Software Qual J (2017) 25:407–435
DOI 10.1007/s11219-016-9308-8

CrossMark

# Testing of model-driven development applications

**Beatriz Marín[1] · Carlos Gallardo[1] · Diego Quiroga[1] · Giovanni Giachetti[2] · Estefanía Serral[3]**

Published online: 10 February 2016
© Springer Science+Business Media New York 2016

**Abstract** Human resource management practices are key for the success of software development projects. Practices that promote knowledge sharing and organizational learning are positively related to development–effort curves, and thus software companies are looking for different alternatives oriented to promoting these practices. The model-driven development (MDD) paradigm is positioned as one of the best alternatives for reutilization of development knowledge. In particular, this paradigm considers the specification of conceptual models that can be used as input for automatic code generation to different target platforms. However, testing of applications developed through MDD solutions is still performed by the manual definition and execution of test cases by testers, which negatively impacts in the time reduction obtained from automatic code generation and the reutilization of knowledge generated during the MDD project execution. To address this issue, this paper presents a testing approach that automatically generates executable test cases for software developed by using MDD technologies.

✉ Beatriz Marín
beatriz.marin@mail.udp.cl

Carlos Gallardo
carlos.gallardo@mail.udp.cl

Diego Quiroga
diego.quiroga@mail.udp.cl

Giovanni Giachetti
giovanni.giachetti@unab.cl

Estefanía Serral
estefania.serralasensio@kuleuven.be

[1] Facultad de Ingeniería, Escuela de Informática y Telecomunicaciones, Universidad Diego Portales, Ejército 441, Santiago, Chile

[2] Facultad de Ingeniería, Escuela de Ingeniería Informática, Universidad Andrés Bello, Santiago, Chile

[3] Department of Decision Sciences and Information Management, Faculty of Economics and Business, KU Leuven, Louvain, Belgium

المنارة للاستشارات

**Keywords** Model-based testing · Model-driven development · Abstract test case · Concrete test case

## 1 Introduction

The success of software development projects has direct relation with the productivity of the human resources involved and the efficiency of the development processes to obtain high-quality products. In this context, model-driven development (MDD) is a well-known and much adopted paradigm for automating the software generation and reutilization of development knowledge. MDD allows the automatic code generation (model compilation) of a software product by using a conceptual model and a set of transformations rules (Selic 2003). Software engineers are focused on the specification of a holistic conceptual model, and later, by using a model compiler, they automatically obtain the code, thus avoiding the complexity and human mistakes of programming the code manually. With MDD, it is therefore possible to reduce the programming effort that is required to obtain a software product and, hence, to maximize the human resources availability (Brambilla et al. 2012).

In the context of MDD, as any other development paradigms, it is necessary to test the software obtained in order to verify and validate the final product. To do this, software engineers face several issues at the moment of testing software products, such as time constrains, technical complexity, code interpretation. (Botteck and Deiß 2008). For this reason, testing is one of the most resource-consuming phases in the software development cycle (Slaughter et al. 1998). It is important to note that the testing costs increase in direct relation to the complexity of the software products to be tested (Thayer et al. 1974). One interesting alternative to reduce these costs is the application of model-based testing (MBT) (Utting and Legeard 2007). The idea behind MBT is to automatically generate test cases by using the models of the system under test (SUT). Thus, testing activities can be performed at early stages in the software development cycle, and the test cases obtained are independent of the implementation details (Iyenghar et al. 2011). However, with current MBT approaches it is still necessary to derive the models from code and manually implement the concrete (executable) test cases in accordance to the final software products developed.

Since MDD uses the conceptual models of a system to generate its concrete (final) code, and MBT also uses models to generate abstract test cases, we state that both techniques (MBT and MDD) can be used as a natural complement. Thereby, we present the specification of a MBT technique that takes advantage of the conceptual models used in MDD solutions to automatically generate both: abstract test cases that are represented by using an XML specification, and concrete test cases implemented in a specific programming language. The abstract test cases are defined independently of the programming platform and the concrete test cases are defined in the same programming language of the SUT. To do this, the proposed MBT technique is based on a set of testing coverage criteria that are defined for UML diagrams (OMG 2011), the standard language used for specifying conceptual models. Thus, with the application of MBT to MDD, the extra effort that demands the development of specific testing models that are depending of the implementation technology of the SUT can be avoided.

The proposed approach has been developed for an industrially applied MDD solution called OO-Method (Pastor et al. 2001). The application of the testing criteria to


المنارة للاستشارات

OO-Method models is performed automatically, which results in a set of abstract test cases. Then, these abstract test cases are automatically concretized for two technical platforms. Thus, the design time and the execution time of test cases are reduced to seconds.

In summary, the main contribution of this work is twofold:

- A model-based testing technique that automatically generates abstract test cases from conceptual models used in MDD environments.
- A model-based testing technique that automatically generates concrete test cases in Java and C#, reducing the testing effort in MDD projects.

The rest of the paper is organized as follow. Section 2 presents the supporting background of this work and its related work. Section 3 presents T4MDD, a model-based testing technique that generates abstract test cases for a MDD approach. Section 4 presents CONT4MDD, a tool that automates the generation of concrete test cases. Section 5 shows the applicability of T4MDD and CONT4MDD on a concrete case study. Finally, Sect. 6 presents some conclusions and further work.

## 2 Background and related works

This section briefly presents: (1) the foundations of model-based testing techniques, (2) the MDD approach that has been selected to put in practice our proposal, and (3) relevant related work.

### 2.1 Model-based testing (MBT)

MBT uses models for the automatic generation of test cases (Timmer et al. 2011). To do this, it is necessary to take into account the reference modeling language, the algorithms for the generation of the test cases, and the tools infrastructure for the automatic generation and execution of test cases (Dalal et al. 1999). Thus, a model-based testing process has the following steps (Utting and Legeard 2007):

1. Creation of a testing model. This model could be specified from the requirements of the system or from design models. This testing model should not add complexity to the software development process.
2. Generation of abstract test cases. In this step, a specific criterion to define test cases must be defined. This criterion can be related to a functionality of the system, to the model structure (for example, state coverage, transition coverage, def-use coverage), to stochastic characterizations such as user profiles, or to a set of well-defined faults.
3. Concretize abstract test cases. In this step, automatic test case generators transform the abstract test cases into executable scripts by formalizing the selected testing criteria.
4. Test case execution. This step includes the execution of the test cases by indicating automation and technical aspects, such as tool support, online platforms, automatic or manual execution.
5. Analysis of results. In this step it is necessary to analyze the results obtained from the execution of test cases and to perform correction activities when necessary.

المنارة للاستشارات

## 2.2 The OO-Method MDD approach

The OO-Method approach is a MDD method that allows the automatic code generation from a conceptual model by means of a set of well-defined transformation rules (Pastor et al. 2001). The software products generated by this approach correspond to management information systems.

The OO-Method conceptual model has four complementary views that allow the specification of the entire software product. These views are: the structural model, the functional model, the dynamic model, and the presentation model. A detailed specification of all the conceptual constructs of each model can be found in Marín et al. (2010).

The structural model represents the static part of the system by means of a class diagram, i.e., by using classes, attributes, services, pre-conditions, post-conditions, aggregations, inheritance. The users of the system are represented by a particular class type: the agent class. This class has specific properties for representing the access and visibility of the users' system over attributes and services of the classes defined in the model.

The functional model represents the way to assign a value to each attribute defined for the classes of the system. To do this, the OO-Method approach uses the OASIS formal language (Pastor et al. 1992). By using this language it is possible to specify how the value of an attribute changes when a service is executed or a condition is met.

The dynamic model represents the valid states and the states transitions for the instances of a class; i.e., the objects of the system. To do this, a state transition diagram is used.

The presentation model represents the graphical user interface by means of a set of interaction patterns: population interaction units, instance interaction units, service interaction units, and master detail interaction units. This last interaction unit is a composition of an instance interaction unit and a population interaction unit. For the correct specification of these interaction units, it is necessary to specify the display set of attributes, the navigations among the interaction units, entry fields, etc.

The industrial implementation of the OO-Method approach is called Integranova M.E.S. (Model execution System) (Pastor et al. 2004). With this technology it is possible to generate fully executable code in a three-tier architecture: the persistent layer, the business logic layer, and the presentation layer. Each layer can be generated in different technological platforms, such as C#, Java, ASP, JSP, and SQL from the same conceptual model.

## 2.3 Related works

MBT is a variant of testing that relies on explicit behavior models that encode the intended behavior of a system, and possibly, the behavior of its environment (Utting et al. 2012). However, which models can be used by MBT techniques? which is the abstraction level of these models? and which is the modeling language involved? Are questions without a clear answer yet (Rodrigues da Silva 2015).

Dias Neto et al. (2007) presents a systematic literature review, where 78 papers related to MBT are analyzed. The main conclusions of this review are (1) that MBT approaches are usually not related to the development process, so that it is necessary to create integration tools to apply the MBT techniques to the software development process; and (2) high knowledge of modeling languages, testing criteria and languages to generate the test scripts are needed to apply these MBT techniques.

Springer

المنارة للاستشارات

There are some works that uses state charts as the behavior model of the system (Botteck and Deiß 2008; Koopman et al. 2008; Reza et al. 2008; Seifert 2008; Yang et al. 2011; Zeng et al. 2009). In these works, behavioral models are only used in early phases of the software development process for documenting user requirements, and they are not used for an automatic code generation; i.e., these models are not use in MDD environments. This also leads to the problem that behavioral models are specified separated from the code, which finally results in dead models (Engels 2009).

In addition, other MBT proposals use structural models of the system (e.g., Blanco and Tuya 2015; Fourneret et al. 2011; Fujiwara et al. 2011; Pérez-Lamancha et al. 2013), use-case diagrams (such as Gutierrez et al. 2009), or activity diagrams (e.g., Farooq and Lam 2009; Yuan et al. 2008) as input for the test case generation techniques. These proposals use a model that represents a single view of the system to generate the test cases. We advocate that a software system model has different complementary views (structural, behavioral, and interaction) that must be specified to automatically generate a complete software system in MDD environments (Marín et al. 2013). Thus, in order to test the holistic software system, test cases should be generated using models related to the different views; however, this is a big pending challenge for MBT approaches.

