

A constructive approach to testing model transformations

Camillo Fiorentini,¹ Alberto Momigliano,¹
Mario Ornaghi,¹ and Iman Poernomo²

¹ Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Italy
{fiorenti,momiglia,ornaghi}@dsi.unimi.it

² Department of Computer Science, King's College London, Strand, London WC2R2LS, UK
iman.poernomo@kcl.ac.uk

Abstract. This paper concerns a formal encoding of the Object Management Group's Complete Meta-Object Facility (CMOF) in order to provide a more trustworthy software development lifecycle for Model Driven Architecture (MDA). We show how a form of constructive logic can be used to provide a uniform semantics of metamodels, model transformation specifications, model transformations and black-box *transformation tests*. A model's instantiation of a metamodel within the MOF is treated using the logic's realizability relationship, a kind of type inhabitation relationship that is expressive enough to express constraint conformance between terms and types. These notions enable us to formalize the notion of a correct model instantiation of a metamodel with constraints. We then adapt previous work on snapshot generation to generate input models from source metamodel specification with the purpose of testing model transformations.

1 Introduction

While model transformations have the potential to radically change the way we develop code, the actual development of transformations themselves should be conducted according to standard software engineering principles. That is, transformations need to be either certified via some kind of formal method or else developed within the software development lifecycle. Currently the latter is the norm and, consequently, effective testing techniques are essential.

However, as observed in [3], the field currently lacks of adequate techniques to support model transformation testing: testing techniques for code do not immediately carry across to the model transformation context, due to the complexity of the data under consideration.

The subject of this paper is the automated generation of test case data for black-box testing of model transformations. In this case, the test case data consists of metaobjects that conform to a given metamodel, that satisfy the precondition of the transformation's specification and conform to further constraints employed to target particular aspects of the implementation's capability. A metamodel itself has a particular structural semantics, consisting of a range of different constraints over a graph of associated metaclasses. It is therefore a difficult task to automatically generate a suitable range of instantiating metaobjects as test data.

We argue that a *trustworthy* method of testing of model transformations is of even greater importance than in ordinary software testing. This is due to the systematic, potentially exponential, range of errors that a single bug in a transformation can introduce into generated code. Within the scope of our problem, we believe it essential to formally guarantee formally that generated test models are actual instances of their, often complex, classifying metamodels and that test models satisfy the transformation specification preconditions and the tester's

given contractual constraints for generation. We provide this guarantee by employing a uniform formalism for representing models and metamodels and their interrelationship, model transformation specification and implementation and test-case generation.

Our work assumes adherence to the transformation development approach of Jezequel et al., where design-by-contract and testing are used as a means of increasing trust [6, 21]. The idea is that a transformation is equipped with a contract, consisting of a pre-condition and a post-condition. The transformation is tested with a suitable data set, consisting of a range of input models that satisfy the pre-condition, to ensure that it always produces output models that satisfy the post-condition. If an input model is produced that violates the post-condition, then the contract is not satisfied by the transformation, and the transformation needs to be corrected.

In essence, our understanding of black-box testing of a model transformation follows the framework given in Fig. 1. A transformation T takes a Platform Independent Model PIM as

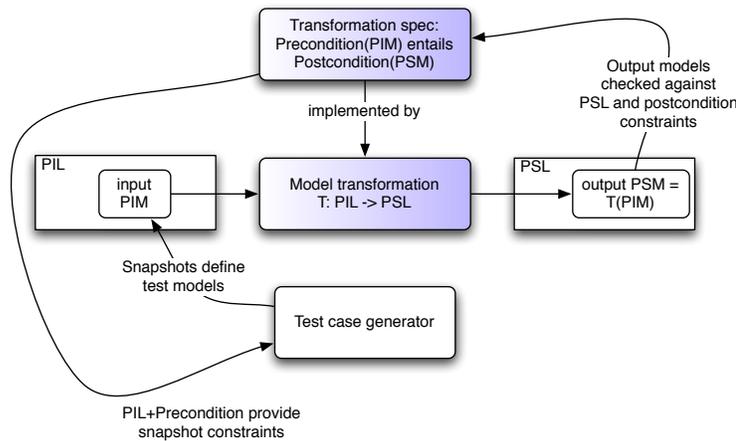


Fig. 1. Framework for testing transformation against contractual specifications and metamodel descriptions.

input, written in a source modelling language PIL , and outputs a Platform Specific Model PSM . The transformation is defined as a general mapping from elements of the language PIL to elements of the language PSL . The transformation is an *implementation* of a particular specification. This specification is a contract involving a pre-condition over any input PIM and prescribing an expected post-condition, required to hold over the resulting PSM . A test case generator produces an appropriate set of $PIMs$: the transformation can then be tested by checking that each of the resulting $PSMs$ preserve the contract.

This paper proposes a solution to open problem in model transformation testing of how to derive an appropriate data set of $PIMs$ for a given PIL metamodel, and a given pre-condition, post-condition and transformation. We will describes how ideas from constructive logic and logic programming enable a form of *meta-object snapshot generation* to solve this problem. Our approach is similar to ordinary object-oriented snapshot generation, where objects are generated to satisfy given class specifications, for the purposes of testing and validation. However, our approach is not concerned with generating instantiating programmatic objects, but with the more complicated problem of *models* that instantiate *metamodels*.

Various metaobject generation techniques have been advocated by a number of researchers for the purposes of testing. Our formalism is unique in its use of constructive logic: this has

the advantage that all components of the framework in Fig. 1 are treated uniformly, within the same language. Constructive logic forms the single representational metalanguage for the syntax and semantics of models, metamodels, transformations, transformation specifications (treated in Section 2 of the paper) and test cases and the constraints for forming test cases (treated in Section 3). As a consequence, our metaobject instantiation, the test case generation approach, contracts and contract testing have a uniform semantics. The *testing process itself* is thus more trustworthy: we have guarantees that the test cases being generated do, in fact, test the transformation specified.

In contrast, if we simply program a way of generating metaobject test cases, even if the generation employs a formal language (such as a logic programming tool), there is no formal relationship between the contract to be verified and the test cases: there is no formal guarantee that the test cases generated do indeed conform to the constraints inherent in the PIM, the preconditions of the contract and further generation constraints.

We illustrate our approach with a non-trivial example. Related work and conclusions are given in Section 4.

2 A constructive encoding of the MOF

Model transformation test cases are generated according to constraints given over MOF-based metamodels. Our approach is to take advantage of an approach to the uniform representation of both metamodel structure and semantics in utilizing constructive logic, after the fashion of CooML’s approach [8, 14] and the constructive encoding of the earlier MOF in [15, 16]. This constructive encoding has been shown to have a number of advantages as a formalisation of the MOF. In particular, realizability semantics can naturally treat higher-order metamodel instantiation, where classifiers are considered as instances of other classifiers: a feature that is prominent in the MOF itself (model instances are classification schemes, but themselves instantiate a metamodel scheme). This section sketches the principle of the constructive encoding, showing how the structure of metamodels can be understood as set theoretic signatures with a constructive *realizability* semantics to formalize instantiation. This realizability semantics will then be exploited in the next sections for test generation. The implication of our final result will be that the encoding presented here is a uniform framework for both reasoning about models and metamodels, for writing model transformations and also for generating test cases.

