Singularity: A methodology for automatic unit test data generation for C++ applications based on Model Checking counterexamples

Eduardo Rohde Eras
eduardorohdeeras@gmail.com
Instituto Nacional de Pesquisas Espaciais
São José dos Campos, São Paulo
Brazil

Valdivino Alexandre de Santiago Júnior
valdivino.santiago@inpe.br
Instituto Nacional de Pesquisas Espaciais
São José dos Campos, São Paulo
Brazil

Luciana Brasil Rebelo dos Santos
lurebelo@ifsp.edu.br
Instituto Federal de Educação, Ciência e Tecnologia de São Paulo
Jacareí, São Paulo, Brazil

ABSTRACT

One of the most challenging task of testing activity is the generation of test cases/data. While there is significant amount of studies in this regard, there is still need to move towards approaches that can generate test case/data based only on source code since many software systems mostly have the source code available and no adequate documentation. In this paper a new methodology, called Singularity, is introduced to generate unit test data for C++ applications based on Model Checking, a popular technique for test case generation. Our approach, which is to be supported by a tool, automatically translates C++ code into a model which resembles a Statechart model and then into the notation of the NuSMV Model Checker. Later, we rely on a technique based on the HiMoST Method, producing counterexamples from the Model Checker that are, in fact, the test cases/data themselves. We have applied our approach to a few C++ case studies analyzing how feasible it is for automatic test data generation.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Model checking**; **Software verification**.

KEYWORDS

Unit test, Model Checking, Counterexamples, Automation, Tool

ACM Reference Format:
Eduardo Rohde Eras, Valdivino Alexandre de Santiago Júnior, and Luciana Brasil Rebelo dos Santos. 2019. Singularity: A methodology for automatic unit test data generation for C++ applications based on Model Checking counterexamples. In *IV Brazilian Symposium on Systematic and Automated Software Testing (SAST 2019), September 23–27, 2019, Salvador, Brazil.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3356317.3356319

SAST 2019, September 23–27, 2019, Salvador, Brazil

ACM ISBN 978-1-4503-7648-8/19/09…$15.00
https://doi.org/10.1145/3356317.3356319

1 INTRODUCTION

Software production is a continuous activity. The omnipresence of technology in the modern era suggests large and constant software production in unlimited different areas. All that code must be tested somehow. Quality of all such produced software systems must be ensured, and hence Verification & Validation (V&V) activities can be employed for this goal [2]. Moreover, software testing [6] is the most used V&V activity in practice. Ideally, adequate software documentation should exist to support code production and enable testing based on established specifications. However, sometimes the code itself is the only documentation that is available. It is not uncommon to find software systems that are in production and whose documentation is extremely poor [5]. In such situations, the use of source code alone to generate tests case/data should be considered, where code robustness needs to be evaluated. This work handles the situation where source code is already available for the software, but functional aspects of the program are perhaps poorly documented or documented but no longer credible.

Unit testing is widely used in industry and developers are familiar with writing test suites [16]. This specific type of test verifies if the functionality of a small part of the program is correct. It is also interesting to note that another topic mentioned in this paper, the C++ programming language, is still one of the most used programming language in the global development scenario, as recently corroborated by the TIOBE index [20].

From this perspective, the present work proposes a new methodology for automatic generation of software test data from existent C++ code. This methodology, called Singularity, presents a collection of processes to read the source code, extract the states and transitions, build a model of the software under test and apply it as an input to a Model Checker tool. Using special properties that force the Model Checker to generate counterexamples, test data are generated aiming at achieving higher code coverage of the source file without the need of documentation or software specification.

The use of counterexamples, i.e. traces of execution where a given propertie is not satisfied, to generate test cases is a very popular approach [9], being a very convenient technique that can be fully automated, quite flexible and, under proper circumstances [1], very efficient. To create unit test case/data without requiring a documentation, based only on the source code, is a very attractive

[1] In scenarios of a limited size and without parallel content to avoid state explosion

approach with great appeal to practitioners. To the best of our knowledge, no other approach generates test cases/data based only on the C++ source code using Model Checking counterexamples. Explore the feasibility of such scenario is one of the goals of the present methodology.

This paper is structured as follows. Section 2 provides a review of the main issues used to develop this work. Section 3 explains the approach to achieve unit test generation from source code, using counterexamples of Model Checking process. This section also brings the details for constructing the models, describing the algorithms. In Section 4 an analysis of the approach is made by applying it to six different case studies and collecting the results to verify the feasibility of Singularity. Related work is discussed in Section 5. Finally, Section 6 concludes the paper and comments the future of this research.