Even though some MBT proposals clearly state the use of conceptual models as input to apply the MBT techniques, most of these proposals just declare that they are using UML (or UML-like) models without indicating the specific diagrams, necessary UML extensions, concrete model mappings, or transformation rules used (e.g., Brucker et al. 2011; Castillos et al. 2011; Lasalle et al. 2011; Mlynarski 2010; Wieczorek et al. 2008). Taking into account (1) that UML does not have enough semantic precision to allow the unambiguous specification of software, which is clearly presented in Berkenkötter (2008), France et al. (2006), Opdahl and Henderson-Sellers (2005) and (2) that UML does not allow the complete specification of a software system (for instance it does not include models to specify the presentation of the final application), MDD approaches (such as Coleman et al. 1994; Moreno et al. 2007; Pastor et al. 2001) have selected a subset of UML models and they have aggregated the needed semantic expressiveness to automatically generate software from conceptual models. However, without knowing the specific UML models used by the MBT techniques, it is difficult to apply these MBT techniques to a sound MDD approach that consider the holistic specification of the conceptual models.

Regarding the tools that implements MBT techniques, there are some tools that generates abstract test cases from state machines models (e.g., Bigot et al. 2003; Conformiq: Conformiq Designer; Elvior: MOTES) and from class models (e.g., Smartesting; SourceForge.net). There are also some tools that generate concrete test cases (e.g., Chen and Miao 2013; da Silveira et al. 2011; Nylund et al. 2011). However, there is a lack of a complete strategy to generate abstract test cases and concrete test cases for MDD environments.

In Xu et al. (2015), authors present a model-based testing approach to automatically generate concrete test cases in java for role-based access control. These authors use predicate/transition nets as the model for the generation of test cases. These models are not complete enough to generate automatically the code of the system. In our approach, we use conceptual models of an MDD approach to generate the test cases, which allow the complete specification of the system, and therefore, a better coverage of the test cases. In addition, since the system can be programmed in different languages, the techniques presented in this paper allow the automatic generation of both abstract test cases and concrete test cases.

المنارة للاستشارات

In Amalfitano et al. (2015) is presented MobiGUITAR, an approach to automatically generate test cases for mobile applications. This approach generates a state machine model from the GUI of the applications, and then generates the test cases from this model. MobiGUITAR do not use an automatic approach to generate the test values. In our approach, the test values are generated automatically.

In summary, we did not found a MBT technique that uses as input the same models that are used by concrete MDD approaches. To solve this important challenge, in the next sections, we present our approach and tools for the automatic generation of abstract test cases and concrete test cases from the conceptual model used by an industrial MDD approach.

## 3 T4MDD: automatic generation of abstract test cases for MDD

Testing for MDD (T4MDD) is a model-based testing technique that allows the automatic generation of abstract test cases by using as input a conceptual model that is ready to be used in an automatic model compilation process. Thus, the test cases can be generated as soon as the conceptual model of the system is done. This conceptual model is composed by the structural model, the functional model, the dynamic model, and the presentation model (see Sect. 2.2). This conceptual model is specified by using the Integranova modeling suite (Integranova 2015), which also stores the models' specification in XML.

The XML representation of the conceptual model is entered in the T4MDD tool, which analyses the model and identifies the different views of the system. At this point, it is important to note that the system under test (SUT) can be the entire system or a part of the system depending on the testing needs. The software engineer that uses the T4MDD tool is the responsible who indicates if the SUT corresponds either to the entire system or only to part of it.

Afterward, the software engineer must select some testing criterion to generate the abstract test cases. T4MDD has implemented nine criteria to generate test cases, which are focused on the coverage of the class diagram, the data coverage, and the transition coverage.

There are 3 testing criteria related to coverage on the class diagram: class attribute (CA), association end multiplicity (AEM), and generalization (GN). CA explores the space of possible values that the attributes of a class can have. To create the test cases, each attribute of a class has their own id in the xml representation of the conceptual model. Each attribute can be of three different kinds: *Constant*, the value assigned to the attribute cannot be changed; *Variable*, it possible to change the value assigned to the attribute with the execution of a specific service; or *Derivate*, the value of the attribute is automatically calculated from a formula specified in the OASIS formal language (Pastor et al. 1992). In addition, each attribute has the specification of type (string, int, float, etc.) and the maximum size allowed. This information is used to generate representative values for each attribute of each class selected in the SUT.

AEM is focused on testing the cardinalities of each association between the classes. OO-Method allows the specification of relationships between two classes, so that, in order to calculate all the valid cardinalities, it is necessary to calculate the Cartesian product using the cardinalities of a class. Then, in order to find faults, it is necessary to generate test cases using cardinalities beyond the valid cardinalities calculated. To do this, techniques of random pairs are used.


المنارة للاستشارات

GN states that all the specializations and generalizations of a class must be generated, and also, the relationships of the superclass with other classes must be also tested with the specialized classes. Thus, these test cases can reveal defects related to the substitution principle, which states that an instance of a superclass can be replaced by an instance of any of its subclasses. The creation of the corresponding test cases is performed by locating superclasses, subclasses, and the related inheritance relationships. All these elements can be located by the corresponding *id* in the xml file of the conceptual model. With this information, it is possible to know the child relationships for each superclass and then generate the test cases by substitution of instances of the corresponding subclasses.

Regarding data coverage, pairwise criterion is used. When a service has too many parameters, the number of combinations of all values could be too big, increasing the time for test case generation. To avoid this problem, pairwise testing states that just using a pair of possible values for the parameters is enough to evaluate the feasibility to test a service.

Regarding transition coverage, the following testing criteria have been taken into account by T4MDD: all-states, all-transitions-pair, all-loop-free-paths, all-one-loop-paths, custom-all-paths. All-states criterion expresses that all the states defined for the objects of a class must be reached. All-transitions-pair criterion is focused in the evaluation of objects in one particular state and the feasibility to reach all the possible next adjacent states. All-loop-free-paths evaluates that all the states are reachable by the execution of the transitions, without passing two times for the same state, i.e., without loops at one state. All-one-loop-paths criterion is similar to all-loop-free-paths, with the difference that it is allowed one loop in the process to reach all the states. Custom-all-paths criterion is focused on reach a percentage of the transitions specified in the state transition diagram. To do this, the software engineer needs to specify the percentage of the transitions that he/she wants to test.

To create the test cases related to transition coverage, the state transition diagram is used. Each class defined in the MDD conceptual model has related a state transition diagram. In this diagram, the initial state, final state, and the intermediate states of the

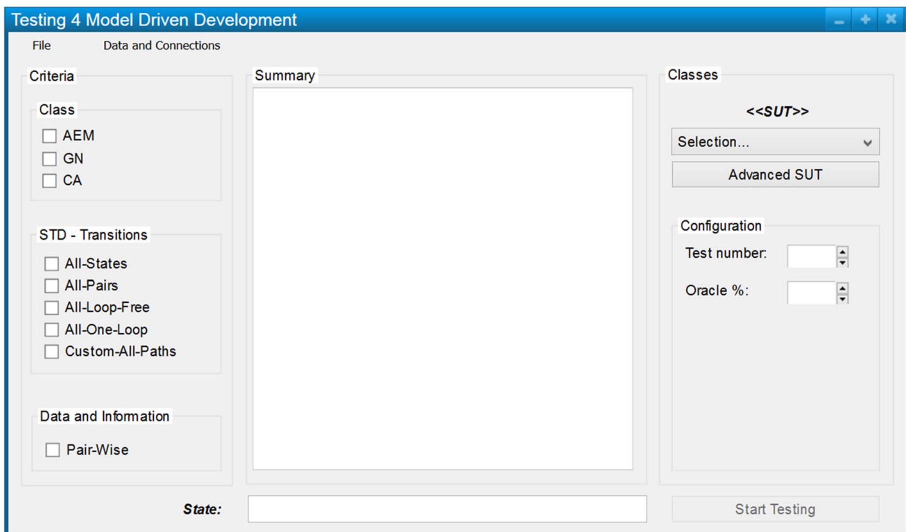


Fig. 1 T4MDD layout

المنارة للاستشارات

objects are specified, and also, the corresponding transitions to change the state of an object are specified. This diagram is automatically generated by the Integranova modeling suite (Integranova 2015). Moreover, the software engineer has the possibility of modify these diagrams in order to provide details regarding the possible states of the objects of a class and the transitions related to these objects.

Once the testing criteria have been selected, the T4MDD tool generates an XML file with the specification of the abstract test cases. It is important to remark that abstract test cases provide the conceptual representation of the test cases to be executed, while the concrete test cases correspond to the executable test cases that are implemented according to the SUT execution platform.

The T4MDD tool has a front end layout that is separated in three parts (see Fig. 1): the selection of the testing criteria (on the left), a summary of the test case generation (on the middle), and the selection of the SUT (on the right). The software engineer has the possibility to select the entire SUT to generate the test cases for the whole system or an advanced SUT to generate test cases only for a set of specific classes. At the bottom of the layout, the tool shows a state bar that indicates the progress in the generation of the test cases. According to the progress, the T4MDD tool can show the following messages and alerts to the software engineer: (1) Document correctly uploaded, (2) There are no data in the database, (3) You must select a number of test cases, and (4) The XML file does not have the proper format.

T4MDD implements the following algorithm to generate the abstract test cases for the CA criterion.

**Algorithm 1. CA criterion**

Let $C$ be the list of classes that are in the model of the SUT selected, $C = \{c1, c2, \ldots, cn\}$, let $A$ be the list of attributes owned by each class c that belongs to $C$, $A = \{a1, a2, \ldots, an\}$. Let *tn* the number of test cases selected.

