Automated Test Case Generation for Safety-Critical Software in Scade

Elson Kurian*, Pietro Braione*, Daniela Briola*, Dario D'Avino#, Matteo Modonato*, Giovanni Denaro*

*University of Milano - Bicocca
Dept. of Informatics, Systems and Communication
Milano, Italy

#Rete Ferroviaria Italiana
R&D Sviluppo Sistemi
Napoli, Italy

{elson.kurian, pietro.braione, daniela.briola, matteo.modonato, giovanni.denaro}@unimib.it
dario.davino@rfi.it

***Abstract*—Software systems for automating safety-critical tasks in application domains like, for example, avionics, railways, automotive, industry 4.0 and healthcare, must be highly reliable. In this paper, we focus on safety-critical software written in Scade, a model-based programming language largely adopted in industry, and we specifically draw on our own experience in a joint industry-university project aimed at developing safety-critical Scade programs for the railways domain. We investigate automated test case generation for Scade programs. We leverage on state-of-the-art test generators based on either symbolic execution, bounded model checking or search-based testing, in order to define an original toolchain for generating test cases for Scade programs. We rely on the toolchain to explore the absolute and relative effectiveness of those mainstream test generation approaches on a benchmark of 37 Scade programs developed as part of an on-board signaling unit for high-speed railway systems.**

***Index Terms*—Safety-critical Scade programs, Automated test generation, Symbolic execution, Bounded model checking, Search-based testing.**

I. Introduction

Safety-critical software must guarantee highly reliable automation in application domains like avionics, railways, automotive, industry 4.0 and patient monitoring, where failures can have catastrophic consequences. Because of these risks, the development process of safety-critical software typically encompasses several quality-oriented requirements, driven by the goal of satisfying the concerned certification authorities [1], [2]. A common tenet is to rely on programming languages that, by their design choices and controlled semantics, may both decrease the chances of introducing subtle faults in the programs, and mitigate the hard work for satisfying the certification requirements [3].

We draw on a joint research and development effort of Rete Ferroviaria Italiana (RFI, the public company that manages the railway infrastructure in Italy) and University of Milano-Bicocca, aimed at developing an on-board signaling unit for high-speed railway systems, compliant with the ERTMS[1] standard specification. With the aim of ensuring the highest degree of software integrity, RFI is relying on the model-based Scade programming language for the development of this safety-critical software. Scade is a synchronous language, like LUSTRE [4] and ESTEREL [5], designed for the development of embedded safety-critical software systems. A Scade program is structured as a collection of communicating components, each designed as a state machine or as a pure dataflow component. The computation of a Scade program proceeds as a sequence of discrete steps referred to as execution cycles. At each execution cycle the outputs and the next state of each component are calculated based on the current inputs and the current state, and at the end of a cycle all components perform an instantaneous transition to the next state as they enter the next cycle. Scade is largely adopted in industry: Ansys, the company that commercializes Scade and the supporting Scade Suite model-based design environment, reports[2] uses of Scade for many safety-critical, embedded applications, including avionics, flight control, automotive, autonomous vehicles and gas turbines [6]–[12].

In this paper we consider the problem of automatically generating unit-level test cases for Scade programs, and we investigate the absolute and relative effectiveness of the mainstream test generation approaches when applied for this purpose. Our work focuses on achieving high structural coverage, as required for certification of safety critical programs [1], [2], and thus differs from existing approaches that generate test cases either at random (e.g., Lutess [13]) or by focusing on invariants or safety properties described in Lustre (e.g., Lurette [14] and Gatel [15]). In particular we focused on the observable modified-condition/decision-coverage (O-MC/DC), the criterion that Scade advises for software that shall work with high integrity level [16].

As the test generation approaches, we consider search-based testing, symbolic execution and bounded model checking. Search-based testing randomly samples the input space of the target program, guided by heuristics based on the improvement of a *fitness* function that represents the extent to which the test cases fulfilled the test objectives [17]–[21]. Symbolic execution and bounded model checking systematically explore

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

[2]www.ansys.com/products/embedded-software/ansys-scade-suite

the execution space of the program under test. Symbolic execution computes the execution conditions of program paths as sets of constraints over symbols that represent the possible program inputs, and then solves these constraints to concrete test data with off-the-shelf constraint [22]–[29]. Bounded model checking encodes the semantics of all execution paths (up to a bounded length) as a boolean formula, and addresses a set of reachability properties of interest by solving the formula in conjunction with the constraints that represent the properties [30]. By encoding coverage objectives as a reachability properties bounded model checking can be exploited to generate test cases.

Our interest in search-based testing, symbolic execution and bounded model checking is motivated by contrasting reasons. On one hand, the heuristics used in search-based testing are mostly based on dynamic program analysis, which generally results in more lightweight approaches than by using static analysis as in symbolic execution and bounded model checking. Nonetheless, the strategies based on random sampling (characterizing search-based testing) notoriously have limited effectiveness when pursuing test objectives that may depend on singular or quasi-singular inputs. In general, these *hard-to-randomly-hit* test objectives often exist in the target programs, making developers of safety-critical software often hesitant to accept the limitations of search-based testing tout court.

On the other hand, symbolic execution and bounded model checking systematically address all test objectives, aiming not to miss any relevant test objective. Our interest in these techniques is further motivated based on the observations that Scade bans some *hard-to-statically-analyze* linguistic features, e.g., by forbidding programmers from allocating memory dynamically, by statically bounding the maximum number of iterations of loops, and by avoiding recursion. As we already commented, embracing these types of restrictions is common in programming languages and coding standards used for developing safety-critical software [3], [31], based on the (empirically motivated) ground that banning them rules out entire classes of subtle failures or risky behaviors, e.g., unbounded consumption of time or space resources at runtime. As a result, we envision the opportunity that symbolic execution and bounded model checking can be more effective for statically analyzing the Scade programs than they are for programs written in other, general purpose programming languages.

Our research method consists in leveraging state-of-the-art test generators that already realize the above test generation approaches for C programs, after transforming the target Scade programs into semantically equivalent C programs. To this end, we rely on the KCG compiler, a cross-compilation facility that is part of the Scade Suite development environment, which indeed allows for rendering a Scade program under test as a semantically equivalent C program. In detail, we designed a toolchain called TECS (Test Engine for Critical software in Scade) that (i) uses KCG to render a Scade program as a C program, (ii) exploits a given test generator to obtain suitable sets of test inputs for the C program, and (iii) recasts the test inputs as test scripts for the original Scade program under test.

