Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Specification-Driven Model Transformation Testing

Esther Guerra[1] *, **Mathias Soeken**[2]

[1] Universidad Autónoma de Madrid (Spain), e-mail: `Esther.Guerra@uam.es`
[2] University of Bremen (Germany), e-mail: `msoeken@informatik.uni-bremen.de`

Received: date / Revised version: date

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 coverage of the source meta-model or the transformation implementation code, whereas oracle functions are frequently defined using query or graph languages. However, these two tasks are usually performed independently regardless of their common purpose, and sometimes there is a gap between the properties exhibited by the generated input models and those considered by the transformations.

Recently, we proposed a formal specification language for the declarative formulation of transformation properties (by means of invariants, pre- and postconditions) from which we generated partial oracle functions used for transformation testing. Here, we extend the usage of our specification language for the automated generation of input test models by SAT solving. The testing process becomes more *intentional* because the generated models ensure a certain coverage of the transformation requirements. Moreover, we use the same specification to consistently derive both the input test models and the oracle functions. A set of experiments is presented, aimed at measuring the efficacy of our technique.

Key words Model Transformation – Model Transformation Specification – Model Transformation Testing – Model Finding – Test Oracle

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 [24]. 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 *trans*ML in the past: a family of modelling languages for the *engineering* of transformations using an MDE approach [24]. The *trans*ML languages provide support for the gathering of requirements, their formal specification, the architectural, high-level and low-level design of transformations, 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 to detect errors in the transformation results.

*trans*ML includes a language with formal semantics called PaMoMo (Pattern-based Model-to-Model Specification Language) [25] 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 such as Z [46] or Alloy [29], PaMoMo's formal semantics enables reasoning at the level of requirements, while being independent of the particular transformation language used for the implementation.

In [25], 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 produce a reasonable set of input test models and build a test script to exercise the transformation with them, using the generated oracle function. Hence, if a specification states

Send offprint requests to:
* *Present address:* Computer Science Department, Universidad Autónoma de Madrid, 28049 Madrid (Spain)

how to transform certain structures (e.g. an inheritance cycle, an unconnected object of a given type, or a particular attribute value), it is important that some input test models include such structures, to assert whether their transformation has been correctly implemented. Unfortunately, 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, as hand-crafted models may focus on certain requirements while leaving others untested.

In this paper, we tackle these problems by deriving not only the oracle function but also a set of input test models from the transformation specification. As a consequence, we ensure the coverage of the properties in the specification. The input models are calculated using SAT solving techniques on OCL expressions generated from the specification [45], and it is possible to choose between seven levels of coverage to obtain different degrees of exhaustiveness when testing. Additionally, 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 the generated 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 [14,42], 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 for different coverage criteria is also novel.

This paper extends [22] by including a more thorough related work section, a formal presentation of our specification language, an integral exposition of the whole framework, and more importantly, we discuss the results of a set of experiments to measure the effectiveness of the different levels of coverage defined for specifications. Input test model generation in these experiments has been carried out using the ocl2smt model finder [45], which has been recently integrated in our transformation testing tool. This has been done for performance reasons, as we noticed that the SMT formalization was advantageous compared to the CSP formalization for the considered model transformations. Instead, in [22], we used the UMLtoCSP [8] solver for model generation, as it is also integrated in our testing tool.

The remainder of the paper is organised as follows. Section 2 reviews related works, focusing on existing approaches to model transformation testing. Afterwards, Section 3 sketches our proposal and introduces a running example that we will use throughout the paper. Section 4 presents 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. Next, in Section 7, we use this tool to study the effectiveness of different coverage criteria, using as a testbed a set of ATL transformations. Finally, we draw some conclusions and lines of future work in Section 8. The paper includes an appendix with a formal definition of the core concepts in PaMoMo, originally introduced in [25,26].

2 State of the art

Validation and verification is an integral task of software development. In the context of model transformations, the works targeting their validation and verification can be classified in three categories: (i) those using a formal language to implement the transformations, so that it is possible to ensure or analyse transformation properties such as termination or determinism [11,12,16,32]; (ii) those translating the transformations into formal domains for analysis, such as Petri nets [23], rewriting logic [3,49] or SAT problems [7]; (iii) and those focusing on testing of transformations. The first two approaches allow for the analysis of general properties such as termination, determinism, rule independence, rule applicability or reachability of system states. In this paper we follow the third approach; hence, in the following, we review related works on model transformation testing, paying special attention to black-box testing approaches as this is the scope of the work that we will present in this paper.

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.

Input models. Most works dealing with the generation of input test models are for black-box testing and only consider the features of the input meta-model but not properties of the transformation. For instance, in [14,42], the authors perform automatic generation of input test models based on the input meta-model and some typical coverage criteria (e.g. partitioning of attribute values and number of classes). Using this approach, in [42], the authors present an experiment where different input test sets are generated for different strategies, obtaining mutation scores ranging from 72% to 87%. In [20], the generation of input test models must be hand-coded using an imperative language with features for randomly choosing attribute values and association ends. Input models are also hand-crafted in [43], although they are only required to conform to a relaxed version of the input meta-model without mandatory references and general constraints. These so-called *partial models* are transformed into Alloy and fed into a constraint solver to find valid instances of the original meta-model which can be used for testing. A similar approach is presented in [14], where the term *model fragment* is used instead of partial model, and some coverage criteria of the input meta-model are also considered.

Closer to our philosophy, contracts in [13] are used both to generate test input models and as oracle functions. Models, meta-models, and specifications must be defined using constructive logic, and the meta-model's OCL constraints must be translated to this logic as well, thereby limiting the applicability of the proposal. As in the previous works, the authors only foresee coverage of the input meta-model.

Finally, there are also a few white-box testing approaches to validate transformations. For example, in [33], the authors propose using all possible overlapping models of each pair of rules in a transformation as input models for testing. More recently, [21] exploits the structure of ATL transformations to generate input models: they extract the dependency graph of the transformations (similar to a control flow graph), and each traversal of the graph is transformed into a set of constraints used to compute an input model by means of constraint solving.

Coverage criteria. Existing black-box testing approaches for model transformations either do not consider coverage criteria [20,43], or support input meta-model coverage (partitioning of attribute values, number of classes and associations, etc.) [13,14, 42]. A drawback in these works is that some transformation properties of interest may remain untested as they are not taken into account when building (manually or automatically) test models.

On their turn, white-box testing approaches usually adopt a mix of classical white-box coverage criteria [21] and others specific for transformations, such as rule coverage or decision coverage [41]. For instance, in [41], the authors measure the decision coverage of input test sets, however, there is no discussion of the correlation between this coverage criterion and the efficacy of a test set.

Oracle function. Regarding the third challenge in model transformation testing, 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 [35] 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 [9,20,37]. The approaches in [15,17] follow a similar philosophy to the xUnit framework, and the oracle functions can be specified as OCL/EOL assertions. There are other approaches that permit the specification of partial oracle functions as graph patterns or model fragments [1]. Finally, in previous works, we presented our visual language PaMoMo to specify contracts for transformations and provided compilations of this language into OCL [25] and QVT-Relations [26], thus enabling the use of PaMoMo contracts as oracles. 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 [15, 17], 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 [14,42], but do not consider transformation properties or different levels of exhaustiveness for testing. In this paper, we 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. As a distinguishing feature, the generated models enable the testing of relevant properties of the transformation, as given by its specification. Moreover, we define a set of specification coverage criteria which enable testing with increasing levels of exhaustiveness.

It is worth noting that, for general software testing, we can find several works where the generation of test cases is performed from specifications, which has been reported to allow a more efficient generation of test cases based on some notion of coverage of properties or error domains [28,38]. Moreover, 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 [4], the authors generate both artefacts 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. Similarly, in [18,47], the authors propose parameterized unit tests (PUTs) to cover the input state space. For this purpose, they are using symbolic execution techniques and constraint solving in order to obtain a high coverage, and to generate new PUTs. In our case, we aim at generating test models ensuring coverage of the transformation requirements; 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. Moreover, automated test case generation such as described in this paper can be compared to techniques outside of software testing, e.g. the testing of executable language definitions. As an example, a grammar-based testing method has been presented in [27,34].

3 A framework for specification-driven testing

Fig. 1 shows the working scheme of our approach. In a first step, the designer specifies the requirements (i.e.

the preconditions, 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 [30], ETL [31], etc.). Indeed, in our experience, we have found that the specification and the implementation are frequently refined iteratively.

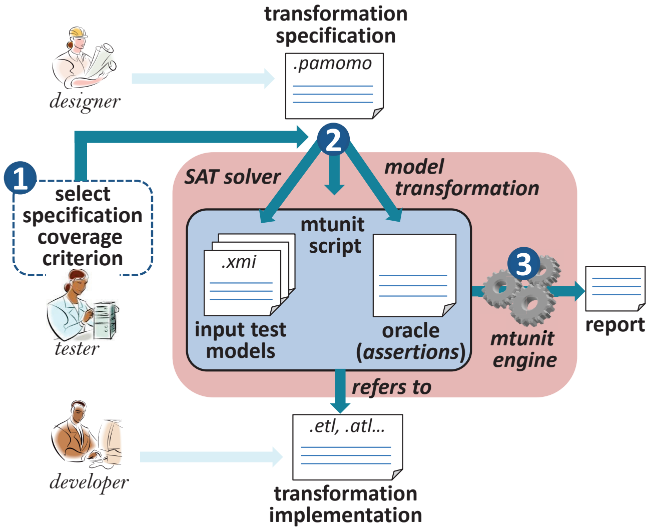


Fig. 1 Framework for specification-driven testing.

Starting from the specification, the transformation tester can automatically generate a complete test suite that 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 [25], (ii) a set of input test models enabling the testing of all requirements in the specification according to certain coverage criteria selected by the tester, 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 the *mtUnit* engine [24]. Roughly, in order to generate the input test models we translate the preconditions and invariants in the specification into OCL invariants, which we combine and feed into a SAT solver to find models that contain certain combination of properties. The way in which the different properties are combined to find models depends on the selected coverage type.