1. *TS={}*; initialize *TS*, which represents the set of test cases generated
2. *cont:=0*; initialize a counter for the test cases that must be generated
3. *tc={};* initializes the next test case to be added to *TS*
4. *DataFactory df = new DataFactory();* initialize the Data Factory library
5. For each class $c \in C$:
   a. Select the name of *c* and stores it in *tc*
   b. For each $a \in A$
      i. Select the *name* of *a* and store it in *tc*
      ii. Select the *datatype* of *a* and store it in *tc*
      iii. If(*df.indexof(a.name) > -1*); searches in DataFactory if exists values for the attribute
         A. while (*cont* < *tn*) *df.getValue()*; selects the value and stores it in *tc*
      iv. else, while (*cont* < *tn*) *a.getRandomValue()*; generates a random value for the *datatype* of the attribute *a* and stores it in *tc*
   c. end for
   d. *TS := TS U tc*; store *tc* in *TS*
   e. *tc={};* initializes the next test case to be added to *TS*
6. end for
7. end

The XML representation of the test cases generated for the CA criterion has the following tags: the testing criterion, the target class, the name of each attribute and its data type, and a set of values generated for each attribute. A DataFactory library (Gibson 2015)



المنارة للاستشارات

has been used to generate these values in order to provide semantically coherent values for the attributes. Figure 2 shows an extract of the XML specification of an abstract test case generated for the CA criterion, which shows the five different values for the attributes *id_Employee* and *empName*.

For the generation of test cases for the AEM criterion, T4MDD analyses the classes of the SUT model to find the class associations. For each association, the minimum and maximum cardinalities of both association ends are identified. Then, the Cartesian product of these four values is calculated in order to obtain all the possible valid combinations of cardinalities values for each association end. Afterward, the creation service is executed for each class of the association as many times as it is indicated in the list of valid cardinality values. Test cases with invalid associations are also generated. The XML representation of the test cases generated for the AEM criterion has the following tags: the testing criterion, the name of the class, the id of the aggregation, a set of cardinalities, and the type (which indicates if the test case correspond to an oracle or it is a test case that must fail). For each set of cardinalities, the services that must be executed with their parameters and the corresponding values are specified.

For the generalization criterion, T4MDD analyses all the inheritance relationships of the SUT to identify the different father and child classes. For each father class, the creation services are executed with testing values, the specialization services that correspond to each child class of the father class analyzed are also executed. Later, for each instance of the child classes, the associations inherited from the father class are analyzed. From these associations, the services of the related classes (that can be executed) are added to the test case and testing values are assigned to their parameters. The XML representation of the test cases generated for the GN criterion has the following tags: the testing criterion, the id of the generalization, the parent class, the child class, and the services that be executed by these classes with the corresponding parameters and values.

For the all-states criterion, the following algorithm is implemented in T4MDD to generate the abstract test cases.

**Algorithm 2. All-States criterion**

Let $C$ be the list of classes that are in model of the SUT selected, $C = \{c1, c2, \ldots, cn\}$. Let $S$ be the list of states defined for each class c belonging to $C$, $S = \{s1, s2, \ldots, sn\}$. Let $t$ the list of transitions that leave an state $s$, $T = \{t1, t2, \ldots, tn\}$.

**Fig. 2** Extract of abstract test case for CA criterion

```
<Test-Suite>
<Criterion name="ClassAttribute">
<Class target="Employee">
<Attribute name="id_employee" dataType="String">
<TestValue dataType="String">87955</TestValue>
<TestValue dataType="String">96524</TestValue>
<TestValue dataType="String">70423</TestValue>
<TestValue dataType="String">39563</TestValue>
<TestValue dataType="String">41842</TestValue>
</Attribute>
<Attribute name="empName" dataType="String">
<TestValue dataType="String">Myron Hayes</TestValue>
<TestValue dataType="String">Laura Richardson</TestValue>
<TestValue dataType="String">Patrick Walker</TestValue>
<TestValue dataType="String">Kaylee Austin</TestValue>
<TestValue dataType="String">Timmy Green</TestValue>
</Attribute>
```

المنارة للاستشارات

1. *TS={}*; initializes *TS*, which represents the set of test cases generated
2. *tc={};* initializes the next test case to be added to *TS*
3. *VS={};* initializes VS, which represents the set of states visited
4. For each class $c \in C$:
   a. Select the name of *c* and store it in *tc*
   b. For each $s \in S$
      i. Select the *name* of *s* and store it in *tc*
      ii. If (T $\neq$ $\emptyset$ ); verify that the state *s* has transitions, otherwise, *s* is the final state and none service can be executed to reach another state.
         A. For each transition $t \in T$, *where* (*t.targetState* $\neq s$ *AND t.targetstate* $\notin$ *VS:*
            a. Select the name of the service *serv* related to *t* and store it in *tc*
            b. For each parameter *p* of *serv*
               i. Select the *name* of *p* and store it in *tc*
               ii. Select the *datatype* of *p* and store it in *tc*
               iii. *p.getRandomValue()*; generates a random value for the *datatype* of the parameter *p* and stores it in *tc*
            c. end for
            d. *VS:= VS U s U t.targetState*; adds both s and target state to the set of visited states
         B. End for
      iii. End if
   c. end for
   d. *TS := TS U tc*; store *tc* in *TS*
   e. *tc={};* initializes the next test cases that will be added to *TS*
5. end for
6. end

The XML representation of the test cases generated for the all-states criterion has the following tags: the testing criterion, the name of the class owner of the state transition diagram, the state that is reached and the service that must be executed to reach that state. All the parameters and the corresponding values for each service are also specified. Figure 3 shows a state transition diagram for the client class, which correspond to a user of archetypical software. An extract of the abstract test cases generated for this *client* class is presented in Fig. 4.

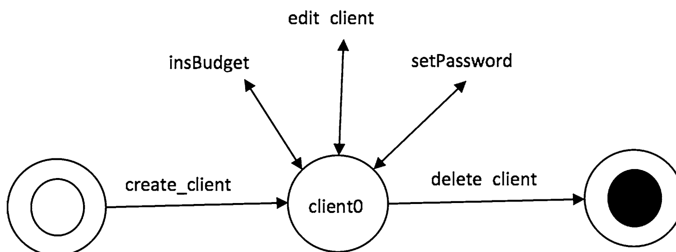


**Fig. 3** State transition diagram for the Client class


المنارة للاستشارات

```xml
<Criterion name="All-States" target="Client">
    <State name="Creation">
        <Service name="create_client">
            <Parameter name="p_atrtelefhone">
                <TestValue dataType="String">djczklzgxv</TestValue>
            </Parameter>
            <Parameter name="p_atraddress">
                <TestValue datatype="String">yirfhxlxkpxkptrssoanajqictnabutmtaohkrvaajuodjnvgqafdtgemhzokauywbdnasfrsfmiy
            </Parameter>
            <Parameter name="p_atremail">
                <TestValue datatype="String">hjdmdjoeiydpujjzizhunpufzymzcf</TestValue>
            </Parameter>
            <Parameter name="p_atrname">
                <TestValue dataType="String">maiqdkhjutwibvorepsxpmebfzarjjgmfpkgaoqftgrmfrpmcdyezvantadxpprzdurkyqjhgbmfh
            </Parameter>
            <Parameter name="p_atrid_Client">
                <TestValue datatype="Autonumeric">44</TestValue>
            </Parameter>
        </Service>
    </State>
    <State name="Client0">
        <Service name="delete_client">
            <Object name="p_thisClient" class="Clas_1384400420864124">
                <Parameter name="p_atrid_Client"><TestValue tipodato="Autonumerico">44</TestValue></Parameter>
            </Object>
        </Service>
    </State>
    <State name="Destrucc"> <StateWithoutServices Service="null"/></State>
</Criterion>
```

**Fig. 4** Extract of abstract test case for all-states criterion

For the test case generation of the all-transitions-pair criterion, the software engineer could select one specific state of the SUT, or it can be automatically selected by T4MDD tool taking into account the state with the bigger number of transitions. For the all-transition-pair criterion, the following algorithm is implemented.

**Algorithm 3. All-transitions-pair criterion**

Let *s* the state selected by the tester or the SUT. Let *t* the list of transitions that leave the state *s*, $T = \{t1, t2, \ldots, tn\}$.

1. *TS={}*; initializes *TS*, which represents the set of test cases generated
2. *tc={};* initializes the next test case to be added to *TS*
3. For each transition $t \in T$:
    a. Select the name of transition *t*, and store it in *tc*
    b. Store s as the initial state of transition t in tc
    c. Select the reach state of transition t and store it in tc
    d. Select the name of the service serv related to t and store it in tc
    e. For each parameter p of serv
        i. Select the *name* of *p* and store it in *tc*
        ii. Select the *datatype* of *p* and store it in *tc*
        iii. *p.getRandomValue()*; generates a random value for the *datatype* of the parameter *p* and stores it in *tc*
    f. end for
    g. *TS := TS U tc*; stores *tc* in *TS*
    h. *tc={};* initializes the next test case that will be added to *TS*
4. end for
5. end

The XML representation of the test cases for the all-transition-pair criterion has the following tags: the testing criterion, the class owner of the state transition diagram, the transition id with the corresponding initial state and target state, and the service that must be executed in order to produce the transition, which has a set of parameters with the corresponding values generated. An example of the test cases generated for all-transitions-pair criterion is shown in Fig. 20.

The algorithm used to generate the test cases of the all-loop-free criterion differs from the all-states criterion in the restriction that establishes that the states must be reached just one time in the execution of a test case. Note that with this criterion, the test suite will not

المنـارة للاستشارات

present the states that are not reached if they are in a loop and the transitions that are not executed if they are in a loop. The XML representation of the test cases generated for the all-loop-free criterion has the same structure than the all-states criterion (see Fig. 4). The test cases generated for the all-one-loop criterion also has the same XML structure as the all-states criterion, with the difference that a state could appear more than once since it could be reached more than once.

The custom-all-paths criterion intends to test all the transitions defined in the state transition diagram. In order to be more effective in the definition of test cases, the software engineer is able to decide which transitions wants to test (see Fig. 5). The XML structure of the test cases is the same as the all-transitions-pair criterion. The algorithm used to create the test cases XML specification is Algorithm 3, which is used by initializing $T$ with the transitions selected by the tester.

In summary, the T4MDD approach and its supporting tool allows the automatic generation of abstract test cases, which are specified in an XML file. These test cases are generated taking into account nine different testing criteria. Therefore, the software engineer just needs to select the SUT and the intended testing criteria to automatically generate all the test cases together with a set of testing values in a few seconds by the T4MDD approach.

## 4 CONT4MDD: generation of concrete test cases for MDD

CONT4MDD is a model-based testing technique that generates concrete test cases from the abstract test cases generated by the T4MDD tool. Thus, CONT4MDD generates executable test cases for the software generated from the conceptual models defined according to the OO-Method MDD approach.

