	**90.90%**	90.90%	88.50%	2.40%	**91.40%**	**0.50%**
message129	**83.50%**	57.90%	**84.20%**	77.40%	6.80%	84.20%	0.00%
runnumber_1	**93.80%**	**93.80%**	93.80%	70.40%	23.40%	93.80%	0.00%
runnumber_2	84.20%	83.20%	**86.10%**	**92.10%**	**-6.00%**	**93.10%**	**7.00%**

on singular inputs or very specific input ranges, which arguably are hard to hit by random mutations. This is a well-known issue in search-based testing.

We further investigated the mutual strengths of the considered approaches by measuring the coverage achieved with the merged test suites. As reported in the Table 4.5 fourth column (TECSK ⟦ TECSC), 15 programs (out of 37) achieved more coverage with TECSK ⟦ TECSC, which are highlighted with a shadowed background. Then, we compared the results of the TECSK ⟦ TECSC with TECSA, (column *di _1.*) and find out that there is only one case (runnumber_2) in which TECSA outperformed TECSK ⟦ TECSC, with a difference in coverage rather limited (6.00%). In further investigation, we realized that it is due to a single MC/DC objective that TECSK ⟦ TECSC did not cover because it missed a specific truth value for a condition that did not belong to the path condition of the corresponding execution path, while AFL could hit by mutating inputs at random.

Later on, we investigated the mutual strengths of three possible ap-

proaches which are TecsK, TecsC and TecsA and measured the coverage achieved with the merged test suites, as reported in the seventh column (TecsK [TecsC [TecsA) of table 4.5. We highlighted the 8 programs (out of 37) were we achieved the maximum coverage only when merging the test suites from all three tools. The column *di _2.* results from the model coverage difference between TecsK [TecsC [TecsA and TecsK [TecsC, and we achieve a delta between 0.20% and 8.40%. These results are comparable apart from few differences, but still there is some complementarity that is important to understand. Moreover, a merged test suite with strictly greater coverage than any contained test suite reveals that each of those test suites hits unique test objectives.

The overall results from Table 4.5 clearly indicate weakness of the random selection approach, which missed many test objectives, further underscoring the beneficial impacts of systematic exploration of the program state space as in symbolic execution and bounded model checking. Anyway, it can be considered as a complementary to the others, so that the combination of TecsK [TecsC [TecsA can hit some unique test objectives. The point to consider is that for the execution of Afl, we consider the time budget as 5 hours and due to the random testing approach, we repeated the execution 3 times. From Table 4.7, the time taken for the TecsK and TecsC is comparatively very low with respect to TecsA. This confirms that systematic test-generation approaches based on either symbolic execution or bounded model checking can effectively address automated test generation for Scade programs, while a search-based approach does not appear per-se to offer comparable advantages. By considering all the points, we think that a tool like AFL does not suit our goal of testing safety-critical software.

RQ1.e Comparison with Manual Testing

In the case of the subject programs `shunting`, `radiohole` and `crossnonlx` we were able to compare the test cases generated with Tecs with manually selected test suites that were already available for those programs at the time of our experiment. These test suites were designed in a functional fashion based on the software requirements specified for the program, using the model-based test criterion of executing at least once all non-cyclic paths of the state machine and all conditions involved in the state transitions. The engineer assigned to this work reported that the analysis of the requirements, the selection of the test cases and their manual implementation in the test suite took overall 16 man-hours (two days of work), 6 man-hours (about half-day) and 9 man-hours (about one day) for `shunting`, `radiohole` and `crossnlnlx` respectively. He tracked the main challenges to (i) devising

a suitable functional partitioning of the relevant cases to be tested (which in turn required reiterating multiple times the study and the analysis of the specification documents), (ii) analyzing the implementation to identify suitable input and test step sequences for exercising the identified set of relevant cases, and (iii) rendering the test cases in the specific language and format required by the SCADE test tool (that we exemplified in Figure 3.7.b).

Table 4.6: Comparison between automatically (TECSK [TECSC) and manually derived test suites

	Manual test suite		**Tecs**	
Program	**time**	**coverage**	**time**	**coverage**
shunting	16 h	95%	8 h 16 m	92.1%
radiohole	6 h	94%	8 m	97.5%
crossnonlx	9 h	80%	37 m	86.8%

Table 4.6 reports the main statistics of the manual test suites (columns *Manual test suite*) for the three considered programs, sided to the statistics of the test suites that TECS generated (columns TECS) for each of the programs. For each test suite, we report the time taken to generate the test suite (column *time*) and the corresponding model coverage (column *coverage*).

The manually derived test suites pay higher costs in terms of working effort (several hours) in comparison with the relatively shorter time that developers must wait to obtain the test cases with TECS (total time required for TECSK and TECSC). In terms of coverage, the manual test suite of `shunting` achieves higher model coverage than the test suite that TECS generated for this program, but TECS achieved higher model coverage than the manual test suites for `radiohole` and `crossnonlx`.

We analyzed the difference in the coverage data, focusing in particular on the items of the coverage domain that either test suite hits and the other one does not. In detail, for `shunting`, the manually designed test suite successfully executed the "degraded behaviours" (since they correspond to specific transitions indicated in the requirements) that TECS missed as we already commented above. On the other hand, the manually designed test suite missed some possible combinations of the conditions that participate in the transition guards, some of which were hit with TECS thanks to the systematic analysis of all execution paths in the program. Instead, we did not find any manually tested behaviour that TECS did not cover in `radiohole` and `crossnonlx`, where TECS was in fact able to cover some additional rare combinations.

4.5.2 RQ2: Efficiency of symbolic execution and bounded model checking for generating test cases for safety-critical programs in Scade

Table 4.7 summarises the data on the execution of TECS in our experiments and the test cases that it generated. For each subject program (column *Program*), the table reports the time in seconds taken to complete the overall test generation process (column *Time*) and the number of test cases generated (column *#Tests*) for TECSK and TECSC. For TECSK, it reports the number of execution paths analysed with symbolic execution (column *#Paths*), and for TECSC it reports if CBMC completed the analysis without specifying the maximum loop unrolling (column *Analysis Completed(Y/N)*).

The technical research hypothesis of our work is that the programming rules fostered by Scade enable the efficiency of symbolic execution and bounded model checking. Hence, they can handle space exploration, the hardness of handling pointer aliases, can solve complex constraints and can systematically address all the test objectives without missing any relevant test objective.

RQ2.a Execution Time

The data in Table 4.7 show that TECS completed in finite time in almost all experiments, supporting our hypothesis that the language restrictions that SCADE embraces to promote safe programs can mitigate the problems of symbolic execution and bounded model checking.