The details of this framework will be presented in Section 5. Before, we introduce our specification language PaMoMo in the next section, and use it to specify a transformation from the Business Process Modeling Notation (BPMN) [5] into Petri nets. The goal is to analyse BPMN models to detect deadlocks, incorrect termination conditions, or tasks that can never be completed. The upper part 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 shown below the BPMN model.

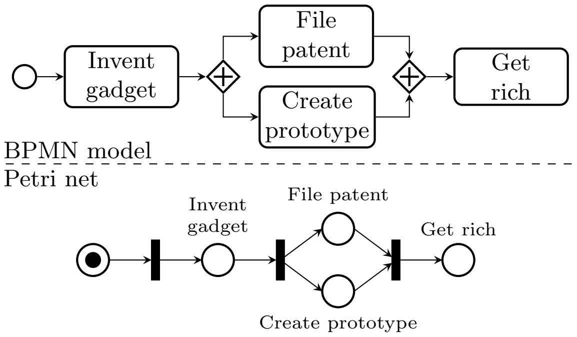


Fig. 2 BPMN model and equivalent Petri net.

Fig. 3 shows an excerpt of the OMG standard BPMN 2.0 meta-model [5] with the main classes and references that we will consider in our example. As the meta-model shows, a BPMN model is made of a set of flow nodes of different types (activities, events and gateways) which can be interconnected through sequence flows, thus defining a process.

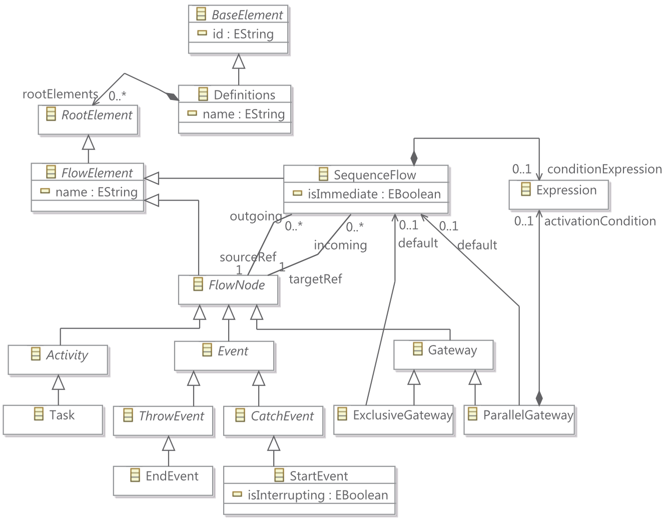


Fig. 3 Excerpt of the BPMN meta-model [5].

4 A specification language for transformations

PaMoMo is a formal, pattern-based, declarative, bidirectional specification language to describe correctness requirements of the transformations and of their input and output models in an implementation-independent way [25]. 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*[1] 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.

Next, we illustrate these concepts in an intuitive manner through our running example. The interested reader can find their formal semantics in the Appendix.

Fig. 4 shows some transformation preconditions for our running example, expressing requirements that any input model should fulfil beyond its meta-model constraints[2]. The name of each precondition is shown in parenthesis and preceded by P or N to denote whether the precondition is Positive or Negative. For instance, our transformation expects models with one start event from which only one sequence flow goes out. This is specified 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.

Preconditions, as well as postconditions and invariants, can include an enabling condition to specify a local satisfaction context. In such a case, the precondition is not evaluated globally in the model, but in the context of each occurrence of its enabling condition. As an example, Fig. 4 shows a precondition, *PathsForGateway*, with an attached enabling condition, *AnyGateway*. The precondition demands that each gateway (enabling condition) defines at least one input and one output flows (precondition). The precondition contains the abstract class *Gateway*, becoming applicable to all concrete gateway types inheriting from it (see BPMN meta-model in Fig. 3).

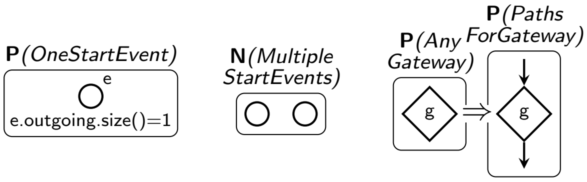


Fig. 4 Some preconditions of the transformation.

Postconditions are similar to preconditions, but they express requirements of the output models. Fig. 5 shows some postconditions for the generated Petri nets, such as the absence of unconnected places (*UnconnectedPlaces*), the existence of input and output places for all transitions (*ConnectedTransition*), and the existence of a single place with one token and without input transitions (*InitialMarking* and *InitialPlace*).

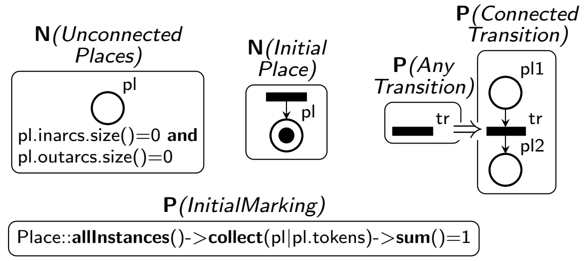


Fig. 5 Some postconditions of the transformation.

Finally, invariants express how certain structures in the input models should be transformed. They are made of a source graph, a target graph, and a formula relating both. Roughly, a positive invariant holds on a pair of source and target models if for each occurrence of its source graph, there is an occurrence of the target graph. If the invariant is negative, then we should not find an occurrence of the target graph.

For instance, Fig. 6 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* includes a disabling condition, named *UnconnectedTask*, which restricts the scope of the invariant to the occurrences of the source part for which the disabling condition is not met. Altogether, this invariant states that if a parallel gateway does *not* have a task

[1] Please note that PaMoMo invariants, which express properties of any pair of input and output models, are sometimes called postconditions [9] or domain/range contracts [37] in the model transformation testing literature. In PaMoMo, a postcondition is just a requirement of the output model alone. Moreover, PaMoMo invariants do not describe conditions to be maintained on the state of the transformation execution, but they are statements that the result of any possible execution of the transformation should satisfy.

[2] 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.

t2 as input (disabling condition), then the place for *t2* cannot be connected to the transition (as the invariant is negative). Preconditions and postconditions can also define any number of disabling conditions. Our specification contains similar invariants for the tasks going out from parallel gateways. The two remaining invariants in Fig. 6 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 such as 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 [26]; (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 [25]. However, the challenge of generating input test models satisfying the meta-model constraints and all preconditions in the specification remains, as tests models 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. For instance, the input test set of the running example should include some model containing an exclusive gateway with several output tasks, in order to check the correct implementation of the invariant *ExclusiveGateway*, and similar for the rest of invariants 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 models satisfying concrete combinations of properties (according to the selected coverage) and all integrity constraints of the input meta-model, 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 there are available solvers that find models satisfying a set of OCL constraints [8,45] 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 the purpose of generating such input models, 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 = o_j if the maximum cardinality of the association is 1, and o_i.link->includes(o_j) otherwise), inequalities for the objects with the 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 because it is subsumed by the invariants. 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. . .*).

```
1  o1.type::allInstances()−>exists(o1 | ...
2   oi.type::allInstances()−>exists(oi |
3    conditions(o1,...,oi)
4    < and not
5       oj.type::allInstances()−>exists(oj | ...
6        ok.type::allInstances()−>exists(ok |
7         conditions(o1,...,oi,oj,...,ok) >∗ ) ...)
```

Listing 1 OCL template for invariants.

As an example, from invariant *ParallelGateway3* in Fig. 6 we generate the expression shown in Listing 2. Lines 1–7 correspond to the encoding of the source pattern of the invariant, whereas lines 8–9 encode its disabling condition.

```
1  Task::allInstances()−>exists(t1 |
2    Task::allInstances()−>exists(t2 |
3      ParallelGateway::allInstances()−>exists(g |
4        SequenceFlow::allInstances()−>exists(s1 |
```

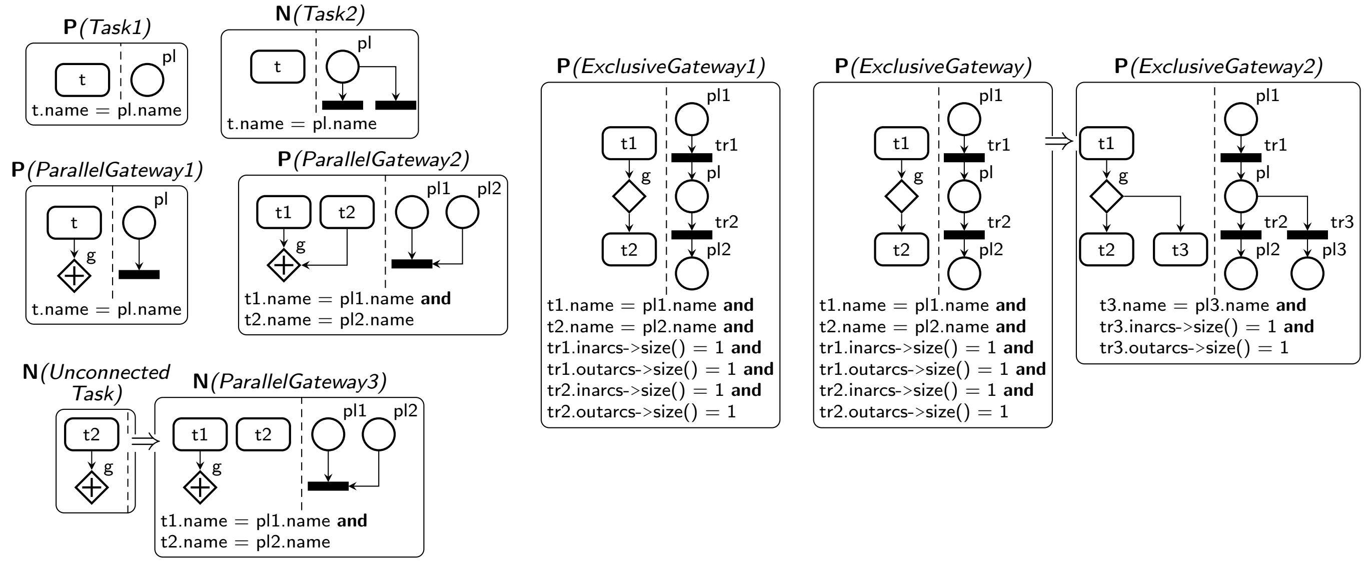


Fig. 6 Some invariants for the transformation.

```
              s1.sourceRef = t1 and
              and s1.targetRef = g
              and t1 <> t2
              and not SequenceFlow::allInstances()−>exists(s2 |
                s2.sourceRef = t2 and s2.targetRef = g)))))
```

Listing 2 OCL expression for invariant *ParallelGateway3*.