Figure 6 schematizes the process to use the CONT4MDD tool. First of all, the software engineer must specify the system model and generate the corresponding software code by using the OO-Method approach and the Integranova tool (see right part of the figure). Then, the software engineer generates the abstract test cases by using the T4MDD tool (see left part of the figure). Once the abstract test cases are generated, the engineer uses CONT4MDD to generate the executable scripts for the concrete test cases. These test cases can be generated in Java (.jar) or C# (.exe), two of the development platforms supported by the Integranova tool (Marín et al. 2008). Thus, the software engineer needs to select the target programming language for the executable test cases before generating them.

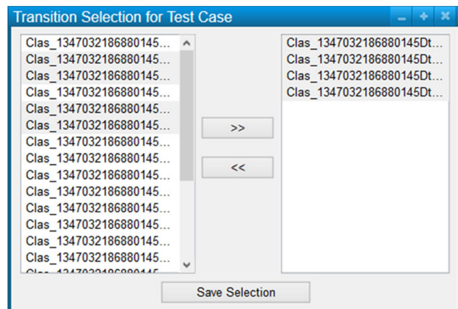


**Fig. 5** Custom-all-paths configuration

Springer

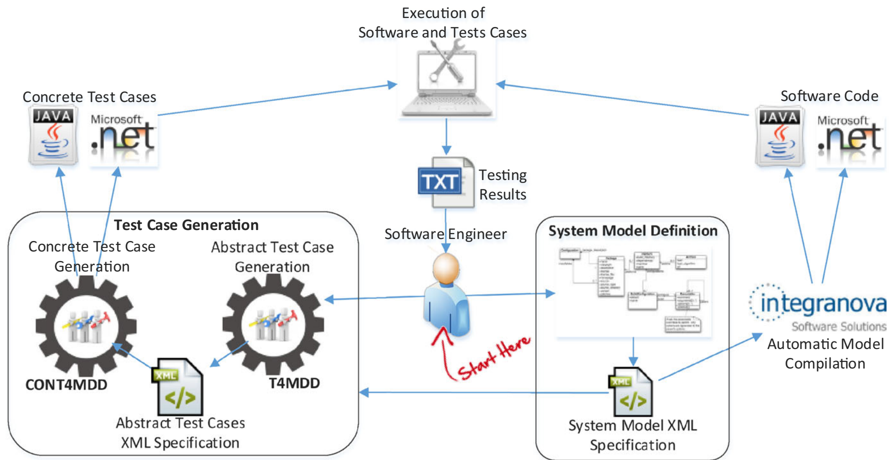


**Fig. 6** Automatic Model-Based Testing process

Finally, the software engineer executes the generated software and the testing scripts (see upper part of the figure), and analyzes the results obtained. It is important to note that the testing scripts are executed over the business logic layer of the software generated since all the criteria implemented in T4MDD are related to functional behavior. Presentation layer is not considered yet by the T4MDD approach.

The CONT4MDD tool receives the XML representation of the conceptual model and the abstract test cases as input (see Fig. 7). These files are verified according to the DTD defined for the OO-Method and T4MDD implementations.

The generation of the concrete test cases has been performed by considering the testing criteria specified in the T4MDD file and the target programming language. For this generation, specific information from the system model has been analyzed. Figure 8 shows an excerpt of the algorithm that is used by CONT4MDD for this analysis.

The testing scripts change depending on the testing criterion selected. Despite this, the part of the code related to the presentation of the results obtained is the same for all the scripts generated. The results of the execution of the test scripts are presented in a text file that shows: the name of the service, the testing criteria, and the results obtained. Following, the analysis performed for each testing criterion is briefly explained.

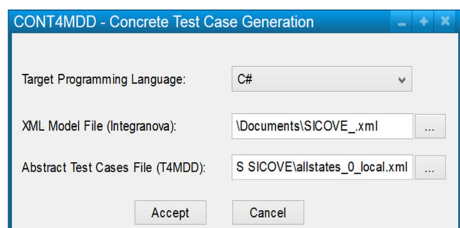


**Fig. 7** CONT4MDD interface

المنارة للاستشارات

```
// ---------- READ THE XML INTEGRANOVA MODEL
String filexml2 = archivoIntegra.getText().replace("\\", "\\\\");
File fXmlFile2 = new File(filexml2);

DocumentBuilderFactory dbFactory2 = DocumentBuilderFactory.newInstance();
DocumentBuilder dBuilder2 = dbFactory2.newDocumentBuilder();
Document doc2 = dBuilder2.parse(fXmlFile2);
doc2.getDocumentElement().normalize();

LinkedList<OrdenClases> orden=ordenAtributos.ordenaAtributos(doc2);

//---------READ THE T4MDD FILE
String filexml = cajaArchivo.getText().replace("\\", "\\\\");
File fXmlFile = new File(filexml);

DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
Document doc = dBuilder.parse(fXmlFile);

doc.getDocumentElement().normalize();

NodeList nList = doc.getElementsByTagName("Criteria");
for (int temp = 0; temp < nList.getLength(); temp++) {
    Node nNode = nList.item(0);

    if (nNode.getNodeType() == Node.ELEMENT_NODE) {
        Element eElement = (Element) nNode;
        criterio=eElement.getAttribute("name");
    }
}
//End read XML
```

**Fig. 8** Excerpt of CONT4MDD algorithm for reading the XML files

## 4.1 CA criterion

In order to implement the CA criterion, the creation service of the classes modeled is used. This service is executed as many times as the number of values defined for each attribute of a class in the T4MDD tool (see Test number in Fig. 1). For each execution of the service, the success or fail is registered in the results file. An extract of the algorithm used for the creation of concrete test cases for java is presented in Fig. 9, where the class and its owned attributes with the test values generated by T4MDD are identified; and Fig. 10, where the file is written.

```
//Get the class
Node nNode2 =nNode.getChildNodes().item(tmp);
for(int tmp2=0; tmp2 < nNode2.getChildNodes().getLength(); tmp2++){

        if(nNode2.getChildNodes().item(tmp2).getNodeType() == Node.ELEMENT_NODE){
              //get the attributes
              Element eElement2 = (Element) nNode2.getChildNodes().item(tmp2);
              Node nNode3 =nNode2.getChildNodes().item(tmp2);
              int var=0;
              LinkedList<String> valoresTemporales= new LinkedList<String>();
              for(int tmp3=0; tmp3 < nNode3.getChildNodes().getLength(); tmp3++){
                  // get the test values
                  if(nNode3.getChildNodes().item(tmp3).getNodeType() == Node.ELEMENT_NODE){
                      valoresTemporales.add(eElement2.getElementsByTagName("TestValue").item(var).getTextContent());
                      var=(tmp3/2)+1;
                  }
              }
             ArgumentosClassAttribute argumentosTemp =
                 new ArgumentosClassAttribute(eElement2.getAttribute("name"),eElement2.getAttribute("datatype"), valoresTemporales);
             argumentos.add(argumentosTemp);
        }
}
ClasesClassAttribute clasesTemp= new ClasesClassAttribute(eElement.getAttribute("target"), argumentos);
clases.add(clasesTemp);
```

**Fig. 9** Excerpt of CONT4MDD algorithm for reading the CA test case



المنارة للاستشارات

```
//name for new file
String nombre="ClassAttribute_"+numero;
File file = new File(nombre+".java");
BufferedWriter output = new BufferedWriter(new FileWriter(file));
PrintWriter wr = new PrintWriter(output);
//header of the file
wr.println("import java.io.BufferedWriter; \nimport java.io.File; \nimport java.io.FileWriter; " +
      "\nimport java.io.IOException;");
wr.println("import java.io.PrintWriter; \nimport java.util.Iterator; \nimport java.util.LinkedList; \n \n \n");
wr.println("public class "+nombre+"{");
wr.println("\tpublic static void metodoPrincipal(){");
wr.println(" ");
wr.println("\t\ttry {");

for(int i=0; i<clases.size(); i++){
// call the creation service
String nombreServicio= clases.get(i).getNombreClase()+"Service.create_"+clases.get(i).getNombreClase();
LinkedList<ArgumentosClassAttribute> atributos = clases.get(i).getAtributos();
LinkedList<String> argumentos= new LinkedList<String>();
for(int j=0; j<atributos.get(j).getValoresAtributos().size(); j++){
    String valores="(";
    for(int k=0; k<clases.get(i).getAtributos().size();k++){
        //assign the values of the attributes to the arguments
        if(clases.get(i).getAtributos().get(k).getTipoAtributo().equals("Nat") ||
                clases.get(i).getAtributos().get(k).getTipoAtributo().equals("Autonumerico") ||
                clases.get(i).getAtributos().get(k).getTipoAtributo().equals("Bool"))
            valores+=(clases.get(i).getAtributos().get(k).getValoresAtributos().get(j));
        else
            valores+="\""+clases.get(i).getAtributos().get(k).getValoresAtributos().get(j)+"\"";
        if(k<clases.get(i).getAtributos().size()-1) valores+=",";
    }
      valores+=");";
      argumentos.add(valores);
  }
```

**Fig. 10** Extract of CONT4MDD algorithm for generating the test case in java

## 4.2 AEM criterion

In order to implement the AEM criterion, the identifier of each classes is used to find the classes involved in an association relationship, and the creation service of each class is used as many times as it is indicated in the cardinalities of the two corresponding association ends. Thus, if the software allows the creation of instances respecting the cardinalities for the oracle test cases, and it does not allow the creation of instances respecting the cardinalities generated in a random way, then the software satisfies the AEM criterion. In contrast, if the software does not allows the creation of instances with the cardinalities of the oracles or allows the creation of instances with the cardinalities generated in a random way, the software does not fulfill the AEM criterion; i.e., the software allows associations that are semantically incorrect.

## 4.3 GN criterion

The GN criterion specifies a list of services of parent and child classes that must be executed. To implement this criterion, the services of the system specified in the T4MDD file are executed. To do this, it is important to know the order or the parameters of each service that have been previously stored. GN criterion is fulfilled when all the services are successfully executed.