2 BACKGROUND

In this section, we present a brief review of the main topics related to this paper.

2.1 Software test

The intricate nature of a computer program requires the use of computational methods to perform tests that efficiently cover the system under evaluation. According to Fraser [11], testing remains the most important approach to verify the quality of a software product, and it is desirable that it is automated because manual testing requires a huge effort and is error-prone.

A testing process can be composed of several activities, being functional or structural. According to Jorgensen [15], *functional tests* are based on the principle that any program can be considered a function that maps values from its input domain to an output range. This type of test is commonly known as "black box test", where the contents of the box (the implementation of the program under test) is unknown, and the operation of the black box is known only in terms of its inputs and outputs. *Structural tests* or white box tests are, in contrast to functional tests, based on knowledge of the implementation of the program being tested. They have the ability to see what's "inside the box" and identify test cases based on how the function is implemented. There is also the Model Based software test technique where the generation of test cases is based on models of the system and allows for system test cases to be elaborated even without the software code itself, early in the software product life cycle.

Another important concept is the level of testing. The level of testing depends on the phase of software development where the test will be accomplished. According to Delamaro [7], it starts at the component level (unit) and is progressively expanded until it encompasses the entire system. In a unit test, a single component is tested and the main goal is to find defects in the component. At the integration level, several components are tested as a group and the interactions among the components are investigated. At the system level, the entire system is tested and it is necessary to ensure that the system works according to its requirements, and furthermore features such as reliability and performance are evaluated. At the acceptance level, the client evaluates the system and issues a verdict if it meets the expected functionality.

2.2 Unit Tests

According to Pressman *et al.* [18], the unit testing is the effort to verify the smallest unit of software, the so-called module. Important control paths are tested to discover errors within the boundaries of the module, limiting the complexity of the tests and the complexity of the errors found within the unit tests restricted field of action. A unit test is always based on white box.

The unit test activity is usually considered part of the coding process. Once the syntax of a source code has been developed and reviewed, a unit test is created for that code. Each test case must be associated with a set of expected results.

2.3 Model Checking

According to Delamaro [7], a model allows the discovery of knowledge about the system, being usable as an oracle for tests, defining what would be an appropriate or erroneous behavior. For this, the correctness of a model must be ensured, i.e., it needs to be tested in the same way as the system itself. Proved that the model is adequate, it becomes of great value for generating test scenarios. An important point is the model format: it is possible to describe the actions performed by a system through a graph where the nodes represent the states and edges represents the transitions between the states. This technique is called the "State Transition System".

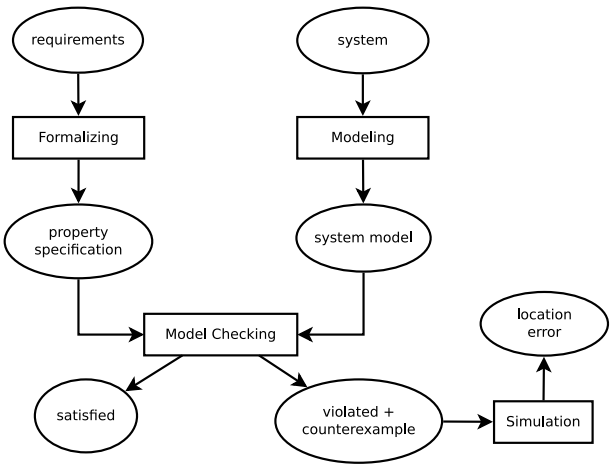

Figure 1: Schematic view of Model Checking. [2]

According to Baier [2], Model Checking requires a precise and unambiguous statement of the properties to be examined. The system model is commonly generated automatically from a model description that is specified in some appropriate programming dialect (such as C++ for example). Specification properties prescribes what the system should and what the system should not do while the model reports what the system actually does. The Model Checker examines all relevant states of the system to see if they satisfy the desired property. A model can satisfy the properties when all states are in agreement with the property under consideration or generating a counterexample if any state violates some property. The counterexample describes an execution path beginning in the initial state of the system towards the violation state. Following

this path, a user can reproduce the scenario, thereby obtaining useful debugging information and adapting the model (or property) according to the detected error. An overview of this process can be seen in Figure 1.

2.4 Using Model Checking Counterexamples as Test Cases

The work of Fraser [11] describe testing data generation with Model Checker as the interpretation of the counterexamples as test cases. Provided a model of the system under test, a set of special properties might be used to force the Model Checker to generate counterexamples. Those properties are called trap properties. A trap property models a characteristic of the system and claims it as false, the generated counterexample exhibits an execution trace where the system characteristic is true. A trace is a sequence of system states, so it's possible to interpret this as a test data. This technique easy produces paths in a system for a given scenario. For example, one may provide a trap property telling to the Model Checker tool that the final state of a model is never reached. The counterexample will show a path thought the system leading from the initial state to the final state.

