A framework for testing model composition engines*

Freddy Munoz, Benoit Baudry

INRIA, Centre Rennes - Bretagne Atlantique
Campus de Beaulieu, F-35042
Rennes Cedex, France
{fmunoz,bbaudry}@irisa.fr

Abstract. Model composition helps designers managing complexities by modeling different system views separately, and later compose them into an integrated model. In the past years, researchers have focused on the definition of model composition approaches (operators) and the tools supporting them (model composition engines). Testing model composition engines is hard. It requires the synthesis and analysis of complex data structures (models). In this context, synthesis means to assembly complex structures in a coherent way with respect to semantic constraints. In this paper we propose to automatically synthesize input data for model composition engines using a model decomposition operator. Through this operator we synthesize models in a coherent way, satisfying semantic constraints and taking into account the complex mechanics involved in the model composition. Furthermore, such operator enables a straightforward analysis of the composition result.

Keywords: Model composition, Model composition engines, Software testing, Data synthesis

1 Introduction

Model-Driven Engineering (*MDE*) aims at tackling the growing complexity of constructing software systems, this by promoting the systematic use of models as primary engineering artifacts [1]. Typically, a software system is not modeled into a single unit capturing all the properties. Instead, it is practical to split the design into several views that capture the specific properties and concerns. Designing a software system with different views enables designers to separate concerns and cope with one complexity dimension at the time. Later, these views are composed into an integrated model representing a general view of the system. This model can be then used to assess consistency of the different views, feed an automatic code generator or an automatic analysis tool to detect conflicts.

Several researchers have addressed the development of composition approaches (*composition operators*) and tools supporting them [2-5]. Such tools, referred as *model composition engines (CE),* aim at composing models as specified in the composition operators they realize.

* This work was partially supported by the European project DiVA (EU FP7 STREP).

Testing whether the model produced by a *CE* is as expected requires: (1) the generation of test data, which are a pairs of composable models covering the input domain of the *CE*, and (2) the expected output model for each input pair.

A pair of models is *composable* with respect to a composition operator, if the operator can unify the elements constituting each model to produce a final well-formed model. In other words, a composable pair of models is one that can be processed by a CE to produce a coherent result.

For testing a *CE*, we can revert the process specified by a composition operator, through a *decomposition operator*. We refer to such process as *decomposition,* and its realization as model decomposition engines (*DE*). Given this *DE* we propose to automatically synthesize models covering input domain of *CE*. We pass these models as input to the *DE* that decomposes them to generate pairs of models, which are composable by *CE*. Since *DE* reverts the process carried out by *CE*, when *DE* generates a pair of models m_1, m_2 from a model *m,* the composition of m_1 with m_2 by *CE* is expected to produce *m*.

The contribution of this paper is describing a strategy to create a decomposition operator, and use it to test a *CE*. This strategy serves as a guideline for creating decomposition operators and their respective decomposition engines. We have created a framework that, given a *DE* and a *CE,* automatically synthesizes input data, decomposes such data, executes the *CE* under test, and decides whether the composition result is correct. That is, an automated process that help tester validating their *CE*. In order to illustrate the usefulness of this framework, we have performed the whole process to test the *kompose* CE [6].

This paper is organized as follows. Section 2 introduces the model composition concepts. Section 3 explains the testing of model composition engines. Section 4 introduces our synthesis technique. Section 5 presents the analysis we perform over the composition result. Section 6 describes the implementation of this approach as a generic testing framework. Section 7 presents experimental results through mutation analysis. Section 8 presents the related work, and finally section 9 concludes.

2 Composing models

Composing models consists in integrating several models representing different modeling dimension into a single one. The remainder of this section introduces a short running example through which we explain the model composition concepts.

2.1 Running example

In model driven engineering, the basis for creating models is specified in a meta-structure (meta-model). Meta-models define the elements, relations, and semantic constraints of the models they specify.

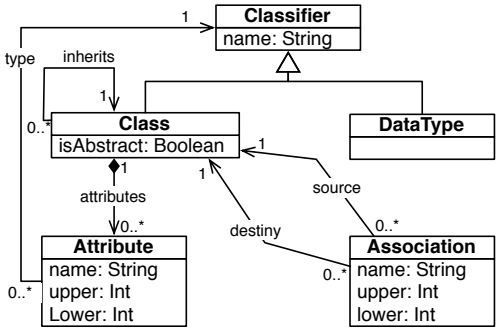

Fig. 1. Reduced version of the UML Class Diagram meta-model (*RCD*)

Figure 1 presents a reduced version of the UML Class Diagram (RCD) meta-model. This version of the meta-model introduces only classes, attributes, and associations between classes and data types used by attributes. All the models we introduce conform to this meta-model.