## 4.4 All-states criterion

This criterion is used to find dead states, which correspond to states that never are reached by the objects of a class. The T4MDD file specifies a set of services that must be executed to reach all the states defined for the objects of a particular class. At this point, it is

المنارة للاستشارات

important to note that only a creation service must be executed to make a transition from the initial state to the next state, and only a destroy service can be executed to make a transition from an intermediate state to the final state. For this criterion, the results file shows the service executed, the state reached and the result of the transition execution. Figure 11 presents an extract of the algorithm used by CONT4MDD to create the test cases for the all-states criterion.

### 4.5 All-transition-pairs criterion

The implementation of all-transition-pairs criterion is similar to all-states criterion since it executes a list of services related to the transitions defined in the abstract test case for a specific state. Thus, the software fulfills the all-transition-pairs criterion when all the services involved are successfully executed.

### 4.6 All-one-loop criterion

Regarding the all-one-loop criterion, the set of transitions specified in the abstract test cases has one state that is visited twice; i.e., there is a loop in the transitions. Thus, to implement this criterion, the services related to the different transitions are executed. If one execution fails, the criterion is not fulfilled.

### 4.7 Custom-all-paths criterion

Regarding the custom-all-paths criterion, the T4MDD file specifies a set of transitions that must be tested. To do this, it is necessary to have an object at the source state of the transition, i.e., it is necessary to create an object and execute the transitions with the corresponding services to reach the source state. Then the service related to the custom transition is executed. If the target state is reached, then this criterion is fulfilled.

```
// obtain the state
   for(int tmp=0; tmp < nNode.getChildNodes().getLength(); tmp++){
       if(nNode.getChildNodes().item(tmp).getNodeType() == Node.ELEMENT_NODE){
        Element eElement2 = (Element) nNode.getChildNodes().item(tmp);

       // obtain the service
        Node nNode2 =nNode.getChildNodes().item(tmp);
        for(int tmp2=0; tmp2 < nNode2.getChildNodes().getLength(); tmp2++){
               if(nNode2.getChildNodes().item(tmp2).getNodeType() == Node.ELEMENT_NODE){
                    Element eElement3 = (Element) nNode2.getChildNodes().item(tmp2);

                    if(eElement3.getTagName().equals("EstadoSinServicios")){
                        continue;
                    }
                    else{
                       // obtain the parameters of the service and their values
                        Node nNode3 =nNode2.getChildNodes().item(tmp2);
                        LinkedList<Atributo> atributos = new LinkedList<Atributo>();

                        for(int tmp3=0; tmp3 < nNode3.getChildNodes().getLength(); tmp3++){
                               if(nNode3.getChildNodes().item(tmp3).getNodeType() == Node.ELEMENT_NODE){
                                    Element eElement4 = (Element) nNode3.getChildNodes().item(tmp3);
                                    Node nNode4 =nNode3.getChildNodes().item(tmp3);

                                    if (nNode4.getNodeType() == Node.ELEMENT_NODE) {
                                           Element eElement5 = (Element) nNode4.getChildNodes().item(1);
                                           Atributo atributoAllStates = new Atributo(eElement4.getAttribute("name"),
                                                   eElement5.getAttribute("datatype"),
                                                   eElement4.getElementsByTagName("TestValue").item(0).getTextContent());
                                           atributos.add(atributoAllStates);
                                       }
```

Fig. 11 Extract of CONT4MDD algorithm for all-states test case



المنارة للاستشارات

### 4.8 Pairwise criterion

To fulfill the pairwise criterion, the services must be executed respecting the order of the parameters defined. If the system accepts the change of values of two parameters of a service, then the system does not fulfill this criterion.

In summary, CONT4MDD automatically generates test cases ready to be executable in C# and Java, thus reducing the programming time just to few seconds.

## 5 Application of T4MDD and CONT4MDD

In the following subsections, we present the application of T4MDD and CONT4MDD to a particular software project related to the domain of management information systems. This project has been performed in order to evaluate the feasibility and the efficiency of our approach of model-based testing techniques in comparison with traditional testing processes.

### 5.1 The SICOVE case

The case corresponds to a system for a trading cars company. Figure 12 shows the conceptual model that has been developed for this system. The model supports the car management, the location management, the client budgets, and the car sales. There are three different types of users of the system: a location manager, a seller, and a client. The

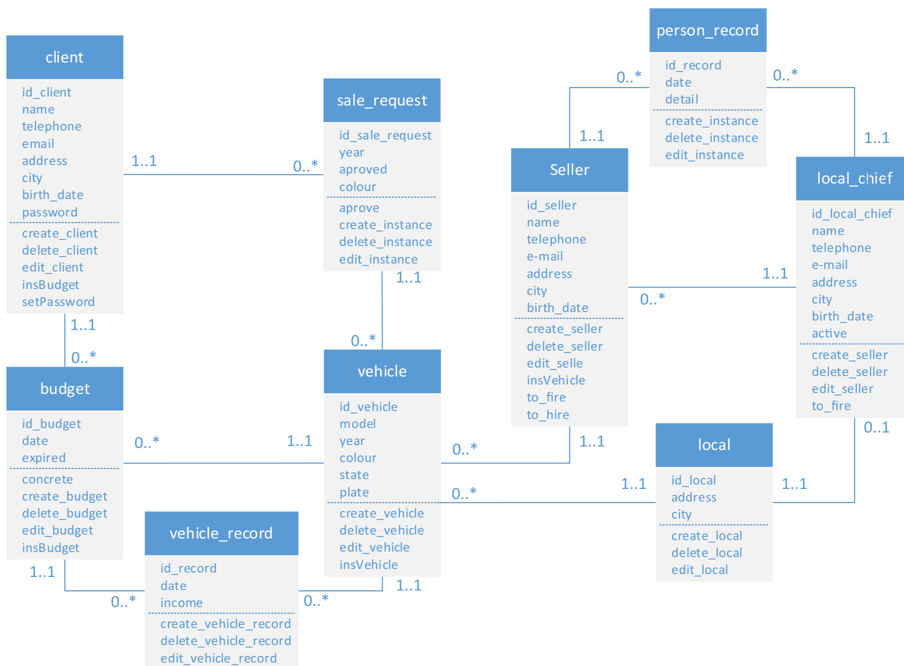


Fig. 12 Conceptual model for SICOVE

المنارة للاستشارات

seller manages historical information of cars and locations. The location manager leads the sellers of a location, the cars of a location, and approves the car budgets. The seller manages the cars, the clients, and the car budgets. The client can see the cars, can see the locations, can require a car sale or a car budget, and also can purchase a car.

Some of the relevant services to understand the system operation are the following: The service *approve* is executed when the location manager approves a car sale request, and hence, the company purchases the car; the service *concrete* is executed when a client asks for a car budget and concretes the purchase of a car; and the service *to_fire* is executed to quit an employee of the company.

### 5.2 Verification of testing criteria

All the testing criteria implemented for the abstract test cases generated by T4MDD and concrete test cases generated by CONT4MDD have been verified by using the case presented above.

Appling the CA criterion to the seller class, the XML file that contains the abstract test cases specification (see Fig. 13) has been generated with three values for each attribute of the class. This XML file is loaded into the CONT4MDD tool, which generates a script with concrete test cases in C#.

The script generated in C# must be opened with the .Net development platform to configure the corresponding reference to the business logic layer of the system that has been generated with the Integranova tool. Figure 14 shows the concrete test cases generated, which contains the script for the CA criterion called *ScriptClassAttribute*, the reference business logic layer called *SICOVE*, and the presentation layer of the system called *SICOVEUI*. Figure 15 shows the concrete test cases generated.

Once the test cases script has been executed, the results obtained are generated in a text file (see Fig. 16). Figure 16 shows that the test cases have been successfully passed, and therefore, the SUT fulfills with the CA criterion, i.e., the class *Seller* selected as SUT allows the creation of objects with different combinations of values for the attributes of each object. Regarding the remaining class diagram coverage criteria, there are not test cases generated for GN criterion due to the conceptual model does not have inheritance relationships defined. The test cases related to the AEM criterion have also been generated

**Fig. 13** CA abstract test cases for SICOVE

```
<Test-Suite>
<Criterion name="ClassAttribute">
<Class target="Seller">
<Attribute name="id_seller" dataType="Autonumeric">
<TestValue dataType="Autonumeric">36</TestValue>
<TestValue dataType="Autonumeric">19</TestValue>
<TestValue dataType="Autonumeric">32</TestValue>
</Attribute>
<Attribute name="name" dataType="String">
<TestValue dataType="String">Tia Fuentes</TestValue>
<TestValue dataType="String">Katherine Patton</TestValue>
<TestValue dataType="String">Charlene Moran</TestValue>
</Attribute>
<Attribute name="telephone" dataType="Nat"></Attribute>
<Attribute name="e-mail" dataType="String"></Attribute>
<Attribute name="address" dataType="String"></Attribute>
<Attribute name="city" dataType="String"></Attribute>
<Attribute name="birth_date" dataType="date"></Attribute>
</Class>
</Criterion>
</Test-Suite>
```



المنارة للاستشارات

**Fig. 14** CA concrete test cases for SICOVE

```
Solution 'SICOVE' (3 projects)
  ScriptClassAttribute
    References
    App.config
    ClassAttribute_0.cs
  SICOVE
  SICOVEUI
```

```
public class ClassAttribute_0{
    public static void metodoPrincipal(){

        try {
            StreamWriter fichero = File.CreateText(\"logClassAttribute_0.txt");
            fichero.WriteLine(" ");
            fichero.WriteLine("Testing results for the ClassAttribute criterion");
            fichero.WriteLine("Class: Seller");
            fichero.WriteLine(" ");
            fichero.WriteLine("---------------------------------------------------------------------------------");
            fichero.WriteLine("Service\t\t\tAttribute N∫\t\t\tResult");
            fichero.WriteLine("---------------------------------------------------------------------------------");
            fichero.WriteLine(" ");

            try{
                sellerService.create_seller(36,"Tia Fuentes",78987654,"dohatch@somema1l.co.uk", "972 Dyer Ave","New York","07/11/2012");
            }
            catch(Exception ex){
                Console.WriteLine( "Error: "+ ex.Message());
                ok=0;
            }

            if(ok==1){
                fichero.WriteLine("sellerService.create_seller\t\t0\t\t\tSuccesful");
            }else{
                fichero.WriteLine("sellerService.create_seller\t\t0\t\t\tFail");
                cumpleCriterio=0;
                ok=1;
            }
```

**Fig. 15** Extract of concrete test case for class attribute criterion at C#

```
CONT4MDD
--------------------------------------------------------------------------------
Testing results for the Class Attribute Criterion
Class Seller
--------------------------------------------------------------------------------
Service                                Attribute Nº                        Result
--------------------------------------------------------------------------------
create_seller                                    0                         Successful
create_seller                                    1                         Successful
create_seller                                    2                         Successful

The system under test fulfills the CA criterion.
--------------------------------------------------------------------------------
```

**Fig. 16** Results for CA concrete test cases for SICOVE

insVehicle
insPersonRecord
setPassword
create_seller
seller0
to_hire
TO_FIRE
fired
delete_seller

**Fig. 17** State transition diagram for the Seller class

المنارة للاستشارات

by the T4MDD and CONT4MDD tools. These test cases have also been successfully evaluated.