However, Model Checker tools are originally meant to formal verification, not test case/data generation. This twisted use of the technique may not be accepted in the industry and some known issues take place. In [10], the author discusses some points relate to this approach, two of then are briefly considered in this section.

The Model Checker tool validates a model against a property, so the validity of the result depends on the validity of the model. A model that does not describe the system correctly does not produce a valid result. This issue is confronted by the Singularity methodology by extracting the model from the code itself. Even if the extraction process may not be perfect, the generated test cases can conceivably be analysed by the user for adjustments.

Model Checkers do exhaustive analysis of an input model. This often leads to state explosion depending on the size of the input. To avoid this trouble, Singularity methodology concentrates test case generation on unit tests. Thus, this problem is minimized by the limited size of the input. The number of model states is proportional to the number of instructions on the code, which is expected to be small most of the time. Parallel programming, a common source of state explosion, is not supported by Singularity, which also contributes to the reduced number of states during the Model Checking process.

3 THE SINGULARITY METHODOLOGY

This section briefly describes Singularity methodology. Its structure is composed by four components: Reader, Extractor, Generator and Constructor. The main workflow is shown in Figure 2.

3.1 The Reader component

The first component is responsible for reading the C++ source code and transform it in some data structure ready to be used by the methodology. A tree structure is created from the source code containing every instruction of the code, organized by the scope blocks.

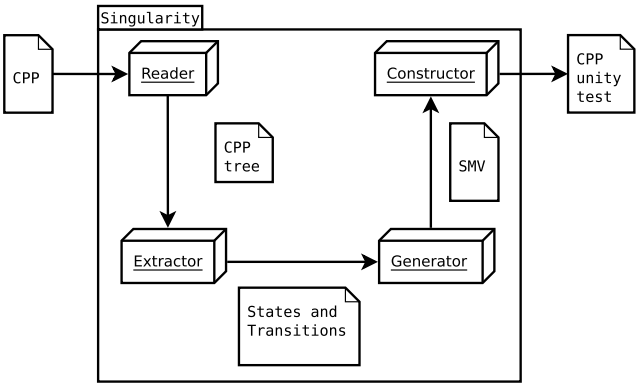

Figure 2: The Singularity methodology structure

3.2 The Extractor component

To use a Model Checker, one must have a system model to be tested in the form of a state transition system. The extractor is responsible to iterate the incoming C++ tree and extracting states and transitions from it. This process generates a metadata representing the structure of the original code using the C++ language scopes as hierarchy levels and each statement line as a state, as well as simulating a code execution in search of a sequence of state transitions. This process is divided into two parts, a state extractor function and a transition extractor function, as shown in Algorithm 1.

Algorithm 1 Extractor

input: C++ tree
output: states, transitions

```
1: states ← stateExtractor(C++ tree)
2: transitions ← transitionExtractor(C++ tree)
3: return states, transitions
```

The transformation of C++ source code into states is accomplished by interpreting each statement line that generates a new scope (block of code that delimits the validity of a variable, usually indented or between braces) as a level state and each line that does not generate a new scope as a plain state. Algorithm 2 describes the procedure for extracting states.

Algorithm 2 stateExtractor

input: C++ tree
output: states

```
1: states ← initializeSet()
2: for node ∈ C++ tree do
3:     if node generates a new scope then
4:         state ← generateLevelState(node)
5:     else
6:         state ← generatePlainState(node)
7:     end if
8:     states.add(state)
9: end for
10: return states
```

For the state diagram to be complete, each state must be connected through a set of transitions. The transitions are inferred simulating an execution of the file being tested. The input C++ file is expected to contain classes and functions. The *trasitionExtractor()* function generates a virtual *main()* function where objects are instantiated for all classes contained in the input file and calls all public functions of those classes. If there is any standalone function in the source code, these will also be called. From the first statement of the main function, a recursive call is made to simulate a code execution. This way, the process forces a coverage of the C++ file. In Algorithm 3, the instruction sequence of the transitions extracting function is shown.

