Specification-driven test generation for model transformations

Esther Guerra

Universidad Autónoma de Madrid (Spain)
`Esther.Guerra@uam.es`

Abstract. Testing model transformations poses several challenges, among them the automatic generation of appropriate input test models and the specification of oracle functions. Most approaches to the generation of input models ensure a certain level of source meta-model coverage, whereas the oracle functions are frequently defined using query or graph languages. Both tasks are usually performed independently regardless their common purpose, and sometimes there is a gap between the properties exhibited by the generated input models and those demanded to the transformations (as given by the oracles).
Recently, we proposed a formal specification language for the declarative formulation of transformation properties (invariants, pre- and postconditions) from which we generated partial oracle functions that facilitate testing of the transformations. Here we extend the usage of our specification language for the automated generation of input test models by constraint solving. The testing process becomes more *intentional* because the generated models ensure a certain coverage of the interesting properties of the transformation. Moreover, we use the same specification to consistently derive both the input test models and the oracle functions.

1 Introduction

Model transformations are the pillars of Model-Driven Engineering (MDE), and therefore they should be developed using sound engineering principles to ensure their correctness [12]. However, most model transformation technologies are nowadays centered on supporting the implementation phase, and few efforts are directed to the specification of requirements, design or testing of transformations. As a consequence, transformations are frequently hacked, not engineered, being hard to maintain, incorrect or buggy.

In order to alleviate this situation, we proposed in the past *trans*ML, a family of modelling languages for the *engineering* of transformations using an MDE approach [12]. *trans*ML provides support for the gathering of requirements, its formal specification, the architectural, high- and low-level design, as well as the specification of test scripts, which themselves are also models. An engine called *mtUnit* is able to execute these test suites in an automated way.

*trans*ML includes a visual language with formal semantics called PaMoMo (Pattern-based Model-to-Model Specification Language) [11] for the contract-based specification of transformation requirements. In this way, the designer may specify requirements of the input models of a transformation (preconditions), expected properties of the output models (postconditions), as well as properties that any pair of input/output models should satisfy (invariants). Similar to software requirement specification languages like Z or Alloy, PaMoMo's formal semantics enables reasoning at the level of requirements, while being independent of the particular transformation language used for the implementation.

In [11], we explored the use of PaMoMo for testing. In particular, we showed how to automatically derive OCL partial oracle functions from PaMoMo specifications, and used these oracles to assert whether a particular implementation satisfied a specification. Still, the transformation tester had the burden to: (a) produce a reasonable set of input test models, (b) build a *mtUnit* script to exercise the transformation with the different input models, and (c) select the partial oracle functions produced from the specification to assert whether the tests passed or failed. In particular, the manual creation of input models is tedious and time-consuming, and it does not guarantee an appropriate coverage of all requirements in the specification.

In this paper, we tackle these problems by deriving, from the transformation specification, not only the oracle functions, but also a set of input test models ensuring a certain level of coverage of the properties in the specification. These input models are calculated using constraint solving techniques. Besides, a dedicated *mtUnit* test suite is generated for the automated testing of the transformation implementation using the generated input models and oracle functions.

While there are several approaches for the automated testing of transformations, ours is unique because our test models aim at testing the requirements and properties of interest as given in a specification. Current approaches either focus on producing input test models ensuring a certain coverage of the input meta-model [7, 21], or do not consider specification-based testing. Hence, our approach is directed to test the *intention* of the transformation. Moreover, the use of the same specification to consistently derive both the input models for testing and the oracle functions is also novel.

Paper organization: Section 2 reviews existing approaches to model transformation testing. Then, Section 3 sketches our proposal. Section 4 introduces our specification language PaMoMo, whereas Section 5 describes our approach to derive input test models with a certain level of specification coverage. We present tool support in Section 6 and discuss some conclusions in Section 7.

2 State of the art

There are three main challenges in model transformation testing [2]: the generation of input test models, the definition of test adequacy (or coverage) criteria, and the construction of oracle functions.

Most works dealing with the generation of input test models consider only the features of the input meta-model but not properties of the transformation (i.e. they support black-box testing). For instance, in [7, 21], the authors perform automatic generation of input test models based on the input meta-model and some coverage criteria (e.g. partitioning of attribute values and number of classes). In [10], the generation of input test models must be hand-coded using an imperative language with features for randomly choosing attribute values and association ends. There are a few white-box testing approaches, like [15] where the authors propose using all possible overlapping models of each pair of rules in a transformation as input models for testing.

Regarding the third challenge, we distinguish between complete and partial oracle functions. The former are defined by having the output models at hand. For instance, the test cases for the C-SAW transformation languages [16] consist of a source model and its expected output model. Partial oracle functions express contracts that the input and output models of a transformation should fulfil. Most proposals to partial oracle functions use OCL to specify the contracts [6, 10, 18]. The approaches in [8, 9] follow a similar philosophy to the xUnit framework, and the oracle functions can be specified as OCL/EOL assertions. Finally, some approaches permit the specification of partial oracle functions as graph patterns or model fragments [1]. None of these approaches provide a mechanism to assert the adequacy of the specified tests and automate their generation.

In conclusion, we observe that some transformation testing approaches provide automated test execution [8, 9], but do not support the generation of input models, and the oracle needs to be specified manually. Other works focus on the automatic generation of input models [7, 21], but without considering transformation properties. Finally, the works proposing contracts for specifying transformations do not use the contracts for input test generation. In this paper, we will present our approach to specification-based transformation testing which automates the generation of the input test models, the oracle function and executable test scripts from the same transformation specification.

