The Journal of Systems & Software 199 (2023) 111629

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Automatically generating test cases for safety-critical software via symbolic execution[☆]

Elson Kurian, Daniela Briola [*], Pietro Braione, Giovanni Denaro

Department of Informatics, Systems and Communication, University of Milano-Bicocca, Viale Sarca 336, Building U14, 20126, Milan, Italy

ARTICLE INFO

Article history:
Received 20 May 2021
Received in revised form 12 December 2022
Accepted 28 January 2023
Available online 2 February 2023

Keywords:
Automated test generation
Symbolic execution
Safety-critical software

ABSTRACT

Automated test generation based on symbolic execution can be beneficial for systematically testing safety-critical software, to facilitate test engineers to pursue the strict testing requirements mandated by the certification standards, while controlling at the same time the costs of the testing process. At the same time, the development of safety-critical software is often constrained with programming languages or coding conventions that ban linguistic features which are believed to downgrade the safety of the 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 bound the complexity of control conditions. As a matter of facts, these linguistic features are also the main efficiency-blockers for the test generation approaches based on symbolic execution at the state of the art.

This paper contributes new evidence of the effectiveness of generating test cases with symbolic execution for a significant class of industrial safety critical-systems. We specifically focus on Scade, a largely adopted model-based development language for safety-critical embedded software, and we report on a case study in which we exploited symbolic execution to automatically generate test cases for a set of safety-critical programs developed in Scade. To this end, we introduce an original test generator that we developed in a recent industrial project on testing safety-critical railway software written in Scade, and we report on our experience of using this test generator for testing a set of Scade programs that belong to the development of an on-board signaling unit for high-speed rail. The results provide empirically evidence that symbolic execution is indeed a viable approach for generating high-quality test suites for the safety-critical programs considered in our case study.

1. Introduction

Safety-critical software systems control life-critical and mission-critical tasks in airplanes, trains, cars, nuclear power plants and patient monitoring tools. Since failures in these systems can have catastrophic consequences, they must be highly reliable (Hatton, 1995). For this reason, the certification authorities of the specific sectors usually impose strict standards on both the development and the quality control activities (RTCA, 2012; CENELEC, 2020), in order to ensure the highest possible confidence in the correct behavior of the developed systems.

In this context, *automated* test case generation can play a crucial role for achieving the testing objectives mandated by the standards, while controlling at the same time the associated costs. While the problem of generating a set of test inputs for an arbitrary software program that cover a given target is undecidable, research aims at producing automatic tools that work well "in practice", i.e., in a sufficient number of common cases. The techniques that are more used by current tools are based either on random testing (Duran and Ntafos, 1984; Chen et al., 2010), or on search-based testing (Ali et al., 2010), or on symbolic execution (Baldoni et al., 2018; Cadar and Sen, 2013). Random and search-based testing sample the input space of the target programs, either in a purely random fashion, or guided by the improvement of a *fitness* function, whose value correlates with the coverage objectives to optimize (Duran and Ntafos, 1984; AFL, 2022; Pacheco et al., 2007; Tonella, 2004; Fraser and Arcuri, 2011). On converse, symbolic execution (Clarke, 1976; King, 1976; Cristian Cadar and Dunbar, 2008; Godefroid et al., 2008; Chipounov et al., 2012; Tillmann and de Halleux, 2008; Braione et al., 2016, 2017) systematically explores the execution paths of the program under test: it computes the execution conditions of

[☆] Editor: Raffaela Mirandola.

[*] Corresponding author.

E-mail addresses: e.kurian@campus.unimib.it (E. Kurian), daniela.briola@unimib.it (D. Briola), pietro.braione@unimib.it (P. Braione), giovanni.denaro@unimib.it (G. Denaro).

https://doi.org/10.1016/j.jss.2023.111629
0164-1212/© 2023 Elsevier Inc. All rights reserved.

the explored paths, and solves these execution conditions with the help of an automatic SMT (satisfiability-modulo-theories) solver as, e.g., Yices (Dutertre, 2014), STP (Ganesh and Dill, 2007) or Z3 (De Moura and Bjørner, 2008). If a solution is found, this is a test input covering the path.

This paper investigates the viability of symbolic execution for automated test generation for safety-critical software. The choice of using symbolic execution 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 should be in principle able to generate at least a test case for every test objective that can be reached on at least an execution path, a goal that the random and search-based techniques cannot generally guarantee. Nonetheless we are also aware of the major challenges of designing test generators based on symbolic execution, which result from common limitations of this technique:

(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 seldom succeeds to analyze all execution paths in finite time. On the contrary, the systematic exploration approach often engages symbolic execution in a very fine-grained analysis of some specific parts of the program execution space, while leaving many other parts entirely 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 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 alias of each other, or yet correspond to distinct objects, respectively (Khurshid et al., 2003). 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*: In symbolic execution, 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 test budget in the analysis of execution paths that depend on unsatisfiable conditions that the constraint solver failed to pinpoint.

This paper contributes new evidence in support of the research hypothesis that, although the above issues hindering the practicality of symbolic execution may hold for many general-purpose programs, they have reduced impact for a significant class of industrial safety-critical systems, where symbolic execution can therefore work effectively. In fact, 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 (Hatton, 1995) and Scade[1] (used in the avionics and in the railway domains, respectively), or coding standards like Misra[2] (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 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.

In particular, this paper reports on a case study drawing on our experience with a recent project aimed to develop an on-board signaling unit for high-speed rail, following the ERTMS[3] standard specification, in which we have been recently involved with an industrial partner. This on-board 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 danger conditions. It is currently being implemented with Scade, a system modeling language and a model-based development environment for embedded software largely adopted in industry[4] (Qian et al., 2015; Beichler et al., 2015; Petit-Doche et al., 2015; Karg et al., 2016; Gudemann et al., 2007; Le Sergent, 2012; Camus, 2015) and certified according to the CENELEC norms (CENELEC, 2020). As we discuss in more detail in Section 2, Scade allows to specify 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 (Fornari, 2010; Berry, 2007) required by ERA, the European Union Agency for Railway.[5]

In the reported case study, we explored whether and to which extent the programming constraints on which the safety-critical software developed in Scade depends enable the exploitation of symbolic execution for effectively generating test cases for such programs. In detail, this paper makes the following contributions:

(i) We introduce an original test generator for Scade programs based on symbolic execution. We refer to this test generator as Tecs (Test Engine for Critical software in Scade). Tecs builds on the symbolic executor Klee (Cristian Cadar and Dunbar, 2008) to render an efficient symbolic analysis of the C programs that the Scade environment compiles out of the original Scade models. While it is true that Tecs builds on existing tools (mainly the Scade translator and Klee), the originality of Tecs is tightly related to the goal of our case study, in that Tecs makes several distinctive design

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

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

[3] www.era.europa.eu/activities/european-rail-traffic-management-system-ertms.

[4] 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 flight control, autonomous vehicles and gas turbines. [www.ansys.com/products/embedded-software/ansys-scade-suite].

[5] www.era.europa.eu.

choices that explicitly exploit the programming constraints guaranteed for programs in Scade.
In particular, we frame the distinctive characteristics of Tecs as follows:

- it intentionally relies on unbounded symbolic execution. Specifically, it defines an analysis algorithm comprised of multiple unbounded symbolic-execution passes of the transition function of the Scade program under test, aiming to systematically analyze the state machine model that the Scade program represents. This analysis algorithm builds on the guarantee that the Scade translator produces C programs with finite execution paths, thanks to the avoidance of unbounded loops and recursive calls, which guarantees the termination of any execution path of each symbolic-execution pass.
- it enforces the initialization of all input data structures at the beginning of symbolic execution with symbolic values assigned to all fields at any nesting level (including the items in all array-typed fields). In this way, it defines a specialized, efficient symbolic execution analysis that shall not cope with discriminating the possible ways of initializing the input data structures and their internal references during the analysis. Tecs exploits the knowledge that all data structures are statically allocated and not recursive, and the size of all arrays is statically specified, which implies that all input data structures are always made of a finite set of statically identifiable distinct fields.

(ii) We report new empirical data that show that Tecs successfully computed test cases that both achieve high model coverage of a set of Scade programs developed by our industrial partner, and revealed (once enriched with suitable assertion-style test oracles) subtle, previously unknown faults for some considered programs. In this way, our case study provides supporting evidence of both the effectiveness of Tecs and the suitability of symbolic execution for generating test cases for the considered class of safety critical programs.

(iii) Furthermore, we report on our experience with using the tool (AFL, 2022), a test generator that is very popular for security vulnerability testing, as a possible replacement of Klee in our tool. AFL is based on random and search-based input selection heuristics. The results clearly indicate the weaknesses of the random selection approach, which missed many test objectives, further underscoring the beneficial impacts of a systematic exploration of the program state space as in our approach.

(iv) Yet, for some considered programs, we were able to compare the test cases that Tecs automatically produced with the ones that were already manually designed by the developers. The comparison revealed interesting complementarities, thus confirming the usefulness and the effectiveness of our test generator, and further supporting the exploitability of symbolic execution to generate test cases for Scade models.

This paper is organized as follows. Section 2 surveys the main characteristics of the Scade programming language, elaborates on the language restrictions that enable our test generation approach, and introduces a sample Scade program that we use as working example in the paper. Section 3 details the design of the test generator Tecs, focusing in particular on the design choices by which Tecs exploits the programming constraints that derive from Scade. Section 4 reports on the case study in which we used Tecs to generate test cases for a set of programs that belong to the on-board train unit developed by our industrial partner. Finally, Section 5 surveys the related work in the field, and Section 6 outlines our conclusions and plans for future work on the topics of this paper.