A metamodel for a modelling language has a definition as a collection of associated MOF meta-classes. For example, the full UML specification, like all OMG standards, has been defined in the CMOF.¹ In Example 1 we present a very simple example of metamodel to metamodel transformation, which will be used through the paper.

Example 1. Fig. 2 (a) and (c) shows two metamodels M_1 and M_2 . The metamodel instances of M_1 represent models for simple composite structures, those of M_2 for tables. We are looking for a transformation of the Component meta-objects ct into Table meta-objects t with columns corresponding to the attributes linked to ct and to the composite containing ct .² For example, the metamodel instance I_2 in Fig. 2(d), modelling a Person according to the Table metamodel, corresponds to I_1 in Fig. 2(b), modelling a Family according to the Composite

¹ EMOF and CMOF are two modelling languages of the MOF 2.0 specification.

² This is inspired by the UML2RDB challenge transformation proposed in [4], for whose details we refer to *op. cit.*

metamodel. Beside the multiplicity constraints shown in the meta-models, other constraints regard the name and id meta-attributes. Those will be discussed in Section 3.

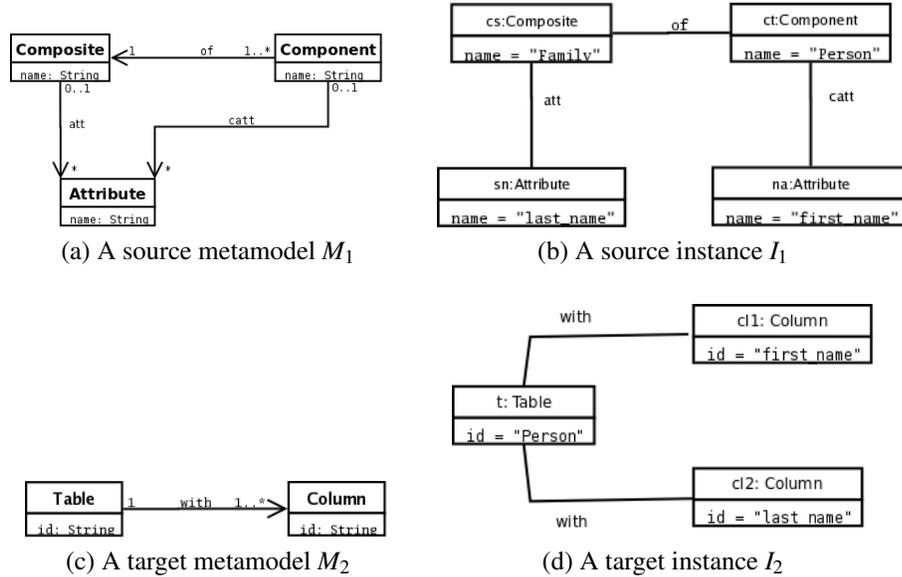


Fig. 2. A source and a target metamodel

2.1 Encoding the structure of metamodels

We now describe a simple set-based encoding of the *structure* of metamodels and metaobject instantiations. Once we have established this encoding, we will turn our consideration to *semantics* and constraint conformance, employing of a constructive logic formalism.

Since we are only concerned with creating metaobject test cases for transformations, we need not deal with representing methods within class types and, for reasons of space, we will consider the subset of the CMOF that permits a UML class-style representation of metamodel grammars.

Definition 1 (Signature for metamodel). Assume Class, Association, AssociationEnd and Property are the metaclasses for metaclasses, associations, association ends and properties in the CMOF. Take any metamodel M that consists of a set of (possibly associated) metaclasses:

$$M : \text{Set}(\text{Class}) \times \text{Set}(\text{Association}) \times \text{Set}(\text{AssociationEnd}) \times \text{Set}(\text{Property})$$

Let M_{Class} denote $\text{Set}(\text{Class})$, $M_{\text{Association}}$ denote $\text{Set}(\text{Association})$ and so on. Then, the signature for M , $\text{Sig}(M)$ is defined as $(\text{Sort}_M, \text{Rel}_M, \text{Op}_M)$, where Sort_M is a set of sort names,³ Rel_M is a set of relations, Op_M is a set of sorted operations and defined to be the minimal tuple of sets satisfying the following conditions:

³ To avoid confusion with other uses of the concept of a Type in the MOF, we use the term “sort” within our formalisation to denote classifying sets.

- $\{T_C \mid C \in M_{\text{Class}}\} \subseteq \text{Sort}_M$, where T_C is a unique sort name corresponding to a metaclass $C \in M_{\text{Class}}$.
- Every datatype T used within a Property of any $C \in M_{\text{Class}}$ or $A \in M_{\text{Association}}$ is taken as a sort in the signature: $T \in \text{Sort}_M$.
- There is a set of distinguished relations $\{isLive_C : T_C \mid C \in M_{\text{Class}}\} \subseteq \text{Rel}_M$.
- For each $A \in M_{\text{Association}}$, such that $A.\text{ownedEnd.Property.type} = T_1$ and $A.\text{memberEnd.Property.type} = T_2$, $A : T_1 \times T_2 \in \text{Rel}_M$.
- For each $C \in M_{\text{Class}}$ and each $at \in C.\text{ownedAttributed}$ such that $at.\text{type} = T$, $at : T_C \times T \in \text{Rel}_M$.

Example 2. The signature for the metamodel M_1 of Fig. 2(a) takes the following form, for $C \in \{Component, Composite, Attribute\}$:

$$\text{Sig}(M_1) = \langle \{T_C, String\}, \\ \{isLive : T_{Composite}, isLive : T_{Component}, isLive : T_{Attribute}, \\ of : T_{Component} \times T_{Composite} \text{ att} : T_{Composite} \times T_{Attribute}, catt : T_{Component} \times T_{Attribute}, \\ name : T_C \times String\}, OPString \rangle$$

Remark 1. Observe that both attributes and associations are formalized in the same way. An attribute of a metaclass is understood as a relationship that holds between an element of the metaclass sort and elements of the data type sort. Sorts $T_C \in \text{Sort}_M$ are intended to denote the range of metaclasses for a given metamodel M . As we shall see, their semantics is taken to range over an infinite domain of possible instantiating metaobjects. However, every given metamodel instance contains a finite number of metaobjects. The predicate $isLive_C$ is consequently intended to always have a finite interpretation, denoting the set of the metaobjects of metaclass C that are operational or are live in the metamodel instance. Note that multiplicities other than 1 or * (including important kinds of multiplicities such as ranges) are not dealt with through the signature. Instead, these will be treated in the same way as metamodel constraints, as part of a larger, logical metamodel specification, defined next. Finally, subclassing can be understood as a subset relation among the live objects and inheritance can be imposed by suitable axioms. We have not consider this issue here, to focus on our constructive logical approach.