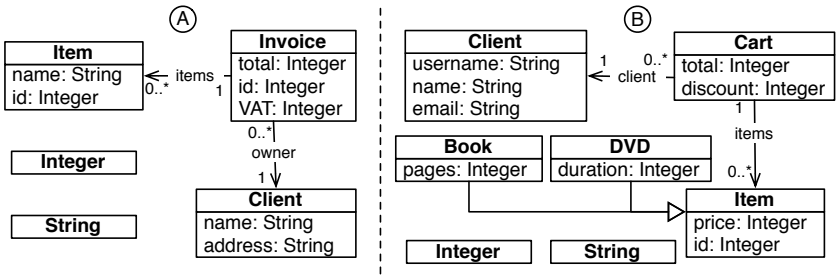

Fig. 2. Class diagrams (models conforming to the *RCD* meta-model) of an (A) invoice service and a (B) sales cart concern

Figure 2 presents two class diagrams of an online sales system. Both diagrams model the same system capturing different features. The diagram on top (A) captures the invoice concern where a client is associated with several invoices, each one containing several items. The diagram on bottom (B) captures the sales cart concern where a client selects items and puts them into the cart to later check out.

2.2 Different ways to compose models

There exists different techniques to compose models, and they can be classified in *symmetric* and *asymmetric*. *Symmetric* model composition consists in the integration of model coming from the same meta-model and representing equivalent concepts [3, 4, 7, 8]. For instance, the composition of two class diagrams into a single one is symmetric because equivalent elements are composed. On the other hand, *asymmetric* model composition integrates models from different meta-models [9]. In this paper we address the symmetric model composition and hence the approach we propose is applicable over any symmetric model composition engine.

2.3 Overview of symmetric model composition

Composing models is a twofold process that integrates different models into a single one. The first step consists in specifying the composition procedure with a *model composition operator*. The second step consists in implementing a model composition engine (*CE*) that realizes the specifications expressed in the composition operator.

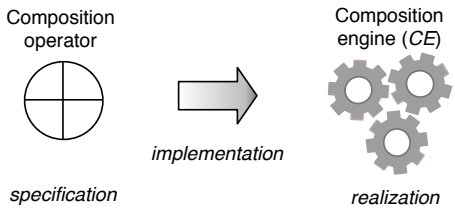

Fig. 3. Specification of a model composition approach as a composition operator and the derivation of its realization as a model composition engine (*CE*)

Figure 3 illustrates these twofold and the link between them. A composition operator specifies the composition in terms of atomic steps at an abstract level. An engineer then takes this specification and implements a *CE* using a specific programming or modeling language such as Kermeta [10]. Notice that the abstractions defined in the specification are not necessarily implemented as single low-level operations. Instead, they are implemented as a sequence of low-level operations that manipulate the model elements. Since the testing activity distinguish specification from realization, we present separately the composition operator (specification) from the composition engine (realization).

A composition operator $\oplus$ specifies how to integrate pairs of models in terms of their constituent elements (model elements). These model elements have an associated semantic that defines whether they can be actually composed or not. The composability of these elements is defined over a meta-model, that is, the meta-classes that these elements instantiate are designated to be composable or not. For instance, the composability semantic associated with the *RCD* meta-model specify that instances of the meta-classes `Classifier`, `Attribute` and `Association` are composable. This knowledge comes from the designer of each particular meta-model and influences the way in which models are composed.

Typically, $\oplus$ is specified by sub operators: (1) a *match operator*, which defines an equivalence relation between the input models. That is, when a pair of model elements are equivalent, then, composed. (2) A *merge operator*, which is responsible for composing the input model elements. It defines, in terms of atomic operations or *composition primitives*, how to compose matched elements and what to do with unmatched elements.

In this paper we use the composition operator $\oplus_{sig}$ proposed by France et al in [3]. The match operator defines that two elements are equivalent when they instantiate the same meta-class and have the same *signature*. A *signature* uniquely identifies an element and is defined on each composable meta-class. For instance, the meta-class `Classifier` is signed by its `name`, which means that two instances of Classifier

that have the same name match, whereas the `name` and `type` signs `Attribute`. The merge operator comprises two composition primitives. (1) A *unify(a, b)* primitive that unifies two model elements *a*, *b* existing in both of the input models into a single one in the resulting model. (2) A *copy(a)* primitive that copies the model element *a* existing only in one of the input models into the resulting model. A formal algebraic notation for model composition is introduced in [11], we reuse this notation to define ⊕.