It is worth noting that the idea of synthesizing both input test data and oracle functions from a specification has been successfully applied to general software testing, if we look at the broader scope of model-based testing. For instance, in [3], the authors generate both artifacts for automated testing of Java programs based on Java predicates from which all possible non-isomorphic inputs (up to a certain size bound) are efficiently generated. This yields complete coverage of the input state space. In our case, we aim at generating test models exhibiting relevant properties; complete meta-model coverage (i.e. generating all meta-model instances of a certain size) does not guarantee this, and may lead to the so-called state explosion problem. The model-based testing approach in [23] uses symbolic execution to generate unit tests ensuring a certain path coverage, i.e., it supports white-box testing. In our case, we follow a black-box testing approach.

3 A framework for specification-driven testing

Fig. 1 shows the working scheme of our approach. First, the designer specifies the requirements (i.e. the pre/postconditions and invariants) of the transformation using our language PaMoMo. The developer can use this specification as a guide to implement the transformation using his favourite language (e.g. ATL [13], ETL [14], etc.). Starting from the specification, the transformation tester can automatically generate a complete test suite which can be directly used to test the transformation implementation. This test suite comprises: (i) an oracle function that encodes the invariants and postconditions in the specification as assertions [11]; (ii) a set of input test models which enables the testing of all requirements in the specification according to certain coverage criteria; and (iii) a test script that automates the execution of the transformation for each test model, checks the conformance of the result using the oracle function, and reports any detected error using our *mtUnit* engine [12].

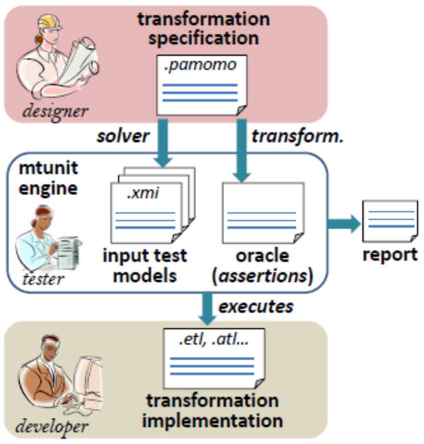


Fig. 1. Framework.

4 A specification language for model transformations

PaMoMo is a formal, pattern-based, declarative, bidirectional specification language to describe, in an implementation-independent way, correctness requirements of the transformations and of their input and output models [11]. These requirements may correspond to *preconditions* that the input models should fulfil, *postconditions* that the output models should fulfil (beyond meta-model constraints), as well as *invariants* of the transformation (i.e. requirements that the output model resulting from a particular input model should satisfy).

Preconditions, postconditions and invariants are represented as graph patterns, which can be positive to specify expected model fragments, or negative to specify forbidden ones. They can have attached a logical formula stating extra conditions, typically (but not solely) constraining the attribute values in the graph pattern. In this paper and in our prototype tool, these formulas are written in OCL. Optionally, patterns can define one enabling condition and any number of disabling conditions, to reduce the scope of the pattern to the locations where the enabling condition is met, and the disabling conditions are not.

Formally, an invariant $I = (C, C_{en}, \{C_{dis}\})$ is made of a main constraint C, an enabling condition C_{en} (which may be empty), and a set $\{C_{dis}\}$ of disabling conditions. The main constraint and conditions $C_x = \langle G_s, G_t, \alpha \rangle$ are made of two graphs typed by the source and target meta-models of the transformation, and a formula α over their elements. A positive invariant holds on a pair of source and target models if: (i) for each occurrence Occ of the source graph of the main constraint C plus the enabling condition C_{en}, (ii) if there is no occurrence of the disabling conditions $\{C_{dis}\}$ in the context of Occ, (iii) then there is an occurrence

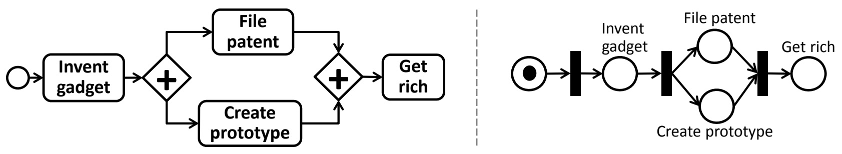


Fig. 2. BPMN model (left) and equivalent Petri net (right).

of the target graph of C in the context of *Occ*. If the invariant is negative, then we should not find an occurrence of the target graph of C in step (iii). Pre- and postconditions have the same structure as invariants, but the target (source) graph in preconditions (postconditions) is always empty. Their interpretation is also different. A pre/postcondition holds if, for each occurrence of the enabling condition, there is an occurrence of the main constraint for which no occurrence of the disabling conditions is found. See [11] for a complete description of the formal semantics of PaMoMo.

As a running example, we use PaMoMo to specify a transformation from the Business Process Modeling Notation (BPMN) [4] to Petri nets. The goal is to analyse BPMN models to detect deadlocks, incorrect termination conditions or tasks that can never be completed. The left of Fig. 2 shows a BPMN model. It specifies a flow initiated in a start event (the circle), and consisting in the completion of different tasks (rounded rectangles). The diamonds in the model are called parallel gateways, and split the execution in several parallel branches (first gateway) which are later synchronized (second gateway). From this BPMN model, our transformation should create a Petri net like the one to its right.

The left of Fig. 3 shows some transformation preconditions, expressing requirements that any input model should fulfil beyond its meta-model constraints[1]. For instance, our transformation expects models with one start event from which only one sequence flow goes out. This is formalised by the positive precondition *OneStartEvent* (i.e. there must exist one start event with one outgoing flow in the input model) and the negative precondition *MultipleStartEvents* (there cannot be several start events). These conditions are not demanded by the BPMN meta-model, which allows models with any number of start events, each one of them with multiple outgoing flows, but are required by our transformation. The figure shows another precondition, *PathsForGateway*, with an enabling condition. It demands that each gateway (enabling condition) defines at least one input and one output flows (main constraint). The precondition contains the abstract class *Gateway*, becoming applicable to all concrete gateway types inheriting from it.