TECS is specifically engineered with APIs that allow its integration with several distinct test generators for C programs, by simply providing an adapter for each new test generator. We integrated TECS with the test generators AFL [18], which implements fuzz testing in search-based fashion, KLEE [24], which is based on symbolic execution, and CBMC [30], which is based on bounded model checking. We presented an initial version of TECS integrated with KLEE in [32], focusing on the usefulness of symbolic execution for testing safety-critical software developed in Scade. This paper extends our previous work by integrating TECS with CBMC and AFL, and reporting on experiments investigating the mutual strengths of search-based and systematic test generation techniques.

Our experiments encompassed a benchmark of 37 Scade programs belonging to the on-board signaling unit for high-speed railway systems mentioned above. For each program, we generated a test suite with each of the three versions of the TECS toolchain, that is, by setting the back-end test generator to AFL, KLEE and CBMC, respectively. We then executed each test suite with Scade Test, the test execution environment which is part of Scade Suite, and measured the amount of observable modified-condition/decision-coverage (O-MC/DC) that each test suite achieves for the corresponding Scade program. We further analyzed to what extent the test suites generated with each approach exercise either the same or distinct O-MC/DC test objectives with respect to the test suites generated with the competing approaches, to unveil possible synergies.

In detail, this paper makes the following contributions:

- We introduce an original toolchain that leverages state-of-the-art test generation tools, in order to generate test suites for Scade programs. The toolchain is designed to facilitate the integration of multiple test generation tools.
- We present three instances of our toolchain, integrated with the state-of-the-art test generators AFL, KLEE and CBMC, respectively. We refer to these instances of the tool chain to investigate the effectiveness of search-based testing (AFL), symbolic execution (KLEE) and bounded model checking (CBMC) in the case of Scade programs.
- We discuss the results of a set of experiments with 37 Scade programs, providing empirical evidence of the mutual strengths of the three considered test generation approaches.

This paper is organized as follows. Section II describes the design of TECS, and its instances based on AFL, KLEE and CBMC. Section III reports the results of our experiments with automatically generating test cases for 37 Scade programs. Section IV surveys the related work. Section V frames our conclusions and research directions.

II. THE TECS TOOLCHAIN

This section describes our toolchain TECS aimed at generating test cases for Scade programs. Figure 1 illustrates the components and the workflow of TECS: the input is a Scade program developed with the Scade Suite development

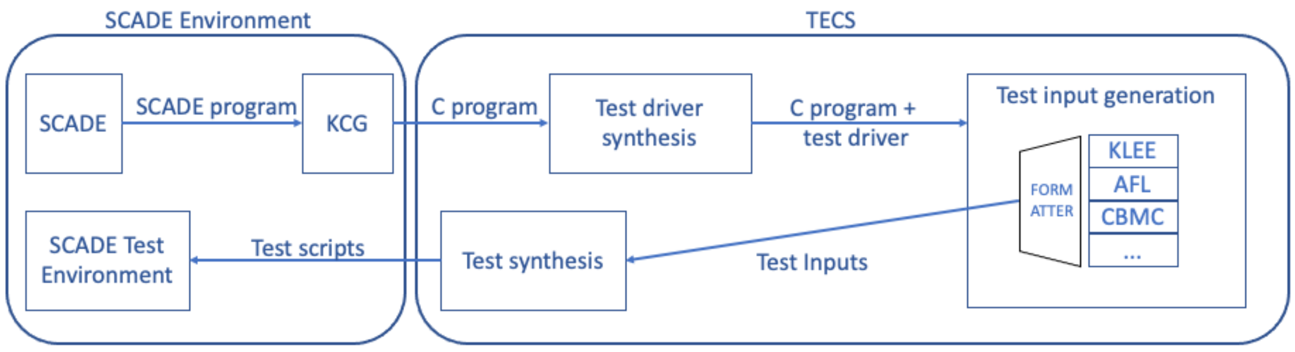

Fig. 1. Components and workflow of TECS

environment (top left part of the figure), and the output is a test suite that can be executed with Scade Test, the test execution environment of Scade Suite (bottom left part of the figure). Below we describe the other components that comprise TECS and its workflow. Since we target unit-level testing, here on in this paper we use the term *Scade program* to generally refer to the Scade component under test, which can be itself part of a larger Scade program.

A. KCG

TECS relies on the KCG compiler, a cross-compilation utility that is part of Scade Suite, to convert the Scade program under test into a semantically equivalent program in the C programming language (Figure 1, *C program*). The C program encodes the execution cycle semantics of the corresponding Scade program: Given current values for the inputs, the outputs and the state, it computes the new values of the outputs and the next state at the end of the execution cycle.

The C programs generated by KCG are provably equivalent to the Scade programs of which they are a translation. In particular, to comply with the semantics of Scade, KCG produces C programs that are deterministic, deadlock-free, and run in bounded space and time[3]. To this end, KCG generates C code that dismisses some constructs out of the expressive power of the C language, specifically such that:

- has unambiguous and precise semantics (i.e., no undefined behaviors);
- conforms to the MISRA C 2012 coding standard rules;
- there is no use of heap memory or of variable-length arrays, all the memory is allocated either globally or automatically on the stack;
- there are no recursive function calls;
- all loops are statically bounded, i.e., their number of iterations is determined by constant values known at code generation time;
- no expression on the right-hand side of assignments or on the guard position of conditional and iterative statements has side effects, e.g., pre- and post-increment operators are not used;
- there is no dynamic address calculation (*pointer arithmetic* expressions), no variable aliasing (including the fact that arrays are always accessed by their declaration names via the array subscript operator), no pathological use of the array subscript operator;
- all array accesses are bounded within the respective array index ranges: the values of the access indices depend only on values known at code generation time.

[3]This also ensures compliance with the most demanding safety levels of certification standards (as, e.g., DO-178C [1], IEC 61508 [33], EN 50128 [2], and ISO 26262 [34]), which require that the program always runs in bounded space and time.