Definition 1. Let ***MM*** be a meta-model defining particular model elements and the relations between them. So ***M*** is the set of all the well-formed models conforming to ***MM***. The models $\boldsymbol{m, m_1, m_2 \in M}$ are particular well-formed models conforming to ***MM***. A symmetric composition operator ⊕ is a function that maps a pair of elements (binary operator) $[\boldsymbol{m_1, m_2}] \in \boldsymbol{M}$ into a single $\boldsymbol{m \in M}$.

$$\oplus : \boldsymbol{M} \; x \; \boldsymbol{M} \rightarrow \boldsymbol{M} \qquad (1)$$

Figure 4 shows the result of composing of the model in figure 2, ⊕(*A, B) = C*. Notice that the model elements `Client` in *A* and *B* are unified because they have the same signature in *A* and *B*. On the other hand, the remainder model elements are copied. It is worth mentioning that the use of signatures is specific to the approach defined by France et al [3]. Other approaches could propose, for example, to use object identifiers as a basis for match operators.

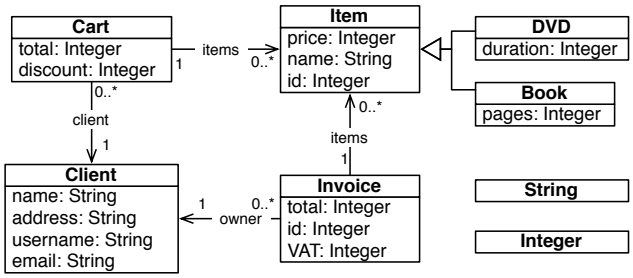

Fig. 4. Model resulting from the composition of the models *A* and *B* in figure 2

3. Testing model composition engines

Testing *CE*s consists in a three-step process. (A) The first is the synthesis of input data, that is, models conforming to a meta-model. These models must fulfill a criterion of coverage for the meta-model. (B) The second is the composition of pairs of model using the *CE* under test (*CEUT*). (C) The third and final is the evaluation of the model produced by the *CEUT*. Figure 5 illustrates this process.

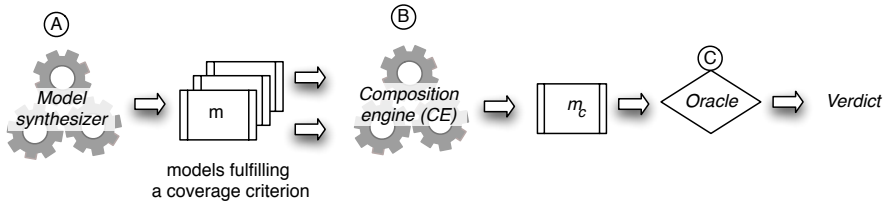

Fig. 5. Testing process for a model composition engine. It consists in generating data (A), composing the generated data (B), and evaluating the result produced in B (C).

This process seems to be straightforward and feasible using existing techniques to generate and evaluate models. Existing techniques such as [12-14] can synthesize models conforming to a given meta-model, and pick a pairs among them to execute the *CEUT*. However, this is tricky because it is not easy to ensure that the selected pairs will be composable. Furthermore, even if the selected pair is composable, there is no guarantee about the evaluability of the expected result.

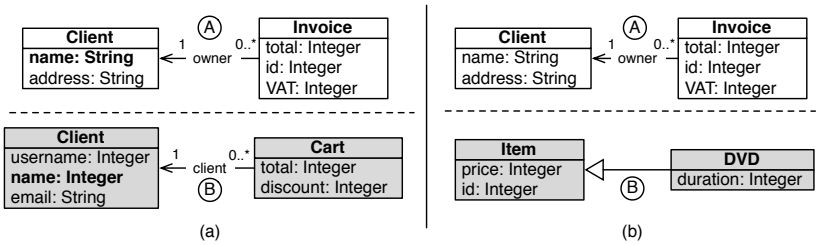

Fig. 6. Models conforming to the *RCD* meta-model, at the left (a), two models that cannot be composed, at the right (b) two models that can be composed.

Figure 6 (a), (b) presents two models *A, B*, conforming to the *RCD* meta-model. At the left (a), composing the models *A* and *B* makes sense because they have the class `Client` in common. However, when composing these two models, the resulting model does not conform to the *RCD* meta-model. This failure is due to an inconsistency between the two models. One defines an attribute `name` with type `String`, whereas the other defines the same attribute, but with type `Integer`. Since the same attribute have different types, it is copied into the unified class. This results in a violation of the class diagram constraint specifying that the name of an attribute is unique for a class. We say that these models are not composable because when composed they do not produce a well-formed model.