Table 4.7 shows the execution time (column *Times*) and the number of test cases (column *#Tests*) for each considered Scade program (column *Program*) for the TECSK and TECSC. We mark with shadowed background the time data in which either TECSK exhausted the time budget before completing the symbolic analysis of all program paths, or TECSC required us to set the maximum loop unrolling depth of CBMC to 1000 for it to work within the time budget. Without considering these shadowed cases, the data indicate that TECSK is faster than TECSC on all subjects but `adfactordmi_1`.

In detail, for most subject programs, TECSK took a few seconds for 29 programs to complete the test generation process. It took more than 1 minute only for 8 out of 7 subject programs and more than 10 minutes only for 5 programs, namely, `shunting`, `crossnonlx`, `trackside`, `adfactordmi_1`, `natvalues`, the maximum time being 300 minutes (18000 seconds) in the experiment with program `shunting`. TECSC completed in less than a minute for 20 programs, and in more than 10 minutes for 7 subject programs,

Table 4.7: TECS: execution time, number of paths and generated test cases

Program	TecsK			TecsC		
	Time(s)	#Paths	#Tests	Time(s)	Analysis Completed(Y/N)	#Tests
shunting	18000	15196	21	11400	no	56
dc_1	2	616	8	23	yes	13
dc_2	<1	2	2	12	yes	1
dc_3	<1	16	6	15	yes	4
dc_4	1	3	2	14	yes	4
dc_5	<1	4	2	13	yes	3
dc_6	<1	3	2	16	yes	5
dc_7	<1	4	2	11	yes	2
dc_8	<1	4	3	12	yes	2
dc_9	2	208	9	16	yes	4
dc_10	1	64	9	15	yes	5
dc_11	<1	3	2	11	yes	2
dc_12	<1	3	3	23	yes	12
dc_13	<1	20	4	16	yes	4
dc_14	<1	4	2	14	yes	4
radiohole	117	45	6	358	no	21
crossnonlx	647	294	13	1618	no	54
baliseinfo	1	3	2	51	yes	10
emergency_1	15	28	14	146	yes	25
emergency_2	29	8	6	747	yes	32
mema	23	17	7	101	yes	24
trackside	1137	3	3	6960	no	10
vbc	164	77	12	2069	no	18
coordfromrbc	41	7	5	74	yes	26
adfactordmi_1	1860	3	3	278	yes	2
adfactordmi_2	1	2	2	34	yes	25
driveridins	5	10	10	60	yes	9
eirene	3	3	3	67	yes	14
ertmslevel	2	3	3	48	yes	9
natvalues	1230	4	4	908	no	7
networkidins	1	3	3	48	yes	4
rbcidins	3	4	4	52	yes	10
trainDataUpdate	47	2	1	68	yes	27
trainDataInsertion	28	4	3	78	yes	39
message129	99	80	10	774	yes	7
runnumber_1	2	3	3	54	yes	9
runnumber_2	3	4	3	73	yes	1

namely `shunting`, `crossnonlx`, `emergency_2`, `trackside`, `vbc`, `natvalues`, `message129`. Comparatively, TECSC executes better than TECSK only for subject program `adfactordmi_1`. We investigated the execution time of TECSC and find out that CBMC takes more time for *unwinding* the maximum loop and its formula generation when compared with symbolic execution.

In all experiments, TECSK used the most computation time to complete the symbolic execution with KLEE and TECSC with CBMC under the guidance of the TECS analysis driver, while the other phases of TECS, i.e.,

synthesizing the analysis driver, and synthesizing the test case in SCADE format, took negligible time.

RQ2.b Size of the Execution Space

We investigated the size of the execution space for TECSK and TECSC explored to accomplish the test generation process. We report that (i) TECSK efficiently explored the execution space of 36 programs (out of 37) under test without the need of specifying custom bounds for the analysis and (ii) TECSC required us to set maximum loop unrolling depth of CBMC to 1000 for its work within the time budget in only some cases.

With respect to symbolic execution, we compute the number of paths for each subject program shown in Table 4.7 (column *#Paths* for TECSK). We analyse the number of execution paths that symbolic execution explores for the subject programs and find out that for 36 programs (out of 37) we are able to efficiently explore the execution space without any custom bound limits, which provide a shred of clear evidence that the symbolic execution is efficient for generating test cases for the Scade programs. We set a maximum time budget of 5 hours for each TECS experiment. For this exceptional case, namely `shunting`, symbolic execution was not able to explore all the paths within the time budget. The reason we identified is that `shunting` is a state machine with 5 states, 2 weak transitions, and 8 strong transitions, in which the states and the transitions are based on computations and conditions that involve 12 input and 14 output variables, respectively and the use of large data structure and arrays. The possible way is to extend the time budget and allow symbolic execution to explore all the path.

For the TECSC, we investigated whether TECSC is able to complete test generation without the need of specifying any loop unrolling bound,[1] as Scade guarantees that all loops are bounded by design. We found that this indeed holds for all subject programs but 6 programs out of 37, namely, `shunting`, `radiohole`, `crossnonlx`, `trackside`, `vbc` and `natvalues`. For these 6 programs executing CBMC without specifying the maximum loop unrolling depth results in exhausting the available memory thus terminating with an error, meaning that, even if the loops are bounded, the execution space is anyway very large.

For these 6 programs, it was anyway possible for us to run CBMC, but we analyzed these subjects by setting the maximum loop unrolling depth of CBMC to 1000, using the command line parameter `--unwind`. Since we had these 6 programs exhausting the memory, we checked them to understand

[1] Recall that CBMC, which underlies TECSC, works by statically unrolling the loops in the programs up to a maximum number of iterations.

why we got this result. We indeed found that these programs entail a very large state space, since they use large data structures and arrays, involving CBMC with unwinding all the items of the arrays.

RQ2.c Complexity of the Constraints to be Solved

We investigated the complexity of the constraints solver queries. In particular, we focused on the TecsK experiments from Table 4.7 in which some subject programs resulted in a (low) number of symbolically executed paths with respect to their execution time, because in these experiments we thought the high execution time could depend on the complexity of constraint solving. For these cases, we investigated whether some complex execution conditions that took a long time for the constraint solver to compute the solutions. To this end, we logged the number of queries that the symbolic executor issued to the constraint solver, and the queries for which the constraint solver took more than a specified time.

Table 4.8 shows these data in particular for the subject programs (column *subject*) for which TecsK executed for a number of seconds (column *time*) higher than the number of symbolically analyzed execution paths (column *#paths*). The table reports the number of the queries issued in total to the solver (column #queries) and is restricted to the ones that took more than a millisecond to be solved (column >1). As the table shows, indeed no query took more than a millisecond, confirming that the execution conditions generated during the analysis of the Scade programs result in simple constraint-solving problems. For these programs, we were able to map the execution time to the large data structures that comprise their inputs, which required the initialization and the handling of many symbolic values during symbolic execution. We also observe that some programs resulted in many queries to the constraint solver, despite the low number of symbolically executed program paths. This happens when the program paths traverse many decision points where only one decision is indeed executable: each of those decision points requires to evaluate of two queries (that is, whether the decision can be true or false, respectively) but, once the constraint solver pinpoints the unsatisfiable query, we interrupt the exploration of the non-executable paths and, as a result, the number of symbolically executed program paths does not increase.