B. Test Driver Synthesis

The C programs generated with KCG cannot be exploited *as-they-are* for the sake of generating proper test cases for the corresponding Scade programs. In fact, a KCG-generated C program implements a single execution cycle of the Scade program from which it was obtained. To exercise the relevant behaviors of a Scade component, we aim at generating test cases that run suitable sequences of execution cycles of the component. A proper Scade test case shall start from an initial state in which all outputs and the state of the Scade component under test are set to default values, and then progress by running multiple execution cycles of the component, setting suitable inputs at each cycle.

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

With reference to Figure 1, the task of generating the code of the test driver is carried out in the *Test driver synthesis* step of the tool chain. This step results in a C program inclusive of the test driver, which can be exploited with test generators to explore the possible sequences of execution cycles of the program under test.

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

When executed, the test driver proceeds as follows. It relies on the init function generated by KCG (Figure 2, line 4) to initialize the values in the output data structure. This structure includes a field for each program output as well the field *out.state* that represents the current program state. Then, the test driver iterates through the loop at lines 7– 12, where it executes the function *program* multiple times (line 10). Each execution of *program*, that is, each execution cycle of the Scade program under test, receives dedicated input values that the test driver sets by calling function *takeInputs* (line 9). The loop iterates as long as the value of field *out.state* corresponds to a program state not yet visited at a previous iteration. This allows for exercising sequences of execution cycles in the scope of the single-state-path-coverage (SSPC) testing criterion, i.e., execution sequences that traverse at most once the states of the state machine that comprises the Scade program under test.

The call to function *takeInputs* (Figure 2, line 9) encapsulates the logic for the test driver to receive new input values at each execution cycle. As explained in the next section, it also allows for generators to exploit the test driver for controlling the test generation process. Figure 2 further specifies the logic of function *takeInputs* at lines 14–17: It first enumerates all fields at any nesting level of the input data structure (function *enumerateFields*, line 15), allocating fresh memory to all pointer-typed fields and returning the references to all leaf, non-pointer fields, and then assigns a new value to each leaf field separately (function *provideInputs*, line 16).

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

Technically, the TECS generates the code of function *enumerateFields* (Figure 2, line 15) by relying on ANTLR4 [35] to parse the type definitions of all fields of the input data

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

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

Fig. 2. Algorithm of the analysis driver

structure, as given in the C program generated by KCG. Each leaf field is then represented as a structure (denoted as *LeafField* at line 15) that includes an identifier label for the field, and a pointer to the memory allocated for containing the value of the field. By naming convention, the identifier label of the leaf fields includes i) the name of the field, ii) the primitive type of the field and iii) the number of the execution cycle in which they will be used. For example, the identifier `in::a::b_int_1` would represent the int-typed field b of the sub-structure a within the input structure in, as assigned at the first execution cycle.

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

C. Test Input Generation

Our tool chain TECS leverages state-of-the-art test generators for C programs (Figure 1, step *Test input generation*), in order to produce test inputs for exercising the Scade programs under test. This step consists in executing the given test generator on the C program that contains the test driver, as follows:

(i) we provide a test-generator-specific implementation of the API *provideInputs* called by the test driver (Figure 2, line 16),
(ii) we compile the C program along with the test driver, the provided implementation of *provideInputs* and a main function that calls the test driver,
(iii) we execute the test generator on the program, and let the test generator generate test data for the program.

(iv) we post-process the test data (that each given test generator produces in its specific output format) to render them in a common format (Figure 1, *formatter*).

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

1) Integrating TECS *with* KLEE*:* KLEE generates test cases based on symbolic execution. Working as a symbolic executor, it models the input values as unconstrained symbols, interprets the statements in the program in function of the input symbols, and computes the execution conditions of the program paths as logic constraints over the input symbols. Finally, KLEE solves the execution conditions of the analyzed program paths with a SMT solver (e.g., STP [36] or Z3 [37]) to obtain concrete inputs that make those program paths execute.

In our setting, KLEE executes the intermediate binary code of the program compiled with LLVM.[4]

To make TECS work with KLEE, we link the program to an implementation of the API *provideInputs* that assigns the relevant inputs with symbolic values by using the primitive *klee_make_symbolic* provided from KLEE. Specifically, the implementation of *provideInputs* calls *klee_make_symbolic* for all primitive inputs enumerated in test driver at each execution cycle (Figure 2, line 15) as follows:

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

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

In this way, running KLEE at step *Test input generation*, we obtain test inputs for the program paths that traverse the program under test through the test driver, i.e., the program paths visited when calling *program* (Figure 2, line 10) multiple times in the while loop of the test driver.

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

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

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

CBMC requires some special care in the way we can associate the non-deterministic inputs with corresponding identifier labels, which is needed for being able to interpret the results from the test generator. As we already explained, this task is carried out in the test driver in the code of function *enumerateFields* (Figure 2, line 15) by producing a string that concatenates the data about the field name, its type, and the number of the current execution cycle. However, since CBMC does not interpret the string operators in its formulas, to work with CBMC, we cannot use string concatenation in the code of the test driver. Conversely, we must produce the code of function *enumerateFields* such that it looks up the identifier labels from a statically unfolded list. This result in the additional requirement of statically specifying the maximum number of execution cycles to be handled. This is arguably a limitation that we incur with CBMC, but not with the other approaches considered in this paper.

3) Integrating TECS *with* AFL*:* AFL works according to search-based testing, and is very popular for security vulnerability testing. It starts by performing random mutations on a set of possible inputs provided by developers, and then progresses in search-based fashion, by considering the newly generated inputs that increase code coverage as candidates for additional mutation attempts.

In our setting, AFL executes the compiled executable of the program, passing input values to its main via the standard input. To make AFL work with TECS, we link the program to an implementation of the API *provideInputs* that simply maps the inputs that AFL passes to the program, to the inputs assigned at each execution cycle of the program under test. This allows AFL to explore the execution space of the program under test, by incrementally executing the program with randomly mutated inputs.

D. Test Synthesis

The last step of the tool chain (Figure 1, *Test synthesis*) renders each generated test input as a test script for Scade Test. This step exploits the identifier labels that the test driver associated with the test inputs, in order to map each input value with specific input fields, correct types and proper execution cycles in the test scripts.