2. Safety-critical development with Scade

In this section, we survey the main characteristics of the Scade programming language, motivate our research hypothesis on the exploitability of symbolic execution to generate test cases for programs in Scade, and introduce a sample Scade program that we use as working example in the subsequent sections of the paper.

2.1. *Scade and characteristics of the Scade programs*

Scade is a system modeling 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, that allows to design embedded cyber–physical systems based on the Scade language, simulate their behavior, and generate qualifiable/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 as, e.g., avionics, automotive and railway.

The Scade modeling language belongs to the family of the synchronous languages, such as Lustre (Halbwachs et al., 1991) and Esterel (Berry and Gonthier, 1992). 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. Each state may have a hierarchical structure, similar in spirit to, but with richer semantics than, the Statecharts (Harel, 1987) or UML state machine languages (Object Management Group, 2017). 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 to edit a model in either format.

Integrated in 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. In this paper we consider the translation to C programs. 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 that 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 (RTCA, 2012), IEC 61508 (IEC, 2010), EN 50128 (CENELEC, 2020), and ISO 26262 (ISO, 2010). To this end, Kcg translates a Scade 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:

- Its semantics is unambiguous and precise (e.g., no undefined behaviors);
- 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, `returns`, 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 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 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. We remark that the nature of Scade models – their 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. We remark that the test generation problems that we consider in this paper refer to the inputs and the outputs of the Kcg-generated programs, i.e., the inputs and outputs of the Scade models, regardless of the possible glue code that binds these inputs and outputs to sensors and actuators of the final system.

2.2. 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 the reader in better understanding how our approach described in Section 3 works.

Fig. 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 behavior 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 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 Fig. 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 behavior 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 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 behavior of 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 behavior 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 behavior 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 Fig. 1. The if-then-else block takes *mirrorData.automaticControl* as 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 Fig. 1 with Kcg yields the C program excerpted in Fig. 2. The program defines the entry function `WingMirrorControl_CarControl` (excerpted at the

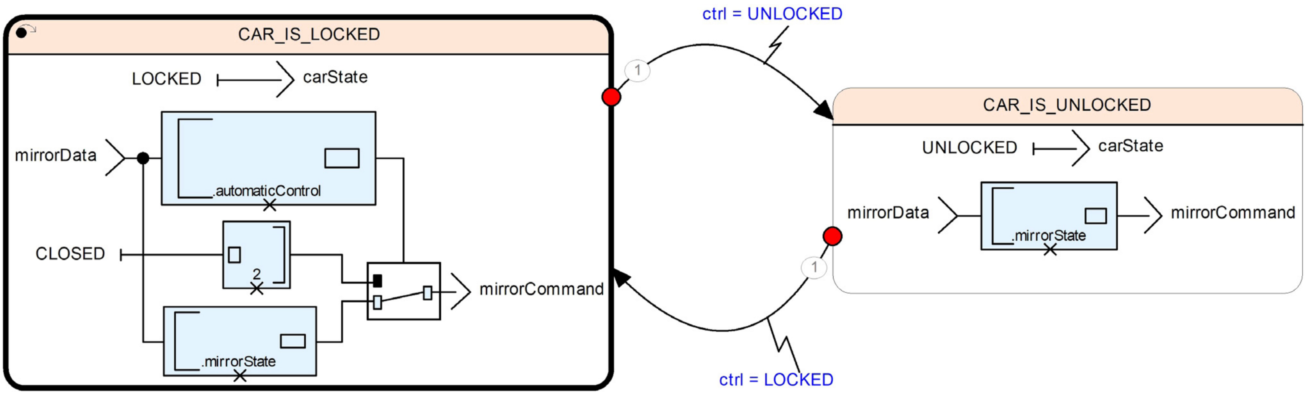


Fig. 1. A sample Scade model for a car wing mirror controller.

```
typedef struct {
  Lock ctrl;
  MirrorData mirrorData;
} inC_WingMirrorControl_CarControl;

typedef struct {
  MirrorStateArray mirrorCommand;
  Lock carState;
  SSM_ST_WingMirrorFSM WingMirrorFSM_state_nxt;
} outC_WingMirrorControl_CarControl;

typedef struct {
  kcg_bool automaticControl;
  MirrorStateArray mirrorState;
} MirrorData;

typedef MirrorState MirrorStateArray[2];

typedef enum {UNLOCKED, LOCKED} Lock;

typedef enum {OPEN, CLOSED} MirrorState;

void WingMirrorControl_CarControl(
  inC_WingMirrorControl_CarControl *inC,
  outC_WingMirrorControl_CarControl *outC);
{
  SSM_ST_WingMirrorFSM WingMirrorFSM_state_act;
  kcg_size idx;

  switch (outC->WingMirrorFSM_state_nxt) {
    case SSM_st_CAR_IS_UNLOCKED_WingMirrorFSM:
      if (inC->ctrl == LOCKED) {
        WingMirrorFSM_state_act =
        SSM_st_CAR_IS_LOCKED_WingMirrorFSM;
      }
      else {
        WingMirrorFSM_state_act =
        SSM_st_CAR_IS_UNLOCKED_WingMirrorFSM;
      }
      break;
    case ...
  }

  switch (WingMirrorFSM_state_act) {
    case SSM_st_CAR_IS_UNLOCKED_WingMirrorFSM:
      kcg_copy_WingMirrorArray(outC->mirrorCommand,
      inC->mirrorData.mirrorState);
      outC->carState = UNLOCKED;
      outC->WingMirrorFSM_state_nxt =
      SSM_st_CAR_IS_UNLOCKED_WingMirrorFSM;
      break;
    case ...
  }
}
```

Fig. 2. Excerpt of the C program that Kcg generates for the Scade model in Fig. 1.

bottom of the figure) that encodes the behavior 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` to `UNLOCKED`, and the output `outC->WingMirrorFSM_state_nxt` to `SSM_st_CAR_IS_UNLOCKED_WingMirrorFSM`.

3. Generating Scade test cases

In this section, we introduce a test generator to automatically generate unit-level test cases for embedded programs written in Scade. Our test generator for Scade programs is built on top of the symbolic executor Klee, and explicitly relies on the C language restrictions that Kcg enforces (as we discussed in Section 2). We designed the test generator with the aim of exploring whether and to which extent these restrictions identify a class of programs that by design mitigate many common sources of open issues for test generators based on symbolic execution. In this section we describe the design of the test generator, while in the next section we report on the effectiveness of the test generator to derive test cases for a set of Scade programs implemented in a recent project in which we are participating along with an industrial partner.

Fig. 3 shows the main components of our test generator, and the workflow that these components comprise. We refer to our test generator as Tecs, the *Test Engine for Critical software in* Scade. Tecs relies on the Kcg compiler, which is part of the cross-compilation tool chain of the Scade Suite, to convert the Scade program under test into an equivalent program written in C.

Then, Tecs includes a *Driver synthesizer* 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 then

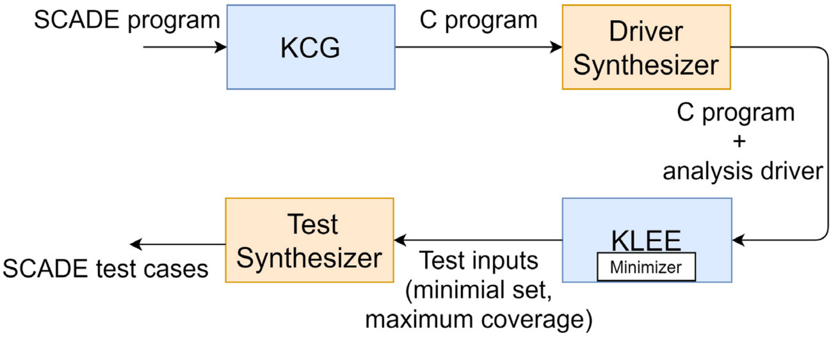


Fig. 3. Workflow of Tecs.

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 with symbolic execution, Tecs steers multiple analysis passes of the execution paths in the program, with each new pass depending on the (symbolically represented) results of the previous pass. As we explain in detail in Section 3.1, the Driver synthesizer tailors the general analysis algorithm to the specific signature of the program under test.

To accomplish symbolic execution according to the analysis algorithm provided with the analysis driver, Tecs relies on Klee, a well known state-of-the-art symbolic executor for programs in C (Cristian Cadar and Dunbar, 2008). Klee generates test inputs for each execution path that the analysis driver induces through the program as follows. First, Klee performs symbolic execution along each execution path to compute the associated *path condition*, being the path condition a (quantifier free) logic formula that represents the conditions that the inputs shall satisfy for the program to execute along the given path. Then, Klee attempts to solve each path condition with the STP (Ganesh and Dill, 2007) constraint solver. If a path condition has a solution, this is a set of concrete inputs for executing the corresponding execution path; Otherwise, if there is no solution, the path is *infeasible*, i.e., no input can drive the execution of the program through it. Yet, a further and unfortunate phenomenon is that some path conditions could be formulas that the STP constraint solver cannot solve within the allowed timeout (1 millisecond in our current experiments), and thus the test generation process might result in either or both missed test cases and wasted analysis time. As we already commented in the introduction section, this phenomenon is a common source of ineffectiveness for test generators based on symbolic execution, but also a phenomenon that we hypothesize to be rare for the safety-critical programs in Scade. Indeed, in the case study that we report in Section 4 we did not experience any unsolved path condition.