Table 4.8: Data on the queries issued to the constraint solver

Program	**time (s)**	**#paths**	**#queries**	**>1 ms**
radiohole	117	45	484	0
crossnonlx	647	294	502	0
emergency_2	29	8	280	0
mema	23	17	122	0
trackside	1137	3	1305	0
vbc	164	77	279	0
coordfromrbc	41	7	155	0
adfactordmi_1	1860	3	1256	0
natvalues	1230	4	1296	0
trainDataUpdate	47	2	215	0
trainDataInsertion	28	4	121	0
message129	99	80	133	0

4.6 Threats to validity

Internal validity concerns whether our conclusions may be wrong due to methodological errors. A possible issue is that we assessed the strengths of symbolic execution and bounded model checking by integrating Tecs with one single test generator for each approach. We mitigated this issue by selecting state-of-the-art test generators that have an acknowledged reputation in the scientific communities of each reference approach. We aim to integrate Tecs with additional test generators in the future. Furthermore, our findings can be affected by the arbitrary choices of limiting the maximum time budget of each experiment to 5 hours and, in the case of Cbmc, of setting the loop unrolling depth to 1000 in the experiments in which Cbmc ran out of memory otherwise. Most of the subject programs are single state so the effectiveness of the SSPC criterion can be evaluated only in the case of three subjects. Hence, in future, we need to do more experiments to evaluate the SSPC criterion.

Tecs generates test cases without oracles, which may limit the practical usefulness of the test suites. In the experiments reported in the thesis, we assessed the effectiveness of the generated test suites based on O-MC/DC code coverage, because it is relevant for satisfying certification standards. Besides, in our project, we found it beneficial to augment those test suites with manually derived oracles, which cost acceptable effort as Tecs produced test suites of a manageable size.

External validity concerns the extent to which our results can generalize to Scade programs other than the ones that we considered in the experiments.

In this respect, the main issue is that all our subject programs belong to the same project. Nonetheless, on one hand, these programs are a representative sample of the safety-critical software that our industrial partner typically develops, following the most prominent certification standards in the railway sector; On the other hand, the restrictions that SCADE embraces to promote the safety of the programs are common to other programming languages for developing safety critical software, e.g., SAFERC. Thus, we believe that our result might in fact generalised. We could not mitigate this threat in the current experiments, and we aim to collect further experimental data in future work.

Chapter 5

Conclusion

The development of safety-critical software must ensure with a high degree of confidence software programs that behave correctly in all operating conditions. To this end, automated software testing can assist in verifying the programs more thoroughly, more quickly, and at a lower cost than traditional, manual testing techniques.

In this thesis, we studied the viability of an automated test generation approach based on symbolic execution and bounded model checking, specifically tailored to the characteristics of a widely adopted programming language for safety-critical software systems (Scade).

We demonstrated our prototype tool Tecs to automated test generation for Scade programs, which can be configured to generate test cases based on symbolic execution and bounded model checking. We provided empirical evidence of the suitability of Tecs on a benchmark of 37 Scade programs that belong to an industrial onboard train signalling system, and discussed the generalized systematical approach for the test generation strategies and the overall effectiveness of the approach as a whole.

In particular, we showed that the systematic test generation strategies of Tecs, based on symbolic execution and model checking, yielded higher structural coverage than the search-based technique. The successfully produced test suites that achieved a high model coverage and that, once augmented with suitable oracles, assist in identifying faults for the considered safety-critical programs in Scade, while keeping the test generation effort under control.

The results of this thesis have been partially published at the International Conference of Software Engineering [34] and in the Journal of System and Software [35].

Future work

We envision many opportunities for future research on the topic.

We plan to extend the experimental assessment by considering further case studies. On one hand, we aim at assessing the scalability of the proposed approach through the analysis of components with growing complexity. So that, we can extend Tecs to be applied for integration testing.

On the other hand, we would like to investigate the possibility of extending our approach to safety-critical software developed in programming languages other than Scade.

We also plan to extend the evaluation of Tecs by assessing the fault detection ability of the generated test suites, e.g. by exploiting a mutation framework for Scade [142].

Lastly, the test cases generated by Tecs currently may contain assertion checks that are usable for regression testing only, but in the future, we would like to integrate Tecs with a component for generating general oracles. Automatic oracle generation is an open research problem, and we are currently studying how to extend techniques to automatically generate oracles from software annotations [156] so that the oracles are generated from the software requirements specification documents.

Appendix: How to use TECS

Installing TECS

Tecs can presently only be installed by building it from the GitHub source. As Tecs is more feature-ready and stable, formal releases will indeed be available.

Dependencies

Tecs has several dependencies. The current version of Tecs support only the Linux platform, Ubuntu [x86-64]. To build Klee, follow the Klee GitHub instructions[1]. The main point to consider while making Klee for Tecs are,

- Install LLVM 9 or above.
- Install constraint solver, STP best option for Tecs.
- Build uClibc and the POSIX environment model.
- Build Klee.

To install Cbmc, follow Cbmc installation guide[2].

- Install Cbmc.

Building TECS

Tecs is built with Maven, which is included in the repository. First, ensure that all the dependencies are present like Klee, Cbmc (see section 5). Then, clone the Tecs GitHub repository.

[1] http://klee.github.io/build-llvm11/
[2] http://www.cprover.org/cbmc/

Usage

- Create a folder for each subject program, create an inner folder (called org_files) and copy the KCG code and its associated files from SCADE suite.
- Run the bash code and follow the instructions in the terminal.
- Output scenario(.sss files) are generated in a folder in the same directory.

List of Publications and Presentations

Publications

1. Kurian, Elson and Briola, Daniela and Braione, Pietro and Denaro, Giovanni. Automatically Generating Test Cases for Safety-Critical Software via Symbolic Execution, *Journal of Systems and Software*, vol. 199, 2023. [35]

2. Kurian, Elson and Briola, Daniela and Braione, Pietro and Denaro, Giovanni and Modonato, Matteo and D'Avino, Dario. Automated Test Case Generation for Safety-Critical Software in Scade, *ICSE, IEEE/ACM International Conference on Software Engineering*, 2023, Paper accepted in ICSE SEIP 2023, to appear. [34]

3. Kurian, Elson and Varghese, Sherwin and Fiorini, Stefano. Towards an innovative model in wearable expert system for skiing, *Metadata and Semantic Research*, March 2021, vol.1355, 403–410.

4. Kurian, Elson and Varghese, Sherwin. Relevance of Bots in Software and Their Impacts on Software Security, *IAENG Proceedings of the World Congress on Engineering* , July 2021, pages 207–212.