Frequently, specifications include invariants with same source and different target. For instance, *Task1* and *Task2* in Fig. 6 have both a task as source, as the former specifies how to translate a task correctly, whereas 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 3. 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 9–12). 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 (line 1).

```
1   < not >?
2   < o1.type::allInstances()−>forAll(o1 | ...
3         oi.type::allInstances()−>forAll(oi |
4           conditions(o1,...,oi)
5           implies >?
6           oj.type::allInstances()−>exists(oj | ...
7             ok.type::allInstances()−>exists(ok |
8               conditions(o1,...,oi,oj,...,ok)
9                 < and not
10                  ol.type::allInstances()−>exists(ol | ...
11                    om.type::allInstances()−>exists(om |
12                      conditions(o1,...,oi,oj,...,ok,ol,...,om) >∗ ...)
```

Listing 3 OCL template for preconditions.

Listing 4 shows the OCL expressions generated from the preconditions depicted in Fig. 4: *OneStartEvent* (line 1), *MultipleStartEvents* (lines 3–5) and *PathsForGateway* (lines 7–14).

```
1   StartEvent::allInstances()−>exists(e | e.outgoing−>size() = 1)
2
3   not StartEvent::allInstances()−>exists(e1 |
4     StartEvent::allInstances()−>exists(e2 |
5       e1 <> e2))
6
7   Gateway::allInstances()−>forAll(g |
8     true
9     implies
10    SequenceFlow::allInstances()−>exists(s1 |
11      SequenceFlow::allInstances()−>exists(s2 |
12        s1.targetRef = g
13        and s2.sourceRef = g
14        and s1 <> s2)))
```

Listing 4 OCL expressions for preconditions in Fig. 4.

From the invariants and postconditions in the specification, we generate a set of OCL assertions that will act as oracle function in the generated test suite. The scheme to generate these assertions will be presented in Section 5.4. Before, the next section presents a set of coverage criteria for specifications.

Table 1 Expressions generated from a specification with 3 invariants $I = \{I_1, I_2, I_3\}$. The terms i_1, i_2 and i_3 represent the OCL code generated from the corresponding invariant in the specification ($i_1 = \text{ocl}(I_1)$, $i_2 = \text{ocl}(I_2)$, $i_3 = \text{ocl}(I_3)$).

property	closed property	*2*-way	closed *2*-way	combinatorial	closed combinatorial	exhaustive (for i_1, i_2)
i_1	i_1	$i_1 \wedge i_2$	$i_1 \wedge i_2$	i_1	i_1	*true*
i_2	i_2	$i_1 \wedge i_3$	$i_1 \wedge i_3$	i_2	i_2	i_1
i_3	i_3	$i_2 \wedge i_3$	$i_2 \wedge i_3$	i_3	i_3	i_2
	$\neg i_1$		$\neg i_1$	$i_1 \wedge i_2$	$i_1 \wedge i_2$	$\neg i_1$
	$\neg i_2$		$\neg i_2$	$i_1 \wedge i_3$	$i_1 \wedge i_3$	$\neg i_2$
	$\neg i_3$		$\neg i_3$	$i_2 \wedge i_3$	$i_2 \wedge i_3$	$i_1 \wedge i_2$
				$i_1 \wedge i_2 \wedge i_3$	$i_1 \wedge i_2 \wedge i_3$	$i_1 \wedge \neg i_2$
					$\neg i_1$	$\neg i_1 \wedge i_2$
					$\neg i_2$	$\neg i_1 \wedge \neg i_2$
					$\neg i_3$	

5.2 Coverage criteria for input model generation

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. 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, combinatorial 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 each level in detail.

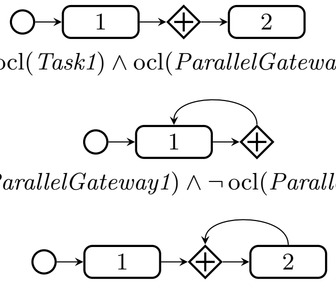
Property coverage. This is the least exhaustive level of coverage, appropriate when the invariants in the specification are independent. It generates as many input models as invariants in the specification, each one including at least one occurrence of the source of an invariant. The rationale is to 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 $I = \{I_1, \ldots, I_n\}$ invariants (with different source), we generate n expressions of the form $i_j = \text{ocl}(I_j)$. Each expression demands the existence of an occurrence of the source of invariant I_j. As an example, Table 1 shows in the first column the expressions generated from a specification with three invariants, where each i_j term represents the OCL code generated from the invariant I_j as explained in Section 5.1.

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 under-specifications, frequently lead to errors in the final implementations, yielding malformed output models. Thus, given a specification with $I = \{I_1, \ldots, I_n\}$ invariants, we also generate n additional expressions of the form $\neg\,\text{ocl}(I_j)$. The second column of Table 1 shows the generated expressions assuming a specification with 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 has 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 [44] 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



(a) ocl(*Task1*) ∧ ocl(*ParallelGateway3*)

(b) ocl(*ParallelGateway1*) ∧ ¬ ocl(*ParallelGateway3*)

(c) ocl(*ParallelGateway1*) ∧ ¬ ocl(*ParallelGateway3*), with failure due to disabling condition

Fig. 7 Generated models for OCL expressions.

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 transformation rule or relation that interacts with other rules in the transformation, e.g. through explicit invocation.

In this case, given a specification with $I = \{I_1, \ldots, I_n\}$ invariants, we generate an expression of the form $\text{ocl}(I_{j_1}) \wedge \cdots \wedge \text{ocl}(I_{j_t})$ for all $\binom{n}{t}$ *t*-tuples 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 equivalent to property coverage. Table 1 shows the expressions generated for pairwise (i.e. *2*-way) testing.

As an example, Fig. 7(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 [39, 40], 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 as well.

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 criterion we generate the same models as in *t*-way coverage, as well as models generated from expressions of the form $\neg\,\text{ocl}(I_j)$, as Table 1 shows for *t=2*.

Combinatorial coverage. It generates all models for *1*-way, *2*-way, ..., *n*-way coverage, where n is the number of invariants in the specification. Thus, here we consider all combinations of properties, including all of them simultaneously (*n*-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 ($\text{ocl}(I_j)$), forbidden ($\neg\,\text{ocl}(I_j)$), or ignored (i.e. the invariant is not taken into account). This yields a number of 3^n potential models, also including the expression *true* in which all invariants are ignored. The last column of Table 1 shows the OCL expressions for a specification with two invariants.

As an example, Fig. 7(b) shows a model generated for the OCL expression

$$\text{ocl}(\textit{ParallelGateway1}) \wedge \neg\,\text{ocl}(\textit{ParallelGateway3}).$$

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 ($\neg\,\text{ocl}(I_j)$). 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 can choose to generate a different OCL expression for each way to disable the property (i.e. 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. 7(c) shows a model used to test invariant *ParallelGateway1* and the absence of *ParallelGateway3*, the latter due to the occurrence of its disabling condition (as both tasks are input to the gateway).

Altogether, this coverage uses a brute-force approach to the generation of test models. Notice that some of the generated OCL expressions may be unsatisfiable if they contain contradicting requirements. For instance, the expression

$$\text{ocl}(\textit{ParallelGateway2}) \wedge \neg\,\text{ocl}(\textit{ParallelGateway1})$$

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.
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). Section 7 presents the results of some initial experiments in this line, although focussed on less exhaustive coverage kinds.

5.3 Customization

Some expressions generated from the above mentioned coverage levels might be unsatisfiable and if many combinations are unsatisfiable, e.g. when using the exhaustive strategy, a lot of unnecessary run-time is spent by the constraint solver. To avoid getting too many unsatisfiable problem instances, we propose to generalise the expressions. For this purpose, one general expression is formalized for each level of coverage which encodes how many invariants shall be considered and if some of them may be negated, but not specifies them precisely. If the constraint solver finds a satisfying assignment, then from this assignment not only the model but also the considered configuration of invariants can be deduced from it. In iterative runs, already found solutions are explicitly blocked until either all combinations have been considered or the constraint solver cannot find a satisfying assignment any longer. Then, the search can be stopped as it is ensured that no other configuration can be satisfiable. For this purpose, we make use of *select variables* s_j and *polarity variables* p_j for each OCL constraint $i_j = \mathrm{ocl}(I_j)$ as well as cardinality constraints. A flexible polarity can be added to an expression by replacing each OCL constraint i_j with the term $(p_j \leftrightarrow i_j)$, i.e. the invariant is inverted if, and only if p_j is assigned 0. In a similar way, select variables are introduced by substituting each i_j with the term $(s_j \rightarrow i_j)$, i.e. the invariant i_j is disabled if, and only if s_j is assigned 0 and i_j must hold only if s_j is assigned 1. Both concepts can be combined to $(s_j \rightarrow (p_j \leftrightarrow i_j))$ embracing both a flexible polarity and the possibility to disable the invariant.

Table 2 shows the generalised expressions for most of the presented levels of coverage. It turns out that the generalised *closed t-way coverage* and *closed combinatorial coverage* are too complex and thus do not justify the additional overhead.