A different situation occurs at the left (b) of figure 6. Notice that the model elements in *A* have no equivalence with those in *B*. That is, the composition of these models produces a model containing their elements separately. None of the model elements are unified because they are not equivalent. In this case the models are composable and produce a result that conforms to the *RCD* meta-model. However, even when the composition produces a result, this result has no further meaning because it is the same as having separated models.

The previous examples show some difficulties that can arise when trying to compose two models. When these models are automatically synthesized, and a pair of

them is selected for composition, it is not possible to guarantee whether they may be composable or not. Furthermore, even if these models are composable, there is no way to ensure that they will produce a meaningful and evaluable result.

It is worth mentioning that although these examples are based on the composition operator defined in section 2.2 and the *RCD* meta-model, the situation they illustrate can arise whatever the meta-model or composition operator.

Automatically synthesizing models is insufficient for testing *CE*s because the relation that makes pairs of models composable is systematically ignored. Furthermore, as the relation between pairs of models is ignored, predicting what the composition result must look like is very difficult.

4. Automatic generation of test data

In the previous section we illustrated the difficulties of current model synthesis techniques to generate test data for *CE*s. This motivates us to propose an approach to generate models in such a way that composability and evaluability is guaranteed.

4.1 Synthesis of composable models

In our approach, we do not synthesize models; instead, we transform the models produced by a model synthesizer into pairs of models suitable to be composed. To achieve this goal, we introduce a specification that captures the relation that pairs of models must bear in order to be composable. We refer to this specification as *decomposition operator,* from which we derive an implementation that actually decomposes models generating pairs of them. We refer to such implementation as *decomposition engine* (*DE*).

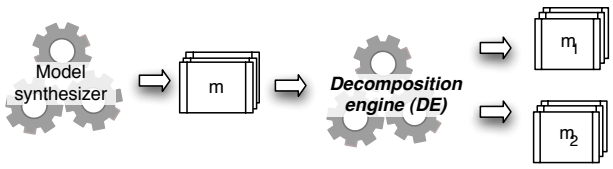

Fig. 7. Generation of composable pairs of model through the decomposition of models generated by a model synthesizer

Figure 7 illustrates our approach. Initially, a model synthesizer generates models conforming to a meta-model. This synthesizer will generate as much models as needed to cover the meta-model structure. Then, a *DE* processes these models and produces several pairs of models and ensures their composability. That is, the *DE* decomposes the models in such a way that the results it produces can be composed. Furthermore, these pairs will also produce predictable results, which make the analysis of the composition result straightforward.

4.2 Decomposing models

Our approach relies on the generation of models in a very particular way, which is specified by a *decomposition operator* that describes how to *generate* models from other models. That is, how to break a single model into other models containing its constituting elements. The process of breaking a model to generate other models containing the elements of the first is called *decomposition*.

More precisely, decomposing a model consists in applying a series of atomic operations (*decomposition primitives*) to break one model into several pairs of model.

Definition 2. Let $m, m_{11}, m_{21}, \ldots, m_{1n}, m_{2n} \in M$ be well-formed models conforming to a meta-model MM. A symmetric decomposition operator $\ominus$ is a function that maps a single model (unary operator) $m \in M$ into a set of pairs of models $M_p = [m_{11}, m_{21}] \ldots [m_{1n}, m_{2n}] \subseteq \wp(M \times M)$.

$$\ominus : M \rightarrow M_p \,/\, M_p \subseteq \wp(M \times M) \qquad (2)$$

The decomposition operation is typically the inversion of the composition operation. The decomposition operator takes a model m and transforms it into a set of model pairs, in such a way that the composition of these pairs will generate m.

$$\ominus = \oplus^{-1} \Leftrightarrow (\forall \{m_1, m_2\} \in M_p), (\exists\, m \in M) \,/\, \oplus(m_1, m_2) = m \wedge \ominus(m) = M_p \qquad (3)$$

Figure 8, illustrates the multiples decompositions for a single model. This shows all the decomposition (on top) for the single class `Item` (on bottom).