Conference/Presentations

1. Kurian, Elson, Effective Testing of Safety-Critical Software with KLEE, *2nd International KLEE Virtual Workshop on Symbolic Execution*, Imperial College London, UK, 10-11 June 2021.

List of Abbreviations

AFL	American Fuzzy Lop
API	Application Programming Interface
ASCII	American Standard Code. for Information Interchange
BDD	Binary Decision Diagrams
BMC	Bounded Model Checking
CBMC	C Bounded Model Checking
CENELEC	European Committee for Electrotechnical Standardization
CNF	Conjunctive Normal Form
ERA	European Union Agency for Railways
ERTMS	European Railway Traffic Management System
ETCS	European Train Control System
IR	Intermediate Representation
IVL	Intermediate Verification Language
LLVM	Low-Level Virtual Machine
Misra	Motor Industry Software Reliability Association
O-MC/DC	Observable-Modified Condition Decision Coverage
RFI	Rete Ferroviaria Italiana
ROBDD	Reduced Ordered Binary Decision Diagrams
RQ	Research Question
SBST	Search-Based Software Testing
SCADE	Safety Critical Application Development Environment
SSPC	Single-State-Path-Coverage
STM	Satisfiability Modulo Theories
SUT	System Under Test
TECS	Test Engine for Critical software in Scade
UML	Unified Modeling Language

Bibliography

[1] RTCA, EUROCAE, DO-178C, Software Considerations in Airborne Systems and Equipment Certification (2012).

[2] CEI, CEI EN 50128, Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems (2020).

[3] M. Brunetto, G. Denaro, L. Mariani, M. Pezzè, On introducing automatic test case generation in practice: A success story and lessons learned, Journal of Systems and Software 176 (2021) 110933. doi:https://doi.org/10.1016/j.jss.2021.110933.
URL https://www.sciencedirect.com/science/article/pii/S0164121221000303

[4] S. Xia, B. Di Vito, C. Muñoz, Automated test generation for engineering applications, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05, Association for Computing Machinery, New York, NY, USA, 2005, p. 283–286. doi:10.1145/1101908.1101951.
URL https://doi.org/10.1145/1101908.1101951

[5] R. Singh, Test case generation for object-oriented systems: A review, in: 2014 Fourth International Conference on Communication Systems and Network Technologies, 2014, pp. 981–989. doi:10.1109/CSNT.2014.201.

[6] G. Fraser, A. Arcuri, A large-scale evaluation of automated unit test generation using evosuite, ACM Trans. Softw. Eng. Methodol. 24 (2) (dec 2014). doi:10.1145/2685612.
URL https://doi.org/10.1145/2685612

[7] I. Hooda, R. Chhillar, A review: study of test case generation techniques, International Journal of Computer Applications 107 (16) (2014) 33–37.

[8] L. Hatton, Safer C: Developing Software for in High-integrity and Safety-critical Systems, McGraw-Hill international series in software engineering, McGraw-Hill, 1995.
URL https://books.google.it/books?id=fbBQAAAAMAAJ

[9] J. Qian, J. Liu, X. Chen, J. Sun, Modeling and verification of zone controller: The scade experience in china's railway systems, in: Proceedings of the First International Workshop on Complex FaUlts and Failures in LargE Software Systems, COUFLESS '15, IEEE Press, 2015, p. 48–54.

[10] B. Beichler, T. Schulz, C. Haubelt, F. Golatowski, A parametric dataflow model for the speed and¬†distance monitoring in novel train control¬†systems, in: M. R. Mousavi, C. Berger (Eds.), Cyber Physical Systems. Design, Modeling, and Evaluation, Springer International Publishing, Cham, 2015, pp. 56–66.

[11] M. Petit-Doche, N. Breton, R. Courbis, Y. Fonteneau, M. Güdemann, Formal verification of industrial critical software, in: M. Núñez, M. Güdemann (Eds.), Formal Methods for Industrial Critical Systems, Springer International Publishing, Cham, 2015, pp. 1–11.

[12] S. Karg, A. Raschke, M. Tichy, G. Liebel, Model-driven software engineering in the openetcs project: Project experiences and lessons learned, in: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MODELS '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 238–248. doi:10.1145/2976767.2976811.

[13] M. Gudemann, A. Angerer, F. Ortmeier, W. Reif, Modeling of self adaptive systems with scade, in: 2007 IEEE International Symposium on Circuits and Systems, 2007, pp. 2922–2925.

[14] T. Le Sergent, Scade: A comprehensive framework for critical system and software engineering, in: I. Ober, I. Ober (Eds.), SDL 2011: Integrating System and Software Modeling, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 2–3.

[15] J.-L. Camus, Esterel scade approach to mbd, Digital Avionics Handbook. Taylor & Francis Group (2015).

[16] X. Fornari, Understanding how SCADE suite KCG generates safe C code, Esterel Technologies, 2010.

[17] G. Berry, SCADE: Synchronous design and validation of embedded control software, in: S. Ramesh, P. Sampath (Eds.), Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems, Springer Netherlands, Dordrecht, 2007, pp. 19–33.

[18] J. W. Duran, S. C. Ntafos, An evaluation of random testing, IEEE Transactions on Software Engineering 10 (4) (1984) 438–444.

[19] American fuzzy lop. (Accessed January 2022).
URL https://lcamtuf.coredump.cx/afl/

[20] C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball, Feedback-directed random test generation, in: Proceedings of the International Conference on Software Engineering, ICSE '07, ACM, 2007, pp. 75–84.

[21] P. Tonella, Evolutionary testing of classes, in: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA '04, ACM, 2004, pp. 119–128.

[22] G. Fraser, A. Arcuri, Evosuite: Automatic test suite generation for object-oriented software, in: Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE '11, ACM, 2011, pp. 416–419.

[23] L. A. Clarke, A system to generate test data and symbolically execute programs, IEEE Transactions on Software Engineering SE-2 (3) (1976) 215–222. doi:10.1109/TSE.1976.233817.

[24] J. C. King, Symbolic execution and program testing, Communications of the ACM 19 (7) (1976) 385–394.

[25] D. E. Cristian Cadar, Daniel Dunbar, Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs, USENIX Association (2008/12/8).

[26] P. Godefroid, M. Y. Levin, D. A. Molnar, Automated whitebox fuzz testing, in: Network Distributed Security Symposium (NDSS), Internet Society, 2008.
URL http://www.truststc.org/pubs/499.html

[27] V. Chipounov, V. Kuznetsov, G. Candea, The s2e platform: Design, implementation, and applications, ACM Transactions on Computer Systems (TOCS) 30 (1) (2012) 2.

[28] N. Tillmann, J. de Halleux, Pex: White box test generation for .NET, in: Proceedings of the International Conference on Tests and Proofs, TAP '08, Springer, 2008, pp. 134–153.