Regarding the transition coverage criteria, Fig. 17 shows an example of the state transition diagram for the *Seller* class. In this diagram it is possible to observe four states: the *creation* state, the *seller0* state, the *fired* state, and the *destruction* state. To reach these four states, it is necessary to execute at least three services related to the transitions between the states. It is clear that from the destruction state is not possible to execute any other transition. Thus, in order to visit all the states, the services that must be executed are the following: *create_seller*, *to_fire*, and *delete_seller*. Other different combinations of three services do not allow testing the all-states criterion, i.e., with other combination of three services, it is impossible to reach the *fired* and the *destruction* states. The T4MDD tool performed the analysis regarding the services that must be executed to validate the all-states criterion. An extract of the concrete test case is shown in Fig. 18. The results obtained by CONT4MDD are shown in Fig. 19.

Regarding the all-transition-pairs criterion, all the transitions must be able to be executed. Figure 3 shows the state transition diagram for the *Client* class. Figure 20 shows the generated abstract test cases, and finally, Fig. 21 shows the results obtained by the concrete test cases. It is important to note that one of the services of the transitions correspond to a service for the destruction of the object. In this case, the CONT4MDD tool executes first the services that are related to transitions to the same state or other intermediate states, leaving the service that is related to the transition to the destruction state at the end.

The remaining transition coverage criteria: all-loop-free, all-one-loop, and custom-all-paths have been also verified with successful results. It is important to remark that CONT4MDD verify in these criteria the correct execution of services that allows the transitions in the state transition diagram. This means that CONT4MDD verify that only creation services can be executed from the creation state, only destruction services can reach the destruction state, and that any other state can only be reached by the execution of a service.

```
using System;
using System.IO;
public class AllStates_0{
    static void metodoPrincipal(){
        int ok=1;
        int cumpleCriterio=1;

        try {
            StreamWriter fichero = File.CreateText("logAllStates_0.txt");
            fichero.WriteLine(" ");
            fichero.WriteLine("Testing results for the All-States criterion");
            fichero.WriteLine("Class seller");
            fichero.WriteLine(" ");
            fichero.WriteLine("----------------------------------------------------------------------------------------");
            fichero.WriteLine("Service\t\t\tState\t\t\t\tResult");
            fichero.WriteLine("----------------------------------------------------------------------------------------");
            fichero.WriteLine(" ");

            try{
                create_seller(24, "jvjebywyypgqorighfawpftjqevznxclmndaawethajnahzxgxzzdurifeubrhhqajaiohnjurnrlpcjxfarpbqzqaphgvjjfhzc",
            }
            catch(Exception ex){
                Console.WriteLine( "Error: {0}", ex.Message );
                ok=0;
            }

            if(ok==1){
                fichero.WriteLine("create_seller\t\tseller0\t\t\tSuccesful");
            }else{
                fichero.WriteLine("create_seller\t\tCreation\t\t\tFail");
                cumpleCriterio=0;
                ok=1;
            }
```

**Fig. 18** Extract of the concrete test case generated by CONT4MDD



المنارة للاستشارات

```
CONT4MDD
--------------------------------------------------------------------------------
Testing results for the All-States Criterion
Class Seller
--------------------------------------------------------------------------------
Service                       State                Result
--------------------------------------------------------------------------------
create_seller                 seller0              Successful
to_fire                       fired                Successful
delete_seller                 destruction          Successful

The system under test fulfills the All-States criterion.
--------------------------------------------------------------------------------
```

**Fig. 19** Results of the concrete test cases for all-states criterion

```
<Test-Suite>
<Criterion name="AllTransitionPairs" target="Client">
<Transition transitionID="class_1402616152064363Dte_1transi_2"
initialState="class_1402616152064363Dte_1est_2"
finalState="class_1402616152064363Dte_1est_3">
<Service name="class_1402616152064363Ser_5"></Service>
</Transition>
<Transition transitionID="class_1402616152064363Dte_1transi_3"
 initialState="class_1402616152064363Dte_1est_2"
 finalState="class_1402616152064363Dte_1est_2">
<Service name="class_1402616152064363Ser_8"></Service>
</Transition>
<Transition transitionID="class_1402616152064363Dte_1transi_4"
initialState="class_1402616152064363Dte_1est_2"
finalState="class_1402616152064363Dte_1est_2">
<Service name="class_1402616152064363Ser_9"></Service>
</Transition>
<Transition transitionID="class_1402616152064363Dte_1transi_5"
 initialState="class_1402616152064363Dte_1est_2"
 finalState="class_1402616152064363Dte_1est_2">
<Service name="class_1402616152064363Ser_9"></Service>
</Transition>
</Criterion>
</Test-Suite>
```

**Fig. 20** Abstract test cases for all-transition-pairs criterion

```
CONT4MDD
--------------------------------------------------------------------------------
Testing results for the All-Transition Pairs Criterion
Class Client
--------------------------------------------------------------------------------
Service                             Transition                         Result
--------------------------------------------------------------------------------
edit_client            clas_1402616152064363Dte1_Transi_3       Successful
insBudget              clas_1402616152064363Dte1_Transi_4       Successful
setPassword            clas_1402616152064363Dte1_Transi_5       Successful
delete_client          clas_1402616152064363Dte1_Transi_2       Successful

The system under test fulfills the All-Transition Pairs criterion.
--------------------------------------------------------------------------------
```

**Fig. 21** Results of the concrete test cases for all-transition-pairs criterion

Regarding the pairwise testing criterion, the abstract test case has a service with two values of parameters that have been changed. For instance, Fig. 22 shows the abstract test case for the pairwise criterion for the *Client* class. The service specified in the abstract test case corresponds to an edit service. Thus, in the concrete test case, it is necessary to first create the object and then execute the edit service with the values specified. Later, when the concrete test case is executed, it fails (see Fig. 23). This means that it is not possible to change the order of the parameters of a service, so that the pairwise criterion is fulfilled.

المنارة للاستشارات

**Fig. 22** Abstract test case for pairwise criterion

```
<Test-Suite>
<Criterion name="PairWise">
<Service name="edit_instance" id="class_1402616152064363Ser_8">
<Argument name="p_this_client">
<Parameter name="p_atrid_client">
<TestValue dataType="Autonumeric">9</TestValue>
</Parameter>
<Parameter name="p_password"></Parameter>
<Parameter name="p_atrname"></Parameter>
<Parameter name="p_atrtelephone"></Parameter>
<Parameter name="p_atremail"></Parameter>
<Parameter name="p_atraddress"></Parameter>
<Parameter name="p_atrcity"></Parameter>
<Parameter name="p_atrbirthdate"></Parameter>
</Argument>
<Argument name="p_name">
<Parameter name="p_name">
<TestValue dataType="String">Rose Bullard</TestValue>
</Parameter>
</Argument>
</Service>
</Criterion>
</Test-Suite>
```

```
CONT4MDD
-------------------------------------------------------------------------------
Testing results for the Pair-Wise Criterion
Class Client
-------------------------------------------------------------------------------
Service              First Attribute      Second Attribute     Result
-------------------------------------------------------------------------------
edit_client          p_atrid_client       p_password           Fail


The system under test fulfills the Pair-Wise criterion.
-------------------------------------------------------------------------------
```

**Fig. 23** Results of concrete test case for pairwise criterion

## 5.3 Efficiency of the techniques

Testing is one of the most used quality assurance technique in industry (Marín et al. 2011). On most software projects testing consumes at least the 30 % of the effort (Collofello and Vehathiri 2005). Hence, by reducing the time required in the testing phase, it could be obtained an important benefit in the development time and effort required to produce high-quality software products.

To measure the specific time reduction, we compared a manual implementation and execution of test cases with the application of the automatic MBT techniques proposed. We consider the manual programming and testing of the SICOVE project as an example of the traditional way to develop software products. We are interested in the percentage of time reduction using the T4MDD and CONT4MDD in order to quantitatively measure the benefits of applying these automatic techniques.

First of all, two software engineers manually programmed the SICOVE system in C# and Java, one for each technical platform, respectively. To do this, they receive the SICOVE specification, which have been defined by following IEEE 830 (IEEE 1984). This specification contains the functionality required, the users of SICOVE, and the use-case diagrams. The software engineers modeled the database, and later they performed the programming tasks by taking into account the SICOVE specification.

The programming tasks took 40 h in the case of C# and 36 h in the case of Java. Then, the subjects performed the testing of both applications in a crossed way, i.e., the software



المنارة للاستشارات

engineer that developed the case in C# performed the testing of the Java application, and vice versa. The testing process consisted on: the specification of the test cases with the corresponding test values, the execution of the test cases, and the analysis of the results obtained. The testing process took 14 h for the C# case, and 16 h for the Java case.

In order to apply T4MDD and CONT4MDD to the SICOVE project, two other software engineers with similar skills to the subjects characterized before developed the case following a MDD approach. They developed the conceptual model for the SICOVE case study using the OO-Method MDD approach. The SICOVE specification given to the MDD developers was the same as the one delivered to the engineers that manually programmed the SICOVE system. In the MDD scenario, it took 2.5 h to correctly and completely specify the conceptual model. In this case, both engineers worked together, and thus, we consider that the work performed for the specification of the conceptual model is equivalent to 5 h of work of one person. This conceptual model was performed by using the Integranova tool, which automatically generates the code in both programming languages, C# and Java; it also generates the scripts to create the system database. Therefore, in just 5 h it was possible to model the entire system. The programming code was automatically generated from the conceptual model, i.e., without programming any line of code. This demonstrates one of the main benefits of the MDD paradigm: a considerable reduction of the development time in relation to traditional software development paradigms.