Postconditions express requirements of the output models, beyond their meta-model constraints. The right of Fig. 3 shows some postconditions for the generated Petri nets, like the absence of unconnected places (*UnconnectedPlaces*), the existence of input and output places for all transitions (*ConnectedTransitions*),

[1] In this section, we use a graphical concrete syntax for the specification. In Section 6, we will show an alternative textual syntax that is supported by our prototype tool.

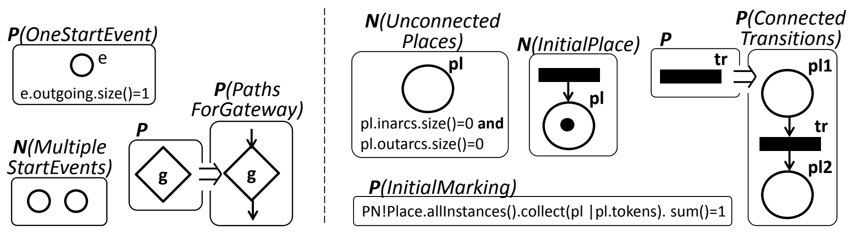


Fig. 3. Preconditions (left) and postconditions (right) of the transformation.

and the existence of a single place with one token and without input transitions (*InitialPlace* and *InitialMarking*).

Finally, Fig. 4 shows some invariants describing the transformation of tasks and gateways. Tasks must be transformed into equally named places (*Task1*). Since tasks can only have one outgoing flow, the corresponding places cannot be connected to two output transitions (*Task2*). Each parallel gateway should be transformed into a transition (*ParallelGateway1*), and the places for all incoming tasks to the gateway should be input to the transition (*ParallelGateway2*). Invariant *ParallelGateway3* states that if a parallel gateway does not have a task *t2* as input (disabling condition), then the place for *t2* cannot be connected to the transition (as the invariant is negative). There are similar invariants for the tasks going out from a parallel gateway. The two remaining invariants state that exclusive (also called choice) gateways should be transformed into an intermediate place, plus one transition for each outgoing branch.

The use of a formal specification language like PaMoMo to specify transformation properties has the following advantages: (i) it enables reasoning on the transformation requirements before their implementation, as well as detecting contradictions in the requirements early in the project; (ii) it provides a high-level notation to specify pre/postconditions and invariants of the transformations; and (iii) it is possible to automate the generation of an oracle function from the specification and use it for automated testing [11]. However, the challenge of generating appropriate input test models remains, as these have to be built by hand, which is a tedious and error-prone task. Moreover, it is difficult to ensure that the input test set will enable the testing of all relevant properties in the specification. To solve this problem, next we present an approach to generate input test models ensuring the coverage of a specification.

5 Specification-driven generation of input test models

Our approach to specification-driven testing consists of the following steps: (1) translation of the properties in the specification into a suitable format for model finding, (2) selection of a level of specification coverage, resulting in a particular strategy to build expressions that demand the satisfaction (or not) of a number of properties in the generated models, (3) use of a constraint solver to find

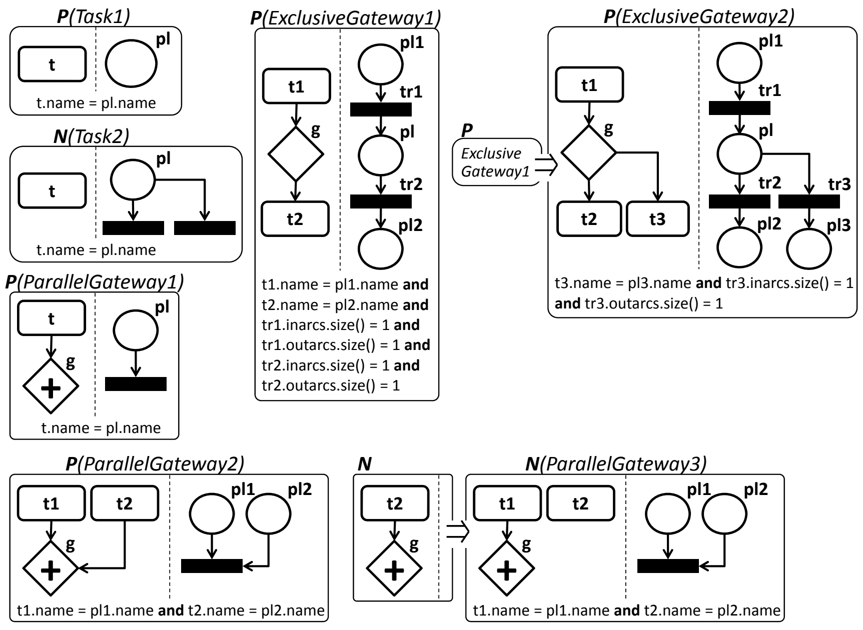

P(Task1)
t
pl
t.name = pl.name
N(Task2)
t
pl
t.name = pl.name
P(ParallelGateway1)
t
g
pl
t.name = pl.name
P(ExclusiveGateway1)
t1
g
t2
pl1
tr1
pl
tr2
pl2
t1.name = pl1.name and t2.name = pl2.name and tr1.inarcs.size() = 1 and tr1.outarcs.size() = 1 and tr2.inarcs.size() = 1 and tr2.outarcs.size() = 1
P
Exclusive Gateway1
P(ExclusiveGateway2)
t1
g
t2
t3
pl1
tr1
pl
tr2
tr3
pl2
pl3
t3.name = pl3.name and tr3.inarcs.size() = 1 and tr3.outarcs.size() = 1
P(ParallelGateway2)
t1
t2
g
pl1
pl2
t1.name = pl1.name and t2.name = pl2.name
N
t2
g
N(ParallelGateway3)
t1
t2
g
pl1
pl2
t1.name = pl1.name and t2.name = pl2.name

Fig. 4. Invariants of the transformation.

models satisfying concrete combinations of properties and the input meta-model integrity constraints, and finally, (4) identification of the assertions that should be checked after testing the transformation with a particular input model. In the remaining of this section we present in detail this procedure.