Before defining our logic, we first formulate a *value-based* semantics for metamodel signatures $\text{Sig}(M)$, based on the usual notion of $\text{Sig}(M)$ -interpretation.

Definition 2 (Values). Let $T \in \text{Sort}_M$. The set of values of T , denoted by $\text{dom}(T)$, is defined as follows: if T is a data type, then $\text{dom}(T)$ is the set of values inhabiting it; if T is the type of a class C , then $\text{dom}(T) = \text{oid}(C)$, where $\text{oid}(C)$ is the set of object identifiers of class C .

Note that here we assume that data type *values* are represented by ground terms of $\text{Sig}(M)$ and that oids are constants of $\text{Sig}(M)$. Values are the same in all the possible metamodel instances. Specific metamodel instances differ according to their representation of specific interpretations of the predicates $isLive_C$, An and at .

Definition 3 (Metamodel interpretation). Take any metamodel M with signature $\text{Sig}(M)$. A metamodel interpretation is a $\text{Sig}(M)$ -interpretation \mathbf{m} such that:

1. sorts are interpreted according to Definition 2 and data type relations and operations are interpreted according to the implemented data types,
2. each predicate $isLive_C$ is interpreted as a finite sub-domain $\mathbf{m}(isLive_C) \subset \text{dom}(T_C)$ – intuitively, $\mathbf{m}(isLive_C)$ contains the metaobjects of class C that constitute \mathbf{m} ,

3. each association $An : T_1 \times T_2$ is interpreted as a relation $\mathbf{m}(An) \subseteq \mathbf{m}(isLive_{T_1}) \times \mathbf{m}(isLive_{T_2})$
4. each attribute $at : T_C \times T$ is interpreted as a functional relation $\mathbf{m}(at) \subseteq \mathbf{m}(isLive_{T_C}) \times \text{dom}(T)$.

We treat the interpretation of sorts and data types in \mathbf{m} as predefined, i.e., independently of the specific metamodel instance. The latter can be reconstructed from the interpretation of $isLive_C$, An and at . We represent this information with the model-theoretic notion of *diagram*:

Definition 4 (Diagram). *Given an interpretation \mathbf{m} and values v_1, \dots, v_n , a diagram Δ of a relation r is the set of closed atomic formulas $r(v_1, \dots, v_n)$ that are true in \mathbf{m} .*

We will use diagrams as a *canonical representation of metamodel instances*.

Example 3. The metamodel instance I_1 of the metamodel M_1 in Figure 2 is represented by the diagram:

$$\Delta_{M_1} = \{ isLive_{Composite}(cs), isLive_{Component}(ct), isLive_{Attribute}(sn), isLive_{Attribute}(na), \\ of(ct, cs), att(cs, sn), catt(ct, na), name(cs, "Family"), name(ct, "Person"), \\ name(sn, "last_name"), name(na, "first_name") \}$$

2.2 Encoding the constraints of metamodels

Signatures formalize the metamodel structure and diagrams formalize the possible instantiations of this structure. However, a metamodel is not just a structure: it also includes constraints. A diagram is correct if it represents an instantiation that satisfies the constraints. How do we formalize a metamodel with constraints and the notion of correct diagram? That is, when can we say that all the information contained in the relational assertions of a diagram yields an actual metamodel instantiation, in the sense of conforming to the structure of the metamodel and its associated constraints?

We do this via a form of constructive logic. Essentially, we define a *realizability* relationship [22] between a logical formula F and what we call an *information term* τ (see [8] for more details), denoted $\tau : F$, yielding a notion of “information content” $IC(\tau : F)$, both to be defined next. Roughly, we will use $\tau : F$ to construct instance diagrams and $IC(\tau : F)$ to validate them. The realizability relationship forms a kind of type inhabitation relationship that is sufficiently powerful to include constraint satisfaction, essential for developing formal notions of provably correct instantiating model.

Definition 5 (Information terms and content). *Given a metamodel signature $Sig(M)$, the set of information terms τ , (well-formed) formulas F over $Sig(M)$ and information content $IC(\tau : F)$ are defined as follows:*

term τ	formula F	$IC(\tau : F)$
\mathfrak{t}	$true(K)$	$\{K\}$
$\langle \tau_1, \tau_2 \rangle$	$F_1 \wedge F_2$	$IC(\tau_1 : F_1) \cup IC(\tau_2 : F_2)$
$j_k(\tau_k)$	$F_1 \vee F_2$	$IC(\tau_k : F_k) \quad (k = 1, 2)$
$(v_1 \mapsto \tau_1, \dots, v_n \mapsto \tau_n)$	$\forall x \in \{y : T \mid G(y)\}. F$	$\{ \{y : T \mid G(y)\} = \{v_1, \dots, v_n\} \} \cup \\ \bigcup_{i=1}^n IC(\tau_i : F[v_i/x])$
$e(v, \tau)$	$\exists x : T. F$	$IC(\tau : F[v/x])$

where \mathfrak{t} is a constant, K any first-order formula, $v \in \text{dom}(T)$, $\{v_1, \dots, v_n\}$ is a finite subset of $\text{dom}(T)$ and G is a generator, namely a special formula true over a finite domain.

Remark 2. Although the intuitive meaning of each of the formulas should be clear, some explanations are in order for $true(\cdot)$ and for bounded universal quantification. The predicate $true(K)$ signifies a metalogical assertion that the formula K is known to be true and that K , used as an assertion about a metamodel, does not require any specific information to be used in the construction of a metamodel instance (the idea was originally introduced in [13]). The information content is K : i.e., K is the minimal assumption needed to prove K . Regarding universal quantification, the associated information term defines both the domain $\{y : T \mid G(y)\} = \{v_1, \dots, v_n\}$ and a map from this domain to information terms for F . In a metamodel signature, the formulas $isLive_C(x)$ is an example of generator. This corresponds to the assumption that a metamodel instance contains finitely many metaobjects.

If we consider universally bounded quantification $\forall x \in \{y : T \mid G(y)\}.F$ as semantically equivalent to $\forall x : T.G(x) \rightarrow F$ and $true(K)$ as equivalent to K , our formulas can be viewed as a subset of the ordinary (many sorted) first order formulas and $\mathbf{m} \models F$ can be defined as usual. Furthermore, we may use any first order formula as an argument of $true$.