[29] P. Braione, G. Denaro, M. Pezzè, JBSE: A symbolic executor for Java programs with complex heap inputs, in: Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE '16, ACM, 2016, pp. 1018–1022.

[30] P. Braione, G. Denaro, A. Mattavelli, M. Pezzè, Combining symbolic execution and search-based testing for programs with complex heap inputs, in: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA '17, ACM, 2017, pp. 90–101.

[31] E. Clarke, D. Kroening, F. Lerda, A tool for checking ANSI-C programs, in: K. Jensen, A. Podelski (Eds.), Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), Vol. 2988 of Lecture Notes in Computer Science, Springer, 2004, pp. 168–176.

[32] S. Khurshid, C. S. Păsăreanu, W. Visser, Generalized symbolic execution for model checking and testing, in: H. Garavel, J. Hatcliff (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 553–568.

[33] C. Cadar, D. Dunbar, D. Engler, Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs, in: Proceedings of the Symposium on Operating Systems Design and Implementation, OSDI '08, USENIX Association, 2008, pp. 209–224.

[34] E. Kurian, D. Briola, P. Braione, G. Denaro, M. Modonato, D. D'Avino, Automated test case generation for safety-critical software in scade, in: Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE): Software Engineering in Practice track, (To Appear), 2023.

[35] E. Kurian, D. Briola, P. Braione, G. Denaro, Automatically generating test cases for safety-critical software via symbolic execution, Journal of Systems and Software 199 (2023) 111629. doi:https://doi.org/10.1016/j.jss.2023.111629.

[36] K. P. Chan, T. Y. Chen, D. Towey, Normalized restricted random testing, in: J.-P. Rosen, A. Strohmeier (Eds.), Reliable Software Tech-

nologies — Ada-Europe 2003, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 368–381.

[37] K. Claessen, J. Hughes, Quickcheck: A lightweight tool for random testing of haskell programs, in: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP '00, Association for Computing Machinery, New York, NY, USA, 2000, p. 268–279. doi:10.1145/351240.351266.
URL https://doi.org/10.1145/351240.351266

[38] M. Oriol, S. Tassis, Testing .net code with yeti, in: 2010 15th IEEE International Conference on Engineering of Complex Computer Systems, 2010, pp. 264–265. doi:10.1109/ICECCS.2010.58.

[39] K. V. Hanford, Automatic generation of test cases, IBM Systems Journal 9 (4) (1970) 242–257. doi:10.1147/sj.94.0242.

[40] G. J. Myers, The Art Of Software Testing, John Wiley & Sons, 1979.

[41] D. L. Bird, C. U. Munoz, Automatic generation of random self-checking test cases, IBM Syst. J. 22 (1983) 229–245.

[42] R. Hamlet, Random Testing, John Wiley and Sons, Ltd, 2002. doi:https://doi.org/10.1002/0471028959.sof268.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/0471028959.sof268

[43] G. J. Myers, C. Sandler, T. Badgett, The art of software testing, John Wiley and amp; Sons, 2012.

[44] I. Ciupa, A. Leitner, M. Oriol, B. Meyer, Experimental assessment of random testing for object-oriented software, in: Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA '07, Association for Computing Machinery, New York, NY, USA, 2007, p. 84–94. doi:10.1145/1273463.1273476.
URL https://doi.org/10.1145/1273463.1273476

[45] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, B. Meyer, Efficient unit test case minimization, in: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, Association for Computing Machinery, New York, NY, USA, 2007, p. 417–420. doi:10.1145/1321631.1321698.
URL https://doi.org/10.1145/1321631.1321698

[46] J. W. Duran, S. Ntafos, A report on random testing, in: Proceedings of the 5th International Conference on Software Engineering, ICSE '81, IEEE Press, 1981, p. 179–183.

[47] K. Sen, Effective random testing of concurrent programs, in: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, Association for Computing Machinery, New York, NY, USA, 2007, p. 323–332. doi:10.1145/1321631.1321679.
URL https://doi.org/10.1145/1321631.1321679

[48] I. Ciupa, A. Leitner, M. Oriol, B. Meyer, Artoo: Adaptive random testing for object-oriented software, in: Proceedings of the 30th International Conference on Software Engineering, ICSE '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 71–80. doi:10.1145/1368088.1368099.
URL https://doi.org/10.1145/1368088.1368099

[49] D. Hamlet, R. Taylor, Partition testing does not inspire confidence (program testing), IEEE Transactions on Software Engineering 16 (12) (1990) 1402–1411. doi:10.1109/32.62448.

[50] S. Ntafos, On random and partition testing, SIGSOFT Softw. Eng. Notes 23 (2) (1998) 42–48. doi:10.1145/271775.271785.
URL https://doi.org/10.1145/271775.271785

[51] B. P. Miller, L. Fredriksen, B. So, An empirical study of the reliability of unix utilities, Commun. ACM 33 (12) (1990) 32–44. doi:10.1145/96267.96279.
URL https://doi.org/10.1145/96267.96279

[52] J. E. Forrester, B. P. Miller, An empirical study of the robustness of windows nt applications using random testing, in: Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4, WSS'00, USENIX Association, USA, 2000, p. 6.

[53] B. P. Miller, G. Cooksey, F. Moore, An empirical study of the robustness of macos applications using random testing, in: Proceedings of the 1st International Workshop on Random Testing, RT '06, Association for Computing Machinery, New York, NY, USA, 2006, p. 46–54. doi:10.1145/1145735.1145743.
URL https://doi.org/10.1145/1145735.1145743

[54] P. McMinn, Search-based software test data generation: a survey, Software Testing, Verification and Reliability 14 (2004) 105–156.

[55] W. Miller, D. Spooner, Automatic generation of floating-point test data, IEEE Transactions on Software Engineering SE-2 (3) (1976) 223–226. doi:10.1109/TSE.1976.233818.

[56] B. Korel, Automated software test data generation, IEEE Transactions on Software Engineering 16 (8) (1990) 870–879.

[57] B. Korel, Dynamic method for software test data generation, Software Testing, Verification and Reliability 2 (4) (1992) 203–213. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.4370020405, doi:https://doi.org/10.1002/stvr.4370020405.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.4370020405

[58] O. BUEHLER, J. Wegener, Evolutionary functional testing of an automated parking system (01 2003).

[59] O. Bühler, J. Wegener, Evolutionary functional testing, Computers and Operations Research 35 (10) (2008) 3144–3160, part Special Issue: Search-based Software Engineering. doi:https://doi.org/10.1016/j.cor.2007.01.015.
URL https://www.sciencedirect.com/science/article/pii/S0305054807000329

[60] P. Puschner, R. Nossal, Testing the results of static worst-case execution-time analysis, in: Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279), 1998, pp. 134–143. doi:10.1109/REAL.1998.739738.

[61] J. Wegener, H. Sthamer, B. Jones, D. Eyres, Testing real-time systems using genetic algorithms, Software Quality Journal 6 (1997) 127–135. doi:10.1023/a:1018551716639.