Algorithm 3 trasitionExtractor

input: C++ tree
output: transitions

```
1: transitions ← initializeSet()
2: if ∄ main() function then
3:     Create main() functiom
4:     Instatiate objects for all classes
5:     Call public methods from all objects and all standalone functions
6: end if
7: initialState ← first line of main() function
8: identifyAllPossibleNextStates(initialState)
9: function IDENTIFYALLPOSSIBLENEXTSTATES(state)
10:    nextStates ← identifyTransitions(state)
11:    for next ∈ nextStates do
12:        transitions.add(create a transition to next state)
13:        if next ≠ finalState then
14:            identifyAllPossibleNextStates(next)
15:        end if
16:    end for
17: end function
18: return transitions
```

3.3 The Generator component

The generator component is responsible for transforming the data structure that represents a state transition diagram into a model ready to be used by the NuSMV Model Checker. This step of the methodology is made by a variation of HiMoST method [6]. The generator component proposed here has as input a state transition diagram generated by the Extractor component, which differs from Statecharts [12] used in HiMoST method because it has no history or parallelism. It is also different in the way CTL properties are generated, instead of using the Dwyer [8] patterns in properties specifications combined with the states of the diagram by the TTR algorithm [3], the properties are generated by trap properties found in the work of Fraser et al. [11]. The complete process performed by the generator module is divided into five functions that can be seen in Algorithm 4.

The Algorithm 5 describes the first of the five functions of the Generator component. Three variables are created to describe a model of the system under test: state variable, event variable and

Algorithm 4 Generator

input: states, transitions
output: NuSMV Model

```
1: variables ← createVariables(states, transitions)
2: initials ← createInitialStates(states, transitions)
3: next ← createNext(initials, states, transitions)
4: properties ← createProperties(variables, states, transitions)
5: return NuSMV model ← generateNuSMV(variables, states, transitions, properties)
```

decision variable. Those variables have a set of values (states) extracted from the C++ code. The state variable holds information about the plain states and the decision structures from level states. A decision structure, when present in the code, uses Boolean events to define its execution flow. The existence of Boolean events sets the decision variable to be created. All other events different from Boolean ones are stored in the event variable.

Algorithm 5 createVariables

input: states, transitions
output: variables

```
1: stateVariable ← initializeSet()
2: for state ∈ states do
3:     if state == 'plainState' then
4:         stateVariable.add(state)
5:     else if state.type == 'decision' then
6:         stateVarable.add(state)
7:     end if
8: end for
9: eventVariable ← initializeSet()
10: Boolean ← False
11: for transition ∈ transitions do
12:     if transition.event ≠ 'TRUE' ∨ transition.event ≠ 'FALSE' then
13:         eventVariable.add(transition.event)
14:     else
15:         Boolean ← TRUE
16:     end if
17: end for
18: if Boolean then
19:     decisionVariable ← initializeSet()
20:     return variables ← stateVariable ∪ eventVariable ∪ decisionVariable
21: end if
22: return variables ← stateVariable ∪ eventVariable
```

Once the variables have been defined, the initial state of each variable must be established by the *createInitialState()* function. During the creation of the transitions by the extractor module, a *main()* function was used to simulate an execution stream. This function however is not tested. Therefore, the initial state of each variable must begin outside the *main()* function. The *decision* variable is obviously an exception, its initial value is simply set to *False*.

The *createNext()* function, seen in Algorithm 6, creates the set of next states for each variable in the NuSMV model. After finding the

initial state in the transition set, a recursive call is made to create transitions for all states, events and decisions, always avoiding visit the *main()* function.

Algorithm 6 createNext

```
input: initials, states, transitions
output: next
1:  next ← initializeSet()
2:  for transition ∈ transitions do
3:      if transition.origin == initialState then
4:          createTransitions(transition)
5:          break
6:      end if
7:  end for
8:  function createTransitions(transition)
9:      next.state.add(create state transition)
10:     if transition.event == binary then
11:         next.event.add(create decision transition)
12:     else
13:         next.event.add(create event transition)
14:     end if
15:     if transition.destiny ≠ finalState then
16:         if transition.destiny ∉ main() function then
17:             createTransitions(transition.destiny)
18:         else
19:             nextValid ← find next transition out of main()
20:             createTransitions(nextValid)
21:         end if
22:     end if
23: end function
24: next ← clear duplicated transitions
25: return next
```

The *createProperties()* function takes as input the states and transitions sets. The work of Fraser *et al.* [11] includes several properties for the generation of test cases regarding the coverage of the tested system. These properties, called trap properties, are a logical negation for each element of a model, forcing a coverage of the path traveled by that element in the given model. Three trap properties are used to cover the model by the Singularity methodology: the negation of events, the negation of transitions, and the negation of decision guards. These properties, called case 1, case 2 and case 3 respectively, cover the three types of variables created in Algorithm 5: states, events, and decisions.