We will use formulas F to formalize constraints over metamodels and information terms to guarantee that F contains all the information needed to construct *valid* diagrams, i.e. if it is the diagram of a metamodel interpretation \mathbf{m} that satisfies the constraints – we shall formalize this in Def. 7. We will represent a metamodel M by a $Sig(M)$ -formula $Spec(M)$, where the latter encodes the constraints of the metamodel M so that if $IC(\tau : Spec(M))$ is (model-theoretically) consistent then the corresponding diagram is valid.

Consequently, with the aim of testing a transformation Tr , with input metamodel M_1 and output metamodel M_2 , if we can generate a set of $\tau_j : Spec(M_1)$ with valid information content, we can generate the corresponding diagrams, that may then be fed into the transformation as test cases. Furthermore, if we specify the precondition of Tr as $true(Pre_{Tr})$ and its postcondition as $true(Post_{Tr})$, then we can use $Spec(M_1)$, $Spec(M_2)$, $true(Pre_{Tr})$ and $true(Post_{Tr})$ to check the correctness of Tr , i.e., to supply a test oracle.

We now formally define $Spec(M)$ and introduce some notation: for an atom B , $B(s^*)$ is an abbreviation of $\forall y : T.B(y) \leftrightarrow y \in s$ and $B(!x)$ an abbreviation of $\forall y : T.B(y) \leftrightarrow y = x$.

Definition 6 (Metamodel specification). *Given a metamodel signature $Sig(M)$, we represent each class C of M by a formula of the following form, where square brackets indicate optional sub-formulae:*

$$\begin{aligned}
Spec(C) = & \forall x \in \{y : T_C \mid isLive_C(y)\}. \\
& \exists z_1 : Set(T_{C_1}).true(A_1(x, z_1^*)) [\wedge true(K_1(x, z_1))] \wedge \dots \wedge \\
& \exists z_m : Set(T_{C_m}).true(A_m(x, z_m^*)) [\wedge true(K_m(x, z_m))] \wedge \\
& \exists v_1 : T_{at_1}.true(at_1(x, !v_1)) \wedge \dots \wedge \\
& \exists v_n : T_{at_n}.true(at_n(x, !v_n)) [\wedge true(K_C)]
\end{aligned}$$

where:

1. $A_i : T_C \times T_{C_i}$ ($0 \leq i \leq m$) are the relations of Rel_M corresponding to the associations;
2. K_i ($0 \leq i \leq m$) are “multiplicity constraints” on the associations A_i ;
3. $at_j : T_C \times T_{a_j}$ ($0 \leq j \leq n$) are the relations of Rel_M corresponding to the attributes of C ;
4. K_C is a formula that we call the “class constraint” of C .

A metamodel M is specified by the $Sig(M)$ -formula

$$Spec(M) = Spec(C_1) \wedge \dots \wedge Spec(C_l) [\wedge true(K_G)] \quad (1)$$

where C_1, \dots, C_l are the metaclasses of M and K_G is an optional formula that we call the “global constraint” of M .

We switch to a “light notation”, where we use generators $isLive_C$ as sorts.

Example 4. The specification of the metamodel M_1 described in Example 2 is $Spec(M_1) = F_{Cmt} \wedge F_{Cms} \wedge F_{Att} \wedge true(K)$, where:

$$\begin{aligned}
F_{Cmt} &= \forall x : isLive_{Component}(x). \\
&\quad \exists sc : Set(Composite). true(of(x, sc^*) \wedge size(sc) = 1) \wedge \\
&\quad \exists sa : Set(Attribute). true(catt(x, sa^*) \wedge \forall a \in sa. catt(!x, a)) \wedge \\
&\quad \exists s : String. true(name(x, !s) \wedge cattNames(x, s)) \\
F_{Cms} &= \forall x : isLive_{Composite}(x). \\
&\quad \exists sa : Set(Attribute). true(att(x, sa^*) \wedge \forall a \in sa. att(!x, a)) \wedge \\
&\quad \exists s : String. true(name(x, !s) \wedge attNames(x, a, s)) \\
F_{Att} &= \forall x : isLive_{Attribute}(x). \exists s : String. true(name(x, !s))
\end{aligned}$$

We omit K for conciseness. The constraint $size(sc) = 1$ in F_{Cmt} encodes the multiplicity 1 of the target association end of of . The constraints $\forall a \in sa. catt(!x, a)$ in F_{Cmt} and $\forall a \in sa. att(!x, a)$ in F_{Cms} encode the multiplicities 0..1 of the source association ends of $catt$ and att . The encoding of multiplicity constraints is standard, i.e., it can be fully automated. Informally, the meaning of K is that different attributes linked to the same component/composite must have different names. The other constraints are encoded in Prolog, for reasons that will be explained at the end of the section. For example, the constraint $cattNames(x, a, s)$ is defined by the clause:

$$false \leftarrow isLive_{Attribute}(a) \wedge catt(c, a) \wedge name(, s)$$

i.e., it is false that there is an attribute a linked to c with the same name s .

Let $\tau : Spec(M)$ be an information term for a specification of a metamodel M . We distinguish between a “*diagram part*” IC_D and a “*constraint part*” IC_C of the information content. The latter contains the multiplicity constraints, the class constraints and the global constraint, while the former contains the *domain formulas* $\{isLive_C(o_i)\}$ for $\{y : T_C | isLive_C(y)\} = \{o_1, \dots, o_n\}$, the *association formulas*, those of the form $true(An(o, s^*))$, and the *attribute formulas*, those of the form $true(at(o, !v))$. The diagram part allows us to define a bijective map δ from the information terms for $Spec(M)$ into the instance diagrams for M , by means of the following conditions:

1. Domain formulas: $isLive_C(o_i) \in \delta(\tau)$ iff $\{isLive_C(o_i)\} \in IC_D(\tau : Spec(M))$;
2. Association formulas: $A(o, o') \in \delta(\tau)$ iff $true(An(o, \{o'_1, \dots, o'_m\}^*)) \in IC_D(\tau : Spec(M))$ and $o' \in \{o'_1, \dots, o'_m\}$;
3. Attribute formulas: $at(o, d) \in \delta(\tau)$ iff $true(at(o, !d)) \in IC_D(\tau : Spec(M))$.