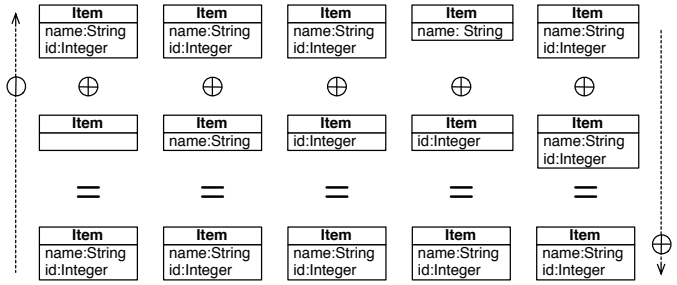

Fig. 8. The same result is generated by the composition of different models. The dotted arrow from bottom to top represents the decomposition of a single model into the pairs.

Analogously to composition operators, we construct decomposition operators in terms of atomic operations or *decomposition primitives*. We derive these decomposition primitives from the primitives defined by a composition operator. For example, the decomposition operator $\ominus_{sig}$ of $\oplus_{sig}$ is constructed as follows. The composition primitive *unify*, and *copy* perform the composition of matched and unmatched elements respectively. The first primitive creates a single element starting

from a model element existing in both of the input models. The second copies an element existing only in one of the input models into the resulting model. Inspired by these composition primitives, we design the decomposition primitive *clone,* and *side*. The primitive *clone(a)* copies the model element *a* into both of the resulting models, whereas the *side(a)* primitive copies the element *a* into only one of the resulting models.

Typically, the match sub-operation of the composition operator defines constraints over the decomposition process. For instance, the match operator of $\oplus_{sig}$ establishes decomposition constraints between the meta-classes in ***MM***. For example, the match operator of $\oplus_{sig}$ for the reduced class diagram meta-model establishes a relation between the meta-classes `Attribute` and `Classifier`. Such relation is determined by the `Attribute`'s signature, which includes the `Classifier`'s signature since it references in the meta-attribute `type`. Whenever the decomposition process may apply a decomposition primitive on the model element `Client.address` in figure 2, it must also apply the same primitive to the model element `String`.

The meta-model designer, based on his domain knowledge declares explicitly the composition constraints[1], for example *there cannot be attributes without a type*. In order to respect these constraints, we propose to check them before performing the decomposition. This guarantees the well formedness of the decomposed models.

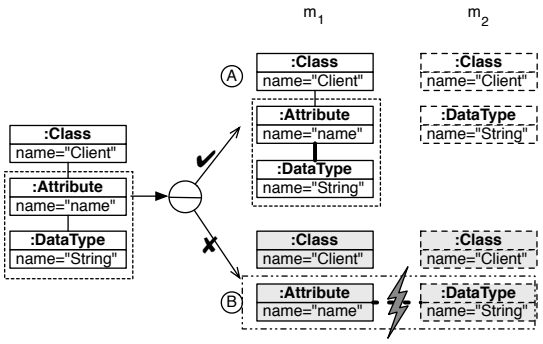

Fig. 9. The figure illustrates a valid, and an invalid decomposition with respect to a decomposition constraint "*an attribute and its data type must not be separated*".

Figure 9 illustrates a decomposition constraint. It displays on the left the instance diagram of a model element `Client` with an attribute `name` of type `String`. On the right, the figure displays the instance diagram of two decompositions of `Client` (*A, B*). We represent the different decomposed models (m_1, m_2) by using dashed, and solid boxes. The dotted box enclosing the attribute named `"name"`, and the data type named `"String"` illustrate the dependence between these two elements. *A* is a valid decomposition of *Client*; it respects the decomposition constraints and does not separated an attribute from its type. On the other hand, *B* is invalid because it separates the attribute `name` from its type `String`, hence generates a malformed

[1] Such constraints can be expressed in languages such as OCL of Alloy [15], and later translated into hard-code.

model. It violates the decomposition constraints and makes the left decomposition instance dependent on the right decomposition instance.

Notice that decomposition operators are derived from the inverse of the different primitive operations constituting the composition operator they reverse. That is, for each primitive *c* in the composition operator, there is at least a primitive *d* in the decomposition operator that can revert (undo) the effect of *c*. Typically, composition operators such as [16, 3, 4, 7, 8], are described in terms of two operations: (1) *match* and (2) *merge*. Defining primitives to inverse these operations could be done in a very similar way to those we previously described in this section. In general, symmetric composition operators with well-defined composition steps such as match and merge are likely to be inversed.

5. Oracle: analyzing the composition result

The decomposition of models eases the analysis of the model produced by a *CE*. Since a DE decomposes pairs of models from a single one (*m*), and a *CE* composes these pairs generating the originally decomposed model (*m*), the input model of the decomposition and the output model of the composition are expected to be the same. The relation between these models enables us to define an *oracle* function that gives true when the produced model is correct, and false otherwise. To define this function we use a reference model (decomposition's input) that is compared with the decomposition result. In this way, when all the model elements of the decomposed model exists or have an equivalent in the composed model, and the relation between these model elements are equivalent on both models, the composition is correct.