The case 1 property in Equation 1 is applicable to the events. Denying each value assumed by the event variable generates a counterexample for each event with a path that leads to the confirmation of that event. This counterexample is used to generate a unit test.

$$\forall \square \neg (event = e), \forall\ e \in events \tag{1}$$

The case 2 property in Equation 2 uses the values of the Boolean transitions, stating that when it is in a particular state of origin and a certain Boolean transition value occurs, the next state will be different from the destination state. This property is written for each Boolean transition, for both *TRUE* and *FALSE* values (guard condition). Thus, this properties will generate a counterexample for both paths of the transition.

$$\forall \square\ origin \wedge guard \rightarrow \exists \bigcirc \neg\ destiny \tag{2}$$

For all non-Boolean events, the case 3 property shown in Equation 3 is used. Note that the first expression of case 3 property is equal to case 2 property. The second part states that a transition from a state other than the origin will always lead to the destination, which is immediately invalid for the model under test. This property guarantees at least one counterexample for each non-Boolean transition.

$$\begin{cases} \forall \square\ origin \wedge guard \rightarrow \exists \bigcirc \neg\ destiny \\ \forall \square\ origin \wedge \neg\ guard \rightarrow \exists \bigcirc\ \ destiny \end{cases} \tag{3}$$

Here, all the elements needed to create a NuSMV model is fulfilled and the *generateNuSMV()* function is responsible for generation of the input file of the Model Checker, following the expected order of elements found in a NuSMV input file. In a linear way, the state and event variables are created from the set of variables, the decision variable is created if it exists, the initial states from the set of initial states, the transitions from the set of transitions, and finally the properties from the set of properties. This final NuSMV file will be consumed by the next component to generate the counterexamples.

3.4 The Constructor component

The last component of Singularity methodology, seen in Algorithm 7, is responsible for generating the unit tests from the counterexamples of the Model Checker. Here the NuSMV Model Checker is called and the model to be tested is verified. It is expected that counterexamples are generated with the execution of the Model Checker. The list of counterexamples is iterated and for each counterexample a unit test is generated. In other words, each counterexample is a test case. It was established that only counterexamples with a number of states greater than 2 will be considered in the generation of unit tests, as suggested by HiMoST [6]. Each counterexample is iterated by its states. These states describe a point in the execution path to be visited in C++ code.

Algorithm 7 Constructor

```
input: NuSMV model, states
output: C++ unit test
1:  C++ unit test ← generate file
2:  counterexamples ← NuSMV(NuSMV model)
3:  for counterexample ∈ counterexamples do
4:      if number of states in counterexample > 2 then
5:          test ← generate C++ function
6:          for state ∈ counterexample do
7:              test ← generate C++ instruction from states
8:          end for
9:          C++ unit test ← test
10:     end if
11: end for
12: C++ unit test ← eliminate duplicated tests
13: return C++ unit test
```

4 EXPERIMENTAL ANALYSIS

This section presents the execution of Singularity over six different case studies. The results obtained describe the efficiency of test case generation and the code coverage using the counterexamples with the larger number of states. A tool is not fully implemented yet, so most of the experiment was conducted manually.

4.1 Description of Case Studies

The methodology was applied to six case studies. Three of them are part of a scientific application called GeoDMA, while the other are simpler academic examples. GeoDMA [17] [14] is a tool developed in C++ under FOSS (free and open source software) license and acts as a plugin for the *TerraView* Geographic Information System [13]. Three class files written in C++ from the GeoDMA source code directory were selected as input to the Singularity methodology for obtaining test cases. The selected case studies will be called *input* in this section.

- input 1 : The "context" class from the GeoDMA source library is a C++ code with approximately 115 lines of code without the comments. It has 8 functions, an operator and the constructors.
- input 2 : The "generalFunction" class from the GeoDMA source library has approximately 137 lines of code without the comments. It contains a unique function with several decision structures.
- input 3 : The "intersectionCache" class from the GeoDMA source library has approximately 128 lines of code without the comments, 3 functions, a destructor and a large *try catch* section with 3 exceptions handler.
- input 4 : The "inheritance" file has 23 lines of code, two classes and describes a simple inheritance scenario in object oriented programming.
- input 5 : The "strategy" file has 34 lines of code, four classes and describes a simple *strategy* design pattern scenario with polymorphic function calls.
- input 6 : The "triangle" file has 46 lines of code, two classes and several decision structures to solve the classic triangle classification problem where given three integers it evaluates if the numbers describe a valid triangle and classifies it according to the sides.

