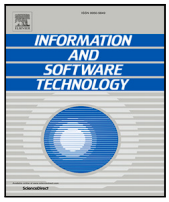
Information and Software Technology 172 (2024) 107467

Contents lists available at ScienceDirect

Information and Software Technology

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



Automatic test cases generation from formal contracts

Samuel Jiménez Gil [a], Manuel I. Capel [b,*], Gabriel Olea Olea [b]

[a] SatixFy Space Systems UK, Trident Unit 2, Styal Road, Manchester, M22 5XB, UK
[b] Department of Software Engineering, ETSIIT, Universidad de Granada, 18071 Granada, Spain

ARTICLE INFO

Keywords:
Automatic test cases generation
Software testing
Formal methods
Software verification

ABSTRACT

Context: Software verification for critical systems is facing an unprecedented cost increase due to the large amount of software packed in multicore platforms generally. A substantial amount of the verification efforts are dedicated to testing. Spark/Ada is a language often employed in safety-critical systems due to its high reliability. Formal contracts are often inserted in Spark's program specification to be used by a static theorem prover that checks whether the specification conforms with the implementation. However, this static analysis has its limitations as certain bugs can only be spotted through software testing.

Objective: The main goal of our work is to use these formal contracts in Spark as input for a test oracle – whose method we describe – to generate test cases. Subsequent objectives consist of a) arguing about the traceability to comply with safety-critical software standards such as DO-178C for civil avionics and b) embracing the best-established software testing methods for these systems.

Method: Our test generation method reads Spark formal contracts and applies Equivalence Class Partitioning with Boundary Analysis as a software testing method generating traceable test cases.

Results: The evaluation, which uses an array of open-source examples of Spark contracts, shows a high level of passed test cases and statement coverage. The results are also compared against a random test generator.

Conclusion: The proposed method is very effective at achieving a high number of passed test cases and coverage. We make the case that the effort to create formal specifications for Spark can be used both for proof and (automatic) testing. Lastly, we noticed that some formal contracts are more suitable than others for our test generation.

1. Introduction

Software verification is a challenging and labor-intensive task for dependable systems that may reach around 60% of the overall software development cost. Thus, the software industry has shown a significant interest in reducing this expense and automation is arguably the most promising solution. Software verification can be carried out either by *code review*, *proof* or *testing* [1].

Code review is normally the least preferred approach as it may involve some manual and sometimes informal justification. On the one hand, software proof applies deductive reasoning through formal specification analysis [1]. These formal specifications usually consist of preconditions to be met by the input data of a program and postconditions for the output one. These verification conditions along with the implementation are used by a *theorem prover* or *verifier* [2] to determine whether postconditions are *always* satisfied.

The latter form induces the concept of "design by contract", which can be understood in simple terms as the mandatory specification of verifiable interfaces for each software component at design time. The definition of these interfaces is done by defining the above-mentioned preconditions, postconditions, and invariants.

The downside of proof is that a design by contract requires significant cost and effort from a business point of view. In addition, the limitations of this approach emerge when it underpins unverified assumptions [1,3] such as the absence of hardware errors, consistency of memory, or even when interfacing with unanalyzable external libraries [4].

Alternatively, software testing is an intrinsic inductive reasoning that consists of collecting observations from real hardware or an equivalent simulator. As a consequence, this test data may be representative of the real behavior of the system when deployed, and run-time bugs or errors such as failure to open a file may become apparent.

Software testing is based on the construction and execution of test cases that are derived from a specification or verifiable requirements. Such test cases consist of a combination of input data, paired with their expected outputs. At the end of a test execution, the actual output is compared with the expected ones to determine whether a test passes

* Corresponding author.
E-mail addresses: Samuel.Gil@satixfy.com (S.J. Gil), manuelcapel@ugr.es (M.I. Capel), gabrieloo@correo.ugr.es (G.O. Olea).

https://doi.org/10.1016/j.infsof.2024.107467
Received 3 May 2023; Received in revised form 2 April 2024; Accepted 3 April 2024
Available online 16 April 2024

or fails. The resulting evidence serves to argue about the confidence in the program's correctness.

A widely accepted method known as Equivalence Class Partitioning (ECP) with Boundary Analysis [5] partitions the input domain into disjoint classes depending on the expected output and then picks values strategically i.e., mainly in the transition of two disjoint classes. The underlying rationale is that these points are more likely to contain bugs [6], e.g., typing $\leq$ instead of $<$ in the implementation of a software component may cause an unforeseen error.

The limitations of software testing become apparent when considering the non-exhaustiveness due to its intrinsic tractability. Consequently, some input–output combinations and paths are untested. Despite the disadvantages, software testing is often deemed as the gold standard for software verification [6].

Automated approaches to generating test input data have gained momentum in recent years. In 2002, Tracey [7] estimated that automating test case generation for jet engine controllers may save between £1 and £1.5 million per project. These savings are, in all likelihood, a lot higher in nowadays embedded systems considering the amount of code that is flashed in chips.

Despite the importance of automatic test generation, only a few works focus on generating true test cases, i.e., providing input data along with expected results in the form of test cases or test suites [3]. Most of the available approaches are code-driven, meaning that source code is read to generate only test data and therefore the requirements or specifications are disregarded. Consequently, these approaches are not acceptable for certain software quality standards, as these tests cannot be traced back to requirements [8–10].

Moreover, the technology implemented in Spark/Ada programming language – perhaps the most popular programming language for safety-critical applications – enables to integrate the formal contracts inside the specification of functions or procedures (also called subprograms in Ada terminology). The most similar state-of-the art work [11] advocated to produce test cases derived from Spark contracts has significant limitations as they do not embrace standard testing methods such as ECP, they have not laid out an explicit method for test cases generation and their approach does not support contracts with First-order Logic relations and quantifiers. The latter are found in modern versions of the Spark programming language.

In this work, we also take the task of generating test cases from formal Spark contracts. More specifically, our contributions are:

- To use standardized ECP supplemented with Parameter Domain Boundary Analysis to generate test cases automatically.
- To propitiate the derivation of test cases from formal contracts that include First-order Logic predicates.
- To provide an empirical study from representative open source benchmarks, the results of which indicate a high success rate in test cases execution and high statement coverage compared to using a random test generator only.

The rest of the paper is organized as follows: Section 2 provides an overview of software testing and formal methods approaches. Section 3 lays out the test generation method followed by simple examples of application, Section 4 explains the results of the experiments and discusses the threats to validity of the conclusions. Section 5 addresses a literature survey. Finally, in Section 6 conclusions are offered followed by future work in Section 7.

2. Background

Software testing is a broad area. A test plan must always have one or more goal properties of the system to evaluate. For instance, *performance testing* [3] is orientated to measure the execution or response time and is often employed in Real-Time Systems to derive the Worst-Case Execution Time of a task. This data is later used to compute the Worst-Case Response Time for a task set. Whilst this testing process was more focused on providing test data to stress paths leading to the largest execution times, more recent approaches to analyze the timing of the multicore processors employ *microbenchmarks* – small low-level programs – which run in parallel with the tasks under analysis [12]. The aim is to collect inflated response or execution times by producing contention on shared hardware resources such as memory buses [12].

In the realm of cybersecurity [13], *penetration testing* aims to determine whether the system allows unauthorized access to restricted data or code. This process is agreed in a consensual manner with the system owner and starts by collecting publicly available information i.e., open source intelligence, and then setting up an attack plan. Next, a vulnerability analysis is carried out to identify weak areas of the system followed by the exploitation phase where malware or pathological input data are fed into it. The process culminates with a post-exploitation stage when an attempt to scale the attack is perpetrated. Lastly, a report of the overall process is filed to improve the security of the overall system [14].

When it comes to requirements-driven testing, probably the most popular testing techniques are: *ECP*, *boundary analysis*, *decision tables* and *combinatorial testing* [6]. ECP is based on splitting the input space of a program into disjoint (and exhaustive) sets depending on the expected output and then choosing test cases located somewhere within and/or at the boundaries of these sets (boundary analysis) [5]. To achieve this, per the Equivalence Class, we must define a corresponding *binary equivalence relation*, this is, a binary relation that is reflexive, symmetric, and transitive. In this way, we ensure we split the domain into disjoint and exhaustive regions. *Decision tables* organize a list of conditions or actions in the first column along with the expected output in subsequent columns. An upside of this approach is the ability to identify software bugs by evaluating impossible combinations of input data [15].

Given that the number of plausible input data combinations is the cartesian product of the range of each input data, *combinational testing* is orientated to handle this exponential explosion when it is not possible to reduce the number of input parameters. This approach carefully picks input data combinations by pairing some parameters to reduce the resulting combinations [15].

Software quality standards provide guidance and objectives to give confidence that the final software product is fit for purpose. They also serve as a legal basis for good practice in the event of an accident or damage. A central software concept for these standards is the Safety Integrity Level (SIL) of a piece of software which takes into account the potential consequences of a software malfunction as well as the likelihood of that event to occur. The ultimate goal is to determine whether the system is acceptably safe.