[62] L. C. Briand, J. Feng, Y. Labiche, Using genetic algorithms and coupling measures to devise optimal integration test orders, in: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, SEKE '02, Association for Computing Machinery, New York, NY, USA, 2002, p. 43–50. doi:10.1145/568760.568769.
URL https://doi.org/10.1145/568760.568769

[63] Z. Li, M. Harman, R. M. Hierons, Search algorithms for regression test case prioritization, IEEE Transactions on Software Engineering 33 (4) (2007) 225–237. doi:10.1109/TSE.2007.38.

[64] J. Wegener, A. Baresel, H. Sthamer, Evolutionary test environment for automatic structural testing, Information and Software Technology 43 (14) (2001) 841–854.

[65] L. C. Briand, Y. Labiche, M. Shousha, Stress testing real-time systems with genetic algorithms, in: Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation, GECCO '05, Association for Computing Machinery, New York, NY, USA, 2005, p. 1021–1028. doi:10.1145/1068009.1068183.
URL https://doi.org/10.1145/1068009.1068183

[66] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Transactions on Software Engineering 37 (5) (2011) 649–678.

[67] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, R. S. Roos, Timeaware test suite prioritization, in: Proceedings of the 2006 International Symposium on Software Testing and Analysis, ISSTA '06, Association for Computing Machinery, New York, NY, USA, 2006, p. 1–12. doi:10.1145/1146238.1146240.
URL https://doi.org/10.1145/1146238.1146240

[68] K. Derderian, R. Hierons, M. Harman, G. Qiang, Automated unique input output sequence generation for conformance testing of fsms, Computer Journal 49 (05 2006). doi:10.1093/comjnl/bxl003.

[69] N. Tracey, J. Clark, K. Mander, J. McDermid, Automated test-data generation for exception conditions, Softw. Pract. Exper. 30 (1) (2000) 61–79. doi:10.1002/(SICI)1097-024X(200001)30:1\%3C61::AID-SPE292\%3E3.0.CO;2-9.
URL https://doi.org/10.1002/(SICI)1097-024X(200001)30:1%3C61::AID-SPE292%3E3.0.CO;2-9

[70] M. Harman, B. F. Jones, Search-based software engineering, Information and Software Technology 43 (14) (2001) 833–839. doi:https://doi.org/10.1016/S0950-5849(01)00189-6.
URL https://www.sciencedirect.com/science/article/pii/S0950584901001896

[71] P. McMinn, Search-based software test data generation: a survey: Research articles, Softw. Test., Verif. Reliab. 14 (2004) 105–156. doi:10.1002/stvr.294.

[72] B. Armin, C. Alessandro, M. C. Edmund, Y. Yunshan, Symbolic model checking without bdds, in: Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems, Springer, 1999, pp. 193–207.

[73] E. M. Clarke, E. A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, in: Workshop on logic of programs, Springer, 1981, pp. 52–71.

[74] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu, Bounded model checking, Vol. 58 of Advances in Computers, Elsevier, 2003, pp. 117–148. doi:https://doi.org/10.1016/S0065-2458(03)58003-2.
URL https://www.sciencedirect.com/science/article/pii/S0065245803580032

[75] E. Emerson, E. M. Clarke, Using branching time temporal logic to synthesize synchronization skeletons, Science of Computer Programming 2 (3) (1982) 241–266. doi:https://doi.org/10.1016/0167-6423(83)90017-5.
URL https://www.sciencedirect.com/science/article/pii/0167642383900175

[76] J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang, Symbolic model checking: 1020 states and beyond, Information and Computation 98 (2) (1992) 142–170. doi:https://doi.org/10.1016/0890-5401(92)90017-A.
URL https://www.sciencedirect.com/science/article/pii/089054019290017A

[77] O. Coudert, J. Madre, A unified framework for the formal verification of sequential circuits, 1990, pp. 126 – 129. doi:10.1109/ICCAD.1990.129859.

[78] R. E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Transactions on Computers C-35 (8) (1986) 677–691. doi:10.1109/tc.1986.1676819.

[79] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, P. Ashar, F-soft: Software verification platform, in: K. Etessami, S. K. Raja-

mani (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 301–306.

[80] M. Kleine Büning, C. Sinz, D. Faragó, Qpr verify: A static analysis tool for embedded software based on bounded model checking, in: M. Christakis, N. Polikarpova, P. S. Duggirala, P. Schrammel (Eds.), Software Verification, Springer International Publishing, Cham, 2020, pp. 21–32.

[81] Z. Rakamaric, M. Emmi, Smack: Decoupling source language details from verifier implementations, Vol. 8559, 2014, pp. 106–113. doi:10.1007/978-3-319-08867-9_7.

[82] C. Lattner, V. Adve, Llvm: a compilation framework for lifelong program analysis and transformation, in: International Symposium on Code Generation and Optimization, 2004. CGO 2004., 2004, pp. 75–86. doi:10.1109/CGO.2004.1281665.

[83] R. DeLIne, R. Leino, Boogiepl: A typed procedural language for checking object-oriented programs, Tech. Rep. MSR-TR-2005-70 (March 2005).
URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-70.pdf

[84] C. Barrett, A. Stump, C. Tinelli, The smt-lib standard - version 2.0, in: Proceedings of the 8th international workshop on satisfiability modulo theories, Edinburgh, Scotland,(SMT '10), 2010.

[85] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, D. A. Nicole, Esbmc 5.0: An industrial-strength c model checker, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 888–891. doi:10.1145/3238147.3240481.
URL https://doi.org/10.1145/3238147.3240481

[86] Domars, Static driver verifier - windows drivers.
URL https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/static-driver-verifier

[87] A. Lal, S. Qadeer, Powering the static driver verifier using corral, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, Association

for Computing Machinery, New York, NY, USA, 2014, p. 202–212. doi:10.1145/2635868.2635894.
URL https://doi.org/10.1145/2635868.2635894

[88] M. K. Ganai, A. Gupta, P. Ashar, Diver: Sat-based model checking platform for verifying large scale systems, in: N. Halbwachs, L. D. Zuck (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 575–580.

[89] N. Eén, N. Sörensson, An extensible sat-solver, in: E. Giunchiglia, A. Tacchella (Eds.), Theory and Applications of Satisfiability Testing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 502–518.

[90] S. Khurshid, C. S. Păsăreanu, W. Visser, Generalized symbolic execution for model checking and testing, in: Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS '03, Springer, 2003.

[91] P. Godefroid, N. Klarlund, K. Sen, Dart: directed automated random testing, in: Proceedings of the Conference on Programming Language Design and Implementation, PLDI '05, ACM, 2005, pp. 213–223.

