Automatic Generation of Test Cases from UML Models

Constanza Pérez
Universidad Diego Portales, Facultad de Ingeniería,
Ejército 441, Santiago, Chile
constanza.perezg@mail.udp.cl

Beatriz Marín
Universidad Diego Portales, Facultad de Ingeniería,
Ejército 441, Santiago, Chile
beatriz.marin@mail.udp.cl

Abstract

[Context] The growing demand for high-quality software has caused the industry to incorporate processes to enable them to comply with these standards, but increasing the cost of development. A strategy to reduce this cost is to incorporate quality evaluations from early stages of software development. A technique that facilitates the process of quality evaluation is the model-based testing, which allows to generate test cases at early phases using as input the conceptual models of the system. [Objective] In this paper, we introduce TCGen, a tool that enables the automatic generation of abstract test cases starting from UML conceptual models. [Method] This paper presents the design and implementation of TCGen, a technique that applies different testing criteria to class diagrams and state transition diagrams making a reality the generation of abstract test cases by using a model-based testing approach. To do that, TCGen uses UML models, which are widely used at industry and a set of algorithms that recognize the concepts in the models and generate abstract test cases. [Results] An exploratory experimental evaluation of TCGen has been performed to compare the TCGen tool with traditional testing techniques. [Conclusions] Even though the exploratory evaluation shows promising results, it is necessary to perform more empirical evaluations in order to generalize the results obtained.

Keywords: **Model-based testing, Abstract test cases, UML models, MBT.**

1 INTRODUCTION

The growing demand for high quality software has caused the industry to use various techniques to ensure the quality of the software developed. Testing corresponds to one of the most used techniques for this purpose [1], however, it is still common to find companies whose definition of the testing process is incipient and often non-existent. In these cases, the design of the test cases is generated spontaneously when problems occur, causing an increase in the cost of development and also in the maintenance of software.

Even though there are companies that have defined a testing process, the responsible for carrying out the tests cases have often to deal with more problems that appears in this phase, for instance, they have limited time to execute the test cases, they need to understand the high complexity of software, they must to interpret the code, among others [2]. All these issues in the testing phase makes this phase as one of the most expensive of the software life cycle [3]. For this reason, there are various testing strategies that aim to reduce the high cost of this phase by reducing the time needed to design the test cases or execute the test cases, for example techniques and methodologies that use search-based testing [4], mutation testing [5], or model-based testing [6]. These testing strategies are used to create test cases that achieve two main goals: to verify the requirements or to find faults.

It is important to note that, when the complexity of the product grows, the cost of the testing phase increases [7]. An interesting alternative to reduce the cost of testing is to use model-based testing (MBT), since this technique uses the models that describe the system as an input to design the test cases. Therefore, this technique allows the generation of test cases automatically using the models of the system under test (SUT). MBT increases the productivity and quality of the software, since it moves the test activities at an early phase in the software development process and it allows to generate test cases independently of the implementation platform [8] due to it is possible to generate abstract test cases with the conceptualization of the test cases, that later can be translated to concrete test cases implemented in a programming language.

Nowadays, there exist diverse tools to support the testing process, whether open source tools as Selenium [9] and Watir [10], or licensed tools as HP QC [11]. These tools enable the automatic generation of test cases by programming the test cases or recording the user actions for interface testing. The amount and variety of available tools that automate testing depends on the programming language that is used by the SUT, so that these tools cannot easily be used in other projects. For a more generalization of the use of tools for generating test cases, these tools should allow the automatic generation of test cases and independently of the programming language of the software, i.e., abstract test cases based on models (MBT) must be generated. The conceptual models are important and a fundamental part in software development, since they constitute the starting point for a good software product. Although there are tools that allow to automatically obtain abstract test cases from models [12], these tools use complex models that are not widely used in industry as UML is, and they do not clearly state which kind of test cases are generated under well-defined testing criteria. So that, the use of these existent MBT tools is difficult to achieve in industry.

In this paper we present TCGen, an open-source tool that enables the automatic generation of test cases starting from UML models [13]. TCGen uses UML models due to UML is easy to understand and it being widely used in industry. For the generation of the test cases, the XML representation of the UML models is used as input by TCGen, which identifies the conceptual constructs in the XML file and then, it generates test cases using different testing criteria for the class diagram and for the state transition diagram. The generated test cases correspond to abstract cases expressed in XML files, that is, the test cases are independent of the programming language of the software to be tested. The contribution of this work is important for academia and industry. Practitioners can use TCGen to generate test cases automatically and then reduce the high cost of designing test cases in the testing process. Scientists can use TCGen to generate the abstract test cases and use these test cases to generate model editors that incorporate the features needed to perform the testing process.

In this paper we extend our previous publication [14] by adding more detail of the systematic literature mapping due to the incorporation of the search process and the retrieved amount of articles; adding a summary of the articles selected in the review indicating the models used and the tool that automatizes the generation of test cases; and adding the analysis of amount of models used that clarifies our motivation to use more than view of the system model to generate the test cases. Moreover, we clarify the different testing criteria that can be supported by an MBT tool, and the model coverage that is used to compare the proposed technique with manual testing. In addition, all the algorithms that are used by TCGen are provided in this version in order to facilitate the replicability of the generation of test cases by TCGen using other input models. Later, we provide more details of the case used to exemplify the use of UML models in the generation of abstract test cases by TCGen, and provide more examples of the test cases generated. Finally, we add more details in the cases used in the evaluation of TCGen and provide more explanations about future lines of research.

The rest of the work is organized as follows: Section 2 presents the background and related work; Section 3 presents the design and implementation of TCGen, and Section 4 presents the evaluation. Finally, Section 5 presents our main conclusions and future work.

2 RELATED WORK

This section briefly presents the foundations of model-based testing (MBT) and some relevant related work that was reviewed by performing a systematic mapping of the literature.

2.1 Model-based Testing (MBT)

MBT seeks to improve the quality of the systems in less time than traditional testing techniques by performing a testing process at early stages of the software development life cycle, with the aim of avoiding the complexity of the manual test case generation from the interpretation of the code [2].

MBT uses models of the system for the automatic generation of test cases of the SUT. Since these system models describe the software requirements and functional specifications, MBT is generally regarded as a black-box testing [15]. The process for the generation of test cases using model-based testing is shown in Fig.1. This process is comprised of the following five steps:

- Creation of the testing model, which can be obtained from the requirements or design models of the SUT.
- Generation of abstract test cases, for which it is necessary to choose a testing criterion, for example a criterion based on requirements, a criterion based on coverage, a criterion based on coverage of transitions, among others.
- Concretization of the abstract cases and the conversion of them into executable scripts. To do this, it is necessary to codify the abstract test cases in order to execute them.
- Execution of test cases automatically or manually, which could be performed online in order to observe the current system outputs, or off-line to encourage the creation of testing repositories.
- Analysis of the results, where the results of the execution of the tests are analysed and corrective actions are performed if they are necessary.

In the context of the software verification based on models, MBT presents the following advantages: (1) it is a practice that derives the information of models created previously, which are usually small in comparison to the entire model of the system; (2) the use of MBT often allows the traceability of the requirements to the test cases; (3) MBT can be used to test the system after the development process and also in its early stages of development; (4) MBT can be used to automatically generates sets of tests that meet a coverage criterion.

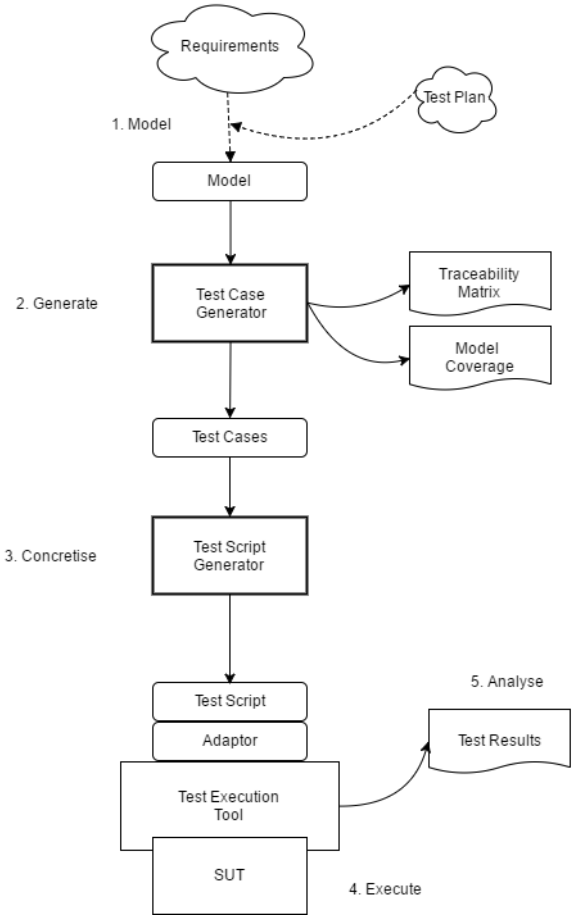

Figure 1: Model-based testing process [15].