These levels are named differently depending on the quality standard. The aerospace standard DO-178C [8] is named 'Design Assurance Level' and it ranges from A to E being A the most critical level, 'Automotive Safety Integrity Level' in the automotive standard ISO26262 [9] ranging from A to D being D the most critical level. Finally 'Criticality' is employed in the ECSS (space standards) [10] using a similar convention to aerospace but removing the last E level.

These SILs have important implications in the level of assurance required and therefore the verification goals. In this respect, *structural code coverage* or just *code coverage* lay out some criteria to quantify the extent to which the requirements are implemented in the code using tests. Though some caveat exists across the standards [16], in general, we can enumerate the following criteria: *statement or block coverage* measures whether all blocks of code have been hit at least once, *branch or decision* coverage measures whether all decisions have been reached at least once, and *Modified Condition/Decision Coverage* (MC/DC), which requires that each condition in a decision to be evaluated at least once to true, once to false; and make this assignment to affect the decision's output [11]. Lastly, MC/DC is only applied for testing software up to the highest SIL criticality level.

Test coverage and code coverage are terms often intertwined. Test coverage refers more to checking the degree to which each requirement has been verified by test cases rather than verifying that all the code has been executed. Logically, a single requirement may have been verified by multiple test cases and a few requirements can be verified by a single test, so there is no bijective relationship between requirements and tests.

Despite code coverage being widely used in software quality standards and providing a systematic and scientific method to verify a software implementation, some renowned software testers [17] have prevented their limitation. For example, detecting bugs related to missing code, or identifying code incompatibility with the platform's hardware or software configuration and memory leaks.

Techniques based on automatic generation of input data have become popular in recent years [3] for automated software testing. The three main surveyed approaches are: (a) Random Testing (RT) consists of simply generating random test data; (b) Search-Based Testing integrates meta-heuristics and requires defining an objective or fitness function that evaluates the goodness of the generated solution. In the scope of our problem, the input to the fitness function comes from the test run. Lastly, (c) Constraint-Based Testing, which collects a list of constraints from the Software Under Test (SUT), is later passed to a *constraint solver* to determine whether some test data can be generated or not.

Aside from that, *software proof* relies on a formal specification that applies preconditions (or hypotheses, H_i), for the input data and postconditions (or conclusions, C_j) for the output one. These are also called Verification Conditions (VCs) or contracts [1]. Barnes [18] argues that the role of the automatic prover is to check whether the following general expression is always true:

$$\bigwedge_{i=0}^{k} H_i \implies \bigwedge_{j=0}^{l} C_j \tag{1}$$

Provided that all hypotheses are evaluable and none of them can be falsified. Designing formal contracts can be challenging and ideally, they should have *weak preconditions* to accept a wide array of inputs and *strong postconditions* i.e., accurate, to guarantee no wrong output is produced [2]. Software proof can be applied both to functional programming and imperative programming [2]. However, in safety-critical systems [6] only the imperative approach is used, as they often have to satisfy strict reliability requirements and real-time constraints.

A recent addition to Spark 2014 [1] has included the notion of *contract cases* which is the extrapolation of ECP to software proof.

Lastly, it is worth mentioning that the Spark compiler provides the option for inserting precondition checks before the body is reached at run-time. This *defensive* code can be enabled by including certain compilation flags.

3. Test cases generation method

Our approach hinges on Constraint-Based Testing. Any constraint satisfaction problem will have some input data list (D_n), in our case, this may match the test vector (some global variables may need to be set as well) along with a list of constraints (C_n). Constraint Optimization Problems also require an optimization function but that is not relevant for our problem domain.

For our test generation process, it is assumed that there is a specification file (Listing 1.1 at Section 3.2 could be an example) that holds the information about the target subprogram variables and includes a formal contract.

One of our main contributions is the domain ECP guided by the contract cases found in such specification: contract cases determine an ECP as they are disjoint and exhaustive. Consequently, we hold that *the local precondition (guard) of each contract case found in the specification of the subprogram serves as a binary equivalence relation (it is reflexive, symmetric, and transitive) to apply ECP, as these constraints indeed lead to splitting the input domain as disjoint and exhaustive regions*. If there are no contract cases, then the domain will be considered to have a trivial partition with a single equivalence class. Note that some equivalence classes may have a single or only a few input values.

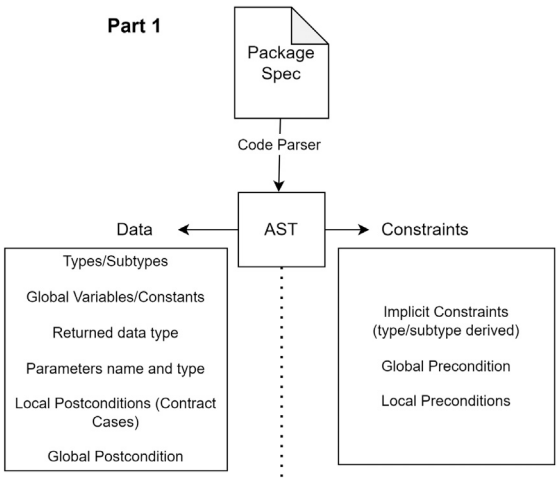


Fig. 1. Data and constraints mapping to generate a constraint satisfaction problem. AST stands for Abstract Syntax Tree.

We would like to remark that the complete process we propose is carried out automatically, from the data/constraints collection to the Constrained Oracle creation. From the specification file, an Abstract Syntax Tree (AST) is derived. Next, our search process traverses the AST to collect the right data for test generation, see Fig. 1, by taking into account the node types of the derived AST.

On the left-hand side, we have depicted the necessary data to document the test vector and the expected output of the test cases. The postconditions are mapped similarly to the pass/fail criterion for the test cases. Another important remark here is that postconditions do not undergo any constraint solving but they are inserted in the resulting unit test. In addition, postconditions can be categorized as *unambiguous* where the exact returned value is known and is equality and *ambiguous* otherwise. The latest case is normally the call to a checker function in the code. In this case, the pass/fail criterion is delegated to a call of the checker function.

On the right-hand side of Fig. 1, we have all constraints data related mostly to preconditions. Based on these definitions, our test generation algorithm selects the test cases using ECP which, if the user wants to, can be supplemented with parameter domain boundary analysis. As it was previously mentioned, this ECP comes directly from the local preconditions of each contract case found in the AST.

For a better understanding of the terms used in Fig. 1, especially to distinguish between local and global preconditions/postconditions, see Fig. 2, where we can observe a possible formal contract for a square root function.

Global preconditions dictate the conditions specified to be met by *all* input data of the subprogram. Similarly, global postconditions impose that the stipulated conditions must be met *all* output data. However, when we refer to local pre/postconditions, these predicates may only consider a subset of the input/output space. Thus, their scope is more local than global. If the conditions are expressed outside of the Contract Cases, they refer to global preconditions/postconditions. Conversely, they allude to local preconditions when reflected on the left part ("guard") of a contract case or local postconditions when appearing on the right part ("consequence") of the contract case.

3.1. Constraint-based test case generation algorithm

Once the previous data and constraints have been automatically gathered from the AST, which correspond to the *Input_Data_list* (D_n) and

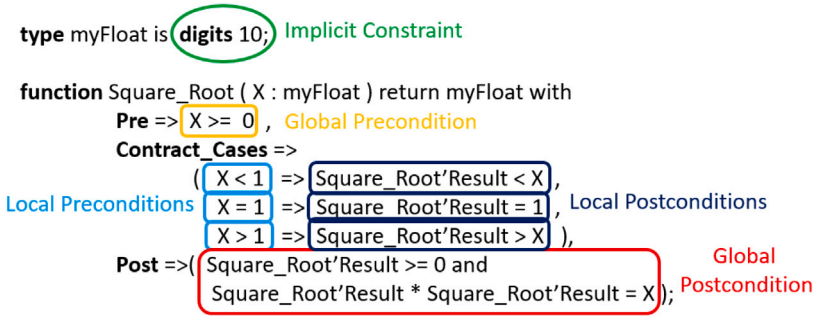


Fig. 2. Elements of a program specification.