Example 5. Consider the specification $Spec(M_1)$ of Example 4 and the information term $\tau_1 = \langle \tau_{1_1}, \tau_{1_2}, \tau_{1_3}, t \rangle$, where:

$$\begin{aligned}
\tau_{1_1} : F_{Cmt} &= (ct \mapsto e(\{cs\}, \langle t, e(\{na\}, \langle t, e("Person", t) \rangle) \rangle))) \\
\tau_{1_2} : F_{Cms} &= (cs \mapsto e(\{sn\}, \langle t, e("Family", t) \rangle)) \\
\tau_{1_3} : F_{Att} &= (sn \mapsto e("last_name", t), na \mapsto e("first_name", t))
\end{aligned}$$

The diagram and constraint part of $IC(\tau_1 : Spec(M_1))$ are:

$$\begin{aligned}
IC_D &= \{ isLive_{Component}(ct), isLive_{Composite}(cs), isLive_{Attribute}(sn), isLive_{Attribute}(sa), \\
&\quad of(ct, \{cs\}^*), catt(ct, \{na\}^*), att(cs, \{sn\}^*), \\
&\quad name(ct, !"Person"), name(cs, !"Family"), name(sn, !"last_name"), name(na, !"first_name") \} \\
IC_C &= \{ size(\{cs\}) = 1, (\forall a \in \{na\}. catt(!ct, a)), (\forall a \in \{sn\}. catt(!cs, a)), \\
&\quad cattNames(ct, "Person"), attNames(cs, "Family") \}
\end{aligned}$$

The diagram $\delta(\tau_1)$ coincides with Δ_{M_1} of Example 3. Clearly, we could start from Δ , reconstruct IC_D and, from the latter, reconstruct τ_1 . We can consider the diagram part as the “encoding” of the pure UML part of an UML model, as depicted in part (b) of Fig. 2.

IC_C does not play any role in the definition of the map δ , but encodes the constraints. In the above example, the constraint part is satisfied: $size(\{cs\}) = 1$ is true, and one can easily see that the other formulas in IC_C are also true. We will say that τ_1 *satisfies the constraints*.

Definition 7 (Constraint satisfaction). *Let $\tau : Spec(M)$ be an information term for a specification of a metamodel M ; τ (and $\delta(\tau)$) satisfies the constraints iff $\Delta = \delta(\tau)$ is the diagram of a metamodel instance \mathbf{m}_Δ of M such that $\mathbf{m}_\Delta \models IC_C(\tau : Spec(M))$.*

Let \mathbf{m}_Δ be as in the above definition. We can prove that $\mathbf{m}_\Delta \models IC(\tau : Spec(M))$ hence the valid metamodel instances of M are models (in the sense of logic) of $Spec(M)$. Finally, the following sufficient condition for satisfying the constraints can be proven:

Theorem 1. *Let $Spec(M)$ be the specification of a metamodel M , $\tau : Spec(M)$ an information term and Ax any set of axioms that are true over all the metamodel-instances. Then:*

- a) *if $Ax \cup IC_D(\tau : Spec(M)) \vdash \bigwedge IC_C(\tau : Spec(M))$, then τ (and $\delta(\tau)$) satisfies the constraints;*
- b) *if $Ax \cup IC_D(\tau : Spec(M)) \vdash \neg \bigwedge IC_C(\tau : Spec(M))$, then τ (and $\delta(\tau)$) does not satisfy the constraints.*

In Ax we have the axioms for the data types (integers, strings, sets, ...) and the general assumptions on the oids and on the metamodel instances.

Assume we have a *possible* term $\tau : Spec(M)$ for a metamodel M . To establish that the diagram $\delta(\tau)$ is a valid metamodel instance we could apply Theorem 1 and we could attempt to derive a proof using a suitable inference system for Ax ; alas, this is hardly predictable if Ax and constraints are full first order formulae. We need a feasible constraint language. The constraints concerning multiplicities are recognized and checked in the generation phase. To express and check the other constraints, we rely on Horn clauses. We distinguish between problem domain and absurdity clauses. The former are definite clauses implementing data types (strings, sets, bags, ...) and encoding the general properties of the MOF. The latter have the form $false \leftarrow Body$, with intended meaning $\neg \exists x. Body$. By the properties of definite programs, one can prove:

- a) *if $false$ finitely fails, then for every absurdity constraint $false \leftarrow Body$ $\neg \exists x. Body$ is satisfied;*
- b) *if $false$ succeeds, then $\neg \exists x. Body$ is falsified for some constraint.*

For example, the constraint part of the information term of Example 5 contains two absurdity constraints, namely $cattNames(ct, "Person")$ and $attNames(cs, "Family")$ and $false$ finitely fails, since there is no attribute with name "Person" or "Family". Had the diagram contained such an attribute, $false$ would had succeeded.

3 Testing via model generation

In this Section we show how our setup allows us to:

1. generate input test models that meet the precondition for a model transformation, and

2. check that, after the input models have been transformed, the transformation postcondition is met.

In this way, we are able to provide a constructive MDA transformation testing framework in accord with the concepts of Fig. 1.

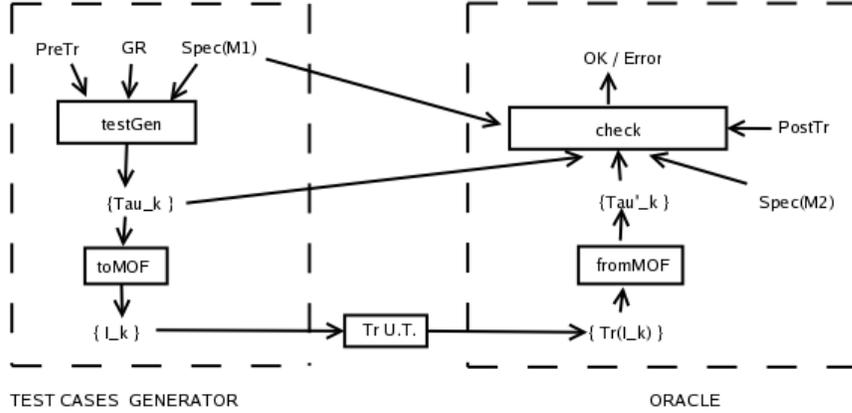


Fig. 3. Testing transformations.

Architecture. The modular architecture of our system is outlined in Fig. 3. The module `testGen` generates a set of information terms that are translated into test cases for the *transformation* undergoing testing Tr U.T. by the module `toMOF`. The inputs to `testGen` are the specification $Spec(M_1)$ of the source metamodel M_1 , the transformation specifications precondition, encoded by a constraint $true(Pre_{Tr})$ and a set GR of *generation requests* provided by the user. This module produces a set of information terms $\tau_k : Spec(M_1) \wedge true(Pre_{Tr})$ that satisfy the GR . The transformation Tr U.T. is then run using the generated test cases $I_k = toMOF(\tau_k)$.

Note that we are agnostic here regarding the way in which Tr U.T. is implemented: we could feed our test cases into a transformation written in any language. However, the transformation can also be assumed to be written in the *same* constructive language as the metamodel specification and information terms, as well as the modules themselves, because this language can include a lambda calculus. The reader is referred to [16] for details of that encoding.

The translated metamodel instances are checked using the module `fromMOF`, which attempts to reconstruct the information terms corresponding to the transformation output model $Tr(I_k)$, and the `check` module, which tries to validate $Post_{Tr}$. The two modules work, together, as a test oracle: if `fromMOF` fails, then the results do not respect the types or the invariants prescribed by $Spec(M_2)$, while if `check` fails, then only $Post_{Tr}$ is violated.