2.2 Systematic literature mapping

A systematic mapping review of literature was performed [16], which makes possible to identify, evaluate, and interpret the current status of literature with respect to the automation in the generation of tests cases starting from conceptual models without bias from the authors of this work. To this end, the following question was formulated:

RQ: What are the existent techniques for the automatic generation of tests cases based on models?

In this literature review, digital libraries were used, since there was no physical access to libraries relating to the theme of model-based testing, and digital libraries provides search tools that speed up the literature searching process. The selected digital libraries were IEEE Xplore, ACM and Springer due to the main testing conferences are indexed in these digital libraries.
In order to obtain a significant number of results, a search string was formulated using the keywords of the research question and their synonyms; and the operators AND and OR were used to join the terms.
Once the search string was executed in the digital libraries, the exclusion criteria were applied for leave out prologues, introductions, systematic reviews, theses, books, and grey literature. In addition, the inclusion criteria were applied to select candidate papers. The inclusion criteria correspond to papers in Spanish or English language, and that include techniques or methodologies for that allows to perform software testing based on models and its automation.
The search process was performed on July of 2015, and it yielded a total of 876 articles (see Table 1). In the first stage of revision, which corresponds to reading the title and abstract of the articles retrieved by the digital libraries, 52 articles were selected after the application of the inclusion and exclusion criteria detailed above. Later, a complete reading of each article was performed to these 52 candidate articles, resulting in 20 articles selected.

Table 1: Number of articles retrieved in the systematic review

Library	Number of articles retrieved	Number of articles first read	Number of articles after fully read
IEEE	550	15	9
ACM	203	28	10
Springer	123	9	1
Total	**876**	**52**	**20**

For each selected article, the models used to generate the test cases, the modeling standard and the tools that support the test case generation were extracted. This information is presented in Table 2, and when this information has not been specified or it cannot be extracted from the article, we use the – symbol.

Table 2: Selected articles in the systematic mapping literature review

Ref.	Models	Standard	Tools
[17]	-	UML	JULE
[18]	Data model and Class model	UML	Conformiq Qtronic and EAST Test Runner
[19]	State machines and State diagram	-	PerformCharts and Condado
[20]	Data model	-	-
[21]	Petri nets and State diagram	-	MISTA
[22]	State transition diagram	UML	TestMaster
[23]	-	UML	-
[24]	Use cases	-	TarGet
[25]	Activity diagram and Use cases	UML	LoadRunner and PLeTsPerf
[26]	Sequence diagram, State machines, and Activity diagram	UML	-
[27]	State transition diagram	UML	-

Ref.	Models	Standard	Tools
[28]	State transition diagram	UML	Conformiq Qtronic and SpecExplorer
[29]	Use cases	UML	-
[30]	Sequence diagram	UML	Eclipse (Plugin EMF) and UML2 Tools
[31]	Interaction diagram	UML	-
[32]	Class diagram	UML	LEIRIOS Smart Testing
[33]	Class diagram, Object diagram, and State machine	UML	LEIRIOS and Test Designer Tool
[34]	Sequence diagram	UML	-
[35]	State transition diagram	UML	-
[36]	State transition diagram	UML	-

Regarding the information obtained, the majority of articles introduces the use of a single model for the generation of test cases and only six articles present the use of more than one model to generate the test cases. Four of them use two models [18], [19], [21], [25], and two use three models [26] [33]. Different types of models represent different views of the software, so that a greater number of models would make it possible to evaluate the software from their different views, thus providing greater coverage of the software. In two articles authors do not mentioned the models used [17] [23], which difficult the generalization of these proposals (see Fig.2).

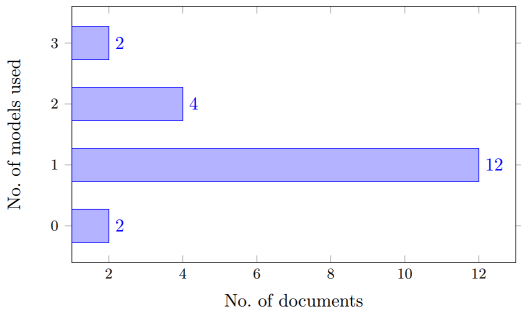

Figure 2: Number of models used by the MBT techniques selected in the mapping.

With respect to the diagrams used, most of the articles use state transition diagrams or state machines. Later, the most used diagrams correspond to interaction diagrams or sequence diagrams. It is important to highlight that 80% of the model-based testing proposals presented the use of UML standard for the models used as input in the generation of test cases. The remaining 20% presents other models, such as Petrinets [21], data models [18], and objects diagrams [33] (see Fig.3).

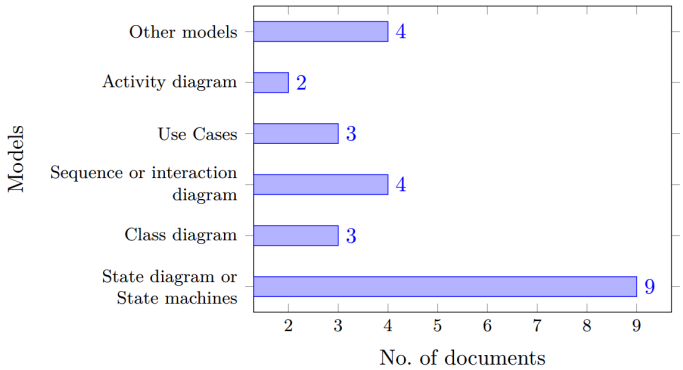

Figure 3: Models used by the MBT techniques selected in the mapping.

Finally, to answer the research question, *RQ: What are the existent techniques for the automatic generation of tests cases based on models?*, we can establish that there exist techniques that allow to generate test cases from UML models, most of which use a single model to generate test cases, such as [20] uses the data model, [22] uses state diagrams, [29] uses use cases, [30] uses sequence diagram, and [32] uses a class diagram. In addition, although there are tools to generate the test cases based on models, for example Selenium [37], SpecExplorer [28], TestMaster [22], PerformCharts [19], and MISTA [21], the documentation of these tools is scarce, without giving more details of how to perform the test case generation or the clearly state the coverage criteria that they are using. For this reason, in this work we present a MBT tool called TCGen that details the coverage criteria used, and that generates test cases using more than a single conceptual model.

3 TCGEN DESIGN

This section introduces the Test Case Generation Tool – called *TCGen* – that automatically generates test cases based on models by using several testing criteria. TCGen is a model-based testing tool that automatically creates the testing model from the UML models and later automatically generates abstract test cases. TCGen uses the UML class diagram and the UML state transition diagram [13] to generate the test cases.

The testing criteria allow to design black-box test suites, that are independent of the SUT code. At this point it is important to mention that coverage in MBT is related to coverage in the model, which is complementary to code coverage [15]. In general terms, the model coverage has appeared as the systematization of good practices that has been done for code coverage. For this reason, in order to correctly test a software product, it is important to use both, model coverage and code coverage. The major concern of MBT tools is to maximize the model coverage taking into account that the models represents the abstraction of the system and the generation of test cases at this level expresses the testing goals of the system behaviour.