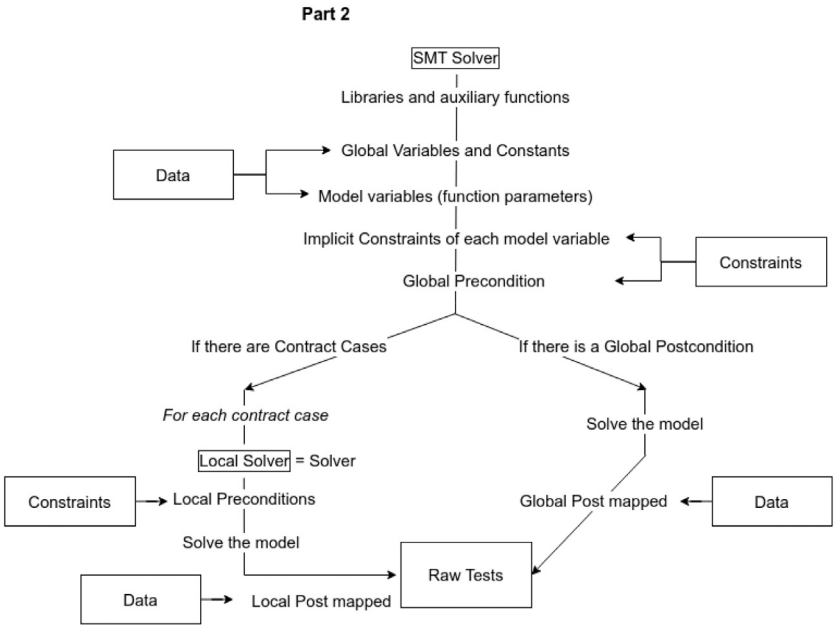


Fig. 3. Algorithm 1 diagram.

the *Constraint_List* (C_n) shown in Algorithm 1, our Constraint Oracle (CO), based on the Satisfiability Modulo Theory (SMT), completes the test vectors with inputs that obey the *Constraint_List* (C_n) and is, therefore, able to generate the test cases that are needed by pairing these concrete inputs with the corresponding mapped outputs.

Fig. 3 shows how our CO integrates an SMT solver provided by the GNAT framework – the Ada toolchain – to produce solutions with the guidance of our method.

This method consists of a natural adaptation of the "*Standard Case Analysis*" from ECP [15]. Such a testing method selects N distinct test cases per equivalence class, found in the input data domain of the software to be tested, where N is a parameter entered by the user. Consequently, since our process is automatic, the outputs provided by the CO consist of N unit tests for each equivalence class found in the input domain, where N is a user-defined parameter. To continue, we provide the steps necessary to create such CO, as can be seen in Algorithm 1:

1. Declare the global variables/constants found in the target function package specification. Usually, these are global constants meant to configure the state of the subprogram and consequently play an important role in the preconditions of the specification under study.
2. Declare the model variables, whose resulting values will be assigned to test vectors. These values correspond to the parameters of the subprogram under test.
3. Add the implicit constraints associated with each model variable, regarding its corresponding parameter type, to the solver. The most typical example of this would be restricting an Integer to a certain range as specified via the user-defined type declaration, such as:

```
type Array_Index is Integer range 0 .. 5;
```

   Failing to comply with these restrictions would lead to an error during the subprogram compilation.
4. Add the explicit constraints or global preconditions. These constraints will be common to all contract cases that may appear later and reflect requirements that input data must meet. Failing to comply with these restrictions would lead to a run-time error.
5. For each contract case, create a local SMT solver to add the local preconditions of the actual contract case so that these local preconditions do not interfere with other (disjoint) contract cases. Note that local preconditions do not restrict the input data i.e., actual subprogram parameter values, as global preconditions do, but from the value detected in the processing of the input data, the possible execution paths of the subprogram under test are obtained.

Algorithm 1: Constraint Oracle Test Case Generation Algorithm

```
Function ConstraintOracleTestGenera-
 tor(D_n, C_n, num_solutions):
    RawTestN ← ∅;
    Solver ← CreateSMTSolver();
    Solver ← IncludeGlobalVariablesAndConstants(Dn);
    /* Target function parameters                    */
    Solver ← Solver ∪ IncludeModelVariables(Dn);
    foreach FunctionParameter ∈ D_n do
        Solver ← Solver ∪
          ParamTypeImplicitConstraints(FunctionParameter, C_n);
    end
    Solver ← Solver ∪ GlobalPrecondition(C_n);
    /* If there are Contract Cases                   */
    if ∃LocalPreconditions ∈ C_n then
        foreach LocalPrecondition ∈ C_n do
            LocalSolver ← Solver;
            LocalSolver ← LocalSolver ∪ LocalPrecondition;
            N ← 0;
            while N < num_solutions ∧
              ConstraintProblemIsSatisfiable(LocalSolver) do
                Model ← SolveConstraintProblem(LocalSolver);
                TestVector ← ∅;
                foreach ModelVariable ∈ Model do
                    TestVector ← TestVector ∪ Modelvariable;
                end
                LocalPostcondition ←
                  LocalPostAssociated(LocalPrecondition, C_n);
                RawTest ← (TestVector, LocalPostcondition);
                RawTestN ← RawTestN ∪ RawTest;
                LocalSolver ← LocalSolver ∪
                  ConstraintsToObtainDifferentSolutions();
                N ← N + 1;
            end
        end
    end
    if ∃GlobalPostcondition ∈ C_n then
        GlobalPostcondition ← GlobalPostcondition(C_n);
        N ← 0;
        while N < num_solutions ∧
          ConstraintProblemIsSatisfiable(Solver) do
            Model ← SolveConstraintProblem(Solver);
            TestVector ← ∅;
            foreach ModelVariable ∈ Model do
                TestVector ← TestVector ∪ Modelvariable;
            end
            RawTest ← (TestVector, GlobalPostcondition);
            RawTestN ← RawTestN ∪ RawTest;
            Solver ←
              Solver ∪ ConstraintsToObtainDifferentSolutions();
            N ← N + 1;
        end
    end
    return RawTestsN;
```

In step 5, it is worth noting that a particular contract case solution should not affect the other contract cases. The disjoint and exhaustive nature of the contract cases allows us to treat each one in isolation, and this compositional feature of the specifications can be exploited in the testing of any software [19], greatly reducing the number of test cases. Also, note that the previous steps (1–4) allude to some common aspects of all contract cases. Once all the constraints have been added to the solver, for each different test the CO will yield:

6. A solution consisting of concrete values for the model variables obtained by running the created solver.
7. Print the test vector obtained (model variables solved) along with the associated expected outcome mapped. They may come from local (contract case) or global postconditions.
8. Ensure different solutions for the next iteration by adding a new constraint to the solver to obtain a different test vector. SMT solvers usually act deterministically so we need to add constraints simply to exclude previously generated values.

A diagram that summarizes, illustrates, and helps to understand the proposed Algorithm 1 is shown in Fig. 3. To clarify the diagram, contract cases, and global postconditions are not mutually exclusive but they should be considered individually as test cases should only focus on "one concern at a time".

3.2. Algorithm application example

To illustrate the algorithm, we will apply it to the following Spark example belonging to the `region_checks` spec package from GNAT-Studio, the current Ada Integrated Development Environment:

Listing 1.1: Sign benchmark with Ada syntax.

```
subtype Sign_Type is Integer range -1 .. 1;

function Sign (X : in Integer) return Sign_Type
with  Contract_Cases =>
        (X < 0 => Sign'Result = -1,
         X = 0 => Sign'Result =  0,
         X > 0 => Sign'Result =  1);
```

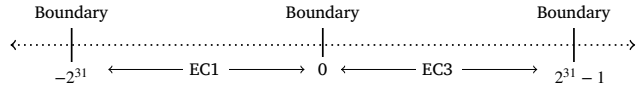
Before the Constrained Oracle creation, we need to gather all the data and constraints needed from the package specification as shown in Fig. 1 by automatically analyzing the node types and content of the AST derived from this specification. Firstly, it is assumed that the current information is available in an AST and the target function is *Sign*. Secondly, the user-defined data subtype *Sign_Type* is processed. Following that, by using the target function name we would isolate the function *Sign* subtree inside the AST and begin with the subprogram declaration/specification analysis.

The first step of this analysis involves collecting the parameter `X` along with its correspondent data type and the subprogram returned data type. Then, three different Equivalence Classes (EC) are automatically identified as the formal contract has 3 different contract cases (note we have a discrete domain), which later on will lead to the creation of three different local SMT solvers:

$$\begin{aligned} \text{EC1} &= \{\mathtt{X} \in \mathbb{Z} \mid \mathtt{X} < 0\} \\ \text{EC2} &= \{0\} \\ \text{EC3} &= \{\mathtt{X} \in \mathbb{Z} \mid \mathtt{X} > 0\} \end{aligned} \tag{2}$$

As a reminder, the mathematical binary equivalence relations used to automatically identify the EC come directly from the local preconditions (left part) of each contract case found in the subprogram specification.

To continue with, if we consider that integers are represented in our experimental set-up using 4 bytes (implicit constraint), an EC diagram would look like:



At this point, we meet the data/constraint requirements shown in Fig. 1 and consequently the CO creation may begin. Firstly, the

Listing 1.2: Binary search benchmark with Ada syntax.

```
type Ar is array (1 .. 10) of Integer;

function Search (A : Ar; I : Integer) return Integer with
 -- A is sorted
 Pre  => (for all I1 in A'Range =>
   (for all I2 in I1 .. A'Last => A (I1) <= A (I2))),
 -- If I exists in A, then Search'Result indicates its position
 Contract_Cases => (
 (for some Index in A'Range => A(Index) = I) =>
    Search'Result in A'Range and A(Search'Result) = I,
 (for all Index in A'Range => A(Index) /= I) =>
    Search'Result = 0);
```

SMT model variable associated with the target function parameter X will be defined as an integer. Following that, considering the implicit constraints of the parameter data type, we would restrict the domain of X to $-2^{31}..2^{31}-1$ (4 bytes). Next, the three previously mentioned local solvers are automatically created. Note that each local solver would return as many values as desired by the user-defined parameter N that aims to emulate Standard Case Analysis.