5.1 Translation of properties in the specification

As a first step, we translate the specification into a language that allows automating the generation of models. In particular, we use OCL as target language because we already had support to generate OCL assertions from our specifications [11], there are available solvers that find models satisfying a set of OCL constraints [5], and we do not need to parse the OCL formulas in the properties of the specification to a different language. Nonetheless, this is our particular option and the framework could be used with a different target language whenever a translation from our specification language is provided.

Although a specification includes preconditions, postconditions and invariants, only preconditions and invariants contain useful information for the input model generation. Postconditions refer to properties of the output models and are only used to generate oracle functions, but not input models.

An invariant expresses a property of the form: *if certain source pattern appears in the input model, then certain target pattern should be present (or not) in the output model.* Thus, it is interesting to generate input models containing instances of the source pattern, to test whether transforming these models actually yields output models containing the target pattern. For this purpose, from each PaMoMo invariant we generate an OCL expression which characterises the source pattern of the invariant. Listing 1 shows a scheme of the generated expression. It iterates on the objects of the source graph of the main constraint (lines 1–3), and checks that there is no occurrence of the source graph of any disabling condition (lines 4–7, this code is generated for each disabling condition). The function `conditions` corresponds to an OCL expression checking the conditions that the traversed objects should fulfil, namely the existence of the links specified in the invariant (o_i.link->includes(o_j)), inequalities for the objects with same type (o_i<>o_j), and all terms in the invariant formula over elements of the input domain only. The enabling condition of the invariants is ignored as we do not want models where the invariant is satisfied vacuously due to the absence of the enabling condition in the models. Moreover, if the invariant is negative, the generated expression is the same (i.e. it is not preceded by the `not` particle) because the source part of the invariant is still positive (*if X appears then...*).

```
o1.type::allInstances()−>exists(o1 | ...
 oi.type::allInstances()−>exists(oi |
  conditions(o1,...,oi)
  < and not
     oj.type::allInstances()−>exists(oj | ...
      ok.type::allInstances()−>exists(ok |
       conditions(o1,...,oi,oj,...,ok) >∗ ) ...)
```

Listing 1. OCL for invariants.

```
< not >?
< o1.type::allInstances()−>forAll(o1 | ...
    oi.type::allInstances()−>forAll(oi |
     conditions(o1,...,oi)
     implies >?
     oj.type::allInstances()−>exists(oj | ...
       ok.type::allInstances()−>exists(ok |
         conditions(o1,...,oi,oj,...,ok)
         < and not
            ol.type::allInstances()−>exists(ol | ...
              om.type::allInstances()−>exists(om |
                conditions(o1,...,oi,oj,...,ok,ol,...,om) >∗ ...)
```

Listing 2. OCL for preconditions.

As an example, from invariant *ParallelGateway3* we generate the expression:

```
Task::allInstances()−>exists(t1 |
  Task::allInstances()−>exists(t2 |
    ParallelGateway::allInstances()−>exists(g |
      t1.outgoing−>includes(g) and t1<>t2 and not t2.outgoing−>includes(g) )))
```

Frequently, specifications include invariants with same source and different target. For instance, *Task1* and *Task2* have both a task as source, whereas the former specifies how to translate a task correctly, the latter identifies an incorrect translation. In this case, generating an input model containing a task enables the testing of both invariants. Thus, from the set of generated OCL expressions, we eliminate redundant source conditions (i.e. equal source in the main constraint and disabling conditions). We do not eliminate subsumptions to allow for the testing of models with different size and context conditions.

Finally, preconditions specify requirements of the input models of a transformation. A transformation is not demanded to work properly for input models that do not satisfy these preconditions. The validity of the input models is hardly ever done by the transformation, but by an external procedure, or otherwise it is ensured by the transformation application context. Thus, we take the convention that all generated input models must fulfil all preconditions in the specification. For this purpose, we generate an OCL constraint from each precondition, and enforce their satisfaction in all generated input models by adding them to the expressions used to generate them (see next subsection). The scheme of the generated OCL code is shown in Listing 2. The expression looks for all occurrences of the enabling condition (lines 2–4), and demands that for each one of them there is an occurrence of the main constraint (lines 6–8) satisfying the disabling conditions (lines 9–10). If the precondition has no enabling condition the resulting expression is the same as the one for invariants, and if it is negative, the generated expression is preceded by `not`.

5.2 Input model generation for different coverage criteria

The model generation process is performed in two steps. First, we compose an OCL expression for each input model to be generated, identifying the properties that this model should fulfil. These expressions are built according to certain specification coverage criteria. Then, we feed each expression, together with the input meta-model and the OCL code generated from the preconditions, to a constraint solver. The solver will try to find a valid input model satisfying the given OCL expression, preconditions and meta-model integrity constraints. For a particular expression, the solver may not find a model in the given scope, or due to some inconsistency in the specification. In such a case, we can either widen the search scope, or do not generate a model for that particular expression.