TCGen is a MBT tool that is related to model coverage. In particular, TCGen is focused on structural model coverage, i.e. the coverage is measured as the part of the model that is covered by the test cases generated. For structural model coverage, the following criteria has been defined in literature [15]: Association-End Multiplicity (AEM), Generalization (GN), Class Atrribute (CA), All-Paths Sequence diagram, All-states, All-Configurations, All-Transitions, All-Transitions-Pair, All-Loop-Free-paths, All-One-Loop-Paths, All-Round-Trips, and All-paths. Above, the testing criteria supported by TCGen are explained.

3.1 Testing criteria supported by TCGEN

Since TCGen uses as input for the test case generation the class diagram and the state transition diagram, All-Paths Sequence diagram is a testing criteria not supported in this version of the tool. Regarding the testing criteria that related to the class diagram, all of them are supported by TCGen, i.e. AEM, GN, and CA are supported. Regarding the testing criteria related to the state transition diagram, All-Paths criterion means that each path must be used at least once, so that it is not practical to use since the state transition diagrams can have loops. For this reason, this criterion is not supported by TCGen. The All-round-trips criterion requires only one iteration of a trip for all the states and transitions, therefore this criterion is not supported by TCGen since can be tested using other criteria. The same occur with the All-configurations criteria, that can be solved with All-states and All-transitions criteria. Therefore, TCGen implements eight testing criteria, three of them are based on the XML representation of the class diagram, four of them are based on the XML representation of the state transition diagram, and the remaining criterion is related to data values.

The AEM (Association-end multiplicity) testing criterion evaluates the cardinality of the associations, in other words, an association must be instantiated with the minimum value of related items, the maximum value of related items and with one or more intermediate values between the range of minimum and maximum value. To generate the test cases, it is necessary to parse the XML representation of the class diagram, and for each pair of associated classes the cardinality is evaluated by creating a set of multiplicities to the maximum, minimum, and one intermediate value. Then, by means of the Cartesian product, a set of valid and invalid configurations is created. Fig. 4 shows the pseudo-code of the algorithm used to generate test cases with the AEM criterion.

```
  Data: Class Diagram
  Result: Test Cases which satisfy the criteria
1 foreach pair of associated classes do
2 |  Get all the associated classes cardinalities;
3 |  Create a multiplicity set by selecting the minimum,
  |   maximum and an intermediate value from both
  |   cardinalities;
4 |  Make cartesian product between both multiplicity sets
  |   to get the valid configurations set, these values will
  |   indicate the number of instances to create of each
  |   associated class;
5 |  Create configurations set with invalid values (out of
  |   cardinal domain) to test the failures;
6 end
```

Figure 4: Pseudo-code of the algorithm for AEM criterion.

The GN (Generalization) testing criterion provides an instance for each specialization and generalization that is defined in the class diagram. To generate test cases for each parent class, the classes that inherit from this are instantiated. Then, for each child class, the methods inherited from the parent class and the classes associated with the parent class are called. Fig. 5 shows the pseudo-code of the algorithm used to generate these test cases.

```
  Data: Class Diagram
  Result: Test Cases which satisfy the criteria
1 foreach Parent Class do
2 |  Instance class;
3 |  Instance all its child classes;
4 |  foreach Child Class do
5 |  |  Call all parent class methods and child class
  |  |   methods;
6 |  end
7 end
```

Figure 5: Pseudo-code of the algorithm for GN criterion.

The CA testing criterion (Class Attribute) generates different combinations of values for the attributes of each class specified in the class diagram. To generate the test cases, a set of significant values using a DataFactory [38] within the software domain is generated for each attribute. Then, a Cartesian product of the set of values of the attributes of each class is calculated. Therefore, each tuple will be a test case for the CA testing criterion. Fig. 6 shows the pseudo-code of the algorithm that implements the CA criterion.

```
  Data: Class Diagram
  Result: Test Cases which satisfy the criteria
1 foreach Class Attribute do
2 |  Generate significant values set using DataFactory [1]
  |   inside the domain;
3 |  Make cartesian product of class attributes values sets;
4 |  Each resulting tuple will be a test case;
5 end
```

Figure 6: Pseudo-code of the algorithm for CA criterion.

The All-States criterion specifies that each state of the model must be visited at least once, in order to doesn´t have states without use. To do this, TCGen receives an XML file with the information of the state transition diagram, from where the initial state and their transitions is selected. Then, the states are visited and they are marked as visited. If all the states have been visited, the criterion finishes successfully. If there are no more states achievable and there still states to visit, these states are reported as they cannot be achieved. Fig. 7 shows the pseudo-code of the algorithm for the all-states criterion.

```
  Data: State Diagram
  Result: Test Cases with All-State Criteria.
1 Criteria start;
2 Get initial state;
3 Set state as visited;
4 Get all transitions from initial state;
5 Stack gotten transitions;
6 while exist no visited states and exist transitions in the
   stack do
7  | Unstack a transition;
8  | Get destination state for that transition;
9  | Set state as visited;
10 | if state does not have transitions then
11 |  | Store test case;
12 | else
13 |  | Get all transitions from current state;
14 |  | Stack all the transitions starting from those that
   |  |  lead to a state already visited;
15 | end
16 end
17 if exist no visited states then
18 | report no reached states;
19 end
20 Criteria End;
```

Figure 7: Pseudo-code of the algorithm for All-States criterion.

The All-Transitions criterion determines that each transition of the model must be travelled at least once. From the XML file that contains the information of the state transition diagram, the initial state and their transitions are selected. Then, the transitions are run by marking them as executed. If all transitions executed are the universe of transitions in the diagram, the criterion successfully finished. Otherwise, it is important to notify the transitions that could not be executed. The pseudo-code for the algorithm that implements All-Transitions criterion is presented in Fig. 8.

```
  Data: State Diagram
  Result: Test Cases with All-Transitions Criteria
1 Criteria Start;
2 Get initial state;
3 Get transitions from initial state;
4 Stack all gotten transitions;
5 while exist transitions in the stack do
6  | Unstack a transition;
7  | Get destination state for that transition;
8  | Set transition as executed;
9  | if state does not have transitions then
10 |  | Store Test Case;
11 | else
12 |  | Get transitions from current state;
13 |  | Stack transitions;
14 | end
15 end
16 if exist not executed transitions then
17 | report not executed transitions;
18 end
19 Criteria End;
```

Figure 8: Pseudo-code of the algorithm for All-Transitions criterion.

The All-Transition-Pairs criterion defines that each pair of adjacent transitions in the model must be travelled at least once. From the XML representation of the state transition diagram, all the states are obtained. For each state, all the incoming and outgoing transitions are obtained. The test cases are created with each pair of incoming – outgoing transitions. If all pairs of transitions are run at the end, the criterion ends correctly. But, if there are pairs that were not visited, they are notified. At this point it is important to note that this testing criteria can be useful when modifications are performed to

the system, for instance including new functionality, in order to generate the test cases just for the new transitions. The pseudo-code for the algorithm that implements All-Transitions-Pair criterion is presented in Fig. 9.

```
  Data: State Diagram
  Result: Test Cases with All-Transitions Pair Criteria
 1 Criteria Start;
 2 foreach estado do
 3 |  Get all incoming transitions;
 4 |  Get all outgoing transitions;
 5 |  foreach Incoming Transition do
 6 |  |  foreach Outgoing Transition do
 7 |  |  |  Create test case with a pair of incoming and
   |  |  |   outgoing transition;
 8 |  |  |  Set pair of transitions as executed
 9 |  |  end
10 |  end
11 end
12 if exist not executed adjacent transitions pair then
13 |  Report not executed transitions pair;
14 end
15 Criteria End;
```

Figure 9: Pseudo-code of the algorithm for All-Transitions-Pair criterion.

The All-Loop-Free-Path criterion specifies that every path loop free must be visited at least once. A path is free of cycles if you do not have repetitions of any configuration. From the XML of the state transition diagram, you get the initial state and their transitions. Then, the transitions are executed. If the transition comes to a state already visited, then this case is discarded; and if the transition arrives at a state without transitions, then saves the case. The approach ends when there are no more transitions in the stack to run. Fig. 10 presents the pseudo-code for the algorithm that implements All-Loop-Free-Path criterion.

```
  Data: State Diagram
  Result: Test Cases with All-Loop-Free-Path Criteria
 1 Criteria Start;
 2 Get initial state;
 3 Set state as visited;
 4 Get all transitions from initial state;
 5 Stack gotten transitions;
 6 while exist transitions in stack do
 7 |  Unstack a transition;
 8 |  Get destination state for that transition;
 9 |  if state has not been visited then
10 |  |  Set state as visited;
11 |  |  if state does not have transitions then
12 |  |  |  Store Test Case;
13 |  |  else
14 |  |  |  Get transitions from current state;
15 |  |  |  Stack transitions;
16 |  |  end
17 |  end
18 end
19 Criteria End;
```

Figure 10: Pseudo-code of the algorithm for All-Loop-Free-Path criterion.