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.

Table 2 Generalised expressions for the different levels of coverage with $\nu s = \sum_{j=1}^{n} s_j$.

Coverage level	Generalised expression
property	$\bigwedge_{j=1}^{n}(s_j \rightarrow i_j) \wedge \nu s = 1$
closed property	$\bigwedge_{j=1}^{n}(s_j \rightarrow (p_j \leftrightarrow i_j)) \wedge \nu s = 1$
t-way	$\bigwedge_{j=1}^{n}(s_j \rightarrow i_j) \wedge \nu s = t$
combinatorial	$\bigwedge_{j=1}^{n}(s_j \rightarrow i_j) \wedge \nu s > 0$
exhaustive	$\bigwedge_{j=1}^{n}(s_j \rightarrow (p_j \leftrightarrow i_j))$

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.

5.4 Linking input models and oracles

As a final step, we automatically derive a test suite to automate the testing of the transformation using the generated models. The test suite includes a test case for each invariant and postcondition in the specification, defining the input models to be used in the test case, and an assertion checking the particular invariant or postcondition. In this way, the test suite will execute the transformation for each generated model, checking in each case whether the output model fulfils the assertions in the test cases. By default, all assertions are checked after transforming each test model. However, if a model was generated from an expression that negated an invariant, then the assertion derived from this invariant is not checked for that particular model, as we already know that the model satisfies the invariant vacuously. This allows for a more efficient testing process, as a given assertion will not be checked in any model for which we already know that the assertion will always hold.

The generation of OCL assertions from the invariants and postconditions in the specification follows the schema shown in Listing 5. Given an invariant, the generated assertion iterates on the objects of its enabling condition and the source of its main constraint (lines 1–3). For all possible bindings of these objects into objects of the checked source and target models, the assertion retains those that do not violate the disabling conditions (lines 4–7). Finally, it iterates on the objects of the target of the invariant main constraint, demanding their existence for each valid binding of the source (lines 9–11) [25].

```
1   o1.type::allInstances()−>forAll(o1 | ...
2     oi.type::allInstances()−>forAll(oi |
3       conditions(o1,...,oi)
4       < and not
5          ol.type::allInstances()−>exists(ol | ...
6            om.type::allInstances()−>exists(om |
7              conditions(o1,...,oi,ol,...,om) >∗ ...)
8       implies
9       oj.type::allInstances()−>exists(oj | ...
10        ok.type::allInstances()−>exists(ok |
11          conditions(o1,...,oi,oj,...,ok)
```

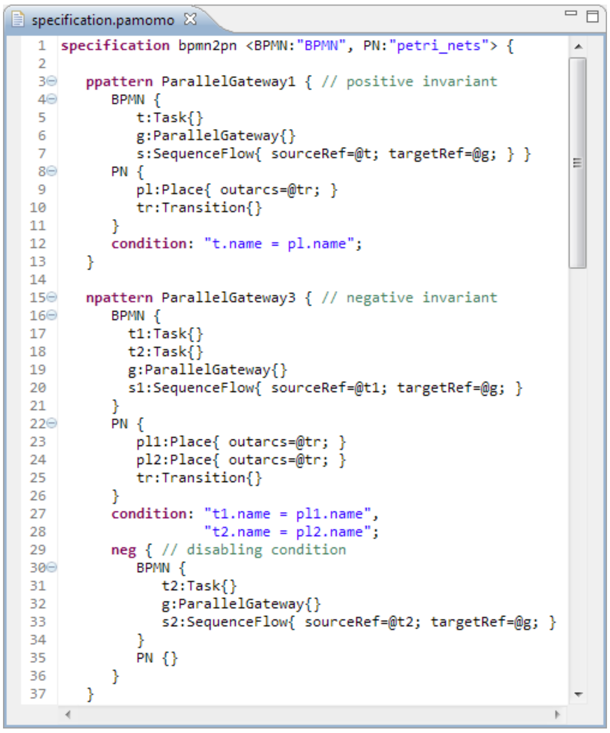
Listing 5 OCL template for oracle function.

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.

Fig. 9 shows an excerpt of the test suite generated from our specification example, using the property coverage level. The test suite is defined using *mtUnit*, which is another language in our *trans*ML family of languages [24]. Lines 4–21 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. Fig. 8 shows part of our specification example using the textual editor, in particular the definition of the invariants *ParallelGateway1* (lines 3–13) and *ParallelGateway3* (lines 15–37).

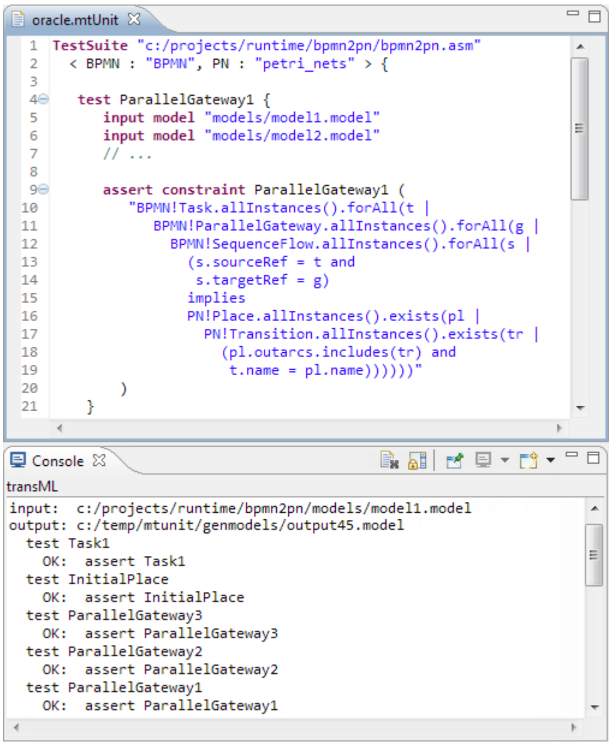


```
1  specification bpmn2pn <BPMN:"BPMN", PN:"petri_nets"> {
2
3     ppattern ParallelGateway1 { // positive invariant
4        BPMN {
5           t:Task{}
6           g:ParallelGateway{}
7           s:SequenceFlow{ sourceRef=@t; targetRef=@g; } }
8        PN {
9           pl:Place{ outarcs=@tr; }
10          tr:Transition{}
11       }
12       condition: "t.name = pl.name";
13    }
14
15    npattern ParallelGateway3 { // negative invariant
16       BPMN {
17         t1:Task{}
18         t2:Task{}
19         g:ParallelGateway{}
20         s1:SequenceFlow{ sourceRef=@t1; targetRef=@g; }
21       }
22       PN {
23          pl1:Place{ outarcs=@tr; }
24          pl2:Place{ outarcs=@tr; }
25          tr:Transition{}
26       }
27       condition: "t1.name = pl1.name",
28                  "t2.name = pl2.name";
29       neg { // disabling condition
30          BPMN {
31             t2:Task{}
32             g:ParallelGateway{}
33             s2:SequenceFlow{ sourceRef=@t2; targetRef=@g; }
34          }
35          PN {}
36       }
37    }
```

Fig. 8 Tool support for PaMoMo specifications.

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 in the upper window of Fig. 9. 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 4–21 correspond to the test case for the *ParallelGateway1* invariant. Two of its input test models are listed in lines 5–6, whereas the OCL assertion generated from the invariant is shown in lines 9–20. 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 window in the figure).