Example. Before delving into the specifics of the generation algorithm behind `testGen`, we illustrate how the architecture would operate over the source metamodel M_1 and the target metamodel M_2 shown in Fig. 2. The specification $Spec(M_1)$ is the one of Example 4, while $Spec(M_2) = F_{Tab} \wedge F_{Col} \wedge K_{Tab}$, where:

$$\begin{aligned}
 F_{Tab} &= \forall t : isLiveTable(t). \\
 &\quad \exists cl : Set(Column). true(with(t, cl^*)) \wedge \exists s : String. true(id(t, !s) \wedge id(!t, s)) \\
 F_{Col} &= \forall cl : isLiveColumn(cl). \exists s : String. true(name(cl, !s)) \\
 K_{Tab} &= true(\forall t : isLiveTable(t). uniqueColName(t))
 \end{aligned}$$

Experiment	Module	Input	Result	time (sec.)
1	Test Case Generator = testGen+toMOF	$Spec(M_1), Pre_{Tr},$ $GR = \{gr_1, gr_2, gr_3\}$	118 test cases I_k	5.9
	Oracle = fromMOF+ check	118 translations $Tr(I_k)$	76 failed	1.9
2	testGen	$Spec(M^*), GR^*, Pre^*$	144 τ_k^*	4.4

Table 1. Experimental results.

where $uniqueColName(t)$ is as follows:

$$\begin{aligned} false &\leftarrow with(t, cl) \wedge id(t, s) \wedge name(cl, s) \\ false &\leftarrow with(t, cl_1) \wedge with(t, cl_2) \wedge name(cl_1, s) \wedge name(cl_2, s) \end{aligned}$$

Let Tr be a transformation mapping M_1 -instances into M_2 -instances defined by the following informal specification:

- *Pre-condition.* For the sake of expositon, we erroneously set the precondition to *true*, i.e., the domain of Tr coincides with the set of all the source metamodel instances. The analysis of the failed test cases will show the need of a less liberal precondition and suggest it.
- *Post-condition.* For every `Composite` object cs and every `Component` ct of cs there is a corresponding `Table` t such that:
 - $t.id$ is equal to $ct.name$;
 - for every `Attribute` a linked to ct by `catt` there is a corresponding `Column` cl linked to t by `with`, so that $cl.id$ is equal to $a.name$;
 - for every `Attribute` a linked to cs by `att` there is a corresponding `Column` cl linked to t by `with`, so that $cl.id$ is equal to $a.name$;

For example, the metamodel instance I_1 of M_1 is transformed into the metamodel instance $I_2 = Tr(I_1)$ of M_2 (see Fig. 2). The relations $toTable(ct, t)$ (“Component ct corresponds to Table t ”), $toCol(a, cl)$ (“Attribute a corresponds to Column cl ”) and the desired association `with` are formalized as follows:

$$\begin{aligned} toTable(ct, t) &\leftrightarrow isLive_{Component}(ct) \wedge isLive_{Table}(t) \wedge \exists s : String.name(ct, s) \wedge id(t, s) \\ toCol(a, cl) &\leftrightarrow isLive_{Attribute}(a) \wedge isLive_{Column}(cl) \wedge \exists s : String.name(a, s) \wedge name(cl, s) \\ with(t, cl) &\leftrightarrow (\exists ct. \exists a. toTable(ct, t) \wedge toCol(a, cl) \wedge catt(ct, a)) \vee \\ &\quad (\exists ct. \exists cs. \exists a. toTable(ct, t) \wedge toCol(a, cl) \wedge of(ct, cs) \wedge att(a, cs)) \end{aligned}$$

Furthermore, we have the constraints:

$$\begin{aligned} false &\leftarrow isLive_{Component}(ct) \wedge \neg(\exists t : isLive_{Table}(t). toTable(ct, t)) \\ false &\leftarrow isLive_{Table}(t) \wedge \neg(\exists ct : isLive_{Component}(ct). toTable(ct, t)) \\ false &\leftarrow isLive_{Attribute}(a) \wedge \neg(\exists cl : isLive_{Column}(cl). toCol(a, cl)) \\ false &\leftarrow isLive_{Column}(cl) \wedge \neg(\exists a : isLive_{Attribute}(a). toCol(a, cl)) \end{aligned}$$

To perform some experiments, we have implemented Tr in Prolog, using diagrams as representations of metamodel instances. To generate our test cases, we supplied (beside $Spec(M_1)$ and the precondition) the following set $\{gr_1, gr_2, gr_3\}$ of generation requests:

$$\begin{aligned} gr_1. & \maxClassSize([composite, 2], [component, 4], [attribute, 8]) \\ gr_2. & \maxMultiplicity([of : component, 2], [catt : attribute, 2]) \\ gr_3. & \text{goal}([name(attribute, o_1, "aa"), name(attribute, o_2, "aa"), o_1 \neq o_2]) \end{aligned}$$

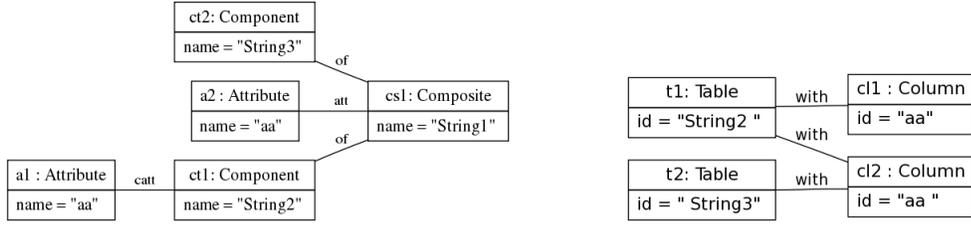


Fig. 4. Failed test cases

With *maxClassSize*, we limit the maximum number of objects to be generated: for example, at most 2 composite objects are generated by [*composite*, 2]. *maxMultiplicity* limits the multiplicity *: for example, by [*of:component*, 2] a composite object may be linked to at most 2 component objects. The *goal* request introduces a goal to be satisfied requiring at least two distinct attribute objects with the same name "aa". The purpose of *gr₃* is to check that the constraint imposing that different columns of the same table have different names is respected by *Tr*. By *GR* above, we obtained 118 test cases that were, in turn, piped into *Tr*. The results have been checked by the oracle module of our architecture. The experimental results are summarized in Table 1. The experiment was conducted with an Intel 2.2 GHz processor T7500 operating with 2Mb of RAM.