Definition 3. For a composition operator $\oplus$ and the decomposition operator $\ominus$ reverting $\oplus$, the oracle function is defined as a mapping between a pair of models and a Boolean value.

$$Oracle: M \times M \rightarrow Boolean \quad (4)$$

$$Oracle(m,m') = \begin{cases} True\ if\ m\ equivalent\ to\ m' \\ False\ otherwise \end{cases} \quad (5)$$

In the particular cases evaluated by this paper, the equivalence of the composed models is replaced by equality with the original model.

6. The *CompTest* framework: Model composition testing framework

We have implemented a generic framework, *CompTest*[2], which supports our testing proposal. It is a collection of tools (some of them third party) automating the testing of *CE*s. That is, this framework can be used to test any *CE* implementing symmetric composition. After briefly introducing the tools in *CompTest*, we detail the activities involved in testing a *CE* using *CompTest*.

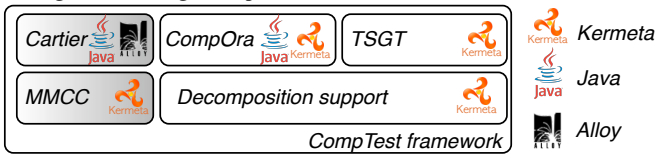

Fig. 10. Elements composing the *CompTest* framework.

Figure 10 presents the different tools in the *CompTest* framework. The icon at right of each box indicates the technology used to build each tool (*Java* , *Kermeta*, and *Alloy*). Grey elements correspond to third party tools.

- *MMCC*[3] [17] is a third party tool that checks whether a set of models cover a meta-model. MMCC generates a set of constraints that should be satisfied by a set of models in order to insure coverage of the input domain.
- *Cartier* [14] is a third party tool that synthesizes models using a constraint solver approach. Actually it constraints derived from *MMCC* to synthesize models.
- *Decomposition support* is the part of the generic framework that provides support for implementing a model decomposition engine. It provides an extensible interface to implement decomposition primitives.
- *CompOra* is a tool that realizes the model analysis (*Oracle*) we proposed in section 5. It is a generic tool that can analyze the differences between two models regardless their meta-model.
- *Test script generator tool (TSGT)* is a tool that generates the necessary scripts needed to execute the entire test using the *decomposed* models.

Figure 11 presents the *CompTest*'s testing activities involved in testing a *CE*. Everything starts with a meta-model for which a *CE* is implemented (1). An engineer derives a decomposition operator from the CE specifications (composition operator). By using this operator he or she extends the Decomposition support and implements a *DE* (2). Once the *DE* is implemented, *CompTest* runs a sanity check that ensures that the elements of the model decomposed by DE (*m*) are at least in one of the decomposed models (m_1, m_2). Once *CE* and *DE* are available, the framework machinery starts working. *MMCC* generates a set of constraints based on a meta-model coverage criterion (*A*). A tester selects such criteria from the catalog offered by *MMCC*. Consequently, *Cartier* takes these constraints (previously generated) and

[2] Available at http://freddy.cellcore.org/research/CompTest
[3] Available at http:// www.irisa.fr/triskell/softwares/MMCC/

generates a set of models satisfying the constraints (*B*). Later on, the *DE* decomposes each model and generates numerous pairs of models (*C*). Once the generation is completed, the *script generator* generates all the scripts for testing the *CEUT* (*D*). These scripts invoke (1) the *CEUT* to compose each pair of models, and (2) *CompOra* to analyze the resulting model. Finally, *CompTest* executes the tests (scripts) and generates a report based on the *CompOra* verdict (*F*). Notice that *CompTest* offers an automated tool chaining, that is, once the composition, decomposition engines, and coverage criteria are available, the framework automatically performs steps from *A* to *F*.