The first local solver takes into account EC1 to produce the test vector, along with its corresponding local postcondition as the first contract case reflects:

$$X < 0 \Rightarrow \texttt{Sign'Result} = -1$$

Consequently, the local precondition $X < 0$ is added. Then, the SMT local solver would be executed and return, for example, $X = -1$, which paired with the corresponding local mapped postcondition will lead to the test case:

$$\texttt{Test\_Vector} = -1, \quad \texttt{Expected\_Output} = -1$$

Next, to obtain a different test vector for the next iteration and eventually supply N different test cases for this EC, it is necessary to exclude the already produced $X = -1$ so the constraint $X \neq -1$ is added back.

A completely similar rationale can be applied to the resting equivalence classes, resulting in the test cases:

$$\texttt{Test\_Vector} = 0, \quad \texttt{Expected\_Output} = 0$$

$$\texttt{Test\_Vector} = 1, \quad \texttt{Expected\_Output} = 1$$

Finally, if parameter domain boundary analysis was enabled, the first and third local solvers will return the test cases:

$$\texttt{Test\_Vector} = -2^{31}, \quad \texttt{Expected\_Output} = -1$$

$$\texttt{Test\_Vector} = 2^{31} - 1, \quad \texttt{Expected\_Output} = 1$$

3.3. First order logic example

The following specification belongs to an array binary search function (see Listing 1.2):

Following the previous example steps, first, we would automatically generate the AST and gather the data/constraints needed to create the constrained oracle. In this case, we would identify the function parameters A, I, and integer as the returned data type.

After that, the specification analysis would report a global precondition that imposes a non-strict ascending order in the array:

$$\forall\ I1 \in \{1, \ldots, 10\},\ A(I1) \leq A(I2)\ \forall\ I2 \in \{I1, \ldots, 10\} \tag{3}$$

Then, 2 different ECs are identified: belonging to the array or not. Next, the CO creation begins by creating an array, A, composed of 10 integer variables and an integer variable I. Since all the model variables are integers, we would consider the implicit constraint of restricting their domains to $-2^{31}..\ 2^{31}-1$.

After that, we would add the global precondition of having a sorted array, and finally, 2 local solvers would be created as there are 2 different contract cases. The first one will consider the local precondition:

$$\texttt{(for some Index in A'Range => A(Index) = I) => Search'Result in A'Range and A(Search'Result) = I}$$

The translation of '`for all/some`' is as simple as imposing an `And/Or` clause respectively to a set of elements. Once the first local solver is finished, it would return a combination of a sorted array and an element belonging to such array, which paired with its corresponding local postcondition would result in, for example:

$$\texttt{Test\_Vector} = ([0, 1, 2, \ldots, 9], 0), \quad \texttt{Expected\_Output} =$$

"`Search'Result in A'Range and A(Search'Result) = I`"

Remember that postconditions are not subject to any translation process, as they are simply mapped to pass/fail criteria through a runtime check inside the unit tests. Therefore, the SMT solver does not consider them. Moreover, note that this first contract case with a local postcondition is ambiguous since the array order is non-strict, repeated elements could exist, and pre-fixing an expected output before calling the function could lead to a mistake if various array indexes comply with the postcondition and the one returned mismatches the pre-fixed.

Applying a completely similar rationale to the second equivalence class, the second local solver would return a test vector composed of a sorted array and an element not belonging to it. When paired with the local postcondition of this contract case, we could for example obtain the following test case:

$$\texttt{Test\_Vector} = ([0, 1, 2, \ldots, 9], -1), \quad \texttt{Expected\_Output} = 0$$

As we can observe, this second contract case does have an unambiguous local postcondition that lets us pre-fix the index 0 as the expected output.

4. Empirical study

The objective of the empirical study is to evaluate the impact of our CO. To achieve that, we follow usual industrial practices for software testing which include (a) running test cases and observing the number of test cases passing or failing i.e., *success rate*, and (b) measuring *code coverage*. The latest is a metric to verify and quantify how the requirements are implemented in the code as well as observing which pieces of the code are untested [6].

More accurately, the two main research questions are:

- **Research Question 1 — Effect of using the Constraint Oracle on the measured success rate for all generated test cases.**

Test cases must conclude whether they pass or fail. What is the success rate of the test cases generated from the CO which take into account the formal contracts?

– **Research Question 2 — Effect of using the Constraint Oracle on statement coverage.** In our study we employ *statement coverage* which was also used in similar research works [11]. Branch coverage was discarded for the reasons laid out in Section 4.3. An effective test oracle should reach a high percentage of statement coverage. Is this hypothesis true in the case of our proposal?

Spark/Ada is a programming language employed in high-criticality applications. Its toolchain is well-equipped with compiler flags, theorem provers, and flow examiners to provide a thorough static analysis. Spark is endowed with formal contracts and they can be integrated into the specification of a software package. In addition, Spark may employ additional run-time checks (implemented by the user) that may be helpful for theorem prover or just as a defensive code to events only discovered at run-time e.g., failure at opening a file [1].

Given that these readable contracts are inserted in the source code, we have implemented a test generation tool to provide some compelling evidence for our test generation method. This tool employs Z3py [20] as a constraint solver. Run-time checks are not read by our test generator since the information contained in them is – to our knowledge – not very useful for the generation of traceable test cases.

By definition, a subprogram can contain several equivalence classes which, in turn, can contain an array of values useful for generating test cases. In this respect, our Oracle creates test cases in two iterative configurations that are set up by the user. In iteration 1 it creates, with a limit of 1000 per EC, test cases using the Standard Case Analysis testing method [15]. The number of tests generated per EC is by default 1 and is regulated with the N parameter set by the user. In iteration 2, Boundary Analysis testing [15] kicks in generating as many test cases as possible depending on the number of boundaries.

To evaluate the statistical significance of the CO, we have implemented a random test generator which consists of using the same SMT solver with minimum guidance from the information of the specification. More precisely, it chooses a random test vector for the input domain and matches it with a random output of the returned data type to build a full test case.

This choice was motivated by the *simplicity* of the implementation given the fact that our libraries were already interfacing with Z3py [20] SMT solver. As a result, random numbers could be generated inside the bounds of the data types that conform to the test vector. These limits are used in the specification of solver execution constraints. Random test oracle is oblivious to any formal contract. The only constraints added from the formal contracts are those of data type ranges i.e., implicit constraints, since assigning values beyond such bounds would result in a compilation error.

This randomized method was evaluated using 15 different seeds to provide confidence in the results to be obtained. By contrast, due to its nature the CO presented here is purely deterministic.

To evaluate the statistical significance we have employed the *Exact Wilcoxon–Mann–Whitney* [21] test as it is a common practice for this kind of evaluation where the resulting random distributions are unknown, this is, a *non-parametrical test*. The idea behind this statistical test is to take advantage of the rank-sum strategy to claim whether two samples come from the same population (null hypothesis) or not. To achieve this, the test procedure involves pooling the observations from the two samples into one combined sample, keeping track of which sample each observation comes from, and then ranking these observations from lowest to highest. The threshold to accept the null hypothesis is the standard p-value ≥ 0.05.

The following subsections are dedicated to discussing the characteristics of the examples followed by a discussion of the empirical results of each research question.

4.1. Benchmarks

The downside of choosing Spark/Ada benchmarks is that it is significantly harder to find open-source examples, in contrast to more popular programming languages such as C or C++. An additional requirement is that Spark benchmarks have to be written with formal contracts. A quick search into Spark examples from AdaCore IDE, GNAT Studio [22], returned that only 26% of the subprograms of this repository contained formal contracts that can be used for our test generation method.

All benchmarks were mathematically verified using the available contracts before generating the tests, with the assistance of the GNATProve – Spark's theorem prover or verifier – provided in GNATStudio. Such *proofs* assure us that the preconditions hold in the function call and the postconditions are always satisfied in the function output.

Benchmarks deployed in the study

– *Accept Offer* is a benchmark described in the previous section and is referenced in software testing books to apply ECP [15] as well as dealing with a fixed-point definition.
– *Sign* and *In Unit Square* also come in a popular Spark book [1] as an example of *"contract cases"*, which is similar to equivalence classes with formal contracts expressions.
– *Binary Search* is a key benchmark for formal contracts [1] that includes universal and existential quantifiers.
– *Compute Speed* is a control system that computes the speed of a rocket exhibiting more complex floating-point management.
– Lastly, the examples *To_Green*, *To_Yellow* and *To_Red* belong to a *Traffic Lights* benchmark from GNAT Studio examples of safety-critical signaling systems that provide an elaborate formal contract including universal and existential quantifiers, complex constraints along with more elaborate data structures.