In the first experiment we used the specifications and the generation requests explained above. The *fromMOF* module of the oracle resulted in a failure message for 76 of the 118 generated test cases. The remaining results passed the *check*. This pointed out the need of a precondition, as illustrated by the errors 1 and 2 shown by the sample test cases in Fig. 4:

1. The column *cl2* corresponding to *a2* occurs in both the tables *t1* and *t2* corresponding to *ct1* and *ct2* resp., i.e., the multiplicity 1 of *with.Table* is violated.
2. The table *t1* corresponding to *ct1* contains two columns with the same id "aa".

Both errors regard the precondition: we have to strengthen it to exclude snapshots of the above kind from the input of *Tr*.

In the *testGen* module, the information terms are generated in two phases. In the first one, some existential parameters are left *generic* (this is possible thanks to the constructive structuring of the information, see rule $\exists\text{exp}$ in the next Section 3). In the second phase, the generic parameters are instantiated randomly or according to some generation requests. As a rule of thumb, one leaves generic those existential parameters that give rise to a combinatory explosion of the number of solutions. In particular we used in the first phase of experiment 1 the generation requests *gr₁*, *gr₂*, which leave the attribute name open; we obtained 18 generic solutions in 0,01 sec. In the second phase the name attribute was instantiated according to *gr₃* (at least two attributes with name "aa" and the other ones distinct): we obtained 118 test cases. As another somewhat larger example, we have formalized the UML metamodel considered in [20] (here denoted *M**) and we have generated 144 generic terms in 4,4 sec.

Snapshot generation. We now describe snapshot generation as an abstract machine operating on the following categories:

$$\begin{aligned}
 \text{Formula Context } \Phi &::= \cdot \mid \Phi, \alpha : F \\
 \text{Eigenvariable Context } \Gamma &::= \cdot \mid \Gamma, y : T \\
 \text{Constraints } K &::= \top \mid K_1 \wedge K_2 \\
 \text{Substitutions } \sigma &::= \varepsilon \mid \sigma, F / \alpha \\
 \text{Configurations } \mathcal{C} &::= \{ \alpha : F \} \mid \{ \Gamma \mid K \mid \sigma \} \mid \{ \Phi \mid \Gamma \mid K \mid \sigma \}
 \end{aligned}$$

Intuitively we start with a formula $\alpha : F$, where *F* encodes a given spec, labeled by a metavariable ranging over info terms. By a non-deterministic application of the following rewrites,

we windup in a final configuration $\{\Gamma \mid K \mid \sigma\}$; the latter yields the info term realizing F in a context Γ by instantiating α with the computed substitution σ and accumulating a set of constraints K . Note that in the underlying implementation, consistency of formulas under a *true* is checked eagerly.

$$\begin{aligned}
init &: & \{\alpha : F\} &\rightsquigarrow \{\alpha : F \mid \cdot \mid \top \mid \varepsilon\} \\
stop &: & \{\cdot \mid \Gamma \mid K \mid \sigma\} &\rightsquigarrow \{\Gamma \mid K \mid \sigma\} \\
true &: & \{(\Phi, \alpha : true(H) \mid \Gamma \mid K \mid \sigma)\} &\rightsquigarrow \{\Phi \mid \Gamma \mid K \wedge H \mid \sigma \oplus t / \alpha\} \\
\forall clo &: & \{(\Phi, \alpha : (\forall x : G.F)) \mid \Gamma \mid K \mid \sigma\} &\rightsquigarrow \{\Phi \mid \Gamma \mid K \mid \sigma \setminus \{x / \alpha\}\} \\
\forall exp &: & \{(\Phi, \alpha : (\forall x : G.F)) \mid \Gamma \mid K \mid \sigma\} &\rightsquigarrow \{(\Phi, \beta : F(y)) \mid \alpha' : (\forall x : G.F) \mid (\Gamma, y : T_G) \mid K \mid \\
& & & \sigma \oplus (y \mapsto \beta, \alpha') / \alpha\} \\
\exists chs &: & \{(\Phi, \alpha : (\exists x : T.F)) \mid \Gamma \mid K \mid \sigma\} &\rightsquigarrow \{(\Phi, \beta : F(g)) \mid \Gamma \mid K \mid \sigma \oplus e(g, \beta) / \alpha\} \\
\exists exp &: & \{(\Phi, \alpha : (\exists x : T.F)) \mid \Gamma \mid K \mid \sigma\} &\rightsquigarrow \{(\Phi, \beta : F(y)) \mid (\Gamma, y : T) \mid K \mid \sigma \oplus e(y, \beta) / \alpha\} \\
\wedge elm &: & \{(\Phi, \alpha : F_1 \wedge F_2) \mid \Gamma \mid K \mid \sigma\} &\rightsquigarrow \{(\Phi, \alpha_1 : F_1, \alpha_2 : F_2) \mid \Gamma \mid K \mid \sigma \oplus \langle \alpha_1, \alpha_2 \rangle / \alpha\}
\end{aligned}$$

Some comments are in order. Rule $\forall exp$ introduces a fresh eigenvariable y and recurs, duplicating the universal formula so that more snapshots can be generated: σ is updated (\oplus) with a new mapping for the new name. A configuration with leading universals can also be (non-deterministically) *closed*, e.g. when *generation requests* have been met. Similarly for existentials, where in $\exists chs$ the ground value g is chosen out of some GR . We skip over the rule for disjunction. Snapshot generation preserves the following: if $\{\alpha : F\} \rightsquigarrow^* \{\Gamma, K, \sigma\}$ and $\tau = \alpha\sigma$, then τ realizes F in the context $\Gamma\sigma$; further, for all interpretation i and every assignment ρ to the free variables of K , if $i \models K\rho$ then $i \models \tau\rho : F$.

4 Related work and conclusions

While full verification of model transformations can be as difficult to achieve as in ordinary programming, the power of model transformations demands some formal guarantee that any generation algorithm actually produces the tests that we expect: in particular, that tests cases are of appropriate metamodel types and satisfy generation constraints. We have developed a constructive encoding of the CMOF that facilitates this via a uniform, single-language treatment of models, metamodels, instantiation, transformation specification, test case generation constraints *and* test cases generation. To the best of our knowledge, a similar approach has not been explored before.

The relevance of snapshot generation (SG) for validation and testing in OO software development is widely acknowledged. The USE tool [9] has been the first one supporting automatic SG; differently from us, SG requires the user to write Pascal-like procedures in a dedicated language. The performances of USE are very sensitive to the *order* of objects and attribute assignments [2]. Other animation and validation tools support different languages. Alloy [10] compiles a formula in first-order relational logic into quantifier-free booleans and feeds to a SAT solver. Alloy is the leading system [2] for generation of instances of invariants, animation of the execution of operations and checking of user-specified properties. Alloy's original design was quite different from UML, but recent work [1] has brought the two together.

A close relative is FORMULA [11], a tool supporting a general framework for model-based development. Similarly to us, it is based on logic programming, more specifically model generation is based on abduction over non-recursive Horn logic with stratified negation. Model transformations are also encoded logically. Since abduction can be computationally expensive, we plan to compare it with realizability-based SG, as the FORMULA setup can give us a hook to more general domain-specific modeling languages.