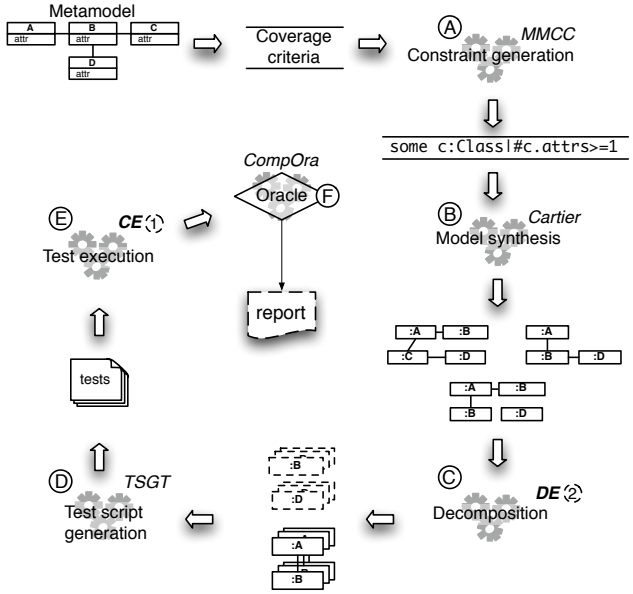

Fig. 11. Activities embodied by CompTest for testing a *CE*. Notice that the arrows denote the order of occurrence for such activities, and (1), and (2) denote non-automated activities.

7. Experiments

7.1 Mutation analysis on Kompose

We have extended the CompTest *CompTest* framework, to test the *CE Kompose* [6], a generic implementation of the composition operator proposed by France et al [3]. It is generic in the sense that it provides a base that needs to be specialized (extended) to a particular meta-model. Such specialization consists in identifying each *mergeable* element in the meta-model and defining their signature. We have *specialized kompose* for the *RCD* meta-model presented in section 2, and using the facilities provided by the *CompTest* we have implemented a *DE* as specified in section 4.

The results obtained after performing the activities illustrated in figure are summarized in the following. The *MMCC* tool generated 37 constraints that models must fulfill in order to fully cover the meta-model using the *all partitions* criterion.

Using these constraints, *Cartier* generated 37 model instances (one per constraint). These instances were feed into the *DE*, which generated in average 10 model pairs per instance (335 model pairs in total). The test script generator generated 335 test scripts, one per model pair. Each test script invoked the *kompose* with a model pair, and using composition result, invoked *CompOra*. The whole testing process takes about 20 minutes in a Macbook pro with a 2.4 Ghz processor and 2GB of main memory.

In order to check the CompTest's effectiveness for detecting faults, we intentionally introduced faults in the *Kompose* engine. This is known as mutation analysis in classical testing literature [18]. In the following we summarize these faults:

F 1. Delete instructions from the code realizing the *match* operator.
F 2. Delete instructions from the code realizing the *clone* primitive.
F 3. Delete instructions from the code realizing the *unify* primitive.
F 4. Delete completely the code realizing the *match* operator.
F 5. Delete completely the code realizing the *clone* primitive.
F 6. Delete completely the code realizing the *unify* primitive.

Notice that when the fault *F4* is introduced, fault *F1* cannot be introduced. Analogously, when either fault *F5* or *F6* is introduced, either fault *F2* or *F3* cannot be introduced.

In order to execute the tests we use the data synthesized in the previous section. Notice that the faults we seeded into the *CE* do not modify or introduce errors in neither the decomposition operator nor the RCD meta-model.

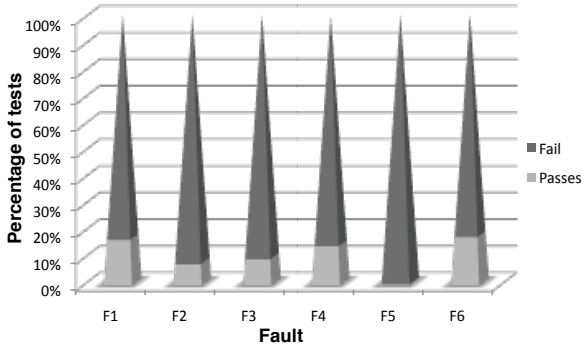

Fig. 12. Results obtained after testing the faulted version of *Kompose*.

Figure 12 summarizes the results we obtained after executing the tests. We have performed 335 tests for each fault, which corresponds to composing 335 pairs of models and check the result of their composition. *Most of these tests were able to detect that something was wrong with the CE. More important, the high mutation score is consequence of the large amount and variety of the testing data, which covers all the meta-model constructions and the possible ways to compose them. This ensures that all the composition paths where exercised by at least one model pair.*

For the fault *F1* we have deleted 3 lines of the code realizing the match operator. The experiments show that, 17% of the tests pass. These tests were passing because a portion of the decomposed models was exercising the non-deleted lines of code. That leaves 73% of tests exercised the deleted lines of code. Notice that deleting three lines of code was equivalent to deleting the complete operator (*F4*). The models that were successfully composed are those having only elements to clone. Remark that when we completely removed the code realizing the clone operator (*F5*), all the tests failed. That means that all the instances we composed were exercising the code realizing the clone operator. When fault *F2* was introduced, only 7% of the tests passed. This is because we deleted only 2 lines of the code realizing the clone operator, and 7% of the tests exercised only the remaining lines of code, whereas 93% of them exercised the deleted code. The results for faults *F3* and *F6* follow the same logic.