The available formal contracts employed on these benchmarks provide some *representative* examples for our CO given the wide array of constructs for a fair evaluation.

To lay out the complexity of the input and subsequently, the formal contracts in our case study the following metrics and their rationale are used in the study:

1. **Number of Equivalence Classes found** to represent the complexity of the ECP of the specification domain. Logically, the more ECs, the more complex the test generation becomes. As it was previously mentioned in Section 3, the absence of contract cases, namely the absence of local preconditions, will lead us to consider the formal contract domain as a trivial ECP with a single EC.
2. **Number of constraints added to the CO** quantifies the number of restrictions that are needed by our CO to yield a single solution, as well as the number of test cases to be produced per class. This is a measure of the burden for the CO test generator. The result is directly proportional to the N parameter defined by the user which dictates the number of test cases per EC and the idiosyncrasies of the Standard Case Analysis.
3. **Universal Quantifiers** indicate the deployment of universal quantifiers in the formal contract specification. Universal quantifiers enhance the expressivity of the boolean predicates, however, they also impose additional challenges from the proof or test generation perspective.
4. **Lines of Code** quantifies the number of lines of code in the implementation of the subprogram. It is a standard measure of the amount of code.

Regarding the first 2 rows of Table 1, benchmarks *Sign* and *In Unit Square* domains contain 3 and 5 EC respectively, according to the number of contract cases found in their specification. Each *Sign* EC adds 1 constraint to the solver (check the local preconditions of Listing 1.1),

Table 1
Complexity of the input space and formal contract for each benchmark. N is the user-defined number of test cases to be generated for each EC and n is the size of the array or list (if any).

Benchmark	EC found	Constraints added	Univ. quantifier	Lines of code
Sign	3	$3{\cdot}N$	✗	5
In Unt. Sqr.	5	$20{\cdot}N$	✗	10
Accept Offer	2	$3 + 2{\cdot}N$	✗	4
Compute Spd.	1	$8 + 5{\cdot}N$	✗	13
To Green	1	$16 + (9n + 1){\cdot}N$	$\forall,\forall$	1
To Red	1	$16 + (4n + 2){\cdot}N$	$\forall,\forall$	1
To Yellow	1	$16 + (4n + 2){\cdot}N$	$\forall,\forall$	1
Binary Srch.	2	$\sum_{i=1}^{n-1} n - i + (2n){\cdot}N$	$\exists,\forall$	16

which repeated N times leads to a total of $3{\cdot}N$ constraints added to the underlying SMT solver. On the other hand, each EC found in *In Unit Square* formal contract adds 4 constraints, leading to $20{\cdot}N$ constraints overall.

With respect to the next 2 rows, *Accept Offer* and *Compute Speed*, a completely analogous reasoning can be applied with the difference of considering the implicit constraints added at the beginning, which are related to the user-defined types range and/or floating-point precision used in the formal contracts. Besides this, *Accept Offer* has 2 EC each adding 1 constraint per iteration, and *Compute Speed* single EC stacks 5 constraints per iteration. Lastly, bearing in mind that implicit constraints (as well as global preconditions) do not stack per iteration of the Standard Case Analysis, the final formulae for the number of constraints added to the underlying SMT solver of our CO by these benchmarks would be: $3 + 2{\cdot}N$ and $8 + 5{\cdot}N$, respectively.

Next, the *Traffic Lights* package benchmarks display implicit constraints too and introduce universal quantifiers, which are especially useful when applying restrictions over an array of elements. Within this package, the benchmark "To Green" implies a higher complexity as its specification must ensure a harder safety property for swapping the traffic lights' color to green, as would be expected.

To continue with, *Binary Search* formal contract (see Listing 1.2) includes 2 different EC as well as a strong global precondition. Firstly, imposing an ascending order over an array of size n implies adding $\sum_{i=1}^{n-1} n - i$ constraints as each element must be smaller than its subsequent array components. Then, checking whether an element I belongs or not to an array of size n means adding another n constraints, joined by "or" clauses if we are looking for a positive answer ($a_1 = I \vee a_2 = I \vee \dots$) or joined by "and" clauses otherwise ($a_1 \neq I \wedge a_2 \neq I \wedge \dots$). Note that the conventional strategy used to reduce the cost of finding an element in an ordered array is part of the implementation of the subprogram, but is not reflected in the specification itself which is our target.

Again, considering that global preconditions do not stack per iteration of the Standard Case Analysis, the final formula for the number of constraints added to the underlying SMT solver of our CO by this benchmark would be: $\sum_{i=1}^{n-1} n - i + (2n){\cdot}N$.

We would like to highlight that even though *In Unit Square* has the highest number of EC found in the domain, the *Traffic Lights* package benchmarks raise more complex problems for the SMT solver as they add more constraints (even for the base case $n = 1$) despite having a single EC.

Finally, the last column is dedicated to listing the lines of code ranging from 1 to 16. Admittedly, they are very sizable subprogram bodies. It is worth noting that when selecting the benchmarks we picked examples with different complexities and features of the input space and the formal contracts, as that is what would give confidence in the validation of our CO method.

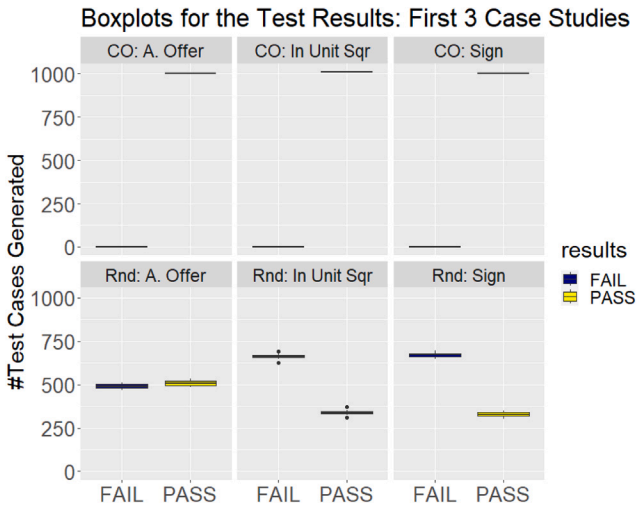


Fig. 4. Number of test cases produced and their execution results (*success rate*) by Constraint Oracle (CO) and Random method (Rnd). These case studies consist of Accept Offer (A.Offer), In Unit Square (Sqr), and Sign.

Table 2
Number of test cases generated for each benchmark by the two testing methods.

Benchmark	A. Offer	Sign	In unit square
Standard case	1000	1000	1009
Boundary A.	2	2	4

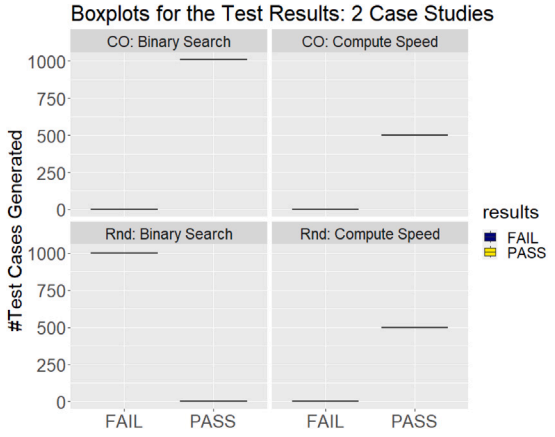


Fig. 5. Number of test cases produced and their execution results (*success rate*) by Constraint Oracle (CO) and Random method (Rnd). CO produced 1069 test cases for Binary Search. Only 500 test cases were produced for Compute Speed as it had a single ECP.

4.2. Success rate: Pass/fail results

The first effect to measure is the success rate i.e., the number of pass vs. fail test cases, of the resulting test suite. Given that the test suite is derived from the formal contracts and the code is also proved, we would expect to see a high ratio of passing test cases for the CO. This process of testing proven code may look redundant, nonetheless, static analysis has its limitations and certain run-time bugs can only be spotted by testing e.g., bugs embedded in the software, the compiler or even faulty hardware. In addition, a theorem prover can only replace testing for safety-critical systems if the tool is qualified [6] which is extremely costly.

Fig. 4 displays the success rate for the first 3 case studies. As expected, CO returned *pass* for *all* test cases. By contrast, the results for the random test generator were roughly the same in the first case

Table 3
Number of test cases generated for each benchmark by the two testing methods.