4.2 Metrics Definition

In order to analyze the results, three metrics were defined. Given the six inputs and three trap properties, several counterexamples were obtained. Only the counterexamples with more than 2 states were considered valid. Each input generates a test scenario with many states, events, and transitions. Each trap property generates several different properties for each input according to its states, events, and transitions. The three metrics are defined as:

- Efficiency: the ratio of generated counterexamples for each trap property
- Effectiveness: the number of valid counterexamples for each input
- Coverage: the code coverage considering the biggest counterexample encountered for each property

	Sta.	Eve.	Dec.	Prop.	Count.	Efficiency
input 1	35	4	yes	88	88	100,00%
input 2	48	0	yes	146	146	100,00%
input 3	55	9	yes	161	161	100,00%
input 4	4	2	no	8	8	100,00%
input 5	4	2	no	10	10	100,00%
input 6	21	5	yes	74	74	100,00%
Total	167	22	66,67%	487	487	100,00%

Table 1: Model variables and efficiency percentage for each input

4.3 Results and Analysis

Each input creates a model to be verified by the Model Checker tool. A model is composed by states, Boolean events or "decisions", non-Boolean events, transitions, and properties. Every input has a different number of states. A state is every node in the C++ tree marked with a unique ID, or every line of code where some flow of execution happens, such as statements and decision structures. Functions and classes declarations, "namespace" declarations, open and close brackets, visibility keywords or any other kind of code line where any significantly compute happens are not considered as a state. Not all inputs have events and decisions. Some inputs, like 4 and 5, have no decision structure so no Boolean event was used. The input 2 shows non-Boolean events. A non-Boolean event is any output string, returning object or exception thrown. Every input has transitions and properties. A property is generate by the trap properties seen in Section 3, named case 1, case 2, and case 3. Each "case" generates several properties to be verified by the Model Checker using the states, events, and transitions.

Table 1 shows the number of states, events, transitions, and properties for each input, as well as the number of generated counterexamples. Dividing the number of properties of each input by its respective number of counterexamples produces the efficiency value. The Singularity methodology achieved 100% efficiency for the six tested input.

Each input generates a number of counterexamples, but only the traces with more than 2 states is considered a valid counterexample. The total (which is called 'bulk') and the valid number of counterexamples tend to be different. The effectiveness of each input is calculated by Equation 4.

$$\text{effectiveness} = \sum_{case=1}^{number\ of\ cases} \left(\frac{valid_{case}}{bulk_{case}} \right) \tag{4}$$

Table 2 shows the number of bulk and valid counterexamples for each input divided by case 1, case 2 and case 3 trap properties. It also shows the effectiveness percentage for each input and the total effectiveness as the sum of every single effectiveness value divided by 6 (simple mean). The Singularity methodology owns 44, 97% of total effectiveness.

The six inputs generates a total of 264 valid counterexamples. It's not feasible to test code coverage of every trace without an automated tool. Thus, the biggest counterexample of each case in each input was considered to achieve code coverage. For code coverage, comments and imports were not considered as lines of

	Bulk			Valid			Effectiveness
input 1	4	76	8	3	36	8	53,41%
input 2	-	124	22	-	60	22	56,16%
input 3	9	130	22	8	58	22	54,66%
input 4	2	-	6	2	-	1	37,50%
input 5	2	-	8	1	-	0	10,00%
input 6	4	52	18	4	23	16	58,11%
Total	21	382	84	18	177	69	44,97%

Table 2: Bulk counterexamples and valid counterexamples for each input and effectiveness percentage

	case 1	case 2	case 3	lines	Coverage
input 1	55	54	43	86	58,91%
input 2	-	39	22	112	27,23%
input 3	18	26	23	82	27,24%
input 4	13	-	18	21	73,81%
input 5	20	-	-	27	74,07%
input 6	29	26	26	49	55,10%
Total	135	145	132	377	52,73%

Table 3: Covered lines for each case, total number of lines and total coverage

code. Table 3 shows the number of code lines that were covered by each case in each input, as well as the total of lines for each input. The coverage was calculated by Equation 5.

$$\text{coverage} = \frac{\sum_{case=1}^{number\ of\ cases} \left(\frac{covered\ lines}{total\ lines} \right)}{number\ of\ cases} \tag{5}$$

The coverage for each input is given by the summation of the covered lines divided by the total lines for each case, divided by the number of cases. The total coverage is given by the simple mean of all individual coverages. Singularity methodology achieved 52.73% of code coverage considering only a unique big counterexample.

5 RELATED WORK

This section presents some work related to the approach used by the Singularity methodology, some using Model Checker counterexamples to generate test cases, other generating unit tests and some just mentioning similar techniques, considering its own particularities.