We illustrate this process by considering the example in Figure 3. In the top part of the figure, the first two columns report a set of test data (column *value*) and the associated identifier labels (column *label*) that TECS could produce for a possible program under test at step *Test input generation*. The next three columns show how, at step *Test synthesis*, TECS decomposes each label as the name of an input field (column *field*), its type (column *type*) and the execution cycle (column *cycle*) in which that field must be set to the given value. Being all generated test data related to leaf fields with primitive type, the type of a value can be either a standard data type (e.g,

[4]https://llvm.org

value	label	name	type	cycle
true	in::a_bool_1	a	boolean	1
10	in::b::x_int_1	b.x	integer	1
2	in::b::y_ESome_1	b.y	enum ESome	1
false	in::a_bool_2	a	boolean	2
11	in::b::x_int_2	b.x	integer	2
0	in::b::y_ESome_2	b.y	enum ESome	2

(a) Test data generated at step *Test input generation*

```
#Test cycle 1
SSM::set a true
SSM::set b.x 10
SSM::set b.y VAL2

SSM::cycle

#Test cycle 2
SSM::set a false
SSM::set b.x 11
SSM::set b.y VAL0

SSM::cycle
```

(b) Test script synthesized at step *Test synthesis*

Fig. 3. Synthesis of a test script

boolean, integer, ...) or an enumerative data type defined in the program under test. For example, the last row indicates that the test generator provided the value 0 to be set, at the second execution cycle, for the input named `b.y` (that is, the field `y` of the data structure `b` that is part of the program inputs). The type of the field is the enumerative `ESome`, meaning that the value 0 must be reconciled with first label that the program under test defines for that enumerative type.

The bottom part of Figure 3 shows the final format of the test script, in compliance with the syntax required in Scade Test. Each input is assigned with a SSM::set test statement, by specifying the name of the input in the Scade program and the corresponding value. Each SSM::cycle statement issues the Scade Test executor to run an execution cycle. For example the last row of the above table maps to the last line of the test script, which prepares the inputs before running the Scade program at the second execution cycle. It sets the input `b.y` to the value `VAL0` (for this example we are assuming that the enumerative type `ESome` defined in the program specifies the sequence of values `VAL0`, `VAL1` and `VAL2`). Note that using the labels (not the values) defined in the enumerative types is mandatory in Scade Test.

III. Experimental Assessment

We used the toolchain Tecs to empirically assess the absolute and relative effectiveness of search-based testing (via Afl), symbolic execution (via Klee) and bounded model checking (via Cbmc) when applied to generate test cases for Scade programs. This section reports on the subject programs that we considered in our experiment, outlines the main research questions that drove the experiments, describes the experimental settings, and presents and discusses the results that we obtained.

A. Subject programs

We performed our experimental assessment on a benchmark of 37 Scade programs, belonging to the on-board signaling unit for high-speed railway systems developed at RFI that we already mentioned in the introduction. This system must comply with the ERTMS/ETCS standard, the European standard aimed at harmonizing the management, control and safety of the European high-speed railway traffic, prescribing how trains, track-side devices (e.g., transponders and radio units) and control stations must interoperate to ensure safety objectives like train separation, speed control and automatic protection upon adverse events. The standard defines functional safety in terms of a set of procedures that suitable ensembles of ERTMS/ETCS subsystems must perform in reaction to specific events and conditions that can be signaled to them during railway operations.

The Scade programs in our benchmark describe how the onboard train computer must perform a set of the necessary procedures. The programs are comprised of combinations of dataflow and state machine models, with a number of states ranging from 1 to 5, a number of transitions ranging from 1 to 10, a number of inputs ranging from 1 to 14, and a number of outputs ranging from 1 to 19. These programs were defined by using the Scade Suite environment version 2020 R2. The corresponding C programs, translated to ISO C18 with the Kcg compiler that is part of the scadetools Suite, have a size ranging between about 30 and about 1000 LOCs, excluding the data types declarations that have a size of about 8000 LOC in each program.

B. Research Questions and Metrics

Our experimental assessment aimed at answering the following research questions:

- *RQ1:* How do the different configurations of Tecs, based on search-based testing (Afl), symbolic execution (Klee) and bounded model checking (Cbmc), respectively, compare with each other?
- *RQ2:* To what extent is the approach proposed in this paper effective for generating test cases for safety-critical Scade programs?

RQ1 aims at quantifying the relative efficiency and effectiveness of using Tecs with each backend test generator, to assess the mutual strengths of the three considered test generation approaches. RQ2 addresses whether the Tecs approach is generally effective for the testing goals of the considered class of safety-critical programs.

To answer these RQs, we considered two performance measures, i.e., the O-MC/DC model coverage that the generated test suites achieve for the Scade programs, as measured with Scade Test, and the execution time of the corresponding Tecs runs. The former metric is directly related to the certification objectives required at the highest integrity level of the safety standards. The latter metric quantifies the efficiency of TECS to obtain the corresponding degree of coverage.

We recall that O-MC/DC is the criterion that Scade advises for software that shall work with high integrity level. Scade Test measures O-MC/DC by considering (i) a test objective for each dataflow connection and block in the program, and (ii) two test objectives for each boolean variable and condition, being that boolean variable/condition true and false in either test objective, respectively, with the requirement that the test execution shall result in a different value of some observable datum in either case, independently from any other inputs [16].

C. Experimental Setting

We ran TECS on a virtual Ubuntu machine with 48 VCPUs and 150 GB of memory. We integrated TECS with version afl-fuzz++4.01a of AFL, version 2.3-pre of KLEE and version 5.6 of CBMC: Hereon in this section, we refer to the configurations of TECS integrated with the backend test generators AFL, KLEE or CBMC as TECSA, TECSK and TECSC, respectively.

We executed all generated test suites and measured the O-MC/DC model coverage with Scade Test version 2020 R2.