Benchmark	Binary search	Compute speed
Standard case	1000	500
Boundary A.	60	1

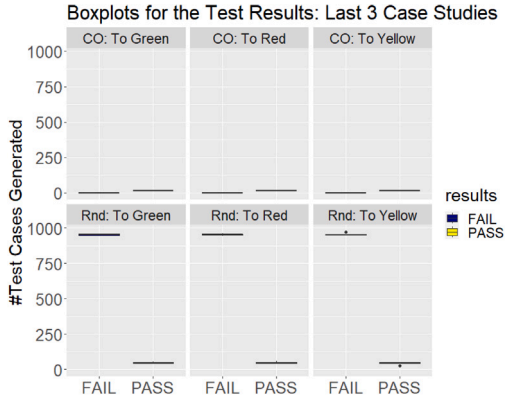


Fig. 6. Number of test cases produced and their execution results (*success rate*) by Constraint Oracle (CO) and Random method (Rnd). Traffic Lights Case Studies. CO generated all (16) plausible (passing) test cases.

because *Accept Offer* only had two equivalence classes, whereas in *In Unit Square* and *Sign* most of the test cases produced failed. This was because both benchmarks had more equivalence classes that made it more unlikely to build a passing test case. Table 2 shows the number of test cases generated in each configuration. In the case of *In Unit Square* and *Sign* for the Boundary Analysis method, a central EC was omitted as there was a single value separating adjacent classes that were previously considered in the Standard Case. For example, *Sign* has 3 different EC as shown in Listing 1.1, where we can appreciate how the central EC is composed of the single value 0. Consequently, Boundary Analysis coincides with Standard Case Analysis in those ECs that are composed of a single element.

The second group of benchmarks are depicted in Fig. 5 which is composed of *Binary Search* and *Compute Speed*. Again, CO produced a test suite that returns *pass* in every case. The accuracy of the CO is evidenced in *Binary Search* where an accepted input entails feeding a sorted numeric vector and an element included in such a data structure. On the contrary, the random method fails to generate a sorted test vector and therefore it did not meet the precondition to call the function.

On the right-hand side in Fig. 5, it can be also observed the results from the *Compute Speed* are the same for both testing methods. This is because there is a single EC along with a widely accepted postcondition that is contrary to design prescriptions that advocate for *precise* postconditions [2].

Table 3 outlines the number of test cases produced by each method. For *Binary Search* 60 test cases evaluating the boundaries were produced which was motivated by the number of resulting bounds for the 11 input parameters. On the contrary, for *Compute Speed* only a single test case was produced given its only EC.

Finally, the last 3 case studies from a traffic light control system are portrayed in Fig. 6. Each benchmark controls the light of a usual traffic light system that takes into account the state of abstract adjacent traffic lights. CO oracle succeeds at generating all plausible test cases (126) and all of them finally report a passing result. In this case, Boundary Analysis did not produce any test cases since Standard Case Analysis achieved exhaustive testing. By contrast, the randomized method struggles to achieve a few passing ones whilst most of them were failing tests. The complexity of the verification conditions was the cornerstone

Table 4
Wilcoxon–Mann–Whitney test results. (H0 hypothesis: there is no significant difference in pass rate between the randomized method and CO).

Benchmark	A. Offer	Sign	In unit square	Binary search
p-value	$6.657 \cdot 10^{-8}$	$6.657 \cdot 10^{-8}$	$6.657 \cdot 10^{-8}$	$6.657 \cdot 10^{-8}$
Benchmark	C. Speed	To Red	To Yellow	To Green
p-value	H_0 accepted	$6.657 \cdot 10^{-8}$	$6.657 \cdot 10^{-8}$	$6.657 \cdot 10^{-8}$

Table 5
Resulting statement coverage for Constraint Oracle (CO) and Random (Rnd) in each case study.

Test generator\Benchmark	Accept offer	In unit square	Sign
CO	100%	100%	100%
Rnd	100%	93.33%	88.89%

to show the difference in the usefulness of the results produced by each of the methods.

Last but not least, Table 4 shows the statistical significance from Wilcoxon–Mann–Whitney test. Results concluded statistical significance when comparing all FAIL and PASS distributions between both test generators except Compute Speed. The reason behind it is that given the wide acceptability of the postcondition (a safety-invariant) and the fact there is a single EC, many test cases produced were successful and subsequently, the success rate distributions were very similar.

Finally, to clarify the p-values obtained in Table 4, we need to realize that the Wilcoxon–Mann–Whitney test is based on the sum of *signs* resulting from computing the subtraction of the data from both distributions, just like every "rank sum" test computes, and if there is a distribution that always acts as an upper unreachable bound for the other, the *sign* sum will be constant and will conclude independence as expected.

4.3. Code coverage

This subsection is dedicated to assessing the results of code coverage, a standardized measure of completeness for requirements-driven testing. For this evaluation, we employed `gcov` [23] as a coverage tool.

As mentioned before, branch coverage was relinquished because it was not a *controllable* variable from the experimental point of view. From the technical point of view, to achieve 100% branch coverage test cases must be run to evaluate *all* decisions once to true and once to false. Some of the benchmarks employed implemented run-time checks whose decisions must be negated to raise branch coverage. After a careful look, we noticed that the conditions to trigger these run-time checks were mutually exclusive with the software contracts, which are proved statically before running our test oracle. As a result, some *dead branches* may become apparent, and reaching 100% branch coverage is not feasible.

Last but not least, branch coverage reports may be inaccurately derived by `gcov`, as the compiler may introduce additional and *hidden* decisions related to run-time checks, exceptions, or subroutines. In conclusion, statement coverage is deemed our main code coverage measure whereas branch coverage is more secondary.

The first case study results are displayed in Table 5 and they encompass *Accept Offer* and *In Unit Square* benchmarks. Firstly, *Accept Offer* achieved 100% statement coverage with just 2 test cases. Branch coverage data also reported 100% for *Accept Offer* for both methods. The simplicity of the benchmark was a key contributing factor to this result. Secondly, *In Unit Square* returned 100% statement coverage for CO and 93.33% for random. The available constraints for this benchmark were dealt with successfully by the CO, whereas the randomized method struggled with one of the branches as the input data were very specific to be generated randomly. Branch coverage resulted in 94.4% for CO and 83.33% for the random approach.

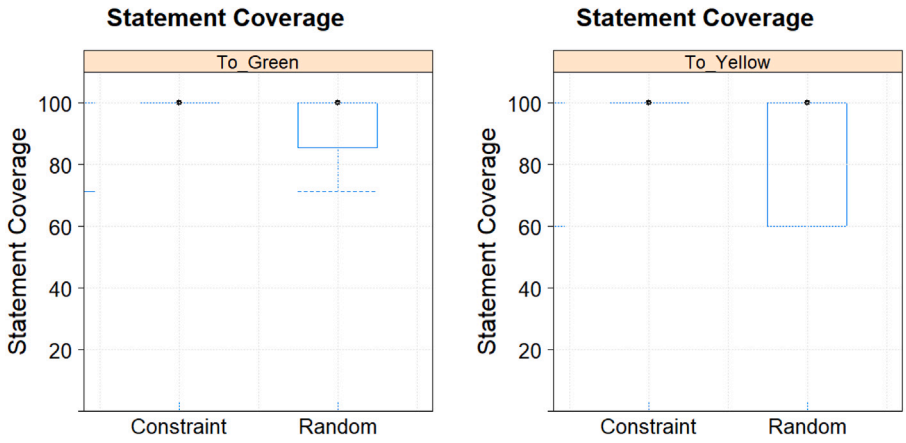


Fig. 7. To green and to yellow statement coverage results in boxplots.

Table 6
Resulting statement coverage for Constraint Oracle (CO) and Random (Rnd) in each case study.

Test generator\Benchmark	Binary search	Compute speed
CO	91.3%	100%
Rnd	21.74%	100%

Table 7
Resulting statement coverage for Constraint Oracle (CO) and Random (Rnd) in each case study.

Test generator\Benchmark	To Red
CO	100%
Rnd	100%

Table 8
Wilcoxon–Mann–Whitney test results. (H0 hypothesis: there is no significant difference in pass rate between the randomized method and CO).

Benchmark	A. Offer	Sign	In unit square	Binary search
p-value	H_0 accepted	$6.657 \cdot 10^{-8}$	$6.657 \cdot 10^{-8}$	$6.657 \cdot 10^{-8}$
Benchmark	C. Speed	To Red	To Yellow	To Green
p-value	H_0 accepted	$6.657 \cdot 10^{-8}$	$6.657 \cdot 10^{-8}$	$6.657 \cdot 10^{-8}$

The next case study data is populated in the third column in Table 5. The statement coverage data was high for both methods but only CO reached 100%. In the *Sign* case study, the random method converges to 88.89% statement coverage because there is a branch $x = 0$ in the code followed by a statement that is never reached. As for branch coverage, CO achieved 100% and random method 80%.

Binary Search case study results are listed in Table 6 where CO did not reach 100% statement coverage but converged to 91.3%. The underlying reason is that there was one statement that was not exercised that contained arithmetic manipulation of the array index. The random method remained constant at 21.74% because the random test generator failed to provide a sorted test vector. Therefore, the precondition check halted the execution before reaching the body of the function. This statement coverage can be considered measurement noise as the input data never reaches the body of the function.