The criterion One Value receives an XML file with the information of the class diagram. For each attribute in a class, it is necessary to get the domain of the attribute and generates a value within this domain by using a DataFactory [38]. Fig. 11 shows the algorithm that implements One Value criterion.

Using the algorithms presented in this section, TCGen clearly state the testing criteria used for to automatically generates the test cases starting from the class diagram and the state transition diagram specified for a software system. This is a big difference regarding the tools of the state of the art that do not specify the testing criteria used to generate test cases starting from conceptual models neither explain how it is possible to generate test cases starting from conceptual models.

```
  Data: Class Diagram
  Result: Test Cases that satisfy the criteria
1 foreach Class Attribute do
2 |  Get attribute domain;
3 |  Generate a value inside domain using DataFactory [1];
4 end
```

Figure 11: Pseudo-code for One Value criterion algorithm.

3.2 Design of test cases

The TCGen technique needs four steps to generate the test cases: (1) generate the model, (2) identify structure and attributes of the models, (3) generate the abstract test cases and (4) generate an executable script.
For the first step, i.e. to generate the model, Eclipse Papyrus [39] will be used. Papyrus is a free tool that allows create UML models, and then save them in an XML file that contains all the specification of the UML models. This step is not automated, it is necessary that the software engineer understand the requirements of the system and specify the solution in UML models in order to satisfy the requirements.
In the second step, TCGen read the XML representation of the UML conceptual models and it identifies the classes, attributes, methods, associations, states, and transitions. This step is performed automatically, i.e. the software engineer just need to enter the XML representation of the UML models in the TCGen tool.
In the third step, TCGen generates the abstract test cases in an XML file. This XML file have the specification of the testing criteria used with the tag <Criterion name="NameOfCriterion">, in addition to the parameters for each criterion selected to generate the test cases. With this, the abstract test cases are generated automatically without the intervention of the software engineer.
Later, in a four step, it is important to generate the executable scripts for the abstract test cases, for which it is possible to use the Selenium tool [9]. In this step, the software engineer translate the abstract test cases to concrete test cases written in the same programing language that the software has been developed in order to automates the execution of the test cases generated.
Fig. 12 shows an example of a class model created for a family pets management system in the Papyrus tool, which allows the registration of cats or dogs as pets of a family.

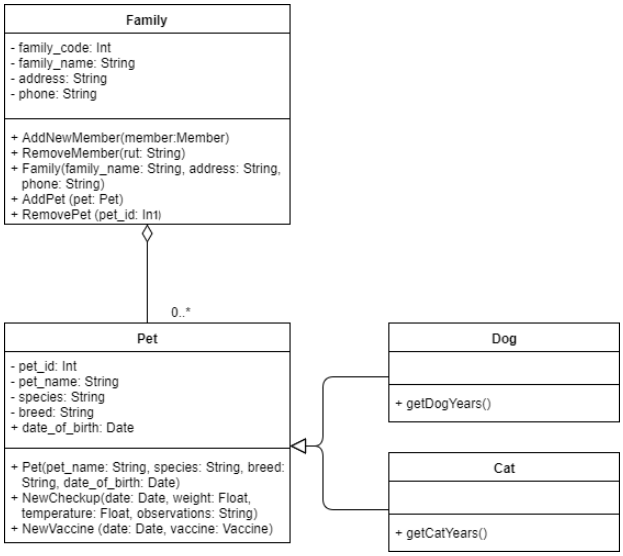

Figure 12: Example class model for a family pets management system.

For the example model shown in Fig. 12, the AEM, GN, and CA testing criteria are used to generate the abstract test cases by TCGen. In Fig. 13 is presented an extract of the XML file generated for the AEM criterion. In this figure, it is possible

to observe that in the XML is clearly indicated the testing criteria and the values of the attributes for the associated classes (*Family* and *Pet*) with a valid cardinality of 1 to 0, and 1 to 1.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<suite name="Test Sisgevet" verbose="10">
    <parameter name="selenium.host" value="localhost"/>
    <parameter name="selenium.port" value="3737"/>
    <parameter name="selenium.browser" value="firefox"/>
    <parameter name="selenium.url" value="http://www.sisgevet.com"/>
    <test>
        <Criterio nombre="AEM">
            <Clase nombre="Family">
                <Agregacion claseDestino="Pet" claseOrigen="Family" nombre="PetFamily" valida="true">
                    <Multiplicidad cardinalidad="1,0">
                        <Clase nombre="Family">
                            <Atributo nombre="family_code" tipo="int" valor="1968649244"/>
                            <Atributo nombre="family_name" tipo="String" valor="when"/>
                            <Atributo nombre="address" tipo="String" valor="as in"/>
                            <Atributo nombre="phone" tipo="String" valor="who"/>
                        </Clase>
                    </Multiplicidad>
                    <Multiplicidad cardinalidad="1,1">
                        <Clase nombre="Family">
                            <Atributo nombre="family_code" tipo="int" valor="1852359517"/>
                            <Atributo nombre="family_name" tipo="String" valor="trying"/>
                            <Atributo nombre="address" tipo="String" valor="white"/>
                            <Atributo nombre="phone" tipo="String" valor="wrong"/>
                        </Clase>
                        <Clase nombre="Pet">
                            <Atributo nombre="pet_id" tipo="int" valor="931933135"/>
                            <Atributo nombre="pet_name" tipo="String" valor="so interest"/>
                            <Atributo nombre="species" tipo="String" valor="list"/>
                            <Atributo nombre="breed" tipo="String" valor="in throw"/>
                            <Atributo nombre="date_of_birth" tipo="EDate" valor="1992-5-13"/>
                        </Clase>
                    </Multiplicidad>
                    <Multiplicidad cardinalidad="1,5">
                        <Clase nombre="Family">
                            <Atributo nombre="family_code" tipo="int" valor="643908251"/>
                            <Atributo nombre="family_name" tipo="String" valor="in find"/>
                            <Atributo nombre="address" tipo="String" valor="words"/>
                            <Atributo nombre="phone" tipo="String" valor="computer"/>
                        </Clase>
                        <Clase nombre="Pet">
                            <Atributo nombre="pet_id" tipo="int" valor="1759108239"/>
                            <Atributo nombre="pet_name" tipo="String" valor="fire"/>
                            <Atributo nombre="species" tipo="String" valor="always"/>
```

Figure 13: Example of abstract test case generated for AEM criterion (extract).

For the GN criterion, where the child classes are instantiated, it is possible to observe the testing criterion in the XML file, and the attributes and methods inherited by the child class from the parent class (see Fig.14).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<suite name="Test Sisgevet" verbose="10">
    <parameter name="selenium.host" value="localhost"/>
    <parameter name="selenium.port" value="3737"/>
    <parameter name="selenium.browser" value="firefox"/>
    <parameter name="selenium.url" value="http://www.sisgevet.com"/>
    <test>
        <Criterio nombre="GN">
            <Clase nombre="Pet">
                <Clase nombre="Pet" tipo="padre">
                    <Atributo nombre="pet_id" tipo="int" valor="897279080"/>
                    <Atributo nombre="pet_name" tipo="String" valor="when"/>
                    <Atributo nombre="species" tipo="String" valor="as in"/>
                    <Atributo nombre="breed" tipo="String" valor="who"/>
                    <Atributo nombre="date_of_birth" tipo="EDate" valor="1924-7-12"/>
                </Clase>
                <Clase nombre="Dog" tipo="hija">
                    <Metodo clase="Pet" name="getDogYears"/>
                    <Metodo clase="Pet" name="Pet">
                        <Argumento nombre="pet_name" tipo="String" valor="trying"/>
                        <Argumento nombre="species" tipo="String" valor="white"/>
                        <Argumento nombre="breed" tipo="String" valor="wrong"/>
                        <Argumento nombre="date_of_birth" tipo="EDate" valor="1947-9-22"/>
                    </Metodo>
                </Clase>
                <Clase nombre="Cat" tipo="hija">
                    <Metodo clase="Pet" name="getCatYears"/>
                    <Metodo clase="Pet" name="Pet">
                        <Argumento nombre="pet_name" tipo="String" valor="so interest"/>
                        <Argumento nombre="species" tipo="String" valor="list"/>
                        <Argumento nombre="breed" tipo="String" valor="in throw"/>
                        <Argumento nombre="date_of_birth" tipo="EDate" valor="1983-4-16"/>
                    </Metodo>
                </Clase>
            </Clase>
        </Criterio>
```

Figure 14: Example of abstract test case generated for GN criterion.