We configured all backend test generators to address the code coverage of the C programs according to the criterion that, among the ones that they support, can work best to address the O-MC/DC model coverage of the Scade programs under test. TECSA exploits the feature of AFL that, upon identifying test inputs that execute yet-uncovered branches, saves those test inputs in the output folder. TECSK runs KLEE with the option `--only-output-states-covering-new` that makes KLEE output only the test inputs covering new statements. TECSC runs CBMC with the option `--cover mcdc` that makes CBMC address MC/DC coverage on the target C programs. We purposely aimed at the strongest coverage criterion addressable with each tool, though acknowledging that the three backend generators may rely on criteria of different strengths.[5]

We established a maximum time budget of 5 hours for all runs of TECS, although we remark that TECSA handles this budget differently from TECSK and TECSC. This because TECSK and TECSC perform a path-based state space exploration, which can terminate earlier than the maximum time budget, if the test generator completes the analysis of all program paths in the scope of the SSPC testing criterion addressed in the TECS's test driver. On converse, TECSA performs a random exploration of the possible inputs, which always continues until exhausting the maximum time budget in the attempt to further improve code coverage. For this reason, in the experiments, we recorded the actual execution time of TECSK and TECSC, but not of TECSA for which we considered a uniform execution time of 5 hours. Furthermore, we repeated the execution of every TECSA experiment three times to account for the randomness of the AFL algorithm. In the paper, we report the results for the best test suite that TECSA computed out of the three runs, that is, the one that achieved the best O-MC/DC coverage score.

[5]KLEE and AFL could be naively used to generate test cases for all symbolically analyzed program paths (KLEE) and all randomly attempted test inputs (AFL). Arguably this would result in test suites of size both unmanageable for Scade Test and questionable from the standpoint of practical use.

TABLE I
TECS: EXECUTION TIME AND NUMBER OF GENERATED TEST CASES

Subject	**Time(s)**		**# Test cases**		
	TECSK	TECSC	TECSK	TECSC	TECSA
shunting	18000	11400	21	56	6
dc_1	2	23	8	13	16
dc_2	<1	12	2	1	1
dc_3	<1	15	6	4	5
dc_4	1	14	2	4	2
dc_5	<1	13	2	3	1
dc_6	<1	16	2	5	3
dc_7	<1	11	2	2	1
dc_8	<1	12	3	2	1
dc_9	2	16	9	4	6
dc_10	1	15	9	5	6
dc_11	<1	11	2	2	1
dc_12	<1	23	3	12	3
dc_13	<1	16	4	4	4
dc_14	<1	14	2	4	1
radiohole	117	358	6	21	4
crossnonlx	647	1618	13	54	6
baliseinfo	1	51	2	10	2
emergency_1	15	146	14	25	2
emergency_2	29	747	6	32	6
mema	23	101	7	24	5
trackside	1137	6960	3	10	5
vbc	164	2069	12	18	3
coordfromrbc	41	74	5	26	7
adfactordmi_1	1860	278	3	2	2
adfactordmi_2	1	34	2	25	3
driveridins	5	60	10	9	2
eirene	3	67	3	14	3
ertmslevel	2	48	3	9	4
natvalues	1230	908	4	7	1
networkidins	1	48	3	4	2
rbcidins	3	52	4	10	2
trainDataUpdate	47	68	1	27	4
trainDataInsertion	28	78	3	39	6
message129	99	774	10	7	8
runnumber_1	2	54	3	9	2
runnumber_2	3	73	3	1	7

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

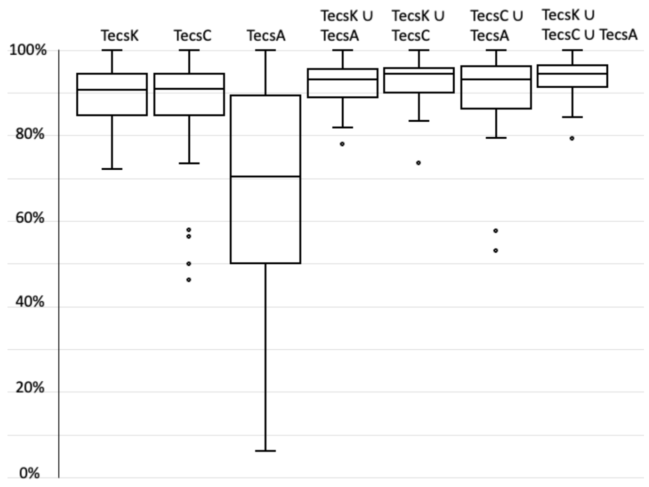

Fig. 4. TECS: Boxplot summaries of the O-MC/DC coverage rates

D. Results

Results for RQ1: Table I reports the execution time (columns *Time*) and the number of generated test cases (columns *Test cases*) for each considered Scade program (column *Program*). The table reports the execution time for TECSK and TECSC, while we omit the execution time of TECSA that is always equal to the maximum time budget of 5 hours. We mark with shadowed background the time data in which either TECSK exhausted the time budget before completing the symbolic analysis of all program paths, or TECSC required us to set the maximum loop unrolling depth of CBMC to 1000 for it to work within the time budget. Without considering these shadowed cases, the data indicate that TECSK is faster than TECSC on all subjects but `adfactordmi_1`. TECSK completed the test generation process in less than a minute for 29 programs, and took more than 10 minutes for 5 programs including the only timeout case. TECSC completed in less than a minute for 20 programs, and in more than 10 minutes for 7 programs including 5 of the 6 cases shadowed in the table. All TECS versions resulted in test suites of manageable size: TECSK, TECSC and TECSA generated test suites ranging between 1 and 21 test cases, between 1 and 56 test cases, and between 1 and 16 test cases, respectively.

Table II reports the O-MC/DC coverage obtained with the test suites generated with TECSK, TECSC and TECSA, respectively (second to fourth columns) and the coverage that we obtained by merging the test suites from different TECS versions (fifth to eighth columns). For example, the column titled TECSK ∪ TECSC reports the coverage of the test suite obtained by merging the test suites that either TECSK or TECSC generated for a program. The rightmost column reports the coverage obtained with the test suites from all three TECS versions. We recall that, in the case of TECSA, the data refer to the test suites with higher coverage out of three runs.

Figure 4 summarizes the main statistics of the coverage data in the columns of Table II with boxplots, highlighting the minimum, the median, the maximum, the (first and third) quartiles, and the outliers of the respective distributions.

Table II (columns TECSK, TECSC and TECSA) highlights in bold typeface the data of the tools that reached the highest coverage for each program, and with underlining the cases in which a single tool achieved strictly higher coverage than all others. TECSK, TECSC and TECSA achieved the highest coverage for 26, 24 and 8 programs (out of 37) respectively. They achieved strictly higher than any other tool for 11, 10 and only 1 program(s), respectively. The first three boxplots in Figure 4 further visualize that TECSK and TECSC achieved comparable coverage performance, with TECSC exhibiting a slightly worse dispersion than TECSK, while TECSA is sensibly the worse.