Branch coverage data resulted in 51.56% for CO and 0% for random for the reason stated above. Perhaps an interesting observation of this case study is the fact that even though the code was proven and, to the best of our efforts, although the CO made full use of the formal contracts, there was still one branch that could not be covered. That is why it belonged to an unspecified subpath included in the *Binary Search* that corresponded to the second contract case which in turn specified that the item to be searched does not belong to the array (see Listing 1.2).

Results for *Compute Speed* displays 100% statement coverage in Table 6 for both methods as it contained a large block of code in the body of the function. This block did not contain branches apart from the preexistent run-time checks. Branch coverage results reported 50% branch coverage for both oracles. Paradoxically, this result is great news considering the implementation is packed with assertions that include secondary branches for run-time checks that were just not stressed.

The last group of benchmarks included the traffic lights benchmarks with 3 subprograms, namely *To_Green*, *To_Yellow* and *To_Red*. The results of the first two subprograms mentioned above are shown in Fig. 7, while Table 7 collects the evidence from the third. Similarly to *Binary Search*, these subprograms were equipped with preconditions that prevented random data from being accepted and reaching the body of the function. In all cases, CO achieved 100% sentence coverage in the first solution, whereas the random technique usually takes several iterations to obtain a result and only gets to execute the function body if it is lucky. Not for all seeds was possible to reach the body of the function and, subsequently, to reach 100% coverage. This is epitomized in the results from the random test generator in Fig. 7.

Branch coverage data reported 100% for CO and 0% for random method. Again, the random method struggled to hit the body of the function due to the elaborate preconditions.

Table 8 shows the statistical significance results of the statement coverage distributions. Except for *Accept Offer* and *Compute Speed*, where statement coverage distributions were similar, Wilcoxon–Mann–Whitney concluded statistical significance for all case studies.

Finally, we should remark that when applying our method based on the information from the specification, if the test cases obtained do not cover the code as expected, then the function programmer should consider refining the specification as it may contain too general contract cases that make the oracle to struggle through a blend of unspecified subpaths. Perhaps alternative implementation of different testing techniques may attain better code coverage results.

4.4. Threats to validity

Almost no empirical study is perfect and this is not an exception to the rule. In this respect, two types of threats can be outlined.

– **Threats to internal validity** comes from the implementation of the random test generator which only considers the input and output ranges. Other possible implementations of the random test generator may consider the contracts. However, to the best of our knowledge, there is no convention on the implementation of random test generators, and to our knowledge, this is the most common [3]. Another threat comes from the statement coverage

results as provided by the `gcov` tool. Some statement coverage was counted even though in some cases the execution flow did not reach the body of the function, e.g., random test generation for binary search. In these cases, statement coverage is not accurate.

– **Threats to external validity.** Perhaps the greatest threat to external validity comes from the benchmarks employed for the case studies. These are open-source benchmarks that are relatively small. Ideally, a more complete study would have encompassed industrial large-scale benchmarks. However, this representative software is protected by Intellectual Property (IP) clauses and is seldom shared with strong non-disclosure agreements. In addition, SatixFy group, does not use at present Spark or Ada for the software of its satellite payloads.

Despite the above threats, in our estimation, the empirical study here presented included a wide array of formal contracts that can be found in industrial benchmarks. Moreover, both coverage and success rate data give some compelling evidence that automated test case generation from formal contracts is possible, effective, and promising.

5. Related work

The seminal works on test data generation date back to the 70s, in one of these papers [24] J.C. King applies (static) *symbolic execution*, which consists of a special software execution by which input data are replaced by symbols and then the constraints are gathered during the process. These constraints are analyzed by a solver to generate input data to stress new paths in the SUT. The analysis was applied to benchmarks with relatively small data sets and, in addition, processing power was quite limited at the time.

In 2002, Henzinger et al. [25] derived input data statically using Constraint-Based Testing (CBT) and a detailed graph derived from SUT written in C. Unfortunately, this resulting graph may contain statements that do not impact the flow of the program and therefore is suboptimal for CBT [3]. Their evaluation included some small drivers for Windows and Linux for which a few bugs were identified in concurrent code. The tool implemented for the evaluation had common static limitations such as the inability to deal with pointers operations.

Later on, Holzer et al. [26] create the so-called *query-driven testing* underpinning Henzinger et al.'s. [25] work. By using the scanned graph from the SUT, this framework enables you to generate test vectors for specific user-defined areas of the code by using queries. The empirical study included both code-generated industrial engine controllers and open-source examples. The evaluation was more orientated to compare the efficiency with a similar and slower test generation tool at achieving block coverage.

Symbolic execution can be applied either statically or dynamically [27]. The static strand reasons at the source code level by using symbols to replace input data. Still, significant issues become apparent when the source code of external libraries is not available (which is common in the software industry due to IP protection), or when constraints are not processable by the constraint solver in the symbolic execution engine e.g., hash functions.

The other strand is known as dynamic symbolic execution or *concolic testing*. The testing process employs concrete values rather than symbols and constraints representing paths or branches. These values are gathered during software execution, fed back to the constraint solver, and new inputs that stress new code paths are computed. This approach is argued to be more accurate than the static counterpart [27] as it has managed to test branches that would have been impossible to analyze statically due to the above-mentioned obscure implementations [28].

The results of the concolic testing achieved significant confidence around 2010 provided the fact that it detected bugs in large-scale software programs. For instance, Godefroid et al. in 2008 [29], successfully tested the popular Microsoft Excel which requires 45.000 bytes as a test vector and nearly a billion x86 instructions from the SUT. Later in 2011, the same author reported that concolic testing unveiled around one-third of bugs in Windows 7 by file fuzzing saving millions of dollars in patches. Unfortunately, the sophisticated instrumentation to gather constraints was tailored for x86 machines and therefore is of very limited use in the embedded systems industry where a wide array of architectures are employed.

Fuzzing testing is a variant of symbolic execution whose purpose is to identify input data that raise exceptions, run-time errors, security vulnerabilities, or similar bugs. This technique has started to be applied in aerospace security by recent commercially available tools for Ada programming with only some demo examples publicly available [13].

So far we have described *code-driven test input generation*. In essence, these testing activities shed light on the behavior of the SUT by revealing pathological input data. In principle, they would not be acceptable since the software quality standards [8–10] impose tracing tests backward to software requirements. Only recently have these test oracles begun to be accepted in the realm of aerospace security [13] because even elaborate requirements and specifications can rarely cover *all* possible run-time behaviors and that security vulnerabilities may be embedded in the code.

The most compliant techniques are concerned with *specification-driven test cases generation* and they are described next. Back in the 2000s Claessen and Hughes [30] developed QuickCheck, a random test generator that considers formal verification conditions in Haskell, a functional programming language, to derive test cases following a distribution. As part of the testing process the programmer was in charge of (a) providing the right predicates to define the correctness of a program and (b) deciding whether the resulting random distribution matches the actual one. It is worth remembering that functional programming is not used in safety-critical systems.

Dafny is another so-called "verification-ready" programming language that employs theorem provers regularly and it supports both functional and imperative programming [2].

In 2023, Fedchin et al. [4] developed an extensive library for testing Dafny programs, which includes unit testing, mocking (code isolation for testing), and test generation modules. Their idea of deriving test cases from formal contracts is similar to the one in this work. Their test case generation derives the expected output from postconditions – similar to our approach – and test vectors from counterexamples produced by the verifier. The latest is produced artificially by inserting some "trap assertions" in strategic decision areas in an intermediate proof language (Boogey) advocated to fail the proof and produce these counterexamples.

The evaluation included two industrial benchmarks from a well-known web services provider, and the automatically generated test cases managed to unveil a significant bug in an external library that could not identified statically by the verifier. The downside of their approach becomes apparent in the extensive need of using mocks i.e., simple functions implementations returning test values to isolate the SUT, and occasional redundant test cases that are later mitigated by an additional form of cleaning process.

Returning to *imperative programming*, the idea of combining Spark/Ada contract test cases is not new, in fact in 2012 Comar et al. [31] investigated the idea of incorporating proofs and tests as part of the verification system in the Ada toolchain, although the test cases, in the only case study presented in this paper, were produced manually.

Later on, in 2017, Sun et al. [11] applied a Constraint-Based Test Generation using an MC/DC method with boundary analysis to derive test cases from Spark contracts. To the best of our knowledge, MC/DC is applied in code coverage analysis but it is not used as a standard testing method. Unfortunately, their test generation does not seem to support universal or existential quantifiers normally available in Spark contracts. Likewise, the evaluation uses a C model checker library that seems coupled with their test generation process. Two industrial case studies were performed. MC/DC was not measured on

the software under test despite using their industrial instrumentation tool supporting MC/DC analysis. Eventually, the statistical significance of the experimental results could not be determined.