[92] Coverity.
URL http://www.coverity.com/

[93] Appscan.
URL http://www-01.ibm.com/software/awdtools/appscan/

[94] Webinspect.
URL www.hp.com/go/appsec/

[95] C. Cadar, D. R. Engler, Execution generated test cases: How to make systems code crash itself, in: Proceedings of the International SPIN Workshop on SPIN Model Checking and Software Verification, SPIN '05, Springer, 2005, pp. 245–264.

[96] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, D. R. Engler, Exe: Automatically generating inputs of death, CCS '06, ACM, 2006, pp. 322–335.

[97] K. Sen, D. Marinov, G. Agha, Cute: A concolic unit testing engine for c, SIGSOFT Softw. Eng. Notes 30 (5) (2005) 263–272. doi:10.1145/1095430.1081750.
URL https://doi.org/10.1145/1095430.1081750

[98] R. Majumdar, K. Sen, Hybrid concolic testing, in: Proceedings of the International Conference on Software Engineering, ICSE '07, IEEE Computer Society, 2007, pp. 416–426.

[99] P. Godefroid, Compositional dynamic test generation, in: Proceedings of the Symposium on Principles of Programming Languages, POPL'07, ACM, 2007, pp. 256–267.

[100] P. Boonstoppel, C. Cadar, D. Engler, Rwset: Attacking path explosion in constraint-based test generation, in: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, 2008, p. 351–366.

[101] J. Burnim, K. Sen, Heuristics for scalable dynamic test generation, in: Proceedings of the International Conference on Automated Software Engineering, IEEE Computer Society, 2008, pp. 443–446.

[102] G.-C. Roman, A. van der Hoek, A. Sigsoft., A. for Computing Machinery., Directed Test Suite Augmentation Techniques and Tradeoffs, ACM, 2010.

[103] M. Ruse, T. Sarkar, S. Basu, Analysis & detection of sql injection vulnerabilities via automatic test case generation of programs, 2010, pp. 31–37. doi:10.1109/SAINT.2010.60.

[104] Y. Kim, M. Kim, N. Dang, Scalable distributed concolic testing: A case study on a flash storage platform, in: Proceedings of the 7th International Colloquium Conference on Theoretical Aspects of Computing, ICTAC'10, Springer-Verlag, Berlin, Heidelberg, 2010, p. 199–213.

[105] M. Baluda, P. Braione, G. Denaro, M. Pezzè, Structural coverage of feasible code, in: Proceedings of 5th Workshop on Automation of Software Testing (AST 2010), 2010, pp. 59–66.

[106] L. De Moura, N. Bjørner, Z3: An efficient SMT solver, in: Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS/ETAPS '08, Springer, 2008, pp. 337–340.

[107] N. Bjørner, N. Tillmann, A. Voronkov, Path feasibility analysis for string-manipulating programs, in: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and

Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,, TACAS '09, Springer-Verlag, Berlin, Heidelberg, 2009, p. 307–321.

[108] K. Lakhotia, N. Tillmann, M. Harman, J. De Halleux, Flopsy: Search-based floating point constraint solving for symbolic execution, in: Proceedings of the International Conference on Testing Software and Systems, ICTSS '10, Springer, 2010, pp. 142–157.

[109] T. Xie, N. Tillmann, P. de Halleux, W. Schulte, Fitness-guided path exploration in dynamic symbolic execution, in: Proceedings of the International Conference on Dependable Systems and Networks, DSN '09, 2009, pp. 359–368.

[110] J. D. Halleux, N. Tillmann, Moles: Tool-assisted environment isolation with closures, in: In TOOLS'10, June–July, 2010.

[111] N. Tillmann, W. Schulte, Parameterized unit tests, Vol. 30, 2005, pp. 253–262. doi:10.1145/1095430.1081749.

[112] V. Ganesh, D. L. Dill, A decision procedure for bit-vectors and arrays, in: W. Damm, H. Hermanns (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 519–531.

[113] J. Yang, C. Sar, P. Twohey, C. Cadar, D. Engler, Automatically generating malicious disks using symbolic execution, in: 2006 IEEE Symposium on Security and Privacy, 2006, pp. 15 pp.–257. doi:10.1109/SP.2006.7.

[114] R. Sasnauskas, J. A. B. Link, M. H. Alizai, K. Wehrle, Kleenet: Automatic bug hunting in sensor network applications, in: Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems, SenSys '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 425–426. doi:10.1145/1460412.1460485.
URL https://doi.org/10.1145/1460412.1460485

[115] C. Zamfir, G. Candea, Execution synthesis: A technique for automated software debugging, in: Proceedings of the ACM SIGOPS EuroSys European Conference on Computer Systems, EuroSys '10, ACM, 2010, pp. 321–334.

[116] V. Chipounov, G. Candea, Reverse engineering of binary device drivers with revnic, in: Proceedings of the 5th European Conference on Computer Systems, EuroSys '10, Association for Computing Machinery,

New York, NY, USA, 2010, p. 167–180. doi:10.1145/1755913.1755932.
URL https://doi.org/10.1145/1755913.1755932

[117] V. Kuznetsov, V. Chipounov, G. Candea, Testing Closed-Source binary device drivers with DDT, in: 2010 USENIX Annual Technical Conference (USENIX ATC 10), USENIX Association, 2010.
URL https://www.usenix.org/conference/usenix-atc-10/testing-closed-source-binary-device-drivers-ddt

[118] T. Avgerinos, S. K. Cha, B. L. T. Hao, D. Brumley, Aeg: Automatic exploit generation, in: NDSS, 2011.

[119] D. Bethea, R. A. Cochran, M. K. Reiter, Server-side verification of client behavior in online games, ACM Trans. Inf. Syst. Secur. 14 (4) (dec 2008). doi:10.1145/2043628.2043633.
URL https://doi.org/10.1145/2043628.2043633

[120] H. Cui, Stable deterministic multithreading through schedule memoization, in: 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10), USENIX Association, Vancouver, BC, 2010.
URL https://www.usenix.org/legacy/events/osdi10/tech/full_papers/Cui.pdf

[121] Dynamic test generation to find integer bugs in x86 binary linux programs, in: 18th USENIX Security Symposium (USENIX Security 09), USENIX Association, Montreal, Quebec, 2009.
URL https://www.usenix.org/conference/usenixsecurity09/technical-sessions/presentation/dynamic-test-generation-find-integer

[122] A. Ermedahl, J. Engblom, Execution time analysis for embedded real-time systems, International Journal on Software Tools for Technology Transfer 4 (2007) 437–455.

[123] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan Kaufmann, 2010.

[124] Object Management Group, OMG® Unified Modeling Language® (OMG UML®) (2017).

[125] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen,

A. J. H. Simons, S. Vilkomir, M. R. Woodward, H. Zedan, Using formal specifications to support testing, ACM Computing Surveys 41 (2) (2015) 18:1–18:41.