In the back-end, we are using the ocl2smt model finder [45] which is based on SMT (Satisfiability Modulo Theories) and in turn uses the Z3 solver [10]. Since the model finder is used as a black box, also other SAT solving tools like UMLtoCSP [8] can be integrated into the flow. In such a case, it may be necessary to transform the input and output formats to the model finders. As an example, since ocl2smt is written on top of USE [19], the EMF meta-model has to be transformed into a USE model and the resulting USE system state given by ocl2smt needs to be transformed into an EMF-conformant representation for its use in *mtUnit*. Currently, we do not provide support for model generation heuristics such as different overlapping degrees or detection of redundant models. Also the generalised expressions as given in Table 2 are not yet implemented as they require adjustments to the internals of the model finder.



```
TestSuite "c:/projects/runtime/bpmn2pn/bpmn2pn.asm"
  < BPMN : "BPMN", PN : "petri_nets" > {

   test ParallelGateway1 {
      input model "models/model1.model"
      input model "models/model2.model"
      // ...

      assert constraint ParallelGateway1 (
         "BPMN!Task.allInstances().forAll(t |
            BPMN!ParallelGateway.allInstances().forAll(g |
              BPMN!SequenceFlow.allInstances().forAll(s |
                (s.sourceRef = t and
                 s.targetRef = g)
                implies
                PN!Place.allInstances().exists(pl |
                  PN!Transition.allInstances().exists(tr |
                    (pl.outarcs.includes(tr) and
                     t.name = pl.name))))))"
         )
    }
```

```
input:  c:/projects/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. 9 Tool support for model transformation testing.

7 Experiments

In this section, we report on some experiments aimed at measuring the effectiveness of our generated input models to detect transformation failures. This is called *vigilance* [48], which is the degree in which contracts can detect faults in the running system. Our goal is identifying which coverage criteria allow for the detection of a higher number of errors with less effort (i.e. with a smaller test data set). In these experiments, we have focused on the following coverage kinds: property, closed property, 2-way, and closed 2-way. These are the simplest coverage criteria, and consequently, the most inexpensive in generation time, and those that yield the smallest test data sets. Thus, it is interesting to know whether even with the simplest criteria, the generated test suites are able to detect a significant number of errors. The sources of these experiments are available at `http://www.miso.es/tools/transML.html`.

7.1 Testbed setup

The testbed for our experiments is a set of ATL model transformations, which includes a transformation of 120 lines of code that we implemented for the running example, as well as some existing transformations from the ATL zoo[3]: one transforming class schema models into relational database models[4] of 107 lines of code, and another one from BibTeX into the XML-based format for document composition DocBook[5] of 261 lines of code. The PAMOMO specification for the running example was created *before* its implementation, whereas in the rest of cases the specification was created *after* the implementation, from the documentation provided in the zoo. This documentation contained a quite detailed description of the transformation rules, in natural language, which we encoded as PAMOMO patterns. We did not add to the specification anything that was not in the documentation, so in this sense, the completeness of the specifications for the transformations in the zoo depended on the available documentation.

Table 3 gathers the number of preconditions, postconditions and invariants in the resulting specifications, as well as the size of the input meta-models for each case. The BPMN meta-model is the most complex in terms of the number of associations between classes, while the class schema meta-model is the simplest of the three, and the BibTeX meta-model has few associations but makes heavy use of inheritance.

Table 3 Size of specifications and input meta-models in the testbed: (a) *Class-to-Relational*, (b) *BPMN-to-Petri nets*, (c) *BibTeX-to-DocBook*.

	Specification			Input meta-model		
	#pre.	#inv.	#pos.	#classes	#assoc.	#inh.
(a)	3	10	1	6	4	5
(b)	12	10	5	18	9	15
(c)	3	18	4	21	1	31

7.2 Test suite generation

From the PAMOMO specifications we derived a test suite for each coverage type under study (property, closed property, 2-way, and closed 2-way). The test suites were automatically generated by using the tool presented in Section 6. Fig. 10 shows the test suite generation time in each case, measured on an Intel Core i7-2600 CPU 3.40 GHz with 12 Gb RAM. The numbers depicted close to the graphic lines correspond to the number of generated models out of the number of sought models (as for some expressions, the solver did not find a solution model in the given scope). In general, a bigger number of invariants implies the generation of more models, and consequently, higher generation times. Nonetheless, the size of the input meta-model and the number of preconditions in the specification have a bigger impact on the model finding time. This is why the time for the *BibTeX-to-DocBook* specification is much lower than the time for *BPMN-to-Petri nets*, even if the former specification defines more invariants and more models are generated, as

[3] `http://www.eclipse.org/atl/atlTransformations/`

[4] `http://www.eclipse.org/atl/atlTransformations/#Class2Relational`

[5] `http://www.eclipse.org/atl/atlTransformations/#BibTeXML2DocBook`

the size of the input meta-model and the number of preconditions to solve in each model search is smaller in the first case.

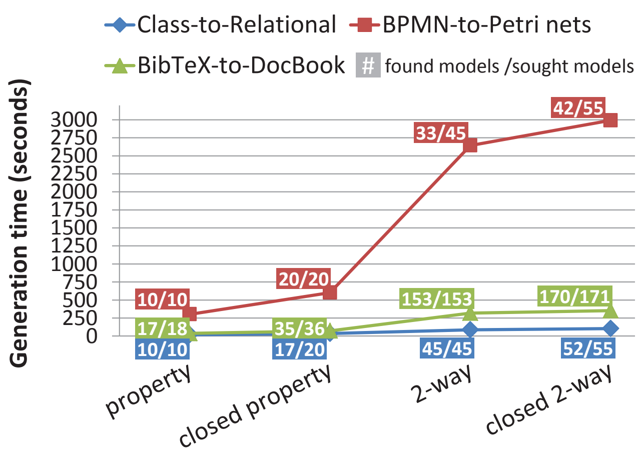


Fig. 10 Test suite generation time.

Let's take a closer look to the generation time per model. The graphic in Fig. 11 shows the median of the time it took our tool to find each test model using the property coverage. We use the median as metric to lessen the effect of outliers. This median was 1.9 seconds for the *Class-to-Relational* specification, 5.9 seconds for *BPMN-to-Petri nets*, and 2 seconds for *BibTeX-to-DocBook*. As previously stated, the reason for this difference in the model generation time is primarily the different size of the input meta-models and the number of preconditions in the specifications. For *BPMN-to-Petri nets*, the maximum generation time was 232.5 seconds in one occasion, 6 models were generated in less than 6 seconds, and 3 models were generated in 14 seconds.

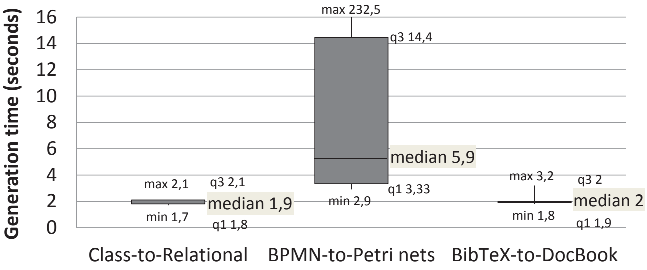


Fig. 11 Model generation time for property coverage.

The graphic in Fig. 12 shows the same metric but for the 2-way level of coverage (i.e. the model finder must consider pairs of invariants). The difference in the generation time with respect to the property coverage is negligible for the *Class-to-Relational* and *BibTeX-to-DocBook* specifications. In contrast, for our running example, the median is more than the double for the 2-way coverage (14.9 seconds compared to 5.9 seconds), and the increment in the value of the first quartile[6] (from 3.33 seconds to 6.3 seconds) indicates that the model generation time was higher in more occasions (i.e. there were more models with higher generation time). We do not show the generation time for the closed variants of the property and 2-way coverage criteria because the results are similar.

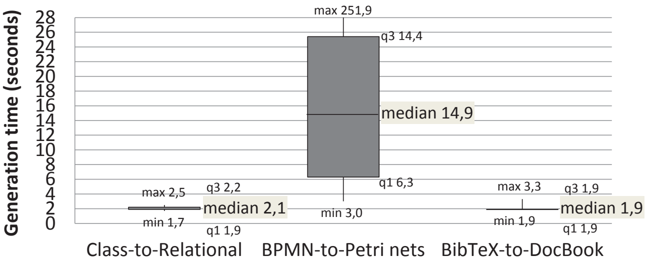


Fig. 12 Model generation time for 2-way coverage.

It is worth mentioning that, even for the least exhaustive coverage, the generated test suites were able to discover non-deliberate errors in the implementation of the *BPMN-to-Petri nets* and *BibTeX-to-DocBook* transformations. In the latter case, the error was in the value of a literal ("Authors List"), which was different in the documentation and the implementation. For the *Class-to-Relational* transformation, the test suites reported some errors as well. In this case, the problem was that the transformation assumed certain preconditions for the input models which were not documented, and therefore were neither included in the specifications nor considered for model generation. For instance, the fact that the transformation only admitted input models with a "String" datatype was not in the documentation, but only mentioned in the code. Thus, we had to refine our specification to include all necessary preconditions and thus generate input models accepted by the transformation.

7.3 Effectiveness of generated test suites

A technique to measure the effectiveness of a test suite and help to improve it is *mutation testing* [6]. In mutation testing, faults are injected in a program to produce erroneous versions of it, which are called *mutants*. Then, each mutant is tested with the test suite. If the test suite detects the error, then the mutant is killed, otherwise the mutant remains alive. The *mutation score*, which is the number of killed mutants divided by the total number of mutants, gives a measure of the quality of the test suite.

We have used this technique to measure the effectiveness of our generated test suites. Thus, we first manually created mutants of the transformations in the testbed

[6] The first quartile $q1$ is the median of the lower half of the data.

by injecting faults that followed the systematic classification of transformation mutations presented in [36]. These mutations are classified in three types: navigation, filtering, and creation mutations. Navigation mutations replace the navigation towards a class with the navigation towards another, remove the last step of a chain of navigations, or add a last step of navigation in a navigation chain. Filtering mutations introduce disturbances in the filters of a collection, either by modifying the attributes used in the filter, or by selecting only some instance types when the collection is defined on a generic class. Finally, creation mutations replace the creation of an object with another with compatible type, delete the creation of a relation between two objects, or add a useless relation between two objects. While navigation and filtering mutations apply to both the input and output of transformations, creation mutations concern only the output.

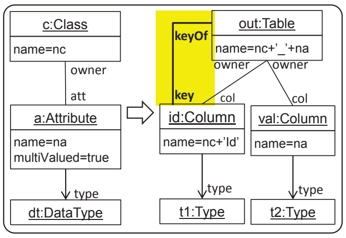
Table 4 shows the mutation operators used to create the mutants in the experiment, which altogether belong to all possible mutation types (navigation, filtering and creation). Each mutant was created by applying one mutation operator once to the original transformation. Thus, each cell in the table corresponds to the number of mutants created using a particular mutation operator. The last column in the table summarizes the number of mutants created for each transformation.

Table 4 Mutation operators [36] used for the creation of transformation mutants: rscc (relation to the same class change), cfca (collection filtering change with addition), cfcd (collection filtering change with deletion), cfcp (collection filtering change with perturbation), caca (classes' association creation addition), cacd (classes' association creation deletion), cccr (class compatible creation replacement). The three rows correspond to (a) *Class-to-Relational*, (b) *BPMN-to-Petri nets*, (c) *BibTeX-to-DocBook*.

	navigation	filtering			creation			number of
	rscc	cfca	cfcd	cfcp	caca	cacd	cccr	mutants
(a)	6	1	1	1	4	12	-	25
(b)	12	4	2	18	2	2	-	40
(c)	-	-	2	6	-	-	4	12

Table 5 shows the number of mutants created from each transformation, as well as the mutation score of the generated test suites. Surprisingly, the mutation score is the same for all levels of coverage under study, and it is not better when using the closed variants. This means that the effectiveness is the same (i.e. we detect the same number of mutants or injected errors) regardless the size of the input test model set, which is much bigger for the 2-way coverage and its closed variant in all cases (see Fig. 10).

Interestingly, for the *Class-to-Relational*, all non-detected errors were of the *Classes' association creation addition* (CACA) mutation type [36]. This mutation type adds a useless relation between two class instances of the output model. Fig. 13 shows an example of one of such mutations, using the ATL syntax to the right and an abstract syntax representation to the left. The injected error is highlighted in a coloured square. In particular, the error consists in the creation of an unnecessary relation of type *key* between the table and one of the columns created by the transformation rule. Actually, the generated test suites did not discover any CACA mutation in the transformation mutants.