Next, Tecs filters the execution paths explored during symbolic execution by computing the minimal set of execution paths that guarantee the maximum coverage of the relevant test objectives (Fig. 3, *Minimizer*). In fact, maintaining test suites that include a test case for each execution path is by far beyond the typical certification requirements, and cannot be generally afforded by producers. The Minimizer aims to produce a test suite of manageable size, while avoiding to miss test objectives.

As a limitation of our current prototype, Tecs delegates the task of the Minimizer to an internal algorithm of Klee, which can be optionally activated to limit the provided test inputs only to the execution paths that improve statement coverage. This is a sub-optimal minimization behavior, and indeed in the experiments that we report later in this paper we observed that some test objectives that are considered in the Scade Test tool (which refers to modified condition/decision coverage, a finer criterion than statement coverage) were missed. We discuss the results of the experiments in detail in Section 4. In future releases, we aim to improve our tool by providing a dedicated Minimizer.

As final step, Tecs constructs a Scade test case (Fig. 3, *Test synthesizer*) 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.

Below we discuss in detail the design of the Driver synthesizer and the Test synthesizer. We then close this section by remarking the core original ideas that our test generator Tecs settles in the analysis algorithm that it instructs with the Driver synthesizer. The Test synthesizer is rather an engineering effort, though important to finalize the generated test suites.

3.1. The driver synthesizer

The goal of the Driver synthesizer is to augment the C translation of the Scade model under test with an *analysis driver*, designed to steer the symbolic analysis of the execution paths in the program. In pure technical terms, the analysis driver provides the entry function that Klee shall symbolically execute in order to generate test inputs for the program under test. The analysis driver assigns symbolic values to the program inputs, and then calls the target program one or multiple times with the symbolic inputs, to unfold the possible sequences of transitions of the state machine that the program represents. Indeed each call to the target program corresponds to firing a transition of the state machine, and thus each execution path through the analysis driver corresponds to a sequence of state transitions that Tecs has to analyze symbolically.

The Driver synthesizer builds an analysis driver that steers Klee to symbolically execute (and thus generate test inputs for) the program paths and the execution sequences that satisfy the single-state-path-coverage (SSPC) testing criterion with respect to the state machine model implemented in the Scade program. The SSPC criterion requires to exercise all paths and execution sequences that traverse states at most once (Pezzè and Young, 2007).

The resulting analysis driver comes in the shape of a general algorithm, representing the overall steering strategy for satisfying the SSPC criterion, and a set of automatically generated hook functions called from the general algorithm, representing the program-specific tailoring of the analysis driver. Algorithm 1 formalizes the general steering algorithm of the analysis driver in pseudo-code, with the calls of the hook functions represented within framed pseudo-code. The hook functions are either functions that already belong to the C program generated with Kcg, or functions that the Driver synthesizer generates and injects in the program. The general algorithm indicates that the analysis driver starts in a state that corresponds to the initial state of the program (line 6), with the program outputs initialized to

Algorithm 1 The algorithm of the analysis driver

```
Let:
1:  program be the program under test,
2:  inputs be a reference to the inputs of the program,
3:  outputs be a reference to the outputs of the program,
4:  $s_0$ be the initial state of the program,

5:  outputs ← $\boxed{default\_values()}$
6:  state ← $s_0$
7:  visited ← $\emptyset$
8:  while state $\notin$ visited do
9:      inputs ← $\boxed{fresh\_symbols()}$
10:     state′, outputs ← $\boxed{program(state, inputs, outputs)}$
11:     $\boxed{save\_symbolic\_expressions(outputs)}$
12:     visited ← visited $\cup$ {state}
13:     state ← state′
14: end while
```

default values by calling the initialization routine that Scade specifically generates as part of the code of each component (line 5), and considering an initially empty set of visited states (line 7). The hook function *default_values* (line 5) for initializing the program outputs with default values is part of the C program generated by Kcg. Then, the driver iterates through the loop at lines 8–14, where it calls the program under test once per iteration (line 10), until the execution of the program leads to an already visited state (line 8). Exiting the loop corresponds to an execution sequence that we must consider according the SSPC testing criterion and for which Klee will then generate a corresponding test input.

At each iteration of the loop at lines 8–14, the analysis driver triggers the possible state transitions of the Scade program by first assigning the program inputs with fresh symbolic values (line 9), and then symbolically executing the program under test to analyze the possible execution paths (line 10). For each analyzed path, it saves the current symbolic values of the outputs to enable the Tecs Test synthesizer to generate regression oracles later on (line 11), updates the set of visited states (line 12), and iterates with the analysis of the next state (line 13). The hook functions at the first three steps inside the loop crucially depend on the restrictions that Kcg enforces to foster dependable safety-critical software. Below, we explain these hook functions in detail.

Function fresh_symbols (Algorithm 1, line 9). This hook function assigns the program inputs with symbolic values. The Driver synthesizer generates the code of the hook function *fresh_symbols* based on the knowledge that Kcg does not generate recursive data structures, dynamic memory allocation or variable-length arrays. This guarantees the viability of unfolding all fields of primitive types that belong to the input data structures at any nesting level, since these fields are necessarily a finite set. Thus, the Driver synthesizer customizes the code of function *fresh_symbols* by assigning each primitive-typed input (received either as an input variable or as a field nested in an input data structure) to a fresh symbolic value, while it initializes all pointer-typed inputs to structures and arrays with the concrete addresses of non-overlapping memory objects suitably allocated by the analysis driver itself.[6] The Driver synthesizer uses the tool ANTLR4 (Bovet and Parr, 2008) to parse the type definitions in the C program for the sake of generating the code of function *fresh_symbols*.

Let us consider, for instance, the working program that we introduced in Fig. 1. With reference to the corresponding C program of Fig. 2, the Driver synthesizer generates the hook function *fresh_symbols* as indicated in Fig. 4. By inspecting the considered C program, the analysis driver synthesizer identifies that the data structure of 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 synthesizer inspects the definition of the array, revealing that it consists of two items of primitive type (again an enumeration). The generated function *fresh_symbols* ultimately consists of C code that initializes a new instance of the data structure in memory (Fig. 4, line 2), relies on Klee (operation *klee_init*) to initialize the primitive field `ctrl` with a new fresh symbol (line 3), initializes the non-primitive field `wingMirrorData` as a new array instance with two items (line 4), and initializes the two items in the array with further fresh symbols (lines 5 and 6).

The operation *klee_init* for initializing the inputs with fresh symbols takes two main parameters: one is the input to be initialized passed by reference, and the other one is a name (a string of characters) to be associated with that symbolic value. For instance, we might define the name "ctrl" for the fresh symbol that function *fresh_symbols* associates with the input `ret->ctrl` at line 2. Upon generating test inputs as possible concrete values of the symbols, Klee will use the provided name to indicate the input data to which those values refer. We postpone to Section 3.2 the discussion on how we specifically define the names for the fresh symbols to facilitate the task of synthesizing Scade test cases out of the test inputs obtained with Klee.

Executing the program under test (Algorithm 1, line 10). The hook function *program* at line 10 represents a call to the program under test, which is already part of the C code generated with Kcg. The function receives the current state, the freshly initialized symbolic inputs and the current values of the outputs, and executes a state transition, possibly yielding a new state and new outputs. When executing *program*, Tecs relies on the knowledge that the C translation of a Scade model consists (by construction) of all deterministic and terminating program paths, and thus the symbolic execution is guaranteed to terminate without need of enforcing any scope bound for the analysis.

Function save_symbolic_expressions (Algorithm 1, line 11). This hook function saves the symbolic expressions associated with the outputs after each execution of the program under test. This enables Tecs to solve (at a later step) these expressions to concrete values that consistently match with the selected inputs, and use those value to define regression oracles within the test cases. At the state of the art, generating regression oracles is a common functionality offered by most test generators (Fraser and Arcuri, 2011): A regression oracle defines the expectation that the outputs shall be equal to the values observed during the test generation process, which is trivially true when executing the test cases against the program that is being considered, but

[6] We remark that, by initializing all pointer-typed inputs with concrete addresses of non-overlapping memory objects, our approach enforces by design that during symbolic execution no pointer dereference can ever result into a *symbolic* memory access, i.e. a memory access in which the memory location itself is a symbolic, non-deterministic value. Accessing arrays with symbolic indices can still lead to symbolic memory accesses, which Klee models with formulas expressed in the theory of arrays (Ganesh and Dill, 2007), consistently with the semantics of the program under test. In this case, our approach guarantees that these formulas predicate on non-overlapping arrays with statically known size, which can be addressed without particular challenges with SMT solvers at the state of the art (Ganesh and Dill, 2007).

```
inC_WingMirrorControl_CarControl* fresh_symbols() {
    inC_WingMirrorControl_CarControl* ret = malloc(sizeof(...));
    klee_init(&ret->ctrl, "...");
    ret->wingMirrorData = malloc(2 * sizeof(...));
    klee_init(&ret->wingMirrorData[0], "...");
    klee_init(&ret->wingMirrorData[1], "...");
    return ret;
}
```

Fig. 4. The hook function *fresh_symbols* for the sample program of Fig. 2.

may provide meaningful insights on possible regressions against future new versions of the program.