[126] M. Utting, A. Pretschner, B. Legeard, A taxonomy of model-based testing approaches, Software: Testing, Verification and Reliability 22 (5) (2012) 297–312.

[127] A. C. Dias Neto, R. Subramanyan, M. Vieira, G. H. Travassos, A survey on model-based testing approaches: A systematic review, in: Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007, WEASELTech '07, Association for Computing Machinery, New York, NY, USA, 2007, p. 31–36. doi:10.1145/1353673.1353681.
URL https://doi.org/10.1145/1353673.1353681

[128] F. Formica, M. Askarpour, C. Menghi, Search-based software testing driven by automatically generated and manually defined fitness functions (2022). arXiv:2207.11016.

[129] D. Balasubramanian, C. S. Păsăreanu, M. W. Whalen, G. Karsai, M. Lowry, Polyglot: Modeling and analysis for multiple statechart formalisms, in: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 45–55. doi:10.1145/2001420.2001427.
URL https://dl.acm.org/doi/10.1145/2001420.2001427

[130] D. Balasubramanian, C. Păsăreanu, M. W. Whalen, G. Karasi, M. Lowry, Improving symbolic execution for statechart formalisms, in: Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation, MoDeVVa '12, Association for Computing Machinery, New York, NY, USA, 2012, p. 47–52. doi:10.1145/2427376.2427385.
URL https://dl.acm.org/doi/10.1145/2427376.2427385

[131] K. Zurowska, J. Dingel, Sauml: A tool for symbolic analysis of uml-rt models, in: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 2011, pp. 604–607. doi:10.1109/ASE.2011.6100136.

[132] C. S. Păsăreanu, N. Rungta, Symbolic pathfinder: Symbolic execution of java bytecode, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 179–180. doi:10.1145/1858996.1859035.
URL https://dl.acm.org/doi/10.1145/1858996.1859035

[133] J.-R. Abrial, Formal methods in industry: Achievements, problems, future, in: Proceedings of the 28th International Conference on Software Engineering, ICSE '06, Association for Computing Machinery, New York, NY, USA, 2006, p. 761–768. doi:10.1145/1134285.1134406.
URL https://doi.org/10.1145/1134285.1134406

[134] J.-R. Abrial, Modeling in Event-B: System and Software Engineering, Cambridge University Press, 2010.

[135] D. Jackson, Software Abstractions: Logic, Language, and Analysis, MIT Press, 2012.

[136] J. M. Spivey, The Z Notation: A Reference Manual, Prentice Hall International Series in Computer Science, Prentice Hall, 1989.

[137] J. Woodcock, J. Davies, Using Z: Specification, Refinement and Proof, Prentice Hall International Series in Computer Science, Prentice Hall, 1996.

[138] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, The synchronous data flow programming language lustre, Proceedings of the IEEE 79 (9) (1991) 1305–1320. doi:10.1109/5.97300.

[139] G. Berry, G. Gonthier, The esterel synchronous programming language: design, semantics, implementation, Science of Computer Programming 19 (2) (1992) 87–152. doi:10.1016/0167-6423(92)90005-V.

[140] C. André, Representation and analysis of reactive behaviors: A synchronous approach, in: Proceedings of Computational Engineering in Systems Applications (CESA'96), IEEE SMC, 1996, p. 19–29.

[141] A. Lakehal, I. Parissis, Lustructu: a tool for the automatic coverage assessment of Lustre programs, in: 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05), 2005, pp. 301–310. doi:10.1109/ISSRE.2005.26.

[142] L. V. Phol, N. T. Binh, I. Parissis, Mutants generation for testing Lustre programs, in: Proceedings of the Eighth International Symposium on Information and Communication Technology, SoICT 2017, Association for Computing Machinery, New York, NY, USA, 2017, p. 425–430. doi:10.1145/3155133.3155155.
URL https://doi.org/10.1145/3155133.3155155

[143] V. Papailiopoulou, Automatic test generation for lustre/scade programs, in: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008, pp. 517–520. doi:10.1109/ASE.2008.96.

[144] P. Raymond, X. Nicollin, N. Halbwachs, D. Weber, Automatic testing of reactive systems, in: Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279), 1998, pp. 200–209. doi:10.1109/REAL.1998.739746.

[145] B. Marre, A. Arnould, Test sequences generation from lustre descriptions: Gatel, in: Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering, 2000, pp. 229–237. doi:10.1109/ASE.2000.873667.

[146] L. M. de Moura, S. Owre, N. Shankar, The sal language manual, in: CSL Technical Report SRI-CSL-OI-02, 2003.

[147] G. Hamon, L. M. de Moura, Automated test generation with sal, in: CSL Technical Note, 2005.

[148] A. Wakankar, A. Bhattacharjee, S. Dhodapkar, P. Pandya, K. Arya, Automatic test case generation in model based software design to achieve higher reliability, in: 2010 2nd International Conference on Reliability, Safety and Hazard - Risk-Based Technologies and Physics-of-Failure Methods (ICRESH), 2010, pp. 493–499. doi:10.1109/ICRESH.2010.5779600.

[149] J. Toennemann, A. Aniculăesei, A. Rausch, Asserting functional equivalence between c code and scade models in code-to-model transformations, in: Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing, SAST 20, Association for Computing Machinery, New York, NY, USA, 2020, p. 60–68. doi:10.1145/3425174.3425213.
URL https://doi.org/10.1145/3425174.3425213

[150] C. Braunstein, A. E. Haxthausen, W.-l. Huang, F. Hübner, J. Peleska, U. Schulze, L. Vu Hong, Complete model-based equivalence class testing for the etcs ceiling speed monitor, in: S. Merz, J. Pang (Eds.), Formal Methods and Software Engineering, Springer International Publishing, Cham, 2014, pp. 380–395.

[151] L. Vu, A. Haxthausen, J. Peleska, A domain-specific language for railway interlocking systems, in: E. Schnieder, G. Tarnai (Eds.), Proceedings of the 10th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems, FORMS/FORMAT 2014, Technische Universität Braunschweig, 2014, pp. 200–209, 10th Symposium on Formal Methods for Automation and Safety i Railway and Automotive Systems, FORMS/FORMAT 2014, FORMS/FORMAT ; Conference date: 10-09-2014 Through 02-10-2014.
URL http://www.forms-format.de/index.html

[152] D. Harel, Statecharts: a visual formalism for complex systems, Science of Computer Programming 8 (3) (1987) 231–274. doi:10.1016/0167-6423(87)90035-9.

[153] IEC, IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems (2010).

[154] ISO, ISO 26262: Road vehicles — Functional safety (2010).

[155] T. Parr, The Definitive ANTLR 4 Reference, 2nd Edition, Pragmatic Bookshelf, 2013.

[156] A. Goffi, A. Gorla, M. D. Ernst, M. Pezzè, Automatic generation of oracles for exceptional behaviors, in: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA '16, ACM, 2016, pp. 213–224.