After generating the code, one of the software engineers entered the xml representation of the SICOVE conceptual model in the T4MDD tool and selected the following options: all the testing criteria, all the classes of the SUT, and the generation of the xml files for the abstract test cases. This took 226 s (less than 4 min) in total to obtain the abstract test cases for the SICOVE case.

To generate the concrete test cases for SICOVE, the engineer entered in the CONT4MDD tool the abstract test case generated with the T4MDD tool, the file of the conceptual model generated with the Integranova tool, and selected the language for each abstract test case. With this input, the tool generates the script in the corresponding target language. This generation process took 126 s (around 2 min). Finally, the software engineer executed the script generated in the selected language. It took 203 s (less than 4 min) for the execution of all the concrete test cases in C#, and 160 s (less than 3 min) for the execution in Java. Therefore, the generation and execution of the concrete test cases took in average 5 min for each platform.

Taking into account that in a software system designed with 11 classes and 60 services the testing process takes around 5 min (0.5 % of the mean time needed using manual testing), we can state that the T4MDD and CONT4MDD are very efficient techniques regarding time reduction in comparison with traditional testing techniques.

### 5.4 Threats to validity

There are some threats to validity of the application of T4MDD and CONT4MDD techniques that are specified in order to denote that the results are not biased by the researchers' point of view. We identify some threats regarding construct, internal, and external validity (Runeson and Host 2009).

Regarding the construct validity, which reflects to which extent the variables that are studied really represents that the researcher have in mind, we identify that the time used to define the test cases depend on the experience of subjects. To mitigate this threat, we select subjects with similar level of expertise.

المنارة للاستشارات

Regarding the internal validity, which expresses that the design and the analysis may be compromised by the existence of other unexpected sources of bias, we identify that the experience of subjects that are specifying the OO-Method model is a threat. People with expert level could take less time to develop the model, so that to mitigate this risk we select people with similar level of expertise than the subjects that apply traditional testing techniques. In addition, there may exist a learning effect of the subjects that manually programmed the SICOVE project functionality and then generated the test cases. To avoid this threat to internal validity, we crossed the subjects, java and C# programmers, after the programming tasks were performed. In any case, if a learning effect appears, it was in favor of the traditional testing method, not for T4MDD and CONT4MDD techniques. Thus, the comparison of results is conservative.

Regarding to the external validity, which reflects to which extend it is possible to generalize the findings, we identify the representativeness of the selected case as a threat, since all the models correspond to a specific MDD approach. This may provoke that the results are valid only on this industrial context. Repeating the application of the proposed testing techniques with other MDD approaches will give more information about the generalization of the results. In addition, the representativeness of the traditional testing technique is a threat. We select manual testing as the traditional testing techniques. However, a repetition with other testing techniques would generate different results.

During the application of T4MDD and CONT4MDD techniques, we learned several lessons such as (1) the MDD approach was selected based on the researcheŕs contact network, and the application of the proposed testing techniques over other MDD approaches would support the generalization of these techniques; (2) the results obtained corresponding to quantitative data, qualitative information (for instance the perception of the ease of use of these techniques) would help to understand the intention to use T4MDD and CONT4MDD techniques.

## 6 Conclusions

Since MDD enables the automatic code generation of a system from a conceptual model specification, MBT seems to be the natural alternative to efficiently perform the testing of applications developed under this paradigm. Thus, we have presented a MBT technique that takes advantage of the conceptual models defined in an MDD approach to automatically generate the test cases.

In this paper, we have demonstrated the applicability of MBT to the MDD paradigm in order to reduce the time needed for software production. To do this, we have presented T4MDD, a MBT technique that automatically generates abstract test cases in an XML representation for a particular conceptual model. T4MDD uses the OO-Method conceptual model, which is a MDD method with more than 10 years of successful application in the market (Integranova 2015). We have also presented CONT4MDD, a MBT technique that generates concrete test cases in two languages: C# and Java. This technique consumes the abstract test cases generates by the T4MDD and automatically generates executable (concrete) test cases. This is an important contribution since MBT proposals of the state of the art are focused in the generation of abstract test cases, and therefore, the concrete test cases must be manually programmed.

Even though the proposed MBT techniques have been defined for a particular MDD approach (OO-Method), it is important to note that the conceptual model of OO-Method



المنارة للاستشارات

has conceptual constructs that are well-known and well-covered by UML (OMG 2011). In addition, by using UML profiles it is possible to project our contributions to any MDD approach that is built using UML diagrams. Thus, other MDD approaches could potentially benefit from the ideas and results presented in this paper.

The contribution presented is useful for researchers and practitioners. Researchers can use the proposed MBT proposal to generate new approaches related to different MDD methods. Practitioners can use the MBT approach to alleviate the time required to design and execute test cases.

Both techniques (T4MDD and CONT4MDD) have been applied to an industrial project, which allows the verification of the results obtained for each testing criterion. The case also demonstrates the efficiency of these techniques regarding traditional testing techniques. In addition, it is recommended that the software engineers that develop a software system do not participate in the testing process in order to avoid possible bias in the design of the test cases. With the proposed techniques we also support this recommendation.

Furthermore, T4MDD and CONT4MDD implement nine testing criteria in contrast to other MBT techniques that are focused in just one testing criterion. In addition, it is possible to extend T4MDD and CONT4MDD to include more testing criteria in order to achieve a quality assurance process as automatic as possible. As future work we plan to add other testing criteria for the presentation view of the OO-Method conceptual model in order to obtain results regarding the usability of the generated applications. We also plan to generate concrete test cases for other programming languages. Finally, we plan to perform further empirical studies to give more soundness to our proposal for testing MDD applications.

**Acknowledgments** This work has been developed with the support of Universidad Diego Portales and FONDECYT under the project TESTMODE 11121395 and the project AMoDDI 11130583.

## References

Amalfitano, D., Fasolino, A. R., Tramontana, P., Ta, B. D., & Memon, A. M. (2015). MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software, 32*(5), 53–59.

Berkenkötter, K. (2008). Reliable UML models and profiles. *Electronic Notes in Theoretical Computer Science, 217*, 203–220.

Bigot, C., Faivre, A., Gallois, J. -P., Lapitre, A., Lugato, D., Pierron, J. -Y., & Rapin, N. (2003). Automatic test generation with AGATHA. In *9th international conference tools and algorithms for the construction and analysis of systems*, Vol. LNCS, pp. 591–596. Berlin: Springer.

Blanco, R., & Tuya, J. (2015). A test model for graph database applications: an MDA-based approach. In: *Proceedings of the 6th international workshop on automating test case design, selection and evaluation (A-TEST 2015)*, pp. 8–15. New York: ACM. doi:10.1145/2804322.2804324

Botteck, M., & Deiß, T. (2008). 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, 10*(4), 285–289.

Brambilla, M., Cabot, J., & Wimmer, M. (2012). *Model-driven software engineering in practice*. London: Morgan & Claypool Publishers. doi:10.2200/S00441ED1V01Y201208SWE001.

Brucker, A., Krieger, M., Longuet, D., & Wolff, B. (2011). A specification-based test case generation method for UML/OCL. In *Workshops and symposia at MODELS 2010: Models in software engineering*, Vol. LNCS, pp. 334–348. Berlin: Springer.

Castillos, K. C., Dadeau, F., & Julliand, J. (2011). Scenario-based testing from UML/OCL behavioral models. *International Journal on Software Tools for Technology Transfer, 13*(5), 431–448.

Chen, R., & Miao, H. (2013). A selenium based approach to automatic test script generation for refactoring javascript code. In *IEEE/ACIS 12th international conference on computer and information science (ICIS)*, pp. 341–346.

المنارة للاستشارات

Coleman, D., Arnold, P., Godoff, S., Dollin, C., Gilchrist, H., Hayes, F., & Jeremaes, P. (1994). *Object-oriented development: The fusion method*. Englewood Cliffs, NJ: Prentice-Hall.

Collofello, J., & Vehathiri, K. (2005). An environment for training computer science students on software testing. In *35th ASEE/IEEE Frontiers in education conference T3E-7*, p. 6. IEEE. doi:10.1109/FIE.2005.1611937

Conformiq: Conformiq Designer. https://www.conformiq.com/products/conformiq-designer/.

da Silveira, M. B., Rodrigues, E. d. M., Zorzo, A. F., Costa, L. T., Vieira, H. V., & de Oliveira, F. M. (2011). Generation of scripts for performance testing based on UML models. In *23rd International conference on software engineering and knowledge engineering*, pp. 258–263.

Dalal, S., Jain, A., Karunanithi, N., Leaton, J., Lott, C., Patton, G., & Horowitz, B. (1999). Model-based testing in practice. In *21st International conference on software engineering (ICSE'99)*, pp. 285–294. New York: ACM Press.

Dias Neto, A. C., Subramanyan, R., Vieira, M., & Travassos, G. H. (2007). A survey on model-based testing approaches: A systematic review. In *1st ACM international workshop on Empirical assessment of software engineering languages and technologies (WEASELTech '07)*, pp. 31–36. New York: ACM.

Elvior: MOTES. http://www.elvior.com.

Engels, G. (2009). Automatic generation of behavioral code-too ambitious or even unwanted? Behavior modeling in model driven architecture. In *Proceedings of first European workshop on behavior modeling in model driven architecture (BM-MDA)*.

Farooq, U., & Lam, C. P. (2009). Evolving the quality of a model based test suite. In *International conference on software testing, verification and validation workshops (ICSTW '09)*, pp. 141–149. IEEE. doi:10.1109/ICSTW.2009.27

Fourneret, E., Bouquet, F., Dadeau, F., & Debricon, S. (2011). Selective test generation method for evolving critical systems. In *REGRESSION'11, 1st international workshop on regression testing—co-located with ICST'2011*, pp. 125–134. IEEE. New York: Computer Society Press. doi:10.1109/ICSTW.2011.95

France, R. B., Ghosh, S., Dinh-Trong, T., & Solberg, A. (2006). Model-driven development using UML 2.0: Promises and pitfalls. *IEEE Computer, 39*(2), 59–66.