The same considerations that we discussed for function *fresh_symbols*, related to the possibility of unfolding the primitive fields in the input data structures at any nesting level in finite steps, hold as well for function *save_symbolic_expressions* with the only change that, in this case, the function unfolds the symbolic expressions associated with all primitive fields that belong to the output data structures of the program. For our working program, Tecs customizes function *save_symbolic_expressions* to save the symbolic expressions associated with the primitive-typed output `carState` and the two primitive outputs that comprise the array `mirrorCommand`.

Technically, function *save_symbolic_expressions* generates a fresh symbol for each primitive-typed output, and informs Klee of the assumption that the new fresh symbol shall be equal to the value of the symbolic expression that is currently associated with the given output. This can be done with the Klee API `klee_assume`. For instance, for saving the symbolic expression associated with the output `carState`, *save_symbolic_expressions* generates a new fresh symbol (say s) and then calls `klee_assume(s==carState)`. This leads Klee to compute a result for the symbol s that reveals the value of `carState` at the moment when the assumption was evaluated during symbolic execution, consistently with the values that Klee computed for all other inputs.

Weak transitions. We now discuss a refinement of the steering algorithm (Algorithm 1) aimed to handle a special types of state transitions, called *weak transitions*, which can be defined in Scade models. When a weak transition is fired, the actions that it defines are activated and the state is updated, but the outputs of the target state become active one execution cycle later. Thus, a weak transition requires two, rather than one, execution cycles to complete. During the second cycle the state machine stays in the destination state of the weak transition, that is therefore visited twice.

Algorithm 2 extends the analysis driver to handle weak transitions. This new algorithm is equal to Algorithm 1, but includes the additional steps highlighted with gray-shadowed background. The algorithm has a new dependency on the predicate $\mathit{weak\_transition}(\mathit{state}_a, \mathit{state}_b)$ (line 5) that indicates whether or not the transition from state_a to state_b is a weak transition. We can deduce this information automatically out of the metadata that Scade associates with the program under test. After each execution step, if a weak transition is fired, i.e., if the predicate $\mathit{weak\_transition}(\mathit{state}, \mathit{state}')$ is true at line 17, then the variable *stutter* memorizes the fact. In this case, at the next iteration, the destination state of the weak transition is not added to the set of the visited states (line 14). This allows the program to complete the weak transition, which requires to visit that state once again, and then progress further on.

3.2. The test synthesizer

The Tecs Test synthesizer uses the test inputs obtained with Klee to construct test cases in Scade Test format. Fig. 5.b reports a sample test case in Scade Test format that was generated with Tecs. It consists of two test steps: The first test step sets (*SSM::set* test statements) `ctrl` to `UNLOCKED`, `automaticControl` to false and `mirrorState` to `OPEN` for both wing mirrors, in order to unlock the car and opening the wing mirrors. Thus, the test case doublechecks (*SSM::check*) that, after this step, `carState` is equal to `UNLOCKED` and the outputs `mirrorCommand` are both assigned to `OPEN`. When the test case executes the statement *SSM::cycle*, Scade Test 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 expecting that the `carState` moves to `LOCKED` while issuing `mirrorCommand` outputs equal to `CLOSED`.

To synthesize the test cases in Scade Test format, the Tecs Test synthesizer renders the test inputs that Klee yielded for a given execution path in the form of suitable *SSM::set* test statements, and renders the regression oracles that Klee yielded for that path in the form of suitable *SSM::check* test statements. For the execution paths that Klee 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 synthesizer 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 synthesizer relies on a set of naming conventions that the analysis driver enforces when defining the names for the symbolic 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 the input field, and (iii) the sequence number of the program call for which the analysis driver instantiated the fresh symbol.

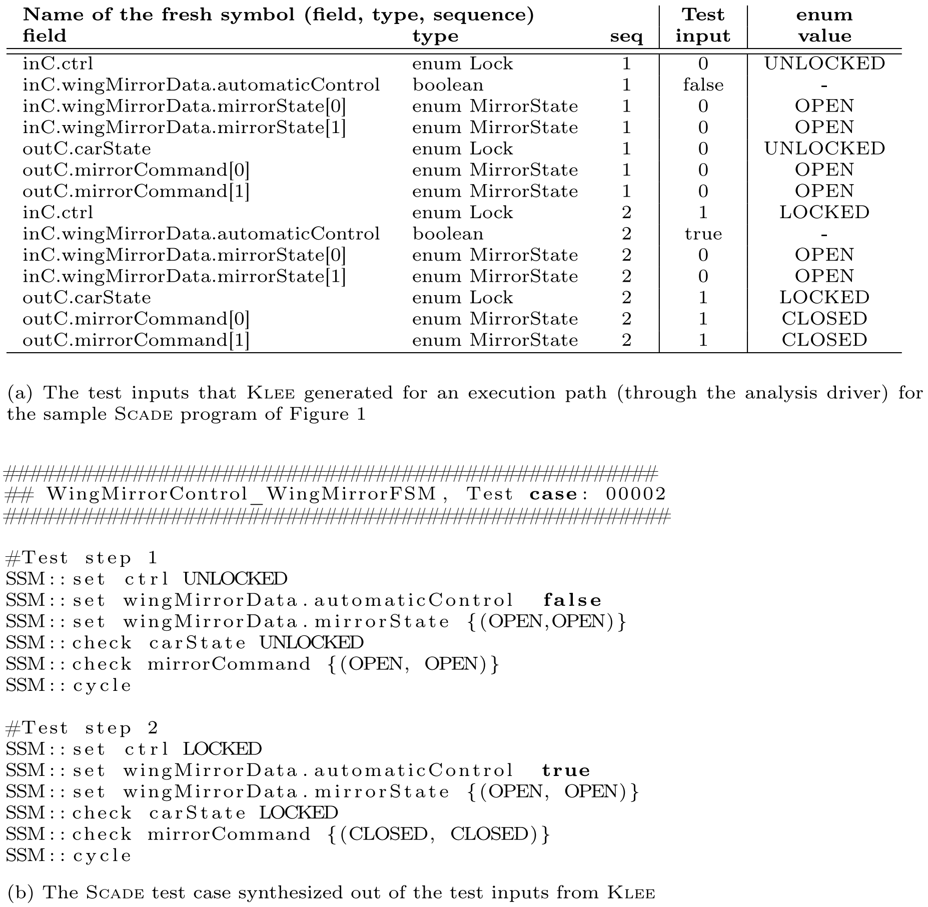
For instance, with reference to the code of function *fresh_symbols* generated for our working program (Fig. 4), the fresh symbol that the analysis driver associates with the input field `ctrl` at the second call of the program under test (for the execution paths that make at least two calls of the program) is named as *"field: inC.ctrl, type: enum Lock, sequence: 2"*. Thus, when Klee yields a test input 1 for that symbol, the test synthesizer understands that the value 1 shall be assigned to the input field `ctrl` at the second program call made in the test case. Moreover, knowing that the field is of type *enum Lock*, it can deduce that the value 1 refers to the second item defined in that enumeration, i.e., the value `LOCKED`. Thus, the test synthesizer generates the assignment `ctrl = LOCKED` at second test step.

Fig. 5 shows the test inputs (Fig. 5.a) that Klee generates for an execution path through the analysis driver for the sample Scade program of Fig. 1, and the Scade test case that Tecs synthesizes correspondingly (Fig. 5.b). The figure indicates the test inputs in tabular form to improve readability. Each row of the table corresponds to a test input from Klee. 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 Klee returned. The fifth column shows the matching enumeration value for test inputs of enumeration types. The test inputs that correspond to the fields of the data structure `inC` were generated in the

Algorithm 2 The algorithm of the analysis driver extended for weak transitions

```
Let:
  $program$ be the program under test,
  $inputs$ be a reference to the inputs of the program,
  $outputs$ be a reference to the outputs of the program,
  $s_0$ be the initial state of the program,
  $weak\_transition$ be a predicate that is true for state-pairs that correspond to weak transitions.

$outputs \leftarrow default\ values$
$state \leftarrow s_0$
$visited \leftarrow \emptyset$
$stutter \leftarrow false$
while $state \notin visited$ do
    $inputs \leftarrow fresh\_symbols()$
    $state', outputs \leftarrow program(state, inputs, outputs)$
    $save\_symbolic\_expressions(outputs)$
    if $\neg$ stutter then
        $visited \leftarrow visited \cup \{state\}$
    end if
    $stutter \leftarrow$ weak_transition(state, state')
    $state \leftarrow state'$
end while
```