The XML file for the CA testing criterion, where different combinations of attribute values are tested is presented in Fig. 15. In this figure, it is possible to observe the testing criterion and the different instantiations for the class family. For each instantiation, data values are provided for each attribute.

```
Total Time: 0.02 seconds.

Number of Test Cases: 50

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<suite name="Test Sisgevet" verbose="10">
    <parameter name="selenium.host" value="localhost"/>
    <parameter name="selenium.port" value="3737"/>
    <parameter name="selenium.browser" value="firefox"/>
    <parameter name="selenium.url" value="http://www.sisgevet.com"/>
    <test>
        <Criterio nombre="CA">
            <Clase nombre="Family">
                <Instancia>
                    <Atributo nombre="family_code" tipo="int" valor="1445922537"/>
                    <Atributo nombre="family_name" tipo="String" valor="numbers"/>
                    <Atributo nombre="address" tipo="String" valor="ending"/>
                    <Atributo nombre="phone" tipo="String" valor="found"/>
                </Instancia>
                <Instancia>
                    <Atributo nombre="family_code" tipo="int" valor="1445922537"/>
                    <Atributo nombre="family_name" tipo="String" valor="numbers"/>
                    <Atributo nombre="address" tipo="String" valor="ending"/>
                    <Atributo nombre="phone" tipo="String" valor="of room"/>
                </Instancia>
                <Instancia>
                    <Atributo nombre="family_code" tipo="int" valor="1445922537"/>
                    <Atributo nombre="family_name" tipo="String" valor="numbers"/>
                    <Atributo nombre="address" tipo="String" valor="room"/>
                    <Atributo nombre="phone" tipo="String" valor="found"/>
                </Instancia>
                <Instancia>
                    <Atributo nombre="family_code" tipo="int" valor="1445922537"/>
                    <Atributo nombre="family_name" tipo="String" valor="numbers"/>
                    <Atributo nombre="address" tipo="String" valor="room"/>
                    <Atributo nombre="phone" tipo="String" valor="of room"/>
                </Instancia>
                <Instancia>
                    <Atributo nombre="family_code" tipo="int" valor="1445922537"/>
                    <Atributo nombre="family_name" tipo="String" valor="throw"/>
                    <Atributo nombre="address" tipo="String" valor="ending"/>
                    <Atributo nombre="phone" tipo="String" valor="found"/>
                </Instancia>
                <Instancia>
```

Figure 15: Example of abstract test case generated for CA criterion (extract)