Yoshida *et al.* [21] introduced KLOVER, a methodology for automatic generating tests for embedded system implemented in C or C++, using an analysis technique called "symbolic execution". The methodology generates unit level tests, and claims to provide excellent test coverage. The proposed approach generates the necessary environment to isolate the input code and delivers unit tests written in Google Test framework, *gtest*. The main difference between KLOVER and Singularity is the use of Model Checking: Yoshida's methodology aims to collect the required data for an execution and ensure the code coverage by using symbolic execution while the solution proposed by the Singularity methodology relies on counterexamples of a Model Checker.

Beyer [4] proposes a variation of *BLAST* Model Checker for automatic generation of test cases from counterexamples with respect to a given property p. Given a code written in C programming language, it finds all paths where the given property p is satisfied, thus generating a tests suite capable of cover all points of the code where p (usually a security property) is true. The proposed approach was used to find, for example, points in a code where access is allowed only within a valid password (root access) and has been tested for locating "dead code" in C applications with up to 3×10^5 lines of code, reporting a false positive free coverage with a fully automated process.

Another work that uses counterexamples of a Model Checker for generating test cases was written by Rayadurgam *et al.*, [19] where a method for automatic test case generation for structural coverage criteria is presented. A complete sequence of tests is automatically generated from Model Checker counterexamples achieving a predefined coverage for any software that can be represented as a finite state machine. The generation of counterexamples uses trap properties to force coverage of the input code, adopting exactly the same properties used by the Singularity methodology presented in that work. The method usage was exemplified with a small critical safety avionics systems.

Paul Ammann [1] proposes the use of Model Checking to generate test cases from the software specifications. The proposed method uses mutants based tests to generate counterexamples, solving the common problem of "equivalent mutants" (mutations in the code that do not affect its operation), since this mutation type is simply satisfied by the Model Checker and does not generate any counterexample. Test case generation uses these counterexamples and reduce the number of outputs by eliminating equivalent tests and prefixes. The method was applied to a code written in Java language.

6 CONCLUSIONS AND FINAL REMARKS

This paper has proposed a new methodology of test case/data generation from C++ code called Singularity. The process is divided in four components and relies on Model Checking counterexamples to generate the test cases and consequently, unit tests. The methodology uses three different trap properties to force the counterexamples generation. The chosen trap properties, named case 1, case 2, and case 3 achieved 100% efficiency in counterexamples generation for the six case studies, which means that all trap properties generates a counterexample in all six inputs. This certainly does not happen all the time, as some input can generate no counterexamples at all despite using a large number of trap properties, but even considering only the present scenario, it is still a promising result for the approach . About 45% of generated counterexamples were considered valid and if specifically only inputs 1 to 3 are analyzed (which are actually real case studies), 54, 74% of effectiveness is achieved. Using only the biggest trace of the delivered counterexamples as test case, an average of 52, 73% of code coverage is reached, a good result since a unique test was applied. The largest inputs 1, 2, and 3 obtained a smaller coverage with a single counterexample but the smaller input obtained a considerable coverage with a single test. If the totality of valid counterexamples could be applied to the case studies a broad code coverage may be achieved. Another positive characteristic of the methodology is about the concern of state explosion within test case generation by the model checker.

This problem is mostly solved in Singularity considering the small scale of a unit test. All scenarios tend to be limited by the size of a single C++ file, considering that each state is equivalent to one line of the code.

The tool to support our methodology is under development and needs to be concluded. The experimentation presented in this paper was basically manually conducted although this does not affect the outcomes of our research. The main contributions of this work are a detailed description of the methodology so that the implementation of a tool to support it is straightforward, and an experimental evaluation showing the feasibility of our approach. Once having a tool, several tests can be conducted and many improvements may be identified in the methodology.

Nevertheless, some challenges need to be addressed within the test case generation approach. When a unit test is written, the test function must be able to test the desired C++ file independent of its context, i.e. a unit test must be a standalone test. In order to execute a C++ file, some resources are needed perhaps from other files, hardware characteristics, specific types of input, some software library or any kind of particularity, known as overhead. To create a unit test for a file with a large overhead is challenging. A deeper study in this area is required, specially topics about stubs or mocks. Another similar problem resides in the functions input types, when for example, some specific numerical value is needed to simulate a desired path through a conditional structure in the code. Guessing a value for a guard condition is not a trivial task. This problem opens a wide field of different research. And finally, the input C++ language is very broad and there is a lot to predict about the entering source code. Things such as templates, pointers, design patterns or regular expressions are not yet supported by the methodology and there are some more functionalities in C++ language to be explored. However, there is still room to improve the methodology in a different direction, by creating compatibility with programming languages different than C++. Other object-oriented languages such as JAVA, Objective C or C# are strong candidates for the Singularity methodology if a demand arises.