In 2021, Jaramillo et al. [32] apply test case generation from formal contracts driven by Meta-heuristics and mutation testing. In particular, they used a method based on Particle Swarm Optimization. However, they did not disclose how they mapped the VCs to the fitness function. Another important issue is the fact that the VCs mutated to influence the mutation score. This decision, in our view, is not acceptable from a methodological point of view, since the VCs serve as a *ground truth* to compare the implementation against. The correct process would have been, if necessary, to change the implementation to fit the specification, but not the opposite.

In the bigger picture of the bug-fixing process, there are some works dedicated to automatic program repair. In his 2019 survey, Gazzola et al. [33] pointed to the abundance and quality of test cases as a key element to spot bugs that can later be fixed automatically. These self-repairing methods rely on properly detecting and isolating bugs as a prerequisite for which a good combination of tests and code coverage data is needed.

In a nutshell, our research work differs in several aspects from the state-of-the-art: First and foremost, our approach targets the *test coverage* of the formal contract from the ECP with boundary analysis perspective, whereas other approaches aim for maximizing code coverage [28], or to find pathological inputs to break the program [13]. Secondly, our method is not bothered with the implementation details (this is often known as black-box testing), which makes it oblivious to the intractable paths handling that are identified as a research challenge [3,27], or any other graph derivation from the SUT.

Thirdly, unlike similar works in functional programming using Dafny [4], our test generator does not rely on a pre-existent verifier to derive test cases. Theoretically speaking, our oracle can produce test cases by supplying a formal contract even though the underlying language toolchain is not equipped with a verifier. Fourthly, our test generation process is intended to comply with industrial software quality standards by embracing ECP with boundary analysis which is perhaps the friendliest testing method [6]. Although, to our understanding, testing methods are not normalized in these standards. The traceability of the testing process is also preserved which makes this activity compliant with the standards.

Regarding our evaluation, it is done systematically and scientifically, choosing open source examples and measuring statistical significance by incorporating random test data. Admittedly, unlike other state-of-the-art work, we have not been able to evaluate our testing method on an industrial scale.

6. Conclusion

Automation is a very promising and probably the only solution to the escalation of software verification costs. Automatic test case generation is a considerable challenge that can help to alleviate this problem. A substantial body of research is dedicated to code-driven input data generation which is normally not accepted in software quality standards for dependable systems, given that test cases – which must include an expected output – must be traced to requirements.

The fact that Spark/Ada programming language enables the integration of a formal specification provides a traceability source for test case generation. This feature, paired with the observation that CBT has become increasingly used and computationally affordable in the last few years, has motivated our work.

As part of our contributions we have derived a Constraint-Based Test Generator Oracle that produces test cases from Spark/Ada formal contracts. Unlike state-of-the-art approaches [11], our method automates a well-established testing method i.e., ECP with Boundary Analysis. In addition, it copes with First-order Logic predicates, it is a reproducible test case generation method, and is independent of the SMT solver. Our evaluation has given some supporting evidence that the test cases derived are *always* successful when it comes to the pass/fail ratio that we named 'success rate' along with statement coverage figures, all of them above 90% indicating a high completion result for software testing. Threats to validity, however, have pointed out the need to evaluate our method to larger industrial benchmarks but this is also very challenging to access due to proprietary and confidential IP.

In conclusion, the reported evidence in this paper along with the work from Sun et al. [11] and Fechin et al. [4] serves as a starting argument that a design by contract can be used both for proving the code and generating test cases automatically. As a result, we believe that the greatest possible level of confidence in the verification process can be achieved by combining software proof for the abstract domain and testing for the empirical one.

7. Future work

Future work can be centered on provisioning formal contracts that are useful both for verifiers and test generators. One line of inquiry can look at scenarios where program proof is not feasible due to unanalyzable external dependencies [4] and study whether this form of contract-driven test generation can detect bugs or whether standard manual testing can be effective enough [1].

Another line of inquiry may also consider a range of formal contracts quality ranging from underspecified contracts to the ones equipped with weak preconditions and strong post-conditions [2]. This is based on our observation that some contract constructs i.e., contract cases from Spark 2014, are more friendly to our approach whereas others with weaker verification conditions were not that useful for testing purposes.

Another approach may consider the C/C++ programming asserts [34] which are similar, to some extent, to Spark/Ada verification conditions and occasionally in "debug" build configurations. A test oracle may feed test cases to assess the available or absent asserts to give confidence in the functional correctness.

Lastly, another research idea consists of extending the usual functionality of test generators (data generation) to generate stubs (data and code generation). Stubs are relatively similar to mocks and consist of testing code that returns a specified value to replace external dependencies. As a result, this increases the isolation of the SUT, raises the controllability of the testing process, and enables greater code coverage results. For example, in C-written Linux Kernel Drivers, isolation is needed to test error-handling code [35]. Ada technologies [22] are equipped with stub function generators (`gnatstub`) and test-harness (`gnattest`) generation but to our understanding the stub returned values are to defined by the user.

CRediT authorship contribution statement

Samuel Jiménez Gil: Writing – original draft, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Manuel I. Capel:** Writing – review & editing, Validation, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Gabriel Olea Olea:** Writing – original draft, Validation, Software, Investigation, Data curation.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Manuel I. Capel reports that travel expenses for the communication of first results, in an initial phase of this study, at a national conference were provided by the Concurrent Systems Research Group of the University of Granada.

Data availability

Data will be made available on request.

Acknowledgments

Funding of the open access fee: University of Granada / CBUA.

References

[1] John W. McCormick, Peter C. Chapin, Building High Integrity Applications with SPARK, Cambridge University Press, 2015.
[2] K. Rustan, M. Leino, Program Proofs, MIT Press, 2023.
[3] Samuel Jiménez Gil, Constraint-Based Testing and Tail Tests for Measurement-Based Probabilistic Timing Analysis, (Ph.D. thesis), University of York, 2020.
[4] Aleksandr Fedchin, A toolkit for automated testing of dafny, 2023.
[5] Stephen Brown, et al., Essentials of Software Testing, Cambridge University Press, 2021.
[6] Leanna Rierson, Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance, CRC Press, 2013.
[7] Nigel James Tracey, A Search-Based Automated Test-Data Generation Framework for Safety-Critical Software, (Ph.D. thesis), 2002.
[8] RTCA, DO-178c, software considerations in airborne systems and equipment certification, 2011.
[9] ISO26262, Road vehicles – functional safety, 2018.
[10] ECSS-E-ST-40C, Space engineering: Software, 2009.
[11] Youcheng Sun, et al., Functional requirements-based automated testing for avionics, CoRR (2017).
[12] Raul de la Cruz, et al., MASTECS multicore timing analysis on an avionics vehicle management computer, 2022.
[13] Paul Butcher, Guidelines and Consideration Around ED-203A / DO-356A Security Refutation Objectives, AdaCore Whitepaper, 2021.
[14] Georgia Weidman, Penetration testing: A hands-on introduction to hacking, 2018.
[15] J.J. Shen, Software testing: Techniques, Princ. Pract. (2019).
[16] https://www.rapitasystems.com/difference-between-decision-coverage-and-branch-coverage.
[17] Cem Kaner, et al., Lessons Learned in Software Testing: A Context Driven Approach, Wiley, 2002.
[18] John Barnes, The Proven Approach to High Integrity Software, Altran Praxis, 2012.
[19] https://www.nasa.gov/content/tech/rse/research/compositional.
[20] https://github.com/Z3Prover/z3.
[21] M. Hollander, D.A. Wolfe, Nonparametric Statistical Methods, second ed., John Wiley and Sons, New York, 1999.
[22] https://alire.ada.dev/.
[23] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[24] James C. King, Symbolic execution and program testing, Commun. ACM (1976).
[25] Thomas A. Henzinger, et al., Lazy abstraction, in: Proceedings of the 29th ACM SIGPLAN-SIGACT, in: Symposium on Principles of Programming Languages, 2002.
[26] Andreas Holzer, et al., FShell: Systematic test case generation for dynamic analysis and measurement, in: Computer Aided Verification, 20th International Conference, in: CAV 2008, Princeton, 2008.
[27] Patrice Godefroid, Test generation using symbolic execution, in: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS, 2012.
[28] Patrice Godefroid, Higher-order test generation, 2011.
[29] P. Godefroid, M.Y. Levin, D. Molnar, Automated whitebox fuzz testing, in: Network and Distributed Systems Security, 2008.
[30] Koen Claessen, John Hughes, QuickCheck: A lightweight tool for random testing of haskell programs, 2000.
[31] Cyrille Comar, et al., Integrating formal program verification with testing, Open-Do (2012).
[32] Román Jaramillo Cajica, et al., Automatic generation of test cases from formal specifications using mutation testing, 2021.
[33] Gazola, et al., Automatic software repair: A survey, IEEE Trans. Softw. Eng. (2019).
[34] G. Kudrjavets, et al., Assessing the relationship between software assertions and faults: An empirical investigation, 2006.
[35] Bai Jia-Ju, et al., Testing error handling code in device drivers using characteristic fault injection, 2016.