```
rule MultiValuedDataTypeAttribute2Column {
  from
    a : UML!Attribute (
        a.type.oclIsKindOf(UML!DataType)
        and a.multiValued
    )
  to
    out : Rel!Table (
      name <- a.owner.name + '_' + a.name,
      col <- Sequence {id, val},
      -- ***** INJECTED ERROR *****
      key <- Sequence{id}
      -- ************************
    ),
    id : Rel!Column (
      name <- a.owner.name + 'Id',
      type <- thisModule.objectIdType
    ),
    val : Rel!Column (
      name <- a.name,
      type <- a.type
    )
}
```

Fig. 13 Mutation example of type *Classes' Association Creation Addition* (CACA). A useless relation *key* is created between the target objects *out* and *id*.

The reason why the generated test suites do not detect this kind of errors is in our specification, as it only considers positive information (i.e. it says which is the correct translation of elements) but it does not specify elements that should not appear. For instance, our specification includes an invariant stating that the correct translation of multivalued attributes with a primitive datatype is a table with two columns (i.e. the case shown in Fig. 13). However, it does not explicitly forbid the created table to have a key. Hence, even if our tool generates a model which allows testing the invariant (i.e. a model that includes multivalued attributes with a primitive datatype), the generated oracle does not check that no useless elements are created, and therefore, it does not notify the error. If we add to our specification negative invariants forbidding useless elements, and then regenerate the oracle function, we obtain a mutation score of 100% for all levels of coverage.

In general, it is frequent to forget specifying negative information (which elements should not appear), also because it is a tedious and error-prone task, as there may be many ways in which useless elements can be added to the target models. Thus, we foresee adding a closed-world semantics to our specification language which automatically enhances the generated oracle with assertions aimed at detecting useless elements. In our context, an element is useless if it is not demanded by any positive invariant or postcondition in the specification. Another possibility is making explicit this semantics in the specification by generating extra negative invariants that handle these cases. The idea is to let the tester se-

Table 5 Mutation score of the generated test suites.

	number of mutants	property	closed property	*2*-way	closed *2*-way
Class-to-Relational	25	21/25 (84%)	21/25 (84%)	21/25 (84%)	21/25 (84%)
BPMN-to-Petri nets	40	27/40 (67.5%)	27/40 (67.5%)	27/40 (67.5%)	27/40 (67.5%)
BibTeX-to-DocBook	12	10/12 (83.3%)	10/12 (83.3%)	10/12 (83.3%)	10/12 (83.3%)

lect whether he wants to add this semantics or not to his specification, as well as how to realise it.

For the other two transformations in our testbed, the undetected errors were of any possible kind, and sometimes, were located in parts not covered by the specification. In most of the cases, it is enough to add new assertions to the generated oracle to detect the errors, being not necessary to increase the number of input test models.

7.4 Effectiveness vs completeness of specifications

A second relevant question in our context is the level of detail required in contracts to find a significant number of errors and obtain high vigilance. In the previous section, we observed that the test suites generated for different levels of exhaustiveness had the same mutation score. Next, we discuss the results of a set of experiments we have performed to study how the degree of completeness in specifications may affect the quality of the generated tests.

Starting from the specifications in our testbed, we have built new specifications where we have only retained a fixed percentage (25%, 50% and 75%) of their invariants, in order to obtain specifications with varying degrees of completeness. Then, we have generated test suites (including both the input models and the oracle function) from these specifications, and we have measured the effectiveness of the test suites by analysing the mutation score. The results are summarized in Fig. 14.

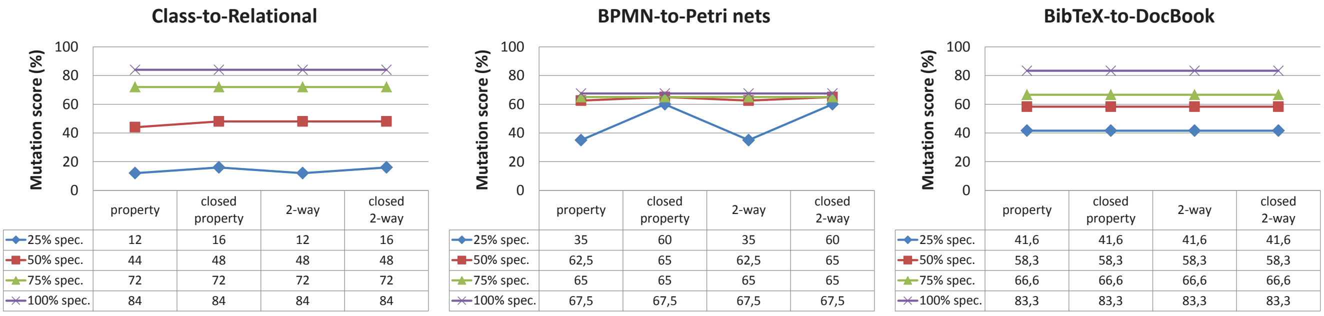
Unsurprisingly, the test suites generated from less complete specifications are less effective. Moreover, whereas in the three cases we obtain the same mutation score if we start from the complete specification, when we move to less complete specifications there are differences in the quality of the generated test suites depending on the selected coverage. For instance, the mutation score for the 50% of the *Class-to-Relational* is 44% if we use the property coverage, and higher (48%) if we use 2-way testing. We can also see that, in some cases, the closed variants help in detecting more errors than the non-closed counterparts. For instance, the obtained mutation score in case of using the property and 2-way coverage on the 25% of the *BPMN-to-Petri nets* is 35%, and this score reaches 60% if we use any of their closed variants.

Altogether, the less complete a specification is, the more important may be moving to more exhaustive coverage criteria in order to obtain test suites with higher vigilance.

7.5 Comparison with random model generation

To conclude, we report on a last experiment that compares our proposal with random model generation.

Random model generation was implemented as follows: first, we generated a random model conforming to the input meta-model (i.e. all generated models were always valid instances of the meta-model), and then, we checked whether the model satisfied the preconditions in the specification, as only these models are valid inputs to the transformation. If the model satisfied the preconditions, it was included in the input test set, otherwise, it was discarded and a new random model was generated. Following this process, we generated test sets of different size, containing as many models as the studied coverage criteria may generate (see Fig. 10). For instance, we generated input test sets of size 10, 20, 45 and 55 random models for the *Bpmn-to-Petri nets* transformation,



Class-to-Relational

	property	closed property	2-way	closed 2-way
25% spec.	12	16	12	16
50% spec.	44	48	48	48
75% spec.	72	72	72	72
100% spec.	84	84	84	84

BPMN-to-Petri nets

	property	closed property	2-way	closed 2-way
25% spec.	35	60	35	60
50% spec.	62,5	65	62,5	65
75% spec.	65	65	65	65
100% spec.	67,5	67,5	67,5	67,5

BibTeX-to-DocBook

	property	closed property	2-way	closed 2-way
25% spec.	41,6	41,6	41,6	41,6
50% spec.	58,3	58,3	58,3	58,3
75% spec.	66,6	66,6	66,6	66,6
100% spec.	83,3	83,3	83,3	83,3

Fig. 14 Mutation score of the test suites generated from specifications with different degrees of completeness. In particular, we consider subsets of size 25%, 50%, 75% and 100% of the original specifications.

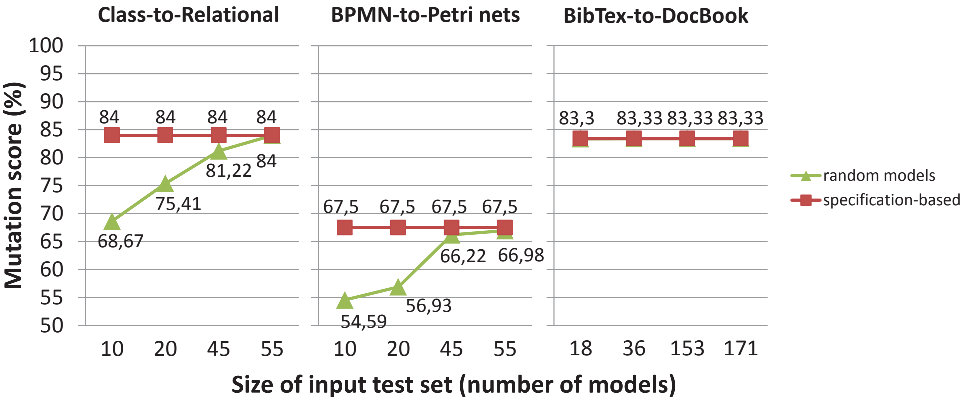


Fig. 15 Mutation scores using random model generation vs specification model generation.

as this is the number of models that one would potentially generate from its specification using the property, closed-property, 2-way and closed 2-way coverage criteria, respectively.

Interestingly, about 1 out of 5 generated Class models satisfied the preconditions and therefore was valid for testing purposes, whereas this ratio for BibTeXML was 1 out 28. However, we were not able to generate any random BPMN model satisfying all preconditions in a reasonable amount of time (i.e. less than 1 hour). In this case, models had to be generated pseudo-randomly, that is, enforcing some of the preconditions a priori, to speed up the generation process. Using this optimisation, we were able to produce 1 valid BPMN model satisfying all preconditions out of 448 generated models. Altogether, fully random model generation do not seem appropriate when the models to be generated need to fulfil many constraints, or these are complex.

To measure the quality of the generated test sets, we calculated the mutation score using the same mutants and oracle functions as in the previous experiments, but with the new test sets. The results are displayed in Fig. 15, as well as their comparison with the mutation scores obtained by our approach using test sets with the same size. Since random generation can yield different models each time, the graphics show the average mutation score for 10 different randomly generated test sets in each case. We obtained the same mutation score for both random and specification-driven approaches in only one case: the *BibTeX-to-DocBook* transformation. In the other two cases, the graphics show that in random model generation, having bigger test sets usually lead to better mutation scores (i.e. test sets with better quality). This does not seem the case in our approach, as we obtained the same mutation score for all coverage criteria analysed. The mutation scores for the randomly generated test sets were never bigger that those obtained using our approach, and indeed, in most cases they were smaller. This is especially the case for the smallest test sets, which in our approach correspond to those generated using the property coverage criterion. For instance, for the *Class-to-Relational* transformation, the average mutation score for 10 test sets of 10 random models was 68.67%, while this score was 84% using our property coverage criterion. This indicates that the test sets generated with our approach are successful in testing the properties in a specification, and are more intentional than those generated randomly, providing some evidence of our starting hypothesis.