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 KLEE generated for an execution path (through the analysis driver) for the sample SCADE program of Figure 1

```
###########################################################
## 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 KLEE

Fig. 5. A test case generated for the sample program of Fig. 1.

hook function *fresh_symbols* of the analysis driver: They indicate the input values for the test case. The ones that correspond to the fields of the data structure `outC` were generated in the hook function *save_symbolic_expressions*: They indicate values for regression oracles.

As the table indicates, KLEE 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 (Fig. 5.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 expecting in the assertions that the `carState` becomes `LOCKED` while issuing `CLOSED` for both `mirrorCommand` outputs.

3.3. Remarks

The method that Tecs realizes to initialize the program inputs with symbolic values, execute the program, and save the values of the outputs, would hardly work if we were addressing the symbolic execution of an arbitrary C program. Thus, our design of the test generator Tecs is tightly connected to the research hypotheses that this paper formulates about the class of programs identified by programming languages for safety-critical software, out of which we refer to Scade as a representative case.

In detail, with reference to the hook functions *fresh_symbols* and *save_symbolic_expressions* that we introduced and discussed in this section, if the inputs and the outputs of the program could be defined of the type of dynamically allocated recursive data structures, the analysis driver that Tecs synthesizes might lead Klee though infinite recursive steps in the attempt to initialize the fields at any nesting level, since the possible nesting levels would be unbounded for a recursive data structure. If dynamic memory allocation had to be considered, symbolic execution should handle pointer-aliases for the pointers present in input data, by considering all the possible alternative initializations in which they could either hold null values, or refer to any compatible memory location that belongs to the input state (Khurshid et al., 2003). If input arrays with non-statically-known length were allowed, there would be no immediate way to initialize them by unfolding their internal items.

With reference to the hook function *program* (which executes the state transitions of the Scade program under test), Tecs relies on the knowledge that the program under test is fully deterministic and does not include unbounded loops or recursion. This assumption guarantees that the analysis driver always analyzes a finite number of execution paths in each pass of the program, and always terminates for each execution path, without need of specifying any customized bound neither in the target program, nor within the symbolic executor. In general, this is impossible for symbolic-execution-based test generators that address arbitrary programs.

4. Case study

In this section we report on a case study where we evaluated the effectiveness of symbolic execution, as instantiated in our tools Tecs described in Section 3, for generating test cases for safety-critical software developed in Scade. We considered a set of Scade programs developed as part of a project for an on-board signaling unit for high speed rail. This project is currently being developed by an industrial partner, with whom we are collaborating. We used Tecs to automatically generate test cases for the considered Scade programs, and we evaluated our approach in terms of both the ability of Tecs to successfully accomplish the test generation process, and the quality of the resulting test suites.

Below we explain the research questions that drove our evaluation, describe the considered Scade programs, present the experimental setting of the case study, report on the results, and discuss the main threats to the validity of our current conclusions.

4.1. Research questions

In the case study we aimed to answer the following research questions:

- RQ1: To what extent is Tecs efficient in accomplishing test generation? This in terms of both the computational effort and the time budget that it requires for completing the test generation.
- RQ2: What is the quality of the test suites that Tecs generates? In particular, we considered the following quality dimensions:
    a. test effectiveness, in terms of both the structural coverage of the programs under test and the ability of detecting the possible faults in those programs;
    b. relative strength with respect to test suites obtained with other approaches, in particular either manually designed test suites or test suites generated with a search-based test generation strategy.

RQ1 aims to produce empirical evidence that we can effectively exploit symbolic execution to generate test cases for safety-critical software in Scade. As we explained in Section 3, Tecs concretizes this hypothesis by tailoring its implementation of symbolic execution on the restrictions by which Kcg fosters by-design safety guarantees in the programs. Thus, as RQ1 states, we aim to empirically study the computational effort and time budget requirements that result from the distinctive design of Tecs.

We answer RQ1 by quantifying how many execution paths Tecs actually analyzes when generating test cases for a set of Scade programs implemented by our industrial partner (presented below in Section 4.2), and how long it takes overall to complete the test generation process for those programs.

RQ2 aims to confirm the merit of generating test cases based on symbolic execution. We answer RQ2.a by measuring the size and the structural thoroughness of the test suites, and by experiencing with the generated test suites to support component-level testing of the considered programs. We answer RQ2.b by comparing with the manually designed test suites that were already available for three of the considered programs, and with test suites automatically derived with the tool AFL that is well known in security testing and implements a search-based test generation approach (AFL, 2022).

4.2. Subject programs

We considered the 37 Scade programs listed in Table 1. The table defines an identifier (first column) that we use to refer to each subject program in the sequel of the paper, and provides a short description (second column) of the task that each program executes. These programs are part of the on-board signaling unit for high speed rail that our industrial partner is currently developing. For example, the first program, `shunting` implements the Shunting procedure. In the 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 subject program shall handle the messages that the train receives from both the driver and the ground signaling equipment, to make decisions on when activating or deactivating the shunting mode. The other programs implement several control tasks, as checking and verify the consistency of the data that the on-board unit receives from the ground components,

Table 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 train is in a radio hole area
crossnonlx	Monitors a level crossing area that is not protected by external authorities
baliseinfo	Renders messages from on-railway transponders to the driver
emergency_1	Updates on-board data when receiving an emergency message
emergency_2	Acknowledges radio control center 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 center 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 center
runnumber_1	Updates the train ID on board
runnumber_2	Notifies changes of the train ID to the radio control center

computations of information for monitoring and controlling the train, rendering appropriate messages to the driver, and sending commands to the actuators.

Table 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 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 3.1 (Algorithm 2). 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 8000 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 on-board 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 on-board 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 1011 lines of code (plus the code defining the data types, i.e., as said, more than 8000 additional lines of code) being program `dc_8` and program `vbc` the smallest and the largest program, respectively.

4.3. Experimental setting

Our case study consisted of a set of experiments, one for each of the subject programs listed in Table 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 cloud facility hosted at our university, using a virtual machine equipped with Linux Ubuntu, 48 cpus, 150 GB of ram memory, which allowed for running multiple instances of TECS in parallel.

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 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, 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.

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

Table 2
Statistics of the subject programs.

Subject	Scade model					C code
	#States	#Transitions		#Inputs	#Outputs	LOC[(a)]
		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	1011
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

[(a)] C code LOC values refer to the lines of code in the C functions specific of each Scade program, but each program includes more than 8000 additional lines of code of data-type declarations, which define the data structures that comprise the inputs and the outputs of the programs.

Table 3
Results of Tecs for the subject programs considered in our case study.

Subject	Time (m s)	#paths	#tests	#test steps		Coverage
				Avg	Max	
shunting	4 m 46 s	3367	20	3	5	86%
dc_1	2 s	616	8	2	2	91%
dc_2	<1 s	2	2	2	2	100%
dc_3	<1 s	16	6	2	2	100%
dc_4	<1 s	3	2	2	2	92%
dc_5	<1 s	4	2	1	2	89%
dc_6	<1 s	3	2	2	2	90%
dc_7	<1 s	4	2	1	2	80%
dc_8	<1 s	4	3	1	2	83%
dc_9	2 s	208	9	2	2	100%
dc_10	<1 s	64	9	2	2	93%
dc_11	<1 s	3	2	2	2	100%
dc_12	<1 s	3	3	2	2	72%
dc_13	<1 s	20	4	2	2	98%
dc_14	<1 s	4	2	1	1	82%
radiohole	1 m 57 s	45	6	3	3	95%
crossnonlx	10 m 47 s	294	13	3	3	84%
baliseinfo	1 s	3	3	1	2	97%
emergency_1	15 s	28	14	1	2	94%
emergency_2	29 s	8	6	1	2	82%
mema	23 s	17	7	1	2	89%
trackside	18 m 57 s	3	3	1	2	99%
vbc	2 m 44 s	77	12	1	2	94%
coordfromrbc	41 s	7	5	1	2	83%
adfactordmi_1	31 m 0 s	3	3	1	2	85%
adfactordmi_2	1 s	2	2	1	1	96%
driveridins	5 s	10	10	1	2	89%
eirene	3 s	3	3	1	1	94%
ertmslevel	2 s	3	3	1	1	94%
natvalues	20 m 30 s	4	4	1	1	90%
networkidins	1 s	3	3	1	1	94%
rbcidins	3 s	4	4	1	2	95%
trainDataUpdate	47 s	2	1	1	2	89%
trainDataInsertion	28 s	4	3	1	1	95%
message129	1 m 39 s	80	10	1	2	83%
runnumber_1	2 s	3	3	1	2	94%
runnumber_2	3 s	22	7	1	2	92%

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 favor 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.

4.4. Results

Table 3 summarizes 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*), the number of execution paths analyzed with symbolic execution (column *#paths*), the number of test cases generated after executing the minimization step (column *#tests*), the average and maximum number of test steps within the test cases (columns *#test steps*), and the model coverage of the test cases (column *coverage*).