We further discuss the experiments in which the tools achieved notably low coverage (below 60%). TECSK scored more than 60% for all subject programs. TECSC scored less than 60% coverage for the programs `crossnonlx` and `vbc`, mostly due the limit on the loop unwinding depth that was needed in these cases, and for programs `message129` `dc_8`, for which the inspection of the data revealed that CBMC was not able to provide suitable solutions for some reachability formulas. TECSA scored less than 60% coverage for 12 programs: In these programs, most untested code is dominated by program branches that can be hit only by singular, or low-cardinality, set of inputs, hard to elicit by random mutations.

Despite the considerations on each specific approach, the data in Table II overall indicate that none of the TECS versions achieved the highest or the worst coverage consistently. Moreover, we remark that even when a given tool achieved higher coverage than another tool, we cannot conclude that it necessarily subsumed all test objectives covered with the latter one. This motivated us to explore their complementary.

We further investigated the mutual strengths of the considered approaches by measuring the coverage achieved with the merged test suites, as reported in the four rightmost columns of Table II. In fact, given two tools (e.g., TECSK and TECSA) the difference between the coverage scores of a tool (e.g., TECSK or TECSA), and the corresponding scores of the merged test suites (TECSK ∪ TECSA) reveals the coverage portions that were uniquely contributed by the other tool (i.e., TECSA or TECSK, respectively). Moreover, a merged test suite with strictly greater coverage than any contained test suites reveals that each of those test suites hit unique test objectives. In the four rightmost columns of Table II, we highlighted these cases in bold typeface, and further marked with shadowed background the maximum coverage. In the rightmost column we highlighted the cases in which we achieved the maximum coverage only when merging the test suites from all three tools.

The experimental data indicate that the test suites TECSK ∪ TECSA, TECSK ∪ TECSC and TECSC ∪ TECSA improved over both respective single tools for many programs, namely, for 14, 15 and 13 programs, respectively. The test suite merged from all three tools was the strongest only for 2 programs. The four rightmost boxplots in Figure 4 confirm that the merged test suites, and in particular the three-tool one, achieved high

TABLE II
TECS: O-MC/DC COVERAGE RATES

Program	TECSK	TECSC	TECSA	TECSK ∪ TECSA	TECSK ∪ TECSC	TECSC ∪ TECSA	TECSK ∪ TECSC ∪ TECSA
shunting	**86.6%**	86.3%	45.3%	86.6%	**92.1%**	86.3%	92.1%
dc_1	90.6%	**93.8%**	89.4%	**96.9%**	**95.0%**	**99.4%**	99.4%
dc_2	**100.0%**	80.0%	**100.0%**	100.0%	100.0%	100.0%	100.0%
dc_3	**100.0%**	**100.0%**	**100.0%**	100.0%	100.0%	100.0%	100.0%
dc_4	92.4%	**95.5%**	86.4%	**95.5%**	95.5%	**98.5%**	98.5%
dc_5	89.3%	**96.4%**	89.3%	89.3%	96.4%	96.4%	96.4%
dc_6	**89.7%**	88.2%	83.8%	**91.2%**	**95.6%**	88.2%	95.6%
dc_7	80.0%	**85.0%**	50.0%	**85.0%**	**90.0%**	85.0%	90.0%
dc_8	**83.3%**	50.0%	41.7%	**91.7%**	83.3%	**83.3%**	91.7%
dc_9	**100.0%**	**100.0%**	**100.0%**	100.0%	100.0%	100.0%	100.0%
dc_10	**93.0%**	**93.0%**	**93.0%**	93.0%	93.0%	93.0%	93.0%
dc_11	**100.0%**	**100.0%**	**100.0%**	100.0%	100.0%	100.0%	100.0%
dc_12	72.1%	**73.5%**	60.3%	**77.9%**	73.5%	**79.4%**	79.4%
dc_13	**98.2%**	85.7%	82.1%	98.2%	**100.0%**	**96.4%**	100.0%
dc_14	81.8%	**90.9%**	63.6%	81.8%	90.9%	90.9%	90.9%
radiohole	**94.9%**	88.4%	68.2%	**95.5%**	**97.5%**	88.4%	97.5%
crossnonlx	**84.5%**	46.3%	19.1%	**86.2%**	**86.8%**	**53.0%**	**87.0%**
baliseinfo	**95.8%**	**95.8%**	44.8%	95.8%	95.8%	95.8%	95.8%
emergency_1	**93.4%**	87.4%	6.2%	**93.6%**	**94.5%**	**87.9%**	94.5%
emergency_2	82.1%	**92.3%**	54.5%	**82.9%**	**92.7%**	**92.5%**	**92.9%**
mema	**88.1%**	82.6%	42.8%	**91.1%**	**93.2%**	82.6%	93.2%
trackside	**98.6%**	**98.6%**	19.6%	98.6%	98.6%	98.6%	98.6%
vbc	**93.7%**	56.3%	36.9%	**94.0%**	**94.8%**	**57.6%**	94.8%
coordfromrbc	**82.6%**	80.1%	56.5%	82.6%	**86.2%**	**80.8%**	86.2%
adfactordmi_1	**84.6%**	**84.6%**	71.2%	84.6%	84.6%	84.6%	84.6%
adfactordmi_2	**96.2%**	**96.2%**	**96.2%**	96.2%	96.2%	96.2%	96.2%
driveridins	**89.2%**	**89.2%**	65.1%	89.2%	89.2%	89.2%	89.2%
eirene	93.7%	**95.8%**	66.3%	93.7%	95.8%	95.8%	95.8%
ertmslevel	**94.4%**	**94.4%**	87.5%	94.4%	94.4%	94.4%	94.4%
natvalues	**90.0%**	**90.0%**	**90.0%**	90.0%	90.0%	90.0%	90.0%
networkidins	**94.4%**	**94.4%**	75.0%	94.4%	94.4%	94.4%	94.4%
rbcidins	**94.5%**	**94.5%**	49.1%	94.5%	94.5%	94.5%	94.5%
trainDataUpdate	89.0%	**94.0%**	60.0%	89.0%	**95.0%**	94.0%	95.0%
trainDataInsertion	88.0%	**90.9%**	88.5%	**90.9%**	90.9%	**91.4%**	91.4%
message129	**83.5%**	57.9%	77.4%	**84.2%**	**84.2%**	**79.3%**	84.2%
runnumber_1	**93.8%**	**93.8%**	70.4%	93.8%	93.8%	93.8%	93.8%
runnumber_2	84.2%	83.2%	**92.1%**	92.1%	**86.1%**	**93.1%**	93.1%

coverage significantly more consistently than the single-tool counterparts.