ACKNOWLEDGMENTS

This research has been supported by *Conselho Nacional de Desenvolvimento Científico e Tecnológico* (CNPq).

REFERENCES

[1] P. E. Ammann, P. E. Black, and W. Majurski. 1998. Using model checking to generate tests from specifications. *Proceedings - 2nd International Conference on Formal Engineering Methods, ICFEM 1998* 1998-Decem (1998), 46–54. https://doi.org/10.1109/ICFEM.1998.730569

[2] C. Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking. MIT Press.* 984 pages. https://doi.org/10.1093/comjnl/bxp025

[3] Juliana Marino Balera and Valdivino Alexandre de Santiago Júnior. 2015. T-tuple reallocation: An algorithm to create mixed-level covering arrays to support software test case generation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9158 (2015), 503–517. https://doi.org/10.1007/978-3-319-21410-839

[4] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2004. Generating tests from counterexamples. *Proceedings - International Conference on Software Engineering* 26 (2004), 326–335. https://doi.org/10.1109/ICSE.2004.1317455

[5] L. C. Briand. 2003. Software documentation: how much is enough?. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.* 13–15. https://doi.org/10.1109/CSMR.2003.1192406

[6] Valdivino Alexandre De Santiago Júnior and Felipe Elias. 2017. From Statecharts into Model Checking : A Hierarchy-based Translation and Specification Patterns Properties to Generate Test Cases. (2017). https://doi.org/10.1145/3128473.3128475

[7] Marcio Delamaro, Mario Jino, and Jose Maldonado. 2017. *Introdução ao teste de software.* Elsevier Brasil.

[8] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1999. Patterns in property specifications for finite-state verification. *Proceedings of the 21st international conference on Software engineering - ICSE '99* (1999), 411–420. https://doi.org/10.1145/302405.302672

[9] Gordon Fraser and Franz Wotawa. 2007. Test-case generation and coverage analysis for nondeterministic systems using model-checkers. *2nd International Conference on Software Engineering Advances - ICSEA 2007* Icsea (2007). https://doi.org/10.1109/ICSEA.2007.71

[10] Gordon Fraser, Franz Wotawa, and Paul Ammann. 2009. Issues in using model checkers for test case generation. *Journal of Systems and Software* 82, 9 (2009), 1403–1418. https://doi.org/10.1016/j.jss.2009.05.016

[11] Gordon Fraser, Franz Wotawa, and Paul E. Ammann. 2009. Testing with model checkers: A survey. *Software Testing Verification and Reliability* 19, 3 (2009), 215–261. https://doi.org/10.1002/stvr.402

[12] David Harel. 1987. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 8, 3 (1987), 231–274. https://doi.org/10.1016/0167-6423(87)90035-9 arXiv:arXiv:1011.1669v3

[13] INPE. 24 de Abril 2019. TerraLib and TerraView Wiki Page. http://www.dpi.inpe.br/terralib5/wiki/doku.php?id=start

[14] INPE. 25 de Março 2019. GeoDMA - Geographic Data Mining Analyst. http://wiki.dpi.inpe.br/doku.php?id=geodma

[15] Paul C Jorgensen. 2013. *Software testing: a craftsman's approach.* Auerbach Publications.

[16] Michal Kebrt and Ondřej Šerỳ. 2009. Unitcheck: Unit testing and model checking combined. In *International Symposium on Automated Technology for Verification and Analysis.* Springer, 97–103.

[17] Thales Sehn Körting, Leila Maria Garcia Fonseca, and Gilberto Câmara. 2013. GeoDMA-Geographic Data Mining Analyst. *Computers and Geosciences* 57 (2013), 133–145. https://doi.org/10.1016/j.cageo.2013.02.007

[18] Roger Pressman and Bruce Maxim. 2016. *Engenharia de Software-8ª Edição.* McGraw Hill Brasil.

[19] S. Rayadurgam and M.P.E. Heimdahl. 2002. Coverage based test-case generation using model checkers. (2002), 83–91. https://doi.org/10.1109/ecbs.2001.922409

[20] TIOBE. 2019. TIOBE Index for April 2019. https://www.tiobe.com/tiobe-index/

[21] Hiroaki Yoshida, Guodong Li, Takuki Kamiya, Indradeep Ghosh, Sreeranga Rajan, Susumu Tokumoto, Kazuki Munakata, and Tadahiro Uehara. 2017. KLOVER: Automatic Test Generation for C and C Programs, Using Symbolic Execution. *IEEE Software* 34, 5 (2017), 30–37. https://doi.org/10.1109/MS.2017.3571576