Fujiwara, S., Munakata, K., Maeda, Y., Katayama, A., & Uehara, T. (2011). Test data generation for web application using a UML class diagram with OCL constraints. *Innovations in Systems and Software Engineering, 7*(4), 275–282.

Gibson, A. (2015). *Data factory*. https://github.com/andygibson/datafactory.

Gutierrez, J., Escalona, M., Mejias, M., Ramos, I., & Torres, J. (2009). An approach for Model-Driven test generation. In *International conference on research challenges in information science*, pp. 303–312.

IEEE (1984). *IEEE 830 guide to software requirements specifications*.

Integranova (2015) Web Page. Last visited June 2015, http://www.integranova.com.

Iyenghar, P., Pulvermueller, E., & Westerkamp, C. (2011). Towards model-based test automation for embedded systems using UML and UTP. In *IEEE 16th conference on emerging technologies & factory automation (ETFA)*, 2011, pp. 1–9. IEEE. doi:10.1109/ETFA.2011.6058982

Koopman, P., Achten, P., & Plasmeijer, R. (2008). Model-based testing of thin-client web applications and navigation input. In *10th International symposium of practical aspects of declarative languages (PADL),* Vol. LNCS, pp. 299–315. Berlin: Springer.

Lasalle, J., Peureux, F., & Fondement, F. (2011). Development of an automated MBT toolchain from UML/SysML models. *Innovations in Systems and Software Engineering, 7*(4), 247–256.

Marín, B., Giachetti, G., & Pastor, O. (2008). Automating the measurement of functional size of conceptual models in an MDA environment. In *Product-focused software process improvement (PROFES). LNCS*, pp. 215–229. Berlin: Springer. doi:10.1007/978-3-540-69566-0_19

Marín, B., Pastor, O., & Abran, A. (2010). Towards an accurate functional size measurement procedure for conceptual models in an MDA environment. *Data & Knowledge Engineering, 69*(5), 472–490.

Marín, B., Pereira, J., Giachetti, G., Hermosilla, F., & Serral, E. (2013). A general framework for the development of MDD projects. In *1st International conference on model-driven engineering and software development—MODELSWARD 2013*, pp. 257–260. Setubal: SciTe Press.

Marín, B., Vos, T., Giachetti, G., Baars, A., & Tonella, P. (2011). Towards testing future web applications. In *5th International conference on research challenges in information science (RCIS 2011), IEEE Computer Society*, pp. 226–237.

Mlynarski, M. (2010). Holistic model-based testing for business information systems. In *Third international conference on software testing, verification and validation (ICST)*, 2010, pp. 327–330. IEEE. doi:10.1109/ICST.2010.35

Moreno, N., Fraternali, P., & Vallecillo, A. (2007). WebML modeling in UML. *IET Software, 1*(3), 67–80.



المنارة للاستشارات

Nylund, K., Ostman, E., Truscan, D., & Teittinen, R. (2011). Towards rapid creation of test adaptation in on-line model-based testing. In *COMPSAC workshops*, pp. 174–179.

OMG (2011). *Unified modeling language (UML) 2.4.1 superstructure specification*.

Opdahl, A. L., & Henderson-Sellers, B. (2005). A unified modelling language without referential redundancy. *Data & Knowledge Engineering, 55*(3), 277–300.

Pastor, O., Gómez, J., Insfrán, E., & Pelechano, V. (2001). The OO-Method approach for information systems modelling: From Object-oriented conceptual modeling to automated programming. *Information Systems, 26*(7), 507–534.

Pastor, O., Hayes, F., & Bear, S. (1992). OASIS: An object-oriented specification language. In: *International conference on advanced information systems engineering (CAiSE)*, pp. 348–363.

Pastor, O., Molina, J. C., & Iborra, E. (2004). *Automated production of fully functional applications with OlivaNova model execution*. ERCIM News no. 57.

Pérez-Lamancha, B., Polo, M., Caivano, D., Piattini, M., & Visaggio, G. (2013). Automated generation of test oracles using a model-driven approach. *Information and Software Technology, 55*(2), 301–319.

Reza, H., Ogaard, K., & Malge, A. (2008). A model based testing technique to test web applications using Statecharts. In *Fifth international conference on information technology: New generations, ITNG 2008*, IEEE. doi:10.1109/ITNG.2008.145

Rodrigues da Silva, A. (2015). Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures, 43*, 139–155.

Runeson, P., & Host, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering Journal, 14*(2), 131–164.

Seifert, D. (2008). Conformance testing based on UML state machines. In *10th International conference on formal engineering methods, ICFEM 2008*, vol. LNCS 5256, pp. 45–65. Berlin: Springer.

Selic, B. (2003). The pragmatics of model-driven development. *IEEE Software, 20*(5), 19–25.

Slaughter, S. A., Harter, D. E., & Krishnan, M. S. (1998). Evaluating the cost of software quality. *Communications of the ACM, 41*(8), 67–73.

Smartesting. *Test designer*. http://www.smartesting.com.

SourceForge.net. ParteG, http://parteg.sourceforge.net.

Thayer, R. H., Slaughter, J. B., Boehm, B. W., Clapp, J. A., Manley, J. H., & Burrows, J. H. (1974). The high cost of software: causes and corrections. In: *Proceedings of the national computer conference and exposition AFIPS '74*, pp. 1009–1009.

Timmer, M., Brinksma, E., & Stoelinga, M. (2011). *Model-based testing. Software and systems safety—specification and verification. NATO science for peace and security series—D: Information and communication security*, Vol. 30. Amsterdam: IOS Press.

Utting, M., & Legeard, B. (2007). Practical model-based testing—A tools approach. Los Altos, CA: Morgan Kaufmann. ISBN: 978-0-12-372501-1.

Utting, M., Pretschner, A., & Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability, 22*(5), 297–312.

Wieczorek, S., Stefanescu, A., Fritzsche, M., & Schnitter, J. (2008). Enhancing test driven development with model based testing and performance analysis. In: *TAIC-PART '08 proceedings of the testing: Academic & industrial conference—Practice and research techniques*, pp. 82–86. IEEE.

Xu, D., Kent, M., Thomas, L., Mouelhi, T., & Le Traon, Y. (2015). Automated model-based testing of role-based access control using predicate/transition nets. *IEEE Transactions on Computers, 64*(9), 2490–2505.

Yang, R., Chen, Z., Xu, B., W. E., W., & Zhang, J. (2011). Improve the effectiveness of test case generation on EFSM via automatic path feasibility analysis. In *IEEE 13th International Symposium on High-Assurance Systems Engineering (HASE)*, pp. 17–24. IEEE. doi:10.1109/HASE.2011.12

Yuan, Q., Wu, J., Liu, C., & Zhang, L. (2008). A model driven approach toward business process test case generation. In *10th International symposium on web site evolution (WSE 2008)*, pp. 41–44. IEEE. doi:10.1109/WSE.2008.4655394

Zeng, F., Chen, Z., Cao, Q., & Mao, L. (2009). Research on method of object-oriented test cases generation based on UML and LTS. In *1st international conference on information science and engineering (ICISE)*, pp. 5055–5058. IEEE. doi:10.1109/ICISE.2009.965

المنارة للاستشارات

**Beatriz Marín** is a full professor at Universidad Diego Portales (UDP). She completed her PhD in Computer Science at Universidad Politécnica de Valencia (2011). She has published papers at top conferences such as CAISE, ESEM, QSIC, MENSURA, EUROSPI, RCIS, and SEKE, and has published articles in journals such as Data & Knowledge Engineering (DKE), Journal of Universal Computer Science (JUCS), International Journal of Software Engineering and Knowledge Engineering (IJSEKE), and ACM Transactions on Software Engineering and Methodologies (TOSEM). She has experience working in industry as project leader by more than five years, and also, has experience in teaching at university level (undergraduate and postgraduate) and in supervision of PhD and Master Students. Beatriz is a member of the Program Committee of the MENSURA conference, the SEKE conference, the CIBSE conference, the ATSE Workshop, and the QUAMES Workshop. Her main research area is software engineering, with specific interest in quality, testing, functional size measurement, model-driven development, and empirical software engineering.

**Carlos Gallardo** is a software engineer who works in industry. He completed his grade on Computer Science and Telecommunications at Universidad Diego Portales (2013). His main research area is software engineering, with special focus at testing and development.

**Diego Quiroga** is a software engineer that works in industry. He completed his studies on Computer Science and Telecommunications at Universidad Diego Portales (2014). His main research area is software testing.



المنارة للاستشارات

**Giovanni Giachetti** Ph.D. is full professor and director of software engineering magister program at Universidad Andrés Bello—UNAB (Chile). He counts with more than 10 years of professional experience in coordination and development of national and international projects at industrial and research level. He has written articles for different conferences and journals, such as CAiSE, RCIS, ESEM, EUROSPI, MENSURA, JUCS, TOSEM, and has participated in relevant research committees. He currently works in model-driven engineering and model-driven interoperability leading the software engineering research area in his university (UNAB). His research experience is centered on the domains of model-driven development, requirements engineering, databases systems, knowledge engineering, software Q&A, software measurement, and software testing.

**Estefanía Serral** is a postdoc researcher at LIRIS in KU Leuven, Belgium. Her main research areas are: business process modeling and execution, pervasive computing, adaptive systems, context-awareness, semantic integration, semantic knowledge modeling and management, model-driven engineering, runtime software evolution, and models at runtime. Before working at KU Leuven, she worked as a postdoc at the CDL-Lab in the technical University of Vienna, Austria, and in the Research Center on Software Production Methods (PROS Research Center) at the Technical University of Valencia, Spain. Dr. Serral has many publications in high-ranking conferences and journals, such as CAiSE, ER, UIC, PMC, SOSYM, MTAP. Moreover, she is also a co-organizer of the International Workshop on Semantic Ambient Media Experience (SAME) since 2011, has been session chair and member of the organizing committee in several international conferences, has collaborated with numerous researchers all around the world, and has participated as a reviewer in relevant international journals. She has a degree in computer science; a aster degree on software engineering, formal methods and information systems; and PhD in computer science.

المنارة للاستشارات

Reproduced with permission of copyright owner. Further reproduction prohibited without permission.

المنارة للاستشارات