In summary, related to the research question *RQ1*, our experiments highlight that the test suites that TECS generated by using KLEE and CBMC, and in particular the combinations of those test suites, achieved the highest coverage for much more subject programs than when using AFL. This confirms that systematic test generation approaches can effectively address automated test generation for Scade programs, while a search-based approach does not appear per-se to offer comparable advantages. At the same time, the data acknowledge that AFL allowed for further improving the coverage rates for more than one fifth of the subject programs, advising the combination of all three approaches as the best choice overall. The paired Student's t-test supports the hypothesis that the distribution of the coverage data in the rightmost column of Table II has significantly greater mean than the distributions in the other columns of the table ($p < 0.02$ in all cases).

Results for RQ2: The research question *RQ2* concerned the effectiveness our approach for generating unit-level test cases for safety-critical Scade programs. We assume that testers shall aim at 100% O-MC/DC coverage since, for safety-critical software that must work at the highest integrity levels, many certification standards (i) indicate MC/DC testing as reference criterion, (ii) require thorough code coverage, (iii) require documented justifications for the uncovered code [1], [2].

According to the highest coverage data that we achieved for each subject program (Table II, rightmost column), TECS as a whole resulted in 90% or more O-MC/DC coverage for 31 programs, including 5 programs in which the coverage rate was exactly 100% and other 10 programs in which it was at least 95%. We observe that coverage rates in the range 90%–100% crucially mitigate the manual effort that can be required from testers,[6] and thus we regard to the results of these 31 experiments as evidences of the effectiveness of our approach.

Nonetheless, on 6 subject programs TECS scored less than 90% coverage. We inspected the uncovered items in these programs, to better understand the reasons why TECS missed these items. We could spot a few infeasible items, typically due to functions called with parameters that specialized their behavior, but we mapped most missed items to the testing

[6]Notice that testers can straightforwardly figure out the uncovered O-MC/DC items while executing the generated test suite in Scade Test, thanks to the information provided by it after the tests execution.

criterion that characterizes the TECS test driver: We do not generate test cases that traverse the same states more than once, whereas those missed items would require more iterations through the execution cycles.

In summary, related to the research question *RQ2*, in our empirical evaluation TECS demonstrated its potential to effectively address unit-level testing of safety-critical Scade programs in 31 out of 37 experiments.

E. Threats to Validity

Internal validity concerns whether our conclusions may be wrong due to methodological errors. A possible issue is that we assessed the strengths of search-based testing, symbolic execution and bounded model checking by integrating TECS with one single test generator for each approach. We mitigated this issue by selecting state-of-the-art test generators that have acknowledged reputation in the scientific communities of each reference approach. We aim to integrate TECS with additional test generators in the future. Furthermore, our findings can be affected by the arbitrary choices of limiting the maximum time budged of each experiment to 5 hours and, in the case of CBMC, of setting the loop unrolling depth to 1000 in the experiments in which CBMC ran out of memory otherwise.

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

External validity concerns the extent to which our results can generalize to Scade programs other than the ones that we considered in the experiments. In this respect, the main issue is that all our subject programs belong to the same project. We could not mitigate this threat in the current experiments, and we aim to collect further experimental data as future work.

IV. RELATED WORK

Our work contributes to the body of knowledge on automatically generating test cases with symbolic execution, bounded model checking and search-based testing. We leveraged on existing techniques to provide empirical evidence of the relative strengths of these approaches to generate test cases for an industrially relevant class of safety-critical programs. Several surveys provide comprehensive reports on the many applications of these baseline approaches for testing tasks [38]–[41].

Scade derives from the synchronous, dataflow programming languages LUSTRE [4] and ESTEREL [5], with further constructs from the graphical, state-machine-based language SyncCharts [42]. Thus our work has relations either with other research that addresses testing automation for these languages [13]–[15], [43], [44], or more in general with model-based testing [45], [46].

Lakehal and Parissis investigated a set of coverage criteria for LUSTRE, in the spirit of data-flow-based testing criteria, and they presented a tool for measuring code coverage accordingly [43]. Other authors defined a set of mutation operators and a corresponding mutation analysis tool for the LUSTRE programming language [44]. Lutess [13] generates test cases at random based on a description of the environment. Lurette [14] and Gatel [15] focused on generating test cases for invariants or safety properties described in Lustre. These approaches could be extended for programs in Scade, but none of them deals with automatically generating test cases for achieving high structural coverage as our approach.

Model-based testing consists in deriving test data by analyzing either program specifications or program behaviors expressed as models, e.g., with class diagrams, state machines or sequence diagrams [47], [48]. Indeed Scade is a model-based programming language that exploits state-machines and data flow models for defining software behaviors [11], [12].

Our approach TECS is particularly related with the work on Polyglot and SAUML, which rely on symbolic execution to generate test cases for state machine models, either statecharts or UML-RT state machines, respectively [49]–[51]. Polyglot is similar to TECS in that it leverages an existing symbolic executor (SPF [52]) after translating the statecharts to Java programs. Indeed the work on Polyglot mostly focuses on how to translate the target statecharts to semantically equivalent Java programs. SAUML defines an ad-hoc symbolic executor that directly analyzes the UML-RT models. Compared to these approaches, TECS distinctively investigates multiple test generation approaches, whereas Polyglot and SAUML focus only on symbolic execution.

A related tool is RT-Tester, which is used for verification and validation activities for automotive, railway and avionic systems, and more recently has been extended with a test generation technique [53]–[55]. It works similarly to bounded model checking, representing the execution semantics with propositional logic, and solving propositional formulas that capture test cases built according to a given testing strategy. We found very limited experimental data on the effectiveness of this approach in the available papers. Since RT-Tester does not directly support Scade, we were not able to compare it with TECS in our experiments. Nonetheless, the results that we achieved with CBMC can be representative to some extent. Duy et al. also exploited model checking in the context of selection of test cases for regression testing of Scade programs [56]. However their approach, based on the model checker LESAR [57], is limited in that they abstract away any numeric computation in the programs.