We identify seven levels of specification coverage for the generated test set, with increasing degrees of exhaustivity: *property*, *closed property*, *t-way*, *closed t-way*, *combinatorial*, *closed combinatorial* and *exhaustive*. The property, t-way and exhaustive levels generate models enabling the testing of a number of invariants in the specification by combining their source models. The remaining levels generate also models that do not contain occurrences of certain invariants. In the following, we present in detail each one of them.

Property coverage. This is the least exhaustive level of coverage, appropriate when the invariants in the specification are independent. It generates as many input models as invariants in the specification, each one including at least one occurrence of the source of an invariant. The rationale is to use each generated model to test one property of the transformation, given by one invariant in the specification. For this purpose, given a specification with $I = \{I_1, ..., I_n\}$ invariants (with different source), we generate n expressions of the form $ocl(I_i)$. Each expression demands the existence of an occurrence of the source of invariant I_i. As an example, Table 1 shows in the first column the expressions generated from a specification with three invariants, where each i_x term represents the OCL code generated from the invariant x.

Closed property coverage. This criterion extends the previous one by generating additional models that do not contain occurrences of the source of some invariant in the specification. The goal is checking whether the transformation under test handles properly the absence of certain patterns in the input models. These limit cases, usually due to underspecifications, frequently lead to errors in the final implementations, yielding malformed output models.

Thus, given a specification with $I = \{I_1, ..., I_n\}$ invariants, we also generate n additional expressions of the form *not* $ocl(I_i)$. The second column of Table 1 shows the generated expressions assuming three invariants.

Interestingly, any model that does not contain the source of an invariant will satisfy the invariant vacuously, as an invariant states the consequences of having some pattern in the source model, but not the consequences of its absence. Nonetheless, the input models generated in this way are still interesting because their transformation have to yield valid target models satisfying the rest of invariants and postconditions in the specification as well as the target meta-model integrity constraints.

Finally, this coverage criterion is also indicated for specifications that use a closed world assumption (i.e. any property not included in the specification is false) by generating models which potentially may not belong to the input language according to the specification. Currently, PaMoMo does not support a closed world semantics.

***t*-way coverage.** Most faults in software systems are due to the interactions of several factors or properties. Based on this observation, *t*-wise testing [22] consists of the generation of test cases for all possible combinations of t properties in the system under test. *Pairwise* testing is a particular case of this kind of testing for $t = 2$ (i.e. the generation of test cases for pairs of properties) which yields smaller test suites than exhaustive generation yet being able to find many errors. In our case, we are interested in detecting errors coming from an incorrect implementation of the combination of several requirements in a specification. These errors are frequent when each requirement is implemented as a rule or relation that interacts with other rules in the transformation, e.g. through explicit invocation.

In this case, given a specification with $I = \{I_1, ..., I_n\}$ invariants, we generate an expression of the form $ocl(I_j)$ *and ... and* $ocl\ (I_k)$ for each *t*-tuple of invariants in the specification, demanding the existence of an occurrence of the source part of each invariant in the tuple. In the limit, *1*-way testing is

Table 1. Expressions generated from a specification with 3 invariants. The terms i_1, i_2 and i_3 represent the OCL code generated from the invariants in the specification.

property	closed property	*2*-way	closed *2*-way	combinatorial	closed combinatorial	exhaustive (for i_1, i_2)
i_1	i_1	i_1 and i_2	i_1 and i_2	i_1	i_1	-
i_2	i_2	i_1 and i_3	i_1 and i_3	i_2	i_2	i_1
i_3	i_3	i_2 and i_3	i_2 and i_3	i_3	i_3	i_2
	not i_1		not i_1	i_1 and i_2	i_1 and i_2	not i_1
	not i_2		not i_2	i_1 and i_3	i_1 and i_3	not i_2
	not i_3		not i_3	i_2 and i_3	i_2 and i_3	i_1 and i_2
				i_1 and i_2 and i_3	i_1 and i_2 and i_3	i_1 and not i_2
					not i_1	not i_1 and i_2
					not i_2	not i_1 and not i_2
					not i_3	

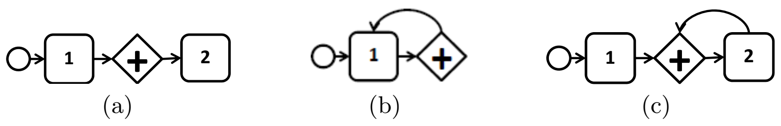


Fig. 5. Generated models for the OCL expressions: (a) *ocl(Task1) and ocl(ParallelGateway3)*; (b) *ocl(ParallelGateway1) and not ocl(ParallelGateway3)*; (c) *ocl(ParallelGateway1) and not ocl(ParallelGateway3)*, with failure due to disabling condition.

equivalent to property coverage. Table 1 shows the expressions generated for pairwise (i.e. *2*-way) testing.

As an example, Fig. 5(a) shows a model generated for pairwise testing, considering the properties *Task1* and *ParallelGateway3*. The model contains two tasks, the first one is input to the gateway, and the second one not (as required by the disabling condition of the second invariant). The solver introduces a start event which does not appear in any of the invariants, as it is required by precondition *OneStartEvent*. Moreover, the tasks in the two invariants are not required to be different in the generated model, hence we obtain a model with two tasks instead of three.

In the MDE community, pairwise testing is being successfully used for software product line testing [19, 20], considering pairs of features in a feature model. In our case there are additional challenges, because our specifications do not explicitly encode dependencies between their requirements, and the model generation procedure has to consider the constraints given by the input meta-model and preconditions.