8 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. Moreover, 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 have conducted some experiments in which we automatically generated test suites from several transformation specifications according to different specification-based coverage criteria, and then measured the efficacy of the generated tests. Our findings can be summarized as follows:

- The models generated with our techniques are useful, even if we use the least exhaustive coverage, as they were able to detect non-deliberate errors in our transformation implementations. Moreover, their generation is effortless and ensures the models will conform to their meta-model and fulfil extra conditions (i.e. preconditions) when required by a transformation. To this respect, the automatic model generation also helped us to identify preconditions for the input models, which were assumed by the transformations but not documented (such in the case of the *Class-to-Relational* transformation).

- 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 *End-Event* 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.
- For rather complete specifications, the most simple coverage criterion (i.e. property) yields test suites as good as other coverage criteria that generate more input models, being these latter more expensive to produce and execute. However, as we move towards less complete specifications, we sometimes obtain more effective tests when using more exhaustive criteria than using simpler ones, as well as when using the closed variant of coverages (i.e. when we include models that purposely do not contain occurrences of some invariant in the specification).
- In our experiments, an error which was recurrently undetected by the generated oracles was the creation of extra classes or relations in the output models, apart from the expected ones. This is due to the fact that, when defining the requirements of transformations, we tend to focus on the elements that the transformation should create (e.g. from each class we should create a table), but forget to specify what should not occur (e.g. the generated table should not have a key). This negative information is tedious to specify by hand, and it is difficult to ensure that any non-allowed configuration has been specified. Thus, we foresee adding a closed-world semantics operation for our specifications which regards any output model not belonging to the language of the specification to be invalid. Two realisations of this operation are under consideration: extending the specification with extra negative information (e.g. negative invariants), or generating oracle functions that take this semantics into account.
  Notice that detecting this kind of errors is not a matter of generating more input test models, but it requires extending the oracle function to make it more precise.

In the future, we plan to perform further experiments with larger case studies to evaluate whether, in such cases, there is more variability in the mutation score for the test sets generated using different coverage criteria. Starting from the results in these experiments, we plan to integrate additional techniques for input model generation, both meta-model based and white-box based. In particular, given a set of input models generated from a specification, we plan to extend them to create new models that enable the testing of a transformation implementation according to some white-box coverage criterion (e.g. rule coverage, OCL expression coverage [21], etc.). The implementation of a mechanism for the detection and elimination of redundant models in the generated test sets is also future work. More in detail, we plan to compare the source model of invariants to eliminate duplicated ones and obtain a smaller input test set that still enables the testing of all invariants in the specification, as well as using model differencing after model generation to detect whether two models are the same. Finally, we plan to develop tool support for the automatic generation of transformation mutants, which is time-consuming, as well as heuristics to produce new input models from live mutants.

Acknowledgements. We thank the referees for their useful comments. This work has been sponsored by the Spanish Ministry of Science and Innovation with project “Go-Lite” (TIN2011-24139), by the R&D program of the Community of Madrid with project “e-Madrid” (S2009/TIC-1650), and by the German Research Foundation (DFG) within the Reinhart Koselleck project (DR 287/23-1).

References

1. A. Balogh, G. Bergmann, G. Csertán, L. Gönczy, Á. Horváth, I. Majzik, A. Pataricza, B. Polgár, I. Ráth, D. Varró, and G. Varró. Workflow-driven tool integration using model transformations. In *Graph Transformations and Model-Driven Engineering*, 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. A. Boronat, J. A. Carsí, and I. Ramos. Algebraic specification of a model transformation engine. In *FASE'06*, volume 3922 of *LNCS*, pages 262–277. Springer, 2006.
4. C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *ISSTA'02*, pages 123–133, 2002.
5. BPMN. `http://www.bpmn.org/`.
6. T. A. Budd. Mutation analysis: Ideas, examples, problems and prospects. In *Proc. of Summer School on Computer Program Testing*, pages 129–148, 1981.
7. J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.
8. 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, 2007.

9. 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.

10. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

11. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer-Verlag, 2006.

12. H. Ehrig and U. Prange. Formal analysis of model transformations based on triple graph rules with kernels. In *ICGT'08*, volume 5214 of *LNCS*, pages 178–193. Springer, 2008.

13. C. Fiorentini, A. Momigliano, M. Ornaghi, and I. Poernomo. A constructive approach to testing model transformations. In *ICMT'10*, volume 6142 of *LNCS*, pages 77–92. Springer, 2010.

14. 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.

15. 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.

16. A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova. Modelling and analysis using GROOVE. *STTT*, 14(1):15–40, 2012.

17. P. Giner and V. Pelechano. Test-driven development of model transformations. In *MoDELS'09*, volume 5795 of *LNCS*, pages 748–752. Springer, 2009.

18. P. Godefroid, J. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating software testing using program analysis. *IEEE Software*, 25(5):30–37, 2008.

19. M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1–3):27–34, 2007.

20. M. Gogolla and A. Vallecillo. *Tract*able model transformation testing. In *ECMFA'11*, volume 6698 of *LNCS*, pages 221–235. Springer, 2011.

21. C. A. González and J. Cabot. ATL-Test: A white-box test generation approach for ATL transformations. In *MoDELS'12*, volume 7590 of *LNCS*, pages 449–464. Springer, 2012.

22. E. Guerra. Specification-driven test generation for model transformations. In *ICMT'12*, volume 7307 of *LNCS*, pages 40–55. Springer, 2012.

23. E. Guerra and J. de Lara. Colouring: Execution, debug and analysis of QVT-Relations transformations through coloured Petri nets. *Software and Systems Modeling*, in press, 2013.

24. E. Guerra, J. de Lara, D. Kolovos, R. Paige, and O. dos Santos. Engineering model transformations with transML. *Software and Systems Modeling*, 12(3):555–577, 2013.

25. E. Guerra, J. de Lara, D. S. Kolovos, and R. F. Paige. A visual specification language for model-to-model transformations. In *VL/HCC'10*, pages 119–126. IEEE Computer Society, 2010.

26. E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Automated verification of model transformations based on visual contracts. *Automated Software Engineering*, 12(1):5–46, 2013.

27. J. Harm and R. Lämmel. Two-dimensional approximation coverage. *Informatica (Slovenia)*, 24(3), 2000.

28. R. M. Hierons. Testing from a Z specification. *The Journal of Software Testing, Verification and Reliability*, 7(1):19–33, 1997.

29. D. Jackson. *Software Abstractions. Logic, Language, and Analysis.* MIT Press, 2006.

30. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008.

31. 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.

32. J. M. Küster. Definition and validation of model transformations. *Software and Systems Modeling*, 5(3):233–259, 2006.

33. 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.

34. R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In *TestCom'06*, pages 19–38, 2006.

35. 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.

36. J. Mottu, B. Baudry, and Y. Traon. Mutation analysis testing for model transformations. In *ECMDA-FA'06*, volume 4066 of *LNCS*, pages 376–390. Springer, 2006.

37. 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.

38. A. J. Offutt and S. Liu. Generating test data from SOFL specifications. *Journal of Systems and Software*, 49(1):49–62, 1999.

39. S. Oster, I. Zorcic, F. Markert, and M. Lochau. MoSoPoLiTe: tool support for pairwise and model-based software product line testing. In *VaMoS'11*, ACM International Conference Proceedings Series, pages 79–82. ACM, 2011.