An extract of the XML file for the All-transitions testing criterion is presented in Fig. 16, where all transitions are travelled until arrive to a target state, and if there are transitions that do not arrive to a target state they are reported.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<suite name="Test Sisgevet" verbose="10">
  <parameter name="selenium.host" value="localhost"/>
  <parameter name="selenium.port" value="3737"/>
  <parameter name="selenium.browser" value="firefox"/>
  <parameter name="selenium.url" value="http://www.sisgevet.com"/>
  <test>
    <Criterio nombre="All-Transitions">
      <Caso estadoFinal="FinalState1">
        <Transicion destino="State1" nombre="Tran1" origen="Initial1"/>
        <Transicion destino="State3" nombre="Tran4" origen="State1"/>
        <Transicion destino="FinalState1" nombre="Tran7" origen="State3"/>
      </Caso>
      <Caso estadoFinal="FinalState3">
        <Transicion destino="State1" nombre="Tran1" origen="Initial1"/>
        <Transicion destino="State3" nombre="Tran4" origen="State1"/>
        <Transicion destino="State3" nombre="Tran5" origen="State3"/>
        <Transicion destino="State4" nombre="Tran6" origen="State3"/>
        <Transicion destino="State5" nombre="Tran9" origen="State4"/>
        <Transicion destino="FinalState3" nombre="Tran10" origen="State5"/>
      </Caso>
      <Caso estadoFinal="FinalState2">
        <Transicion destino="State1" nombre="Tran1" origen="Initial1"/>
        <Transicion destino="State3" nombre="Tran4" origen="State1"/>
        <Transicion destino="State3" nombre="Tran5" origen="State3"/>
        <Transicion destino="State4" nombre="Tran6" origen="State3"/>
        <Transicion destino="State5" nombre="Tran9" origen="State4"/>
        <Transicion destino="FinalState2" nombre="Tran8" origen="State4"/>
      </Caso>
      <TransicionesNoVisitadas/>
```

Figure 16: Example of abstract test case generated for All-Transitions criterion (extract)

The test cases presented above correspond to the abstract test cases automatically generated by TCGen starting from the class model of a management system for family pets presented in Figure 12. These test cases are generated without the intervention of the software engineer in few seconds, so that the time needed to design test cases is reduced considerably with regard to thinking about the test cases from scratch.

3.3 Implementation of TCGen

TCGen is a tool that was implemented using IntelliJ IDEA [40], which is a framework for the development of Java applications that facilitates the completion, refactoring and debugging of code. The TCGen tool can be freely downloaded from https://cperezg@bitbucket.org/cperezg/tcgen.git.
To develop the graphical interface of TCGen, JavaFX library was used [41] since it allows to create desktop applications, mobile applications, web applications, etc. Fig. 17 shows the resulting interface for the TCGen tool, which is divided into 6 parts.

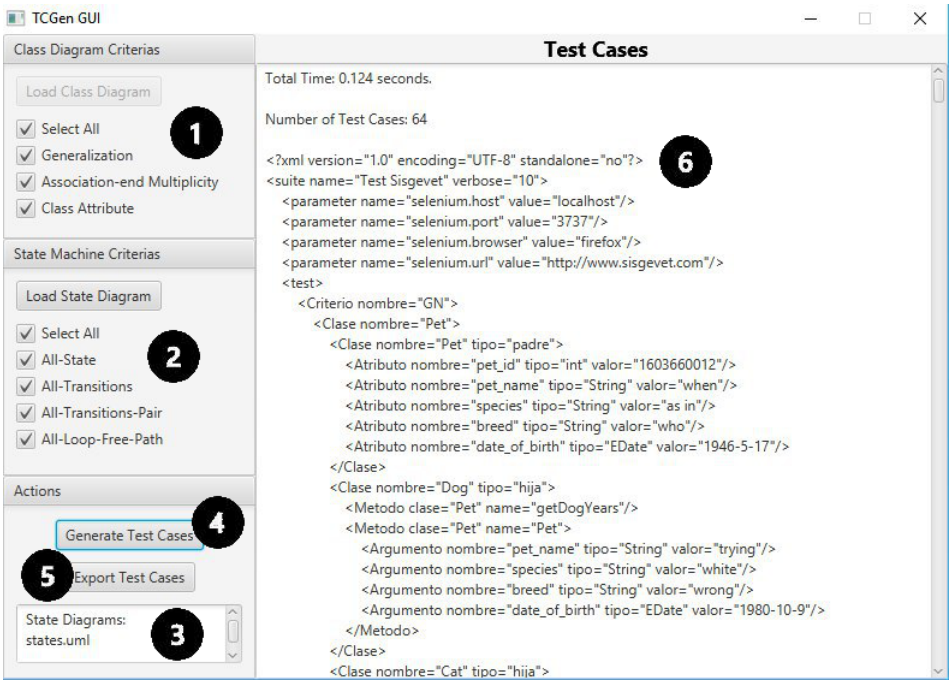

Figure 17: TCGEN graphical user interface.

In the top left corner, it is located the panel for the class diagram (number 1 in Fig. 17). In this panel, there is a button to load a class model, which will open a file selection window to select the model. Then, the checkbox to select the criteria to generate the test cases are enabled. The panel for the state transition diagram is located at number 2 of Fig. 17. This panel has a button for the selection of the model, and when the model is selected, the checkbox with the testing criteria are enabled.
In contrast to the class diagram panel, in the state transition diagram panel it is possible to select multiple state transition diagrams, which will be shown in number 3 of Fig. 17 (which displays the list of state transition diagrams opened). This is because for a software solution one class diagram is used to specify the entire solution, but it is possible to have several state transition diagrams to represent the valid lives of the objects related to the different classes in the solution.
The button *Generate Test Cases* (see number 4 in Fig. 17) initiates the process of creating test cases for each selected criterion. The result will be displayed on the right side of TCGen (see number 6 in Fig. 17). This panel will display the number of cases generated, the total time that TCGen takes in generating the test cases, and the test cases generated. The

button *Export Test Cases* (see number 5 in Fig. 17) will open a window for saving an XML file with the test cases generated.

3.4 Comparison of TCGen with other MBT tools

This section has presented TCGen, an open source tool that puts in practice the model-based testing by using UML models as input in the automatic generation of abstract test cases. Table 3 shows a comparison of TCGen regarding other tools that we found in our literature review.

Table 3: Comparison of TCGen with other tools

Tool	Input models	testing criteria	Type of test case	Open source
JULE [17]	-	-	-	-
Conformiq Qtronic [18]	Behavioural and state diagrams	-	abstract	no
EAST Test runner [18]	-	-	concrete	-
Condado [19]	State transition diagram	-	-	-
Mista [21]	Petri nets and state transition diagrama	Coverage criteria	-	yes
TestMaster [22]	State transition diagram	-	concrete	no
TaRGet [24]	Use case diagram	-	-	-
PLeTsPerf [25]	Use case diagram and activity diagram		concrete	-
SpecExplorer [28]	State transition diagram	-	Concrete	no
LEIRIOS Smart Testing [32], [33]	Class diagrams, objects diagrams, and state transition diagrams	-	Abstract	-
TCGen	**Class diagram and state transition diagram**	**Structural model coverage**	**abstract**	**yes**

As is shown in Table 3, there are some tools that do not specify the type of diagram used as input in the process of test case generation, such as JULE and EAST Test runner, which impede their use in other projects different than the projects reported at [17] and [18], respectively.

Regarding the models used to generate test cases it is possible to observe that the majority of tools uses state transition diagrams, and just one tool also uses the class diagram as input in the generation of test cases [32], [33]. As we stated before, it is important to mention that using just one view of software provokes that the coverture of the generated test cases is not enough to validate the requirements or to find failures in the software. We advocate that since a software product has different complementary views (structural, behavioral, and interaction) that must be specified to completely describe a software system, therefore, test cases should be generated using models related to the different views, in order to test the software holistically. For this reason TCGen uses as input models the class diagram and the state transition diagram, which represents the structural and behavioral views of software. Nevertheless, the generation of test cases that hollistically test the software systems is a still big challenge of MBT approaches.

Regarding the testing criteria, TCGen and Mista identifies the testing criteria used to generate the test cases. This is important to understand the test cases generated and the model coverage that can be achieved using the test cases generated by these tools.

Regarding the type of test case generated, in some cases are generated as concrete test cases, i.e. test cases written in the programing language of the software system. The concretization of test cases should help in the execution of test cases. Nevertheless, it is important to have abstract test cases in order to facilitate the understanding of the design of test cases. For that reason, this version of TCGen generates abstract test cases.

Regarding the license agreement for these MBT tools, only TCGen and Mista are open source tools, which make possible different companies to use them without paying a license fee, and also give the opportunity to other researchers to programme extensions of these tools.

4 EXPERIMENTAL EVALUATION OF TCGEN

In order to assess the functionality of TCGen, a quasi-experiment was performed in a controlled environment, which was designed and implemented following the guidelines presented at [42, 43]. The purpose of the quasi-experiment is to evaluate the coverage and efficiency in the generation of test cases by TCGen in comparison with manual generation of the test cases.

The quasi-experiment consists of two cases of similar difficulty, where two groups of people of similar knowledge must complete the class diagrams and state transition diagrams using Papyrus, in order to finally generate test cases. One of the groups must generate the test cases manually and the other using TCGen.

The independent variable corresponds to the *method* for generating the test cases: manual or automatic using TCGen. The dependent variables correspond to the *coverage* and *efficiency*. Coverage is calculated according to the number of test cases generated for different testing objects. Efficiency is calculated with the time used in the generation of test cases.

For the realization of the experiment, students of the career of Informatics and Telecommunications Engineering at Diego Portales University were selected. Students meet the requirement of having passed the Software Engineering course, where they acquire the foundations and knowledge of conceptual modeling and software testing. The participation in the activity was voluntary, without a qualification associated to their participation. This was decided in order to subjects were not afraid to honestly evaluate the TCGen tool. Since the participation in the study was voluntary, we do not recruit many participants, but we aware that the participants will perform the tasks consciously. With a total of 13 participants, two working groups were formed, six and seven members, according to how they were sitting in the classroom.