Closed *t*-way coverage. As discussed previously, sometimes it is desirable to test also that the input models that do not contain occurrences of the source of invariants are handled correctly. Hence, in this criteria we generate the same models as in *t*-way coverage, as well as models generated from expressions of the form *not* $ocl(I_i)$, as Table 1 shows for *t=2*.

Combinatorial coverage. It generates all models for *1*-way, *2*-way, ... *t*-way coverage, where t is the number of invariants in the specification. Thus, here we consider all combinations of properties, including all of them simultaneously (*t*-way case). A total of $2^n - 1$ models are generated (see Table 1).

Closed combinatorial coverage. It generates the same models as in combinatorial coverage, and a model from each negated invariant (see Table 1).

Exhaustive coverage. This is the most exhaustive level of coverage, generating models for all combinations of the occurrence or absence of the source of the invariants in a specification, or their obliteration. For this purpose it generates different OCL expressions where the existence of the source of each invariant can be either mandatory ($ocl(I_i)$), forbidden (*not* $ocl(I_j)$) or ignored (i.e. the invariant is not taken into account). This yields a number of 3^n potential models. The last column of Table 1 shows the OCL expressions for a specification with two invariants.

As an example, Fig. 5(b) shows a model generated for the OCL expression $ocl(ParallelGateway1)$ *and not* $ocl(ParallelGateway3)$. In particular,

invariant *ParallelGateway3* is not satisfied in the model as there is no occurrence of its main constraint (i.e. there are not two different tasks).
For a more exhaustive coverage we can enforce the absence of a property in the generated models in several ways. Up to now, this was achieved by negating the source of the invariant ($not\ ocl(I_j)$). However, there are different ways in which we can "disable" the testing of a particular invariant: either because there is no occurrence of the source of its main constraint, or because there are occurrences of the main constraint but these do not satisfy some disabling condition. Thus, we generate an OCL expression for each way to disable the property (the source of the main constraint of the invariant is not found, or it is found but it does not fulfil some disabling condition). Fig. 5(c) shows a model used to test invariant $ParallelGateway1$ and the absence of $ParallelGateway3$, the latter due to the occurrence of its disabling condition (both tasks are input to the gateway).
Altogether, this coverage uses a brute-force approach to the generation of test models. Unfortunately, many of the generated OCL expressions are unsatisfiable because they contain contradicting requirements. For instance, the expression $ocl(ParallelGateway2)\ and\ not\ ocl(ParallelGateway1)$ has no solution because it looks for an input model with two tasks connected to a gateway (first invariant), and simultaneously forbids having tasks connected to gateways (negation of the second invariant). The problem is that the negated invariant is included in the required one. Given that the generation of models using constraint solving is time-consuming, it is advisable to discard unsatisfiable expressions prior to model generation. Thus, if the source of one invariant is included in another one, no expression requiring the invariant with bigger source and negating the other should be considered.
Finally, it is for us an open question whether such a deep degree of exhaustivity is worth for certain kinds of specifications, or whether it is more effective to use less exhaustive types of coverage as the previous ones, enriched with heuristics that allow for the generation of bigger sets of test input models (for instance, generating several models from the same OCL expression, or demanding more than one occurrence of the invariants).

Regardless the chosen level of coverage, there are some configurable aspects (or heuristics) in the model generation process, which may affect the size and number of generated models. For example, when looking for models aimed at testing several invariants with non-empty intersection, different levels of overlapping between them can be considered, ranging from non-overlapping (the source of the invariants is taken to be disjoint) to a maximal overlap. Second, for specifications with a high number of requirements or for exhaustive testing, we can minimise the size of the generated test set by skipping the generation of a model for a particular combination of properties if this combination is already present in a model previously generated.

Finally, as the reader may have noticed, the solver may yield the same model for the resolution of two different OCL expressions. For instance, if the input meta-model for our running example requires exclusive gateways to have at least two output tasks, then the solver will always try to complete the source model in invariant *ExclusiveGateway1* with a new task connected to the gateway, i.e., it will try to find a model like the one in invariant *ExclusiveGateway2*. Thus, the expressions $ocl(ExclusiveGateway1)$ and $ocl(ExclusiveGateway2)$ are likely to produce the same input model. At this point, we can simply remove one of the generated input models from the test set and continue processing the next OCL expression, as the model enables the testing of both invariants.

5.3 Linking input models and oracles

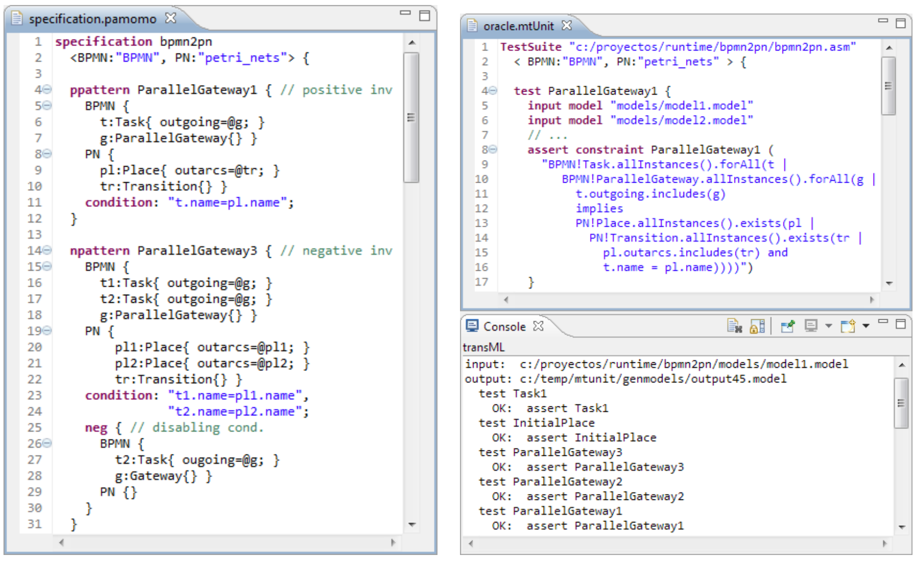
As a final step, we automatically generate an *mtUnit* script – another language in our *trans*ML family of languages [12] – to automate the testing of the transformation using the generated models. The script includes a test case for each invariant and postcondition in the specification, defining the input models to be used in the test case, and the oracle function checking the particular invariant or postcondition. By default, all generated models are added to all test cases. However, if a model was generated from an expression that negated an invariant, then it is not added to the test case for that invariant, as we already know that such a model satisfies the invariant vacuously. This allows for a more efficient testing process, as a given oracle function will not be checked in any model for which we already know that the oracle function will always hold.