RQ1: Efficiency

The data in Table 3 show that Tecs completed in finite time in all experiments, supporting our hypothesis that, thanks to the language restrictions that Scade embraces to promote safe programs, we can exploit symbolic execution to efficiently explore the execution space of the programs under test without need of specifying custom bounds for the analysis.

In detail, for most subject programs, Tecs took a few seconds to complete the test generation process. It took more than 1 min only for 8 out of 37 subject programs, and more than 10 min only for 4 programs, namely, `crossnolx`, `trackside`, `adfactordmi_1` and `natvalues`, the maximum time being 31 min (1860 s) in the experiment with program `adfactordmi_1`.

In all experiments Tecs used most computation time to complete the symbolic execution with Klee, under the guidance of the Tecs analysis driver, while the other phases of Tecs, i.e., synthesizing the analysis driver, and synthesizing the test cases in Scade format, took negligible time.

We investigated in further detail the experiments in which the time budget was not justified by the (low) number of symbolically executed paths. For these cases, we investigated whether the time budget was bounded by some complex execution conditions

Table 4
Data on the queries issued to the constraint solver.

Subject	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

that took 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 shows these data in particular for the subject programs (column *subject*) for which TECS 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 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 two constraint solver 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.

RQ2.a: Structural coverage

Table 3 shows that the test suites that TECS generated in our experiments consist of a minimum of 1 test case, for program `trainDataUpdate`, up to a maximum of 20 test cases for `shunting`. The number of corresponding test steps is either 1 or 2 in all test cases generated for the programs that consist of only a strong or a weak transition, respectively, while it is higher for the three programs that define SCADE models with more states and transitions, i.e., `shunting` (5 states, 10 transitions), `radihole` (3 states, 3 transitions) and `crossnonlx` (3 states, 3 transitions). For these programs, the generated test cases consist of 3 test steps on the average, up to a maximum of 5, 3 and 3 test steps for `shunting`, `radihole` and `crossnonlx`, respectively.

The generated test suites achieved a model coverage of 100% for 4 subject programs, at least 90% for 19 further programs, at least 80% for 13 programs, and 72% in the only case of program `dc_12`.

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. We tracked the uncovered items to four distinct motivations:

- Items that depend on infeasible program paths: In fact, many subject programs include infeasible paths, the most frequent case being the one of 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 to misleadingly classify 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 behaviors out of the scope the single-state-path-coverage testing criterion that TECS uses for steering the test generation process: We observed uncovered functional behaviors 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 behavior* 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 facts, these behaviors 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 3 while discussing the Minimizer step of TECS, a limitation of the current implementation is to select test cases based on statement coverage, which is a grosser grained criterion than the modified condition/decision coverage of transition guards considered in SCADE Test. 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 release.

Out of the above cases, only the last two map to limitations of our approach. While the former of these limitations suggests the strategy of complementing the automatically generated test cases for the programs with missing coverage (by searching for functional behaviors that require iterating multiple times though the same state), the latter could be mitigated by improving the implementation of TECS. We evaluated the room for the coverage improvement that we might achieve with a different strategy for selecting test cases out of the symbolically analyzed execution paths. To this end, we re-executed TECS after disabling the Minimizer option in KLEE, thus making TECS compute exactly one test case for each symbolically analyzed execution path. Table 5 compares the number of test cases and the coverage results that we achieved with and without the Minimizer, respectively (but for program `shunting` for which the high number of test cases computed without Minimizer – 3367 test cases – exceeded the capability of SCADE Test to execute the test suite). In the table, we highlighted in bold the 6 cases in which the coverage rate improved without using the Minimizer. The amount of improvement was 1% for `radiohole` and `message129`, 2% for `crossnolx` and `emergency_2`, 4% for `mema`, and up to 7% for `dc_1`.

RQ2.a: 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 requirement 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 6 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.

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

RQ2.b: Comparison with manually derived test suites

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 engineers reported to us that the analysis of the requirements, the selection of the test cases and their manual implementation in a the test suite took overall 16 man-hours (two days of work), 3 man-hours (about half day) and 9 man-hours (about one day) for `shunting`, `radiohole` and `crossnlnlx` respectively. They tracked the main challenges to (i) devising a suitable functional partitioning of the relevant cases to be tested (which in turn required to reiterate multiple times the study and the analysis of the requirement 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 Fig. 5.b).

Table 7 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*), the number of test cases (column *#tests*) and the corresponding model coverage (column *coverage*).

The manually derived test suites are sightly more compact in terms of number of test cases than the automatically generated counterparts, but it is clear that 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. 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`.

Table 5
Results of Tecs with and without the Minimizer.

Subject	With Minimizer		No Minimizer	
	#tests	Coverage	#tests	Coverage
shunting	20	86%	3367	n.a.
dc_1	8	91%	616	**98%**
dc_2	2	100%	2	100%
dc_3	6	100%	16	100%
dc_4	2	92%	3	92%
dc_5	2	89%	4	89%
dc_6	2	90%	3	90%
dc_7	2	80%	4	80%
dc_8	3	83%	4	83%
dc_9	9	100%	208	100%
dc_10	9	93%	64	93%
dc_11	2	100%	3	100%
dc_12	3	72%	3	72%
dc_13	4	98%	20	98%
dc_14	2	82%	4	82%
radiohole	6	95%	45	**96%**
crossnonlx	13	84%	294	**86%**
baliseinfo	3	97%	3	97%
emergency_1	14	94%	28	94%
emergency_2	6	82%	8	**84%**
mema	7	89%	17	**93%**
trackside	3	99%	3	99%
vbc	12	94%	77	94%
coordfromrbc	5	83%	7	83%
adfactordmi_1	3	85%	3	85%
adfactordmi_2	2	96%	2	96%
driveridins	10	89%	10	89%
eirene	3	94%	3	94%
ertmslevel	3	94%	3	94%
natvalues	4	90%	4	90%
networkidins	3	94%	3	94%
rbcidins	4	95%	4	95%
trainDataUpdate	1	89%	2	89%
trainDataInsertion	3	95%	4	95%
message129	10	83%	80	**84%**
runnumber_1	3	94%	3	94%
runnumber_2	7	92%	22	92%

cov = n.a., if Scade Test failed due to too many test cases.

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 behaviors (since they correspond to a 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 behavior that Tecs did not cover in `radiohole` and `crossnonlx`, where Tecs was in fact able to cover some additional rare combinations.

RQ2.b: Comparison with search-based testing

We investigated if our approach could work also by using search-based random testing heuristics in place of symbolic execution. To this end, we implemented an alternative version of Tecs that used the test generator AFL (2022) instead of Klee 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 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

Table 6
Faults identified in the subject programs considered in our case study.

Subject	Fault
dc_10	Wrong amount of data written in a queue
	Wrongly defined algorithm
coordfromrbc	Missing update of a state variable
	Array updated with 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 wrong sequence
Total	7 faults

Table 7
Comparison between automatically and manually derived test suites.

Subject	Manual test suite			TECS		
	Time	#tests	Coverage	Time	#tests	Coverage
shunting	16 h	15	95%	286 s	20	86%
radiohole	6 h	1	94%	117 s	6	95%
crossnonlx	9 h	3	80%	647 s	13	84%

Table 8
Comparison between Tecs and AFL.

Subject	TECS		AFL		Diff.
	#tests	Coverage	#tests	Coverage	
shunting	20	86%	12	46%	40%
dc_1	8	91%	16	89%	2%
dc_2	2	100%	1	100%	0%
dc_3	6	100%	5	100%	0%
dc_4	2	92%	2	86%	6%
dc_5	2	89%	1	89%	0%
dc_6	2	90%	3	84%	6%
dc_7	2	80%	1	50%	30%
dc_8	3	83%	1	50%	33%
dc_9	9	100%	6	100%	0%
dc_10	9	93%	6	93%	0%
dc_11	2	100%	1	100%	0%
dc_12	3	72%	3	60%	12%
dc_13	4	98%	4	82%	16%
dc_14	2	82%	1	64%	18%
radiohole	6	95%	4	68%	27%
crossnonlx	13	84%	6	19%	65%
baliseinfo	3	97%	2	45%	52%
emergency 1	14	94%	1	6%	88%
emergency 2	6	82%	6	54%	28%
mema	7	89%	5	41%	49%
trackside	3	99%	5	20%	79%
vbc	12	94%	3	40%	54%
coordfromrbc	5	83%	7	57%	26%
adfactordmi 1	3	85%	2	71%	14%
adfactordmi 2	2	96%	3	96%	0%
driveridins	10	89%	2	65%	24%
eirene	3	94%	3	66%	28%
ertmslevel	3	94%	4	88%	6%
natvalues	4	90%	–	–	–
networkidins	3	94%	2	75%	19%
rbcidins	4	95%	2	49%	46%
trainDataUpdate	1	89%	4	60%	29%
trainDataInsertion	3	95%	6	89%	6%
message129	10	83%	8	77%	6%
runnumber 1	3	94%	2	70%	24%
runnumber 2	7	92%	7	93%	−1%

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 next steps. Thus, for each subject program, we proceeded as follows: we executed AFL for 5 h; 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.

Table 8 reports on test cases generated with AFL for the programs considered in our case study (but program `natvalues` for which AFL unexpectedly generated a broken instrumentation that made the program crash deterministically at runtime). The table indicates the information of the test cases generated with Tecs when equipped with Klee (columns Tecs), in comparison with the number of test cases and corresponding model coverage data achieved with AFL (columns *AFL*), and shows the difference between the coverage measurements in either case (column *diff*).

The data in the table indicate that the two approaches led to generating test suites of comparable size in most cases, but the model coverage achieved with AFL was often significantly lower than the coverage achieved with Tecs. AFL achieved the same amount model coverage as Tecs for 7 programs (namely, `dc_2`, `dc_3`, `dc_5`, `dc_9`, `dc_10`, `dc_11` and `adfactordmi` 2), achieved more coverage than Tecs only for 1 program (namely, `runnumber` 2), and achieved less coverage than Tecs for the remaining 28 programs. In the 28 cases in which Tecs outperformed AFL, the difference in coverage ranged between 2% and 88%, with a median of 26%. In the only case in which AFL outperformed Tecs, the difference in coverage was rather limited (1%) due to a single MC/DC objective that Tecs 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.