The cases correspond to a post office management system and a computers' repairing management system. The post office management system allows the interaction of the practitioners of the post office and the citizens. Each type of user has different permissions in the system, and to access the functionality they must be logged into the system. The practitioners of the post office can be management people or postal man people, which must specify the days and hours of their working week. The citizens must register the address and the contacting e-mail. The main functionality of this system is the registration of a postal card, the registration of a package, the reception of a postal or package, and the tracking of the packages in their different states: received, in road, and delivered.

The computer's repairing system allows the interaction of practitioners of the computers repairing company and the clients that leave their computers in order to be repaired. Each type of user has different permissions and functionality in the system, so that they must be logged into the computer's repairing system in order to access the functionality. The clients must register the problem of the computer and the contacting information, such as phone number and e-mail. The registration of a computer to be repaired must content the type of the computer: PC or notebook, in addition to the information specific for each type of computer. PC must be registered with of brand or hand-built, operative system, ram, processor and size of hard disc. The notebooks must be registered with brand, model, and screen size. The main functionality of this system is the registration of computers to be repaired, the tracking of the reparations of each computer, and the return of the repaired computer to the client.

It should be noted that both cases, the office management system and the computer's repairing system, were reviewed by the authors of this work on repeated occasions in order to reduce the risk of these cases were not understood by the subjects, and also in order to state that they had a similar level of difficulty. Moreover, the experiment was piloted with practitioners prior to the execution with the subjects in order to adjust the times allocated to the activities of the assessment and the material of the experiment.

Table 4 shows the activities carried out in the experiment by each group of subjects (the one that generates the test cases manually and the one that generates the test cases automatically by using TCGen) and the duration of each activity.

Table 4: Activities in the TCGEN quasi-experiment

Activity	Group1	Group2
Introduction and motivation	12 minutes	12 minutes
Read the case	7 minutes	8 minutes
Specification of diagrams	100 minutes	80 minutes
Test case generation	0,108 seconds	30 minutes
Total	**2 hours (120 minutes)**	**2 :10 hours (130 minutes**

Once the objective of the evaluation was explained, each group read the case and made the class diagrams and the state transition diagrams in the Papyrus tool. Subsequently, each group generated test cases. In both groups, the diagrams and the test cases were specified all-together. In the case of Group1, it was performed through the use of TCGen, and the Group2 generate the test cases manually. In the case of Group2, they had 30 minutes to generate the greatest amount of test cases as possible.

The participants of the Group1 recognized 8 classes for the post office system: User, Customer, Official, Work_Day, Package, Reception, and Track. Group1 obtained 702 test cases generated using all the criteria that are supported by TCGen in 0.108 seconds. These test cases were generated using the eight system classes and the eight testing criteria. For example, for the testing criterion CA, 354 test cases were created that cover all the attributes of all classes in the software. With this test cases, it is possible to assure the creation of all the objects specified in the UML class model. Therefore, this case meets with the full coverage of the test objects.

The participants of the Group2 recognized 5 classes for the repairing computers system: Customer, Computer, Desktop, Notebook, and Income. For this diagram, Group2 obtained 16 test cases in 30 minutes. This group focused on the verification of the attributes, i.e. the CA criterion that is also covered by TCGen. The generated test cases were performed for the Customer class (12 test cases) and for the Income class (4 test cases). Here it is clearly notice that the generated test cases have not been covered all the test objects (only 2 of the 5 classes specified in the class diagram), so that the coverage in this criterion would be as maximum of 40%. A limitation of the test cases generated manually is that these test cases cannot be automatically used by some tool to execute the test cases. Thus, more time is needed later to execute these test cases. Moreover, the remaining testing criteria supported by TCGen was not even in mind in Group2.

With the results obtained in the quasi-experiment, TCGen proved to generate functional test cases with greater coverage than the coverage obtained generating test cases manually. In addition, TCGen demonstrated that the generation of test cases was more efficient than carried out manually. We are aware that there are test cases that cannot be generated by TCGen, for example, testing of temporal behaviour of the software, testing about the usage of hardware resources, or testing the usability of the software. However, with regard to the functional testing, TCGen has proven to be an alternative which facilitates the process of testing, and that can be complemented later with other testing techniques.

After the quasi-experiment, a post experimental activity was performed by the subjects that had to use TCGen. Thus, a questionnaire was applied to evaluate the ease of use and the utility of TCGen to Group1. This questionnaire has 15 questions that participants must respond by using a LIKERT scale from one to five, with one being a total disagreement, and five being a full agreement. The following questions are used to assess the ease of use:

a) I found the procedure to apply TCGen simple and easy to follow.
b) In general, I find easy-to-use TCGen.
c) I could explain TCGen easily to any person who does not know it.
d) In my opinion, TCGen is easy to use for the application that I have modeled.
e) TCGen was clear and easy to understand.
f) I am sure that I have now the skills necessary to use TCGen in practice.

In Fig. 18 it is possible to note that in general the TCGen tool was considered easy to use, but a majority neutral was identified in relation to the ease to explain the tool to another person.

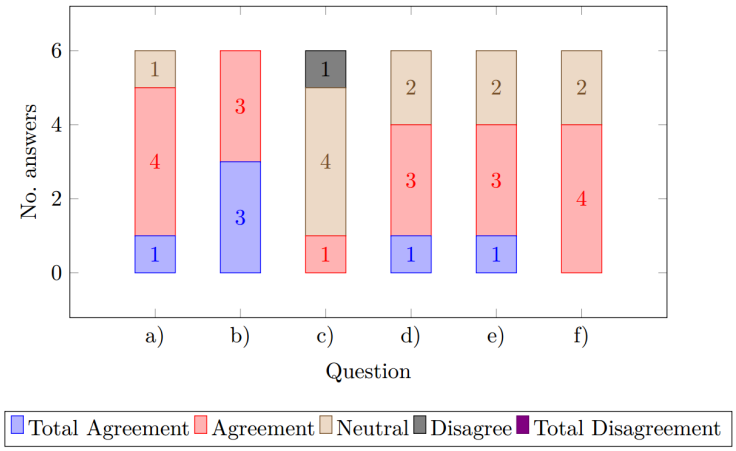

Figure 18: Results in the questionnaire to evaluate the ease-to-use of TCGEN.

In the case of the utility, the questionnaire has the following questions:

a) I think that TCGen reduces the effort required to generate test cases.
b) TCGen facilitates the tester detect faults in an application.
c) In general, I found TCGen useful.
d) In general, TCGen is practical to test the needs of users of an application.
e) Definitely, I would use TCGen to generate test cases.
f) In general, I think that TCGen provides an effective solution for the generation of test cases.
g) Using TCGen it is possible to test large applications efficiently.
h) In general, I think that TCGen is an improvement with respect to other methods for generating test cases.
i) I will try to use TCGen preferably with respect to other methods for generating test cases if I have to prove other applications in the future.

In Fig. 19 it is possible to note that in general the TCGen tool was consider to be useful. It is important to note that a large majority finds that TCGen provides an effective solution for the generation of test cases (question f).

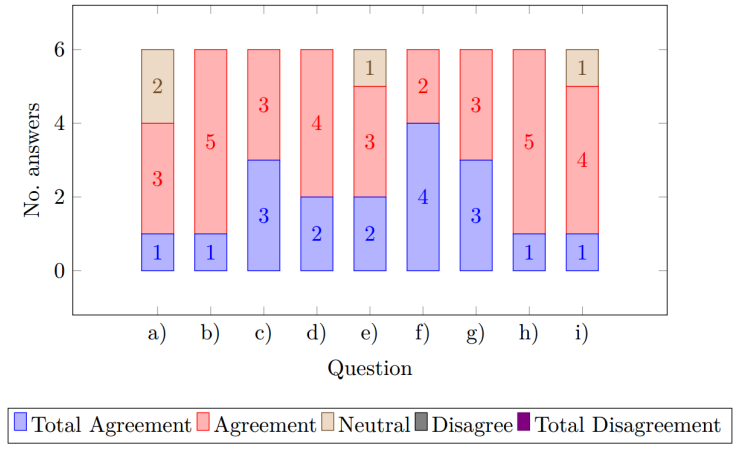

Figure 19: Results in the questionnaire to evaluate the utility of TCGEN.

In general terms, subjects stated that the use of TCGen was quick and easy to learn, and that the use of TCGen facilitated the generation of test cases quickly for a system due to its interface is simple. One of the major difficulties encountered by participants was to carry out the diagrams, since they had no experience using the Papyrus tool even though they have

experience modeling by using other tools. In addition, this difficult the participants that can explain the use of TCGen to other people.

4.1 Threats to validity

There are some threats to the validity of the evaluation of TCGen that may have biased the results of the evaluation. With regard to the *external validity*, we selected UML as the modeling language because it is widely known and it is widely used in the industry, which could facilitate the use of TCGen. With respect to the subjects, it was difficult to obtain practitioners to carry out the evaluation, so were used students that can be seen as novice professionals as they were in the final year in their career.
With regard to *internal validity*, the cases used were reviewed on several occasions by the authors of this work, and they were piloted with practitioners in order to assure that they understand both cases and that the cases have similar difficulty. To avoid the effect of learning, we divided the population of subjects into two groups that used different cases and different testing techniques. Despite of these decisions, one case could be perceived as more difficult than the other case by the subjects.
The *conclusion validity* is threatened by the small number of subjects who conducted the study, which impede to applied statistical test in order to generalize the results obtained in the evaluation of TCGen.
Finally, the *construction validity* is threatened because we choose the Papyrus tool to make the diagrams and the subjects have not knowledge of using this tool in spite of that if they had similar knowledge in UML, so we may have added a confusion factor to the experiment. However, this effect occurs in both groups so that it threats in the same way the generation of test cases manually and automatically using TCGen.

5 CONCLUSION

In this work we have presented TCGen, a tool that support the testing process by the automatic generation of abstract test cases starting from conceptual models. The testing process is crucial when developing software, and its application from early stages (using conceptual models) allows to find defects in advance, when the cost to fix them is lower.