Altogether, the generation of input models, oracle functions and scripts is done automatically from the same specification. This has the advantage that the transformation tester does not need to build them separately by hand, identify the oracle functions to be used for each input model, and build a script to execute the test. More importantly, being generated from the same specification, both the models and the oracle functions will work together to validate the same properties of interest: the models will enable the testing of these properties, and the oracle functions will check their satisfaction.

The right of Fig. 6 shows an excerpt of the *mtUnit* script generated from our specification example, using the *property* coverage level. Lines 4–17 in the upper window contain the definition of the test case generated from the invariant *ParallelGateway1*. For space constraints, the figure only shows two of the input models for this test case (lines 5–6). Below, the figure partially shows the result of running the test.

6 Tool support

The presented framework is supported by an Eclipse, EMF-based prototype tool which allows building PaMoMo specifications using a textual editor, and automates the generation of input models and *mtUnit* test scripts for them. The left of Fig. 6 shows part of our specification example using the textual editor, in particular the definition of the invariants *ParallelGateway1* (lines 4–12) and *ParallelGateway3* (lines 14–31). The generation of the test suite and input models from this specification is *push-button*. In our case, it yields the *mtUnit* file that is partially shown to the right of the figure, in the upper right window. The first two lines declare the file with the transformation to be tested (either ATL or ETL) and the source and target meta-models. Lines 4–17 correspond to the test case for the *ParallelGateway1* invariant (for brevity we only show two of its input test models in lines 5–6). Executing the test suite will run the transformation for each input model and report whether the result verifies the assertions in the different test cases (see lower right window in Fig. 6).



```
specification bpmn2pn
  <BPMN:"BPMN", PN:"petri_nets"> {

  ppattern ParallelGateway1 { // positive inv
    BPMN {
      t:Task{ outgoing=@g; }
      g:ParallelGateway{} }
    PN {
      pl:Place{ outarcs=@tr; }
      tr:Transition{} }
    condition: "t.name=pl.name";
  }

  npattern ParallelGateway3 { // negative inv
    BPMN {
      t1:Task{ outgoing=@g; }
      t2:Task{ outgoing=@g; }
      g:ParallelGateway{} }
    PN {
        pl1:Place{ outarcs=@pl1; }
        pl2:Place{ outarcs=@pl2; }
        tr:Transition{} }
    condition: "t1.name=pl1.name",
               "t2.name=pl2.name";
    neg { // disabling cond.
      BPMN {
        t2:Task{ ougoing=@g; }
        g:Gateway{} }
      PN {}
    }
  }
```

```
 1 TestSuite "c:/proyectos/runtime/bpmn2pn/bpmn2pn.asm"
 2   < BPMN:"BPMN", PN:"petri_nets" > {
 3
 4   test ParallelGateway1 {
 5     input model "models/model1.model"
 6     input model "models/model2.model"
 7     // ...
 8     assert constraint ParallelGateway1 (
 9       "BPMN!Task.allInstances().forAll(t |
10          BPMN!ParallelGateway.allInstances().forAll(g |
11            t.outgoing.includes(g)
12            implies
13            PN!Place.allInstances().exists(pl |
14              PN!Transition.allInstances().exists(tr |
15                pl.outarcs.includes(tr) and
16                t.name = pl.name))))")
17     }
```

```
input:  c:/proyectos/runtime/bpmn2pn/models/model1.model
output: c:/temp/mtunit/genmodels/output45.model
  test Task1
    OK:  assert Task1
  test InitialPlace
    OK:  assert InitialPlace
  test ParallelGateway3
    OK:  assert ParallelGateway3
  test ParallelGateway2
    OK:  assert ParallelGateway2
  test ParallelGateway1
    OK:  assert ParallelGateway1
```

Fig. 6. Tool support for PAMOMO specifications (left) and testing (right).

In the back-end, we are using the UMLtoCSP constraint solver [5] for model finding. UMLtoCSP receives an Ecore meta-model and a file with OCL invariants, and generates a ".dot" file with a model that satisfies the meta-model integrity constraints and the OCL invariants. Then, we parse this file into an EMF-conformant representation for its use in *mtUnit*. Currently, we do not provide support for model generation heuristics like different overlapping degrees or detection of redundant models.

7 Discussion and lines of future work

As discussed in Section 2, most black-box testing approaches use meta-model coverage criteria to ensure that the generated input models will include, altogether, instances of all classes and associations in the meta-model, and extreme values for the attributes. However, it is difficult to ensure that the generated models will include certain structures enabling the testing of relevant transformation properties, whereas unimportant class instances or model fragments may appear repeatedly in the generated models.