Model-based testing has been successfully applied to derive test cases and complement verification for functional specifications expressed in formal languages as B, Z or VDM [58]. The structural testing approach of TECS is naturally complementary and could be profitably integrated with test cases generated by exploiting functional model-based testing.

V. Conclusions

This paper presented an approach to automated test generation for Scade programs, embodied by the prototype tool TECS, which can be configured to generate test cases based on either a search-based strategy, symbolic execution or bounded model checking. We provided empirical evidence of the suitability of TECS on a benchmark of 37 Scade programs that belong to an industrial on-board train signaling system, and discussed both the relative effectiveness of the considered test generation strategies and the overall effectiveness of the approach as a whole. In particular, we showed that the systematic test generation strategies of TECS, either based on symbolic execution or on model checking, yielded higher structural coverage than the search-based version of TECS in most cases. Moreover, by considering also the test cases generated with the search-based strategy, we could successfully improve the thoroughness of the test suites in a statistically significant amount of cases. Our research activity is currently focusing on extending TECS to be applied for integration testing.

Acknowledgement

This work is partially supported by the SISMA national research project (MIUR, PRIN 2017, Contract 201752ENYB).

References

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

[2] CENELEC, *EN 50128, Railway Applications – Communication, Signaling and Processing Systems – Software for Railway Control and Protection Systems*, 2020.

[3] L. Hatton, *Safer C: developing software in high-integrity and safety-critical systems*. McGraw-Hill international series in software engineering, McGraw-Hill, 1995.

[4] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[5] G. Berry and G. Gonthier, "The Esterel synchronous programming language: design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

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

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

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

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

[10] M. Gudemann, A. Angerer, F. Ortmeier, and W. Reif, "Modeling of self-adaptive systems with SCADE," in *2007 IEEE International Symposium on Circuits and Systems*, pp. 2922–2925, 2007.

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

[12] J.-L. Camus, "Esterel SCADE approach to MBD," *Digital Avionics Handbook. Taylor & Francis Group*, 2015.

[13] V. Papailiopoulou, "Automatic test generation for lustre/scade programs," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 517–520, 2008.

[14] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber, "Automatic testing of reactive systems," in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pp. 200–209, 1998.

[15] B. Marre and A. Arnould, "Test sequences generation from lustre descriptions: Gatel," in *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, pp. 229–237, 2000.

[16] ANSYS, *SCADE Test User Manual.* Esterel Technologies S.A.S., 2020.

[17] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 438–444, 1984.

[18] "American Fuzzy Lop (AFL).," Accessed January 2022.

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

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

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

[22] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 3, pp. 215–222, 1976.

[23] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[24] D. E. Cristian Cadar, Daniel Dunbar, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," *USENIX Association*, 2008/12/8.

[25] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Network Distributed Security Symposium (NDSS)*, Internet Society, 2008.

[26] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, p. 2, 2012.

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

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

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

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

[31] MISRA C++ Working Group, *MISRA Compliance: Achieving compliance with MISRA Coding Guidelines*, 2020.

[32] E. Kurian, D. Briola, P. Braione, and G. Denaro, "Automatically generating test cases for safety-critical software via symbolic execution," *Journal of Systems and Software*, vol. To appear, 2023.

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

[34] ISO, *ISO 26262: Road vehicles — Functional safety*, 2010.

[35] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd ed., 2013.

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

[37] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the International Conference on Tools and Algorithms*

for Construction and Analysis of Systems, TACAS/ETAPS '08, pp. 337–340, Springer, 2008.

[38] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Communications of the ACM*, vol. 56, pp. 82–90, Feb. 2013.

[39] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, May 2018.

[40] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[41] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys*, vol. 45, no. 1, p. 11, 2012.

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

[43] A. Lakehal and I. Parissis, "Lustructu: a tool for the automatic coverage assessment of LUSTRE programs," in *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pp. 301–310, 2005.

[44] L. V. Phol, N. T. Binh, and I. Parissis, "Mutants generation for testing LUSTRE programs," in *Proceedings of the Eighth International Symposium on Information and Communication Technology*, SoICT 2017, (New York, NY, USA), pp. 425–430, Association for Computing Machinery, 2017.

[45] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model based testing approaches," *Software: Testing, Verification and Reliability*, vol. 22, pp. 297–312, Aug. 2012.

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

[47] M. Utting and B. Legeard, *Practical Model-Based Testing: A tools approach*. Morgan Kaufmann, 2010.

[48] Object Management Group, *OMG® Unified Modeling Language® (OMG UML® (version 2.5.1))*, 2017.

[49] D. Balasubramanian, C. S. Păsăreanu, M. W. Whalen, G. Karsai, and M. Lowry, "Polyglot: Modeling and analysis for multiple statechart formalisms," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, (New York, NY, USA), pp. 45–55, Association for Computing Machinery, 2011.

[50] D. Balasubramanian, C. Păsăreanu, M. W. Whalen, G. Karasi, and M. Lowry, "Improving symbolic execution for statechart formalisms," in *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa '12, (New York, NY, USA), pp. 47–52, Association for Computing Machinery, 2012.

[51] K. Zurowska and J. Dingel, "SAUML: a tool for symbolic analysis of UML-RT models," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 604–607, 2011.

[52] C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: Symbolic execution of Java bytecode," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, (New York, NY, USA), pp. 179–180, Association for Computing Machinery, 2010.

[53] J. Peleska, "Industrial-strength model-based testing-state of the art and current challenges," *arXiv preprint arXiv:1303.1006*, 2013.

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

[55] L. Vu, A. Haxthausen, and J. Peleska, "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 2014* (E. Schnieder and G. Tarnai, eds.), pp. 200–209, Technische Universität Braunschweig, 2014.

[56] T. C. Duy, N. T. Binh, and I. Parissis, "Automatic generation of test cases in regression testing for lustre/scade programs," *Journal of Software Engineering and Applications*, vol. 06, no. 10, p. 27–35, 2013.

[57] N. Halbwachs, *Lustre program verification: the tool Lesar*, pp. 139–147. Boston, MA: Springer US, 1993.

[58] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Computing Surveys*, vol. 41, pp. 18:1–18:41, Sept. 2015.