The selection of testing criteria are essential to meet the greater amount of functionality with the test cases generated. TCGen uses algorithms based on the coverage of transitions criteria, coverage of structure criteria and coverage of data criteria. The large amount of testing criteria supported by TCGen allows to generate a large number of functional tests, allowing to detect faults in the early stages of software development. In addition, TCGen facilitates the generation of test cases without requiring knowledge of functional testing strategies, which help novice testers in their work.
One of the limitations on the use of TCGen is related to the diagrams, since the class diagram and the state transition diagram need to be correctly specified in order to be used as input to generate test cases. That is to say, if you don't have the knowledge for the correct specification of diagrams, it is not possible to generate appropriated test cases. Another limitation of TCGen is related to the data factory used, because it presents the appropriate data for a limited number of concepts of a domain and they could be expanded according to other domains of application in order to generate meaningful data in the test cases.
As future work, we planned to generate a unified tool to be able to develop models, generate the test cases and run the test cases. To do this, it is necessary to develop two components: a tool for generation of models and a tool for the execution of test cases, and then unify these tools with TCGen in order to generate a complete and functional tool to support the design and testing phases of the development cycle of software systems. All the interoperability of these tools must be specified in a metamodel that comprises the conceptual constructs of modeling as well as the conceptual constructs of model-based testing, and the restrictions and transformations rules needed to accelerate the software production process. The construction of the metamodel is therefore the first step in our future work, we advocate that counting with the metamodel for model-based testing it could be possible to generate UML profiles and model restrictions including OCL rules. In addition, we plan to do more empirical studies in order to generalize the results provided by the evaluation of TCGen and to investigate the advantages of TCGen for practitioners in comparison with the techniques that they are traditionally using for software testing at industry. Moreover, there is further work related to systems in production phase, since we planned to create a strategy to capture data from the system in production and include these data in the data factory in order to facilitate the generation of meaningful test cases when these systems are modified.

References

[1] T. E. J. Vos, F. F. Lindlar, B. Wilmes, A. Windisch, A. I. Baars, P. M. Kruse*, et al.*, "Evolutionary functional black-box testing in an industrial setting," *Software Quality Journal,* vol. 21, pp. 259-288, 2013.

[2] M. Botteck and T. Deiß, "Introduction of TTCN-3 into the product development process: considerations from an electronic devices developer point of view," *International Journal on Software Tools for Technology Transfer,* vol. 10, pp. 285-289, 2008.

[3] S. A. Slaughter, D. E. Harter, and M. S. Krishnan, "Evaluating the cost of software quality," *Communications of the ACM* vol. 41, pp. 67-73, 1998.

[4] F. Lindlar, "Search-Based Functional Testing of Embedded Software Systems," Doctoral Symposium in conjunction with IEEE International Conference on Software Testing, Verification and Validation (ICST 2009), Denver, USA, 2009.

[5] P. Reales Mateo, M. Polo, J. L. Fernández Alemán, A. Toval, and M. Piattini, " Mutation Testing," *IEEE Software,* vol. 31, pp. 30-35, 2014.

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

[7] R. H. Thayer, J. B. Slaughter, B. W. Boehm, J. A. Clapp, J. H. Manley, and J. H. Burrows, "The high cost of software: causes and corrections," Proceedings of the national computer conference and exposition AFIPS '74 1974.

[8] P. Iyenghar, E. Pulvermueller, and C. Westerkamp, "Towards Model-Based Test automation for embedded systems using UML and UTP," Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on, 2011.

[9] Selenium. (last visited on January 2017). Available: http://docs.seleniumhq.org/

[10] Watir. (last visited on January 2017). Available: http://watir.com/

[11] H. P. Q. Center. (last visited on January 2017). Available: http://www8.hp.com/us/en/software-solutions/quality-center-quality-management/

[12] B. Marín, C. Gallardo, D. Quiroga, G. Giachetti, and E. Serral, "Testing of model-driven development applications," *Software Quality Journal,* pp. 1-29, 10ht Feb. 2016 2016.

[13] OMG, "Unified Modeling Language (UML) 2.4.1 Superstructure Specification " 2011.

[14] C. Pérez and B. Marín, "Generación automática de casos de prueba basados en modelos UML," XX Iberoamerican Conference on Software Engineering (CIBSE), Software Engineering Track (SET), Buenos Aires, Argentina, 2017.

[15] M. Utting and B. Legeard, *Practical Model-Based Testing - A Tools Approach*: Morgan Kaufmann, 2007.

[16] B. Kitchenham, "Guidelines for performing Systematic Literature Reviews in Software Engineering," UK, EBSE Technical ReportJuly 9 2007.

[17] P. Bunyakiati and A. Finkelstein, "The compliance testing of software tools with respect to the UML standards specification - The ArgoUML case study," ICSE Workshop on Automation of Software Test, 2009.

[18] K. Nylund, E. Östman, D. Truscan, and R. Teittinen, "Towards Rapid Creation of Test Adaptation in On-line Model-Based Testing," IEEE 35th Annual Computer Software and Applications Conference Workshops (COMPSACW), 2011.

[19] V. Santiago, A. Martins-do-Amaral, N. Vijaykumar, M. Mattiello-Francisco, E. Martins, and O. Cuesta-Lopes, "A practical approach for automated test case generation using statecharts," 2nd International Workshop on Testing and Quality Assurance for Component-Based Systems - IEEE COMPSAC Conference, 2006.

[20] Y. Yang, H. Zhang, M. Pan, J. Yang, F. He, and Z. Li, "A model-based fuzz framework to the security testing of tcg software stack implementations," International Conference on Multimedia Information Networking and Security, 2009.

[21] D. Xu, W. Xu, M. Kent, L. Thomas, and L. Wang, "An automated test generation technique for software quality assurance," *IEEE Transactions on Reliability,* vol. 64, pp. 247–268, 2015.

[22] C. Doungsa-ard, K. Dahal, A. Hossain, and T. Suwannasart, "Test data generation from uml state machine diagrams using gas," International Conference on Software Engineering Advances, 2007.

[23] Z. Wei and W. Xiaoxue, "Graph theory model based automatic test platform design," 2nd International Conference on Software Enginee- ring and Data Mining (SEDM), 2010.

[24] C. Bertolini and A. Mota, "A framework for gui testing based on use case design," Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), 2010.
[25] E. Rodrigues, R. Saad, F. Oliveira, L. Costa, M. Bernardino, and A. Zorzo, "Evaluating capture and replay and model-based performance testing tools: an empirical comparison," 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2014.
[26] M. Shirole and R. Kumar, "Uml behavioral model based test case gene- ration: a survey," *ACM SIGSOFT Software Engineering Notes,* vol. 38, pp. 1–13, 2013.
[27] M. Shirole, A. Suthar, and R. Kumar, "Generation of improved test cases from uml state diagram using genetic algorithm," 4th India Software Engineering Conference, 2011.
[28] M. Sarma, P. Murthy, S. Jell, and A. Ulrich, "Model-based testing in industry: a case study with two mbt tools," 4th India Software Engineering Conference, 2011.
[29] M. Mlynarski, B. Güldali, M. Späth, and G. Engels, "From design models to test models by means of test ideas," 6th International Workshop on Model-Driven Engineering, Verification and Validation, 2009.
[30] B. Pérez-Lamancha, P. Mateo, I. Guzmán, M. Usaola, and M. Piattini, "Automated model-based testing using the uml testing profile and qvt," 6th International Workshop on Model- Driven Engineering, Verification and Validation, 2009.
[31] A. Nayak and D. Samanta, "Model-based test cases synthesis using uml interaction diagrams," *ACM SIGSOFT Software Engineering Notes,* vol. 34, pp. 1–10, 2009.
[32] F. Bouquet, C. Grandpierre, B. Legeard, and F. Peureux, "A test generation solution to automate software testing," 3rd international workshop on Automation of software test, 2008.
[33] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting, "A subset of precise uml for model-based testing," 3rd international workshop on Advances in model-based testing, 2007.
[34] L. Wang, E. Wong, and D. Xu, "A threat model driven approach for security testing," Third International Workshop on Software Engineering for Secure Systems, 2007.
[35] P. Murthy, P. Anitha, M. Mahesh, and R. Subramanyan, "Test ready UML statechart models," ICSE Workshop on Automation of Software Test, 2006.
[36] L. Chang, H. Miao, and G. Lu, "An Implementation Framework for Optimizing Test Case Generation Using Model Checking," in *Structured Object-Oriented Formal Language and Method*, ed, 2015, pp. 3-16.
[37] R. Chen and H. Miao, "A selenium based approach to automatic test script generation for refactoring javascript code," IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS), 2013.
[38] A. Gibson. (Last visited October 2015). *Data Factory*. Available: https://github.com/andygibson/datafactory
[39] Eclipse-Papyrus, last visited on January 2017.
[40] IntelliJ IDEA. (last visited on January 2017). Available: https://www.jetbrains.com/idea/
[41] JavaFX. (last visited on January 2017). Available: http://www.oracle.com/technetwork/java/javase/downloads/sb2download-2177776.html
[42] N. Juristo and A. Moreno, *Basics of Software Engineering Experimentation*: Springer, 2001.
[43] P. Runeson and M. Host, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering Journal,* vol. 14, pp. 131–164, 2009.