By inspecting the coverage objectives that were missed with AFL we tracked most untested code to program branches that depend 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 interpret these data as clear evidence that a tool like AFL does not suit for our goal of testing safety-critical software.

In summary, our case study indicated that the test generation approach that we propose in this paper, as instantiated in the tool Tecs, successfully exploits symbolic execution to generate high-quality test suites for safety-critical programs in Scade, thus confirming the main research hypothesis of this paper. The test suites that we automatically generated with Tecs in our experiments readily satisfied most domain-relevant test objectives, unveiling at the same time small portions of test objectives that require dedicated handling. This straightforwardly suggests a combined approach in which the testers of safety-critical software can efficiently start working with the test cases automatically computed with Tecs, and then complementarily concentrate on the yet-missed behaviors, thus crucially improving both costs and the effectiveness of their test-design efforts.

4.5. Threats to validity

The main internal threats to the validity of our findings are concerned with the risk of implementation errors in Tecs (that could bias our results), and with use of coverage indicators to evaluate the quality of the test suites.

We extensively tested Tecs to ascertain its correctness, and manually crosschecked several result samples. For the implementation of the symbolic execution phase, which is at the core of the results that we computed with Tecs, we relied on Klee, a state of the art symbolic executor actively maintained and largely used in the community. Thus we are confident in the validity of the results that we obtained with Tecs.

We evaluated the quality of the test suites that Tecs computed in our experiments based on the model coverage indicators obtained with the tool Scade Test. We drew on the documentation provided from Ansys and on the advising of the industrial partner with whom we are collaborating, to reckon that the coverage indicators computed with Scade Test correspond to domain-relevant coverage requirements. However, we are well aware that any coverage measurement is just a proxy of the effectiveness of the test cases, and we cannot take for granted that high coverage rates necessarily correspond to high fault-detection power. We attacked this issue by showing that the test suites generated by our tool, once complemented with assertion-style test oracles, succeeded in revealing component-level faults of the considered Scade programs. We look forward to experiencing Tecs on further Scade programs that are currently being developed in the project, to collect further data on which faults we can indeed detect with the help of the test cases generated with Tecs.

The external threats to validity relate to the extent to which our finding can generalize. So far, we experienced Tecs against the set of subject programs considered in this paper, which are admittedly only a small sample of the possible safety-critical programs. 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 generalize.

5. Related work

We surveyed the most relevant techniques for automated test generation for software programs in the introduction of this paper, encompassing test generators based on random testing, search-based testing and symbolic execution. Random testing and search-based testing derive test cases by either randomly sampling the possible program inputs or based on dynamic data about the execution of the programs. Symbolic execution systematically unfolds the execution space of the programs under test and generates test cases by solving the execution of the possible program paths. Then we have described our approach to test case generation for safety-critical programs in Scade, which is based on symbolic execution, and compared our approach with a analogous embodiment based on search-based testing by referring to the tool AFL. Other results on the effectiveness of automated test generation in safety critical systems are provided in Enoiu et al. (2016) and Gay et al. (2015).

The research that we described in the paper is also related to other approaches for automated model-based testing, to methods for formally specifying and verifying safety-critical software, and to other pieces of research on verifying programs in Scade.

5.1. 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 behaviors expressed in suitable modeling languages, e.g., UML class diagrams, state machines or sequence diagrams (Utting and Legeard, 2010; Object Management Group, 2017). Model-based testing has been successfully applied to complement verification of formal specifications expressed in languages as B, Z or VDM (Hierons et al., 2015). For a comprehensive survey of model-based testing we refer the readers to the work of Utting et al. (2012) and Dias Neto et al. (2007).

The approach that we presented in this paper addresses the test generation problem based on the analysis of the execution paths in the programs, and naturally lends itself to complement or be complemented with further test cases generated either manually or yet automatically in model-based fashion.

In particular our approach shares similarities with the ones of Polyglot (Balasubramanian et al., 2011, 2012) and SAUML (Zurowska and Dingel, 2011), which exploit symbolic execution to generate test cases for systems modeled with statecharts and UML-RT state machines, respectively. Polyglot is similar to Tecs in that it translates statecharts to programs (specifically programs in Java) and then exploits symbolic execution (by means of the symbolic executor SPF (Păsăreanu and Rungta, 2010) 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. Our approach differs from both these approaches in the way Tecs distinctively uses symbolic execution within an analysis algorithm tailored on the characteristics of the Scade models, which foster programs with finite path spaces and input data structures comprised of finite sets of distinct fields.

5.2. 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 (Abrial, 2006) 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 method define mathematically rigorous procedures to ensure that the artifacts produced at every step of a development process *refine* the artifacts produced at earlier steps, thus preserving all their relevant properties. The downside of these approaches is the degree of mathematical sophistication that they demand to 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 with tools that assist in performing their tasks, with various degrees of automation, which anyway hardly balance the aforementioned complexity.

Formal methods differ for the breadth of their scope. At one end of the spectrum, methods like B or its successor Event-B (Abrial, 2010) 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 (Jackson, 2012), a language and a tool for modeling systems that is suited to assist the specification and abstract design activities. Similarly, Z (Spivey, 1989) is customarily used as a system modeling language, although there also exists a well-established theory of refinement for Z (Woodcock and Davies, 1996).

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. 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 (Hierons et al., 2015).

5.3. Automated test generation for Scade models

Scade can be regarded as a formal modeling 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 (Halbwachs et al., 1991), with some programming constructs derived from the programming language Esterel (Berry and Gonthier, 1992) and from the graphical, state-machine-based language SyncCharts (André, 1996). 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).

To the best of our knowledge, the only research works that address automated test generation for Scade or Lustre programs is the work proposed in Lakehal and Parissis (2005). 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 (Phol et al., 2017).

The above approach, however, differs in aim and scope from ours. Our approach systematically analyzes the C code that corresponds to the Scade programs, while the approach of Lakehal and Parissis (2005) does not consider the generated C code. The authors of Lakehal and Parissis (2005) propose dedicated coverage measures, specific for synchronous dataflow programming languages, while we aim at covering all execution paths in the programs.

An interesting tool is RT-Tester (Braunstein et al., 2014; Vu et al., 2014), 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 behavior of the system under test, renders the models into a set of expressions in propositional logic, and then solves the formulas with a SMT solver to generate test cases. Bounded model checkers, like CBMC, take a similar approach (Clarke et al., 2004). They represent programs with boolean formulas, which they then check for satisfiability by using a SAT solver, to generate test cases as counterexamples of verification properties. In the future, we aim to compare with these approaches.

6. Conclusions

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 paper, we studied the viability of an automated test generation approach based on symbolic execution, specifically tailored on the characteristics of a programming language for safety-critical software systems. We instantiated the proposed approach with the tool Tecs, a test generator for programs written the Scade language. The case study that we reported in this paper indicates that the proposed approach was able to successfully produce 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.

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. 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 (Phol et al., 2017).

Lastly, the test cases generated by Tecs currently 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 (Goffi et al., 2016) so that the oracles are generated from the software requirements specification documents.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data that has been used is confidential.

Acknowledgments

This work is partially supported by the SISMA national research project (MIUR, Italy, PRIN 2017, Contract 201752ENYB). We thank Raffaele Saponara for his contribution in running some of the experiments reported in the paper, our colleagues of the development team and our industrial partner for their help.

References

Abrial, J.-R., 2006. Formal methods in industry: Achievements, problems, future. In: Proceedings of the International Conference on Software Engineering, ICSE '06. ACM, pp. 761–768. http://dx.doi.org/10.1145/1134285.1134406.

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