Baudry et al. have developed a tool that maps metamodel descriptions in the Ecore framework into Alloy, delegating to the latter the generation of model snapshots to test model transformations [19]. Their approach is focused on generation of models solely from metamodel encoding. Our approach could be used to enhance their result, through a uniform treatment of the wider problematics of satisfying metamodel, model transformation specification and formation constraints to produce test cases. On the other hand, they have extensively investigated the issue of the quality and adequacy of test models, both in term of generation strategies and of mutation analysis [19]. We have already adopted a version of their *domain partition* strategy to guide test generation and plan to refine it towards the filtering of *isomorphic* and thus useless test models.

A number of authors have attempted to provide a formal understanding of metamodelling and model transformations. Ruscio et al. have made some progress towards formalizing the KM3 metamodelling language using the Abstract State Machines [18]. Rivera and Vallecillo have exploited the class-based nature of the Maude specification language to formalize metamodels written in the KM3 metamodelling language [17]. Their treatment of the dual, object- and class-based, representation of metamodels is similar to ours, involving an equivalence mapping. The intention was to use Maude as a means of defining dynamic behaviour of models, something that our approach also lends itself to. Their work has the advantage of permitting simulation via rewriting rules. A related algebraic approach is given by Boronat and Meseguer in [5]. These formalisms are useful for metamodel verification purposes, but are currently not amenable to the test case generation problem.

Rule-based model transformations (in contrast to a procedural/functional ones found in language such as Kermeta and Converse), have a natural formalization in graph rewriting systems [12]: for example, Ehrig et al. have equipped graph grammars with a complex notion of instance generation: essentially adding a means of generation directly into graph rewriting [7]. As with our work, their formalism permits a uniform semantics of model transformations, specification and test generation, although with a fairly heavy mathematical overhead. However, their approach is by definition applicable within for rule-based paradigm: in contrast, because our tests are contractual and based in the very generic space of constructive logic, we need not restrict ourselves to rule-based transformations.

Our implementation consists of a low level encoding and needs future work to be readily integrated into other tool sets: in particular, we need to investigate how standard visual representations of metamodels and transformations might complement the approach, together with better translations from OCL into our logic for constraint representation. There are further optimisation that can be done to improve the constraint solver: for example, isomorphic solutions are reduced but, currently, not eliminated, and divide and conquer strategies such as modularization of tests could be employed to overcome the potential combinatory explosion for large metamodels.

Our approach can currently be considered as one way of integrating formal metamodelling perspectives with snapshot generation for testing. While formal metamodelling opens up the possibility of full transformation verification, from a practical perspective, testing is likely to remain an integral component of transformation development for the medium term. However, by following an approach such as ours, formal metamodelling can still be exploited to generate test data in a way that is guaranteed to preserve consistency with required constraints. For this reason, we see this work as opening up a very promising line of research for the formal metamodelling community. More detail of the implementation and examples can be found at <http://cooml.dsi.unimi.it/SnapMOF>.

References

1. K. Anastakis, B. Bordbar, G. Georg, and I. Ray. On challenges of model transformation from UML to Alloy. *Software and System Modeling*, 9(1):69–86, 2010.
2. E. G. Aydal, M. Utting, and J. Woodcock. A comparison of state-based modelling tools for model validation. In R. F. Paige and B. Meyer, editors, *TOOLS (46)*, volume 11 of *Lecture Notes in Business Information Processing*, pages 278–296. Springer, 2008.
3. B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. L. Traon. Model transformation testing challenges. In *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing*, 2006.
4. J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt. Model transformations in practice workshop. In J.-M. Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 120–127. Springer, 2005.
5. A. Boronat and J. Meseguer. An algebraic semantics for the MOF. In J. L. Fiadeiro and P. Inverardi, editors, *FASE 2008. Proceedings*, volume 4961 of *LNCS*, pages 377–391. Springer, 2008.
6. E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. L. Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *ISSRE 2006*. IEEE Computer Society, 2006.
7. K. Ehrig, J. M. Küster, and G. Taentzer. Generating instance models from meta models. *Software and System Modeling*, 8(4):479–500, 2009.
8. M. Ferrari, C. Fiorentini, A. Momigliano, and M. Ornaghi. Snapshot generation in a constructive object-oriented modeling language. In A. King, editor, *LOPSTR*, volume 4915 of *LNCS*, pages 169–184. Springer, 2007.
9. M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and System Modeling*, 4(4):386–398, 2005.
10. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
11. E. Jackson and J. Sztipanovits. Formalizing the structural semantics of domain-specific modeling languages. *Software and Systems Modeling*, 2009.
12. A. Königs and A. Schürr. Multi-domain integration with mof and extended triple graph grammars. In J. Bézivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005.
13. P. Miglioli, U. Moscato, M. Ornaghi, and G. Usberti. A constructivism based on classical truth. *Notre Dame Journal of Formal Logic*, 30(1):67–90, 1989.
14. M. Ornaghi, M. Benini, M. Ferrari, C. Fiorentini, and A. Momigliano. A constructive object oriented modeling language for information systems. *ENTCS*, 153(1):67–90, 2006.
15. I. Poernomo. A type theoretic framework for formal metamodelling. In R. H. Reussner, J. A. Stafford, and C. A. Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*, pages 262–298. Springer, 2004.
16. I. Poernomo. Proofs-as-model-transformations. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *ICMT 2008, Proceedings*, volume 5063 of *LNCS*, pages 214–228. Springer, 2008.
17. J. Rivera and A. Vallecillo. Adding behavioural semantics to models. In *EDOC 2007*, pages 169–180. IEEE Computer Society, 2007.
18. D. D. Ruscio, F. Jouault, I. Kurtev, J. Bézivin, and A. Pierantonio. Extending AMMA for supporting dynamic semantics specifications of DSLs. TR 06.02, LINA, Nantes, France, Apr. 2006.
19. S. Sen, B. Baudry, and J.-M. Mottu. On combining multi-formalism knowledge to select models for model transformation testing. In *ICST*, pages 328–337. IEEE Computer Society, 2008.
20. S. Sen, B. Baudry, and J.-M. Mottu. Automatic model generation strategies for model transformation testing. In R. F. Paige, editor, *ICMT*, volume 5563 of *LNCS*, pages 148–164. Springer, 2009.
21. Y. L. Traon, B. Baudry, and J.-M. Jezequel. Design by contract to improve software vigilance. *IEEE Trans. Softw. Eng.*, 32(8):571–586, 2006.
22. A. S. Troelstra. Realizability. In S. R. Buss, editor, *Handbook of Proof Theory*, chapter IV, pages 407–473. Elsevier, 1998.