40. G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Traon. Pairwise testing for software product lines: Comparison of two approaches. *Soft. Qual. J.*, 20(3-4):605–643, 2012.

41. J. A. M. Quillan and J. F. Power. White-box coverage criteria for model transformations. In *1st International Workshop on Model Transformation with ATL*, 2009.

42. 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.

43. S. Sen, J.-M. Mottu, M. Tisi, and J. Cabot. Using models of partial knowledge to test model transformations. In *ICMT'12*, volume 7307 of *LNCS*, pages 24–39. Springer, 2012.

44. 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.

45. M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying UML/OCL models using boolean satisfiability. In *DATE'10*, pages 1341–1344. IEEE, 2010.
46. J. M. Spivey. An introduction to Z and formal specifications. *Softw. Eng. J.*, 4(1):40–50, 1989.
47. N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, 23(4):38–47, 2006.
48. Y. L. Traon, B. Baudry, and J.-M. Jézéquel. Design by contract to improve software vigilance. *IEEE Trans. Software Eng.*, 32(8):571–586, 2006.
49. J. Troya and A. Vallecillo. A rewriting logic semantics for ATL. *Journal of Object Technology*, 10:5: 1–29, 2011.

Appendix

This appendix presents the formal semantics of PaMoMo. Further details of this formalisation, as well as additional examples, are available in [25,26].

The building blocks in PaMoMo are *constraints* of the form $C = \langle G_s, G_t, \alpha \rangle$, where G_s is a graph that conforms to the source meta-model of the transformation, G_t is a graph that conforms to the target meta-model of the transformation, and α is a Boolean formula over the elements and attributes in both graphs. We say that a constraint C is *empty* if it has empty graphs G_s and G_t, and its formula α is true. We write $\alpha|_s$ (resp. $\alpha|_t$) to refer to the terms of the formula α that contain only objects and variables in G_s (resp. G_t). Given a constraint C, we can build the constraints $C|_s = \langle G_s, \emptyset, \alpha|_s \rangle$ and $C|_t = \langle \emptyset, G_t, \alpha|_t \rangle$.

We define relations between constraints by means of *morphisms*. Given two constraints $C^1 = \langle G_s^1, G_t^1, \alpha^1 \rangle$ and $C^2 = \langle G_s^2, G_t^2, \alpha^2 \rangle$, a morphism $m\colon C^1 \to C^2$ is given by two embeddings $G_s^1 \hookrightarrow G_s^2$ and $G_t^1 \hookrightarrow G_t^2$, such that $\alpha^2 \Rightarrow \alpha^1$, $\alpha^2|_s \Rightarrow \alpha^1|_s$ and $\alpha^2|_t \Rightarrow \alpha^1|_t$.

Finally, we define a *gluing construction* which merges two constraints through a common intersection. In particular, given the constraints $C^1 = \langle G_s^1, G_t^1, \alpha^1 \rangle$ and $C^2 = \langle G_s^2, G_t^2, \alpha^2 \rangle$ to be merged, a constraint $D = \langle D_s, D_t, \alpha^D \rangle$ identifying the common elements in C^1 and C^2, and the morphisms $C^1 \leftarrow D \to C^2$, we define the gluing construction $C^1 +_D C^2 = \langle G_s^1 +_D G_s^2, G_t^1 +_D G_t^2, \alpha^1 \wedge \alpha^2 \rangle$, which yields a constraint built componentwise by gluing the graphs in C^1 and C^2 through the elements identified in D (formally a pushout construction [11]) and taking the conjunction of the formulas in C_1 and C_2 [25]. For simplicity, we use the shortcut $C^1 + C^2$ (instead of $C^1 +_D C^2$) when there is no room for misunderstanding.

As an example, Fig. 16 shows four constraints. The left compartment in each constraint corresponds to the source graph G_s, the right compartment corresponds to the target graph G_t, and the formula α is depicted below (we hide the formula if it is the *true* formula). The morphisms between constraints are depicted as arrows, and the identification of elements across constraints is given by equality of identifiers (e.g. we identify all tasks in the constraints by using the same identifier `t`). The gluing of constraints C^1 and C^2 via D yields constraint $C^1 +_D C^2$, where object `t` appears once as it was identified in D, and the formula is the conjunction of the formulas in C^1 and C^2.

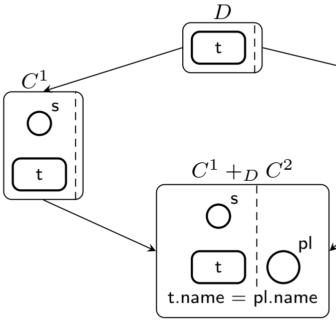


Fig. 16 Gluing of constraints.

A PaMoMo invariant $I\colon C_e \xrightarrow{e} C \xrightarrow{d_i} C_{d_i}$ is made of a *main constraint* C, a constraint C_e (which may be empty) called *enabling condition*, a set of constraints C_{d_i} called *disabling conditions*, and morphisms $e\colon C_e \to C$ and $d_i\colon C \to C_{d_i}$ from the enabling condition to the main constraint, and from this latter to each disabling condition. Moreover, invariants can be positive or negative.

As an example, the upper part of Fig. 17 shows a hypothetical invariant with a non-empty enabling condition C_e and a disabling condition C_{d_1}, just for illustrative purposes. Below, we show the same invariant using a more compact notation that we use in this paper, where the disabling condition only contains the elements that do not appear in the main constraint (and their context), and we omit repeated terms in formulas across constraints. The letter N on the main constraint indicates that the invariant is negative. Altogether, the purpose of this invariant is specifying the following property: if we have a task `t` and a place `pl` with the same name (enabling condition), and `t` is not the first task after the start event (disabling condition), then there cannot be a place with one token connected through a transition to `pl` (because the invariant is negative).

Let $I\colon C_e \xrightarrow{e} C \xrightarrow{d_i} C_{d_i}$ be an invariant, with α, β and γ_i being the formulas defined in C, C_e and C_{d_i}, respectively. If the invariant is positive, then it will hold on a pair of source and target models if: (i) for each occurrence *occ* of the enabling condition C_e plus the source graph of the main constraint C, (ii) if there is no occurrence of any disabling condition C_{d_i} which embeds *occ*, (iii) then there is an occurrence of the target graph of C which embeds *occ*. We formalise this semantics as follows:

$$\begin{aligned}&\forall\, \mathrm{occ}(C|_s + C_e)\\ &\quad \text{s.t. } \beta \wedge \alpha(C|_s + C_e) \text{ holds } \wedge\end{aligned}$$

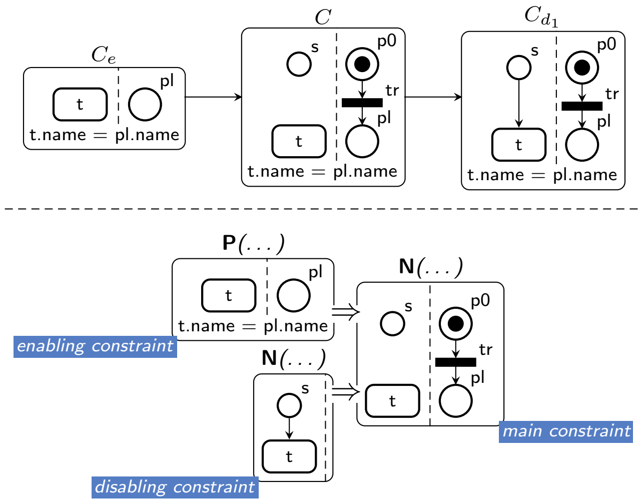


Fig. 17 Invariant in theoretical and compact formats.

$$
\begin{aligned}
&\quad(\forall\ C_{d_i}\ \nexists \mathrm{occ}(C|_s + C_e + C_{d_i}|_s)\\
&\qquad \text{s.t. } \gamma_i(C|_s + C_e + C_{d_i}|_s) \text{ holds}),\\
&\exists\,\mathrm{occ}(C) \text{ s.t. } \alpha \text{ holds}
\end{aligned}
$$

where occ(x) represents an occurrence of the constraint x, and $\alpha(x)$ is the part of the formula α that contains only objects, attributes and variables defined in the constraint x (and similar for $\gamma_i(x)$). If the invariant is negative, its semantics forbids finding and occurrence of the target graph of C, for which purpose we need to change the $\exists$ by $\nexists$ in the last line of the previous formula. For invariants without enabling and disabling conditions we can simplify the formula as follows:

$$
\begin{aligned}
&\forall\,\mathrm{occ}(C|_s) \text{ s.t. } \alpha|_s \text{ holds},\\
&\exists\,\mathrm{occ}(C) \text{ s.t. } \alpha \text{ holds}
\end{aligned}
$$

Fig. 18 illustrates the satisfaction checking procedure for the invariant shown in Fig. 17. First, we look for all occurrences of the constraint that results from gluing $C|_s$ and C_e. There are two occurrences in the models: one made of the start event and the task and place "Invent gadget", and another one made of the start event and the task and place "Get rich". Secondly, we retain only the occurrences that do not violate the disabling condition (i.e. those which are not embedded in a bigger occurrence of the gluing of $C|_s$, C_e and the source part of the disabling condition). Since the start event in the BPMN model is connected to task "Invent gadget", we discard this occurrence; on the contrary, we retain the other occurrence that contains task "Get rich" because this is not connected to the start event. Thirdly, for the retained occurrences, we check whether they cannot be completed to the main constraint (because the invariant is negative). Since the place "Get rich" is not connected to a place with a token, we conclude that the invariant holds in this pair of models.

Pre- and postconditions have the same structure as invariants, but the target (source) graph in preconditions (postconditions) is 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. Formally:

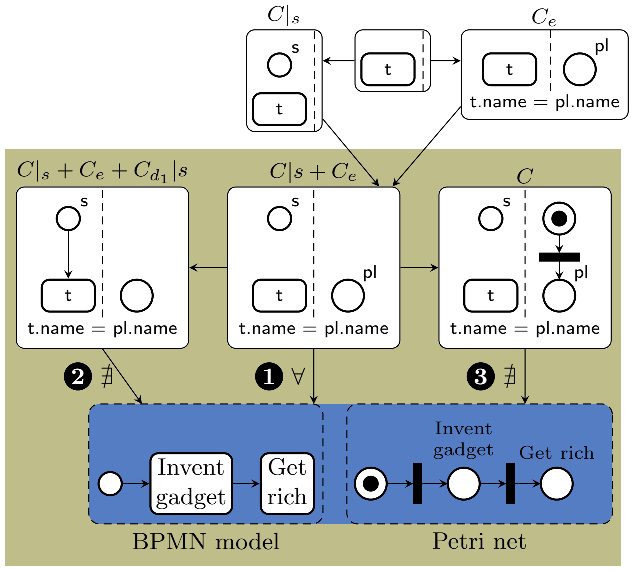


Fig. 18 Checking satisfaction of a negative invariant.

$$
\begin{aligned}
&\forall\,\mathrm{occ}(C_e) \text{ s.t. } \beta \text{ holds},\\
&\exists\,\mathrm{occ}(C) \text{ s.t. } \alpha \text{ holds} \wedge\\
&\quad(\forall\ C_{d_i}\ \nexists\mathrm{occ}(C_{d_i}) \text{ s.t. } \gamma_i \text{ holds})
\end{aligned}
$$

In case of negative pre- and postconditions, we should not find occurrences of the main constraint (i.e. there should be $\nexists$ in the second line of the formula). Moreover, for pre- and postconditions without enabling and disabling conditions, we obtain the simpler formula:

$$\exists\,\mathrm{occ}(C) \text{ s.t. } \alpha \text{ holds}$$