American Fuzzy Lop (AFL), 2022. (Accessed January 2022). URL https://lcamtuf.coredump.cx/afl/.

Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K., 2010. A systematic review of the application and empirical investigation of search-based test case generation. IEEE Trans. Softw. Eng. 36 (6), 742–762. http://dx.doi.org/10.1109/TSE.2009.52.

André, C., 1996. Representation and analysis of reactive behaviors: A synchronous approach. In: Proceedings of IMACS/IEEE Multiconference on Computational Engineering in Systems Applications, CESA '96. IEEE, pp. 19–29.

Balasubramanian, D., Păsăreanu, C., Whalen, M.W., Karasi, G., Lowry, M., 2012. Improving symbolic execution for statechart formalisms. In: Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation, MoDeVVa '12. ACM, pp. 47–52. http://dx.doi.org/10.1145/2427376.2427385.

Balasubramanian, D., Păsăreanu, C.S., Whalen, M.W., Karsai, G., Lowry, M., 2011. Polyglot: Modeling and analysis for multiple statechart formalisms. In: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA '11. ACM, pp. 45–55. http://dx.doi.org/10.1145/2001420.2001427.

Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I., 2018. A survey of symbolic execution techniques. ACM Comput. Surv. 51 (3), 50:1–50:39. http://dx.doi.org/10.1145/3182657.

Beichler, B., Schulz, T., Haubelt, C., Golatowski, F., 2015. A parametric dataflow model for the speed and distance monitoring in novel train control systems. In: Proceedings of the International Workshop on Design, Modeling, and Evaluation of Cyber Physical Systems, CyPhy '15. Springer, pp. 56–66. http://dx.doi.org/10.1007/978-3-319-25141-7_5.

Berry, G., 2007. SCADE: Synchronous design and validation of embedded control software. In: Proceedings of the GM R & D Workshop: Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems. Springer, pp. 19–33. http://dx.doi.org/10.1007/978-1-4020-6254-4_2.

Berry, G., Gonthier, G., 1992. The Esterel synchronous programming language: Design, semantics, implementation. Sci. Comput. Program. 19 (2), 87–152. http://dx.doi.org/10.1016/0167-6423(92)90005-V.

Bovet, J., Parr, T., 2008. ANTLRWorks: An ANTLR grammar development environment. Softw. - Pract. Exp. 38 (12), 1305–1332. http://dx.doi.org/10.1002/spe.872.

Braione, P., Denaro, G., Mattavelli, A., Pezzè, M., 2017. 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, pp. 90–101. http://dx.doi.org/10.1145/3092703.3092715.

Braione, P., Denaro, G., Pezzè, M., 2016. 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, pp. 1018–1022. http://dx.doi.org/10.1145/2950290.2983940.

Braunstein, C., Haxthausen, A.E., Huang, W.-l., Hübner, F., Peleska, J., Schulze, U., Vu Hong, L., 2014. Complete model-based equivalence class testing for the ETCS ceiling speed monitor. In: Proceedings of the International Conference on Formal Engineering Methods: Formal Methods and Software Engineering, ICFEM '14. Springer, pp. 380–395. http://dx.doi.org/10.1007/978-3-319-11737-9_25.

Cadar, C., Sen, K., 2013. Symbolic execution for software testing: Three decades later. Commun. ACM 56 (2), 82–90. http://dx.doi.org/10.1145/2408776.2408795.

Camus, J.-L., 2015. Esterel SCADE Approach to MBD, Digital Avionics Handbook. Taylor & Francis Group.

CENELEC, 2020. EN 50128, railway applications – communication, signaling and processing systems – software for railway control and protection systems.

Chen, T.Y., Kuo, F.-C., Merkel, R.G., Tse, T.H., 2010. Adaptive random testing: The ART of test case diversity. J. Syst. Softw. 83 (1), 60–66. http://dx.doi.org/10.1016/j.jss.2009.02.022.

Chipounov, V., Kuznetsov, V., Candea, G., 2012. The S2E platform: Design, implementation, and applications. ACM Trans. Comput. Syst. 30 (1), 1–49. http://dx.doi.org/10.1145/2110356.2110358.

Clarke, L.A., 1976. A system to generate test data and symbolically execute programs. IEEE Trans. Softw. Eng. SE-2 (3), 215–222. http://dx.doi.org/10.1109/TSE.1976.233817.

Clarke, E., Kroening, D., Lerda, F., 2004. A tool for checking ANSI-C programs. In: Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS '04. Springer, pp. 168–176. http://dx.doi.org/10.1007/978-3-540-24730-2_15.

Cristian Cadar, D.E., Dunbar, Daniel, 2008. 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, pp. 209–224.

De Moura, L., Bjørner, N., 2008. 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, pp. 337–340. http://dx.doi.org/10.1007/978-3-540-78800-3_24.

Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H., 2007. 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, WEASELTech '07. ACM, pp. 31–36. http://dx.doi.org/10.1145/1353673.1353681.

Duran, J.W., Ntafos, S.C., 1984. An evaluation of random testing. IEEE Trans. Softw. Eng. 10 (4), 438–444. http://dx.doi.org/10.1109/TSE.1984.5010257.

Dutertre, B., 2014. Yices 2.2. In: Proceedings of the International Conference on Computer Aided Verification, CAV '14. Springer, pp. 737–744. http://dx.doi.org/10.1007/978-3-319-08867-9_49.

Enoiu, E.P., Cauevic, A., Sundmark, D., Pettersson, P., 2016. A controlled experiment in testing of safety-critical embedded software. In: Proceedings of the International Conference on Software Testing, Verification and Validation, ICST '16. pp. 1–11. http://dx.doi.org/10.1109/ICST.2016.15.

Fornari, X., 2010. Understanding how SCADE Suite KCG Generates Safe C Code. Esterel Technologies.

Fraser, G., Arcuri, A., 2011. 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, pp. 416–419. http://dx.doi.org/10.1145/2025113.2025179.

Ganesh, V., Dill, D.L., 2007. A decision procedure for bit-vectors and arrays. In: Proceedings of the International Conference on Computer Aided Verification, CAV '07. Springer, pp. 519–531. http://dx.doi.org/10.1007/978-3-540-73368-3_52.

Gay, G., Staats, M., Whalen, M., Heimdahl, M.P.E., 2015. The risks of coverage-directed test case generation. IEEE Trans. Softw. Eng. 41 (8), 803–819. http://dx.doi.org/10.1109/TSE.2015.2421011.

Godefroid, P., Levin, M.Y., Molnar, D.A., 2008. Automated whitebox fuzz testing. In: Proceedings of Network and Distributed Systems Security Symposium, NDSS '08. Internet Society, pp. 151–166.

Goffi, A., Gorla, A., Ernst, M.D., Pezzè, M., 2016. Automatic generation of oracles for exceptional behaviors. In: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA '16. ACM, pp. 213–224. http://dx.doi.org/10.1145/2931037.2931061.

Gudemann, M., Angerer, A., Ortmeier, F., Reif, W., 2007. Modeling of self-adaptive systems with SCADE. In: Proceedings of the IEEE International Symposium on Circuits and Systems, ISCAS '07. IEEE, pp. 2922–2925. http://dx.doi.org/10.1109/ISCAS.2007.377861.

Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D., 1991. The synchronous data flow programming language LUSTRE. Proc. IEEE 79 (9), 1305–1320. http://dx.doi.org/10.1109/5.97300.

Harel, D., 1987. Statecharts: A visual formalism for complex systems. Sci. Comput. Program. 8 (3), 231–274. http://dx.doi.org/10.1016/0167-6423(87)90035-9.

Hatton, L., 1995. Safer C: Developing Software in High-Integrity and Safety-Critical Systems. In: McGraw-Hill International Series in Software Engineering, McGraw-Hill.

Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R., Zedan, H., 2015. Using formal specifications to support testing. ACM Comput. Surv. 41 (2), 18:1–18:41. http://dx.doi.org/10.1145/1459352.1459354.

IEC, 2010. IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems.

ISO, 2010. ISO 26262: Road vehicles — Functional safety.

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

Karg, S., Raschke, A., Tichy, M., Liebel, G., 2016. 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. ACM, pp. 238–248. http://dx.doi.org/10.1145/2976767.2976811.

Khurshid, S., Păsăreanu, C.S., Visser, W., 2003. 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/ETAPS '03. Springer, pp. 553–568. http://dx.doi.org/10.1007/3-540-36577-X_40.

King, J.C., 1976. Symbolic execution and program testing. Commun. ACM 19 (7), 385–394. http://dx.doi.org/10.1145/360248.360252.

Lakehal, A., Parissis, I., 2005. Lustructu: A tool for the automatic coverage assessment of LUSTRE programs. In: Proceedings of the International Symposium on Software Reliability Engineering, ISSRE '05. IEEE, pp. 301–310. http://dx.doi.org/10.1109/ISSRE.2005.26.

Le Sergent, T., 2012. SCADE: A comprehensive framework for critical system and software engineering. In: Proceedings of the International SDL Forum: Integrating System and Software Modeling, SDL '12. Springer, pp. 2–3. http://dx.doi.org/10.1007/978-3-642-25264-8_2.

Object Management Group, 2017. OMG® unified modeling language® (OMG UML® (version 2.5.1)).

Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T., 2007. Feedback-directed random test generation. In: Proceedings of the International Conference on Software Engineering, ICSE '07. ACM, pp. 75–84. http://dx.doi.org/10.1109/ICSE.2007.37.

Petit-Doche, M., Breton, N., Courbis, R., Fonteneau, Y., Güdemann, M., 2015. Formal verification of industrial critical software. In: Proceedings of the International Workshop on Formal Methods for Industrial Critical Systems, FMICS '15. Springer, pp. 1–11. http://dx.doi.org/10.1007/978-3-319-19458-5_1.

Pezzè, M., Young, M., 2007. Software Testing and Analysis: Process, Principles and Techniques. Wiley.

Phol, L.V., Binh, N.T., Parissis, I., 2017. Mutants generation for testing LUSTRE programs. In: Proceedings of the Eighth International Symposium on Information and Communication Technology, SoICT '17. ACM, pp. 425–430. http://dx.doi.org/10.1145/3155133.3155155.

Păsăreanu, C.S., Rungta, N., 2010. Symbolic PathFinder: Symbolic execution of Java bytecode. In: Proceedings of the International Conference on Automated Software Engineering, ASE '10. ACM, pp. 179–180. http://dx.doi.org/10.1145/1858996.1859035.

Qian, J., Liu, J., Chen, X., Sun, J., 2015. 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, pp. 48–54. http://dx.doi.org/10.1109/COUFLESS.2015.15.

RTCA, 2012. DO-178C, software considerations in airborne systems and equipment certification.

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

Tillmann, N., de Halleux, J., 2008. Pex: White box test generation for .NET. In: Proceedings of the International Conference on Tests and Proofs, TAP '08. Springer, pp. 134–153. http://dx.doi.org/10.1007/978-3-540-79124-9_10.

Tonella, P., 2004. Evolutionary testing of classes. In: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA '04. ACM, pp. 119–128. http://dx.doi.org/10.1145/1007512.1007528.

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

Utting, M., Pretschner, A., Legeard, B., 2012. A taxonomy of model based testing approaches. J. Softw.: Test. Verif. Reliab. 22 (5), 297–312. http://dx.doi.org/10.1002/stvr.456.

Vu, L., Haxthausen, A., Peleska, J., 2014. A domain-specific language for railway interlocking systems. In: Proceedings of the 10th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems, FORMS/FORMAT '14. Technische Universität Braunschweig, pp. 200–209.

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

Zurowska, K., Dingel, J., 2011. SAUML: A tool for symbolic analysis of UML-RT models. In: Proceedings of the International Conference on Automated Software Engineering, ASE '11. pp. 604–607. http://dx.doi.org/10.1109/ASE.2011.6100136.