In contrast, the presented specification-driven approach aims at testing the intention of the transformation, and ensures that the generated models will allow testing transformation properties of interest. In this sense, the quality of the generated test set highly relies on how complete a specification is. If a specification only covers part of the transformation requirements, then the generated models may not enable the testing of the underspecified parts. For instance, our running example does not include invariants over the *EndEvent* BPMN class, and therefore the generated test models may not include instances of this type, leaving its transformation untested. Thus, we foresee complementing our techniques with additional coverage criteria, also meta-model based.

Finally, the models we generate with our technique tend to be small. This has the advantage that the test models remain intentional: they are generated for testing a particular combination of transformation invariants, which will be checked by the oracle function more efficiently.

We are currently conducting some experiments of our approach with promising results. For the specification in this paper, we have implemented an ATL transformation of 120 lines of code, and performed pairwise testing. To test the effectiveness of the generated test set, we manually created 40 mutants of this transformation by injecting faults that followed the systematic classification in [17] (i.e. navigation, filtering and creation mutations), and then used the test set on the transformation mutants. The test discovered the faults in 28 out of the 40 mutants, which gives a mutation score (or effectiveness) of 70% (28/40).

Starting from the results in this and subsequent experiments, in the future we plan to investigate the effectiveness of our generated input models to detect transformation failures. This is called *vigilance*, which is the degree in which contracts can detect faults in the running system. A relevant question is the level of detail required in contracts to find a significant number of failures and obtain high vigilance. Another interesting issue is whether the size of the generated input test models has an influence on the effectiveness of the test set. In order to obtain "bigger" test models, we are considering (a) the possibility of including extra constraints, stating that e.g., models should have a certain number of instances of each class, and (b) extending the coverage criteria to allow several instances of the same invariant. Regarding tool support, the most critical factor is the constraint solver, which is time-costly, and therefore we are currently working towards a domain-specific constraint solver for models.

Acknowledgements: Work funded by the Spanish Ministry of Economy and Competitivity (TIN2011-24139) and by the R&D programme of Madrid Region (S2009/TIC-1650).

References

1. A. Balogh et al. Workflow-driven tool integration using model transformations. volume 5765 of *LNCS*, pages 224–248. Springer, 2010.

2. B. Baudry, S. Ghosh, F. Fleurey, R. B. France, Y. L. Traon, and J.-M. Mottu. Barriers to systematic model transformation testing. *CACM*, 53(6):139–143, 2010.
3. C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *ISSTA'02*, pages 123–133, 2002.
4. BPMN. `http://www.bpmn.org/`.
5. J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *ASE'07*, pages 547–548.
6. E. Cariou, R. Marvie, L. Seinturier, and L. Duchien. OCL for the specification of model transformation contracts. In *ECEASST*, volume 12, pages 69–83, 2004.
7. F. Fleurey, B. Baudry, P.-A. Muller, and Y. Traon. Qualifying input test data for model transformations. *Software and Systems Modeling*, 8:185–203, 2009.
8. A. García-Domínguez, D. Kolovos, L. Rose, R. Paige, and I. Medina-Bulo. Eunit: A unit testing framework for model management tasks. In *MoDELS'11*, volume 6981 of *LNCS*, pages 395–409. Springer, 2011.
9. P. Giner and V. Pelechano. Test-driven development of model transformations. In *MoDELS'09*, volume 5795 of *LNCS*, pages 748–752. Springer, 2009.
10. M. Gogolla and A. Vallecillo. *Tract*able model transformation testing. In *ECMFA'11*, volume 6698 of *LNCS*, pages 221–235. Springer, 2011.
11. E. Guerra, J. de Lara, D. Kolovos, and R. Paige. A visual specification language for model-to-model transformations. In *VL/HCC 2010*, pages 119–126.
12. E. Guerra, J. de Lara, D. Kolovos, R. Paige, and O. dos Santos. Engineering model transformations with transML. *Software and Systems Modeling*, in press, 2012.
13. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008.
14. D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Transformation Language. In *ICMT'08*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008.
15. J. M. Küster and M. Abd-El-Razik. Validation of model transformations - first experiences using a white box approach. In *MoDELS Workshops*, volume 4364 of *LNCS*, pages 193–204. Springer, 2007.
16. Y. Lin, J. Zhang, and J. Gray. A framework for testing model transformations. In *Model-driven Soft. Devel. - Research and Practice in Sof. Eng.* Springer, 2005.
17. J. Mottu, B. Baudry, and Y. Traon. Mutation analysis testing for model transformations. In *ECMDA-FA 2006*, volume 4066 of *LNCS*, pages 376–390. Springer.
18. J. Mottu, B. Baudry, and Y. Traon. Reusable MDA components: A testing-for-trust approach. In *MoDELS'06*, volume 4199 of *LNCS*, pages 589–603. Springer, 2006.
19. S. Oster, I. Zorcic, F. Markert, and M. Lochau. MoSo-PoLiTe: tool support for pairwise and model-based software product line testing. In *VaMoS'11*, ACM International Conference Proceedings Series, pages 79–82. ACM, 2011.
20. G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Traon. Pairwise testing for software product lines: a comparison of two approaches. *Soft. Qual. J.*, in press, 2011.
21. S. Sen, B. Baudry, and J. Mottu. Automatic model generation strategies for model transformation testing. In *ICMT'09*, volume 5563 of *LNCS*, pages 148–164. Springer, 2009.
22. G. B. Sherwood, S. S. Martirosyan, and C. Colbourn. Covering arrays of higher strength from permutation vectors. *J. of Combinat. Designs*, 14(3):202–213, 2005.
23. N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, 23(4):38–47, 2006.