These results sketch the usefulness of *CompTest*, when the testers can provide a faultless *DE* and a well specified meta-model.

7.2 What if *DE* is faulty?

The previous experience assumed that testers are able to obtain, or develop a faultless decomposition engine. This not always the case, sometimes coding the *DE* could be hard and prone to errors. However, even in such situations having a *DE* serves as a specification to which a *CE* can be faced. Moreover, ensuring that a *DE* engine produces a coherent result is likely to be less difficult than doing the same with a *CE*.

A straightforward way to test whether the decomposition engine produced a coherent result is checking whether the element and relations in the original model *m* exist in either of the decomposed models m_1, m_2, or both. This ensures that the decomposition is actually conserving the model structure and relations. Unsurprisingly, this does not ensure that the *DE* is faultless, but possible *DE* faults will lay problems with the coverage of the composition paths. That is, faults in the decomposition engine will lead to loss in the meta-model coverage with respect to the possible ways it can be composed, i.e. there will be *possible compositions* that will never be tested.

Faults in a decomposition engine that produces coherent decomposed models will result in loss of resolution. Moreover, this kind of faults will be unlikely to be detected through the previously presented testing approach.

7.3 Threats to validity

Experiments suffer from threats to validity. We have made attempts to reduce these, however, we outline the major threats here. With respect to external validity, we acknowledge that we have studied only a single composition engine with a single and reduced meta-model. We have no evidence of how difficult could be to implement a decomposition engine for other symmetric composition approaches / engines. With respect to internal validity, we have tried to ensure that the fault we introduced into the *CE* reflect the faults that developers could commit. We have also made an effort to check that both, composition and decomposition engines were faultless before doing the experiments, and validated that the oracle verdict was the right one. As far

as construction validity, we acknowledge that there could be faults in the third party tools that *CompTest* use (including the *kermeta* platform), and that these faults could leak into the experiments.

8. Related work

In the last years, researchers have paid little attention to the validation of model composition engines, and specially on achieving this through testing. Some researchers have studied the validation through comparing the consistency of different views before and/or after composing them [19-21]. In [22] the authors propose to compose symmetrically (structural models) and asymmetrically (behavioral models) models representing different views of the system. Checking the consistency of both compositions, symmetric and asymmetric, then validates the composition result.

As we mentioned before, model composition can be seen as a model transformation where the input and the output meta-model is identical, besides it receives two input models to produce one. Several authors have addressed model transformation testing. Some authors have addresses the synthesis of testing data for model transformation [12-14]. These approaches can be used to synthesize input data to test model composition engines, however, they cannot ensure composability. In this paper we have introduced an approach that can synthesize composable data and can reuse these approaches. Other authors have addressed the analysis of expected transformation (oracle). Some of them have proposed to check the composed model against a set of constraints [23, 24]. Automatically synthesizing such constraints is challenging and an open issue. Other approaches use a reference model or search for a reference pattern in the resulting model [25, 26]. In [27] Mottu et al. study the issues associated with different mechanisms used to evaluate models. Out of these mechanisms, we use the equivalent to an inverse transformation (decomposition operator) that generates a reference model. Comparing the resulting and the reference model is challenging because of their complex structure. Several authors have addressed this problem and proposed mechanisms for comparing models [28-31]. Our approach makes use of these mechanisms in order to know whether two models differ.

Another important issue related to model transformation validation is model validation. Several works address this validation by supporting model testing. In [32] the authors define a set of coverage criteria for UML design models. Using these criteria, the authors propose to automatically generate input data for testing models [33]. The models for testing comprise class diagrams with OCL pre / post conditions for methods, and activity diagrams to specify the behavior of each method [34].

9. Conclusions

Model composition helps designer to better manage complexity in MDE. Model composition engines (*CE*) are complex programs, which receive complex inputs to produce complex output. These engines need to be reliable in order to have MDE deliver its promises.

In this paper we have studied the validation of composition engines through testing, and particularly we have addressed the automatic generation of testing data. We proposed through the introduction of a *decomposition operator* and its associated *decomposition engine (DE)*, to *synthesize* instances conforming to a meta-model. Such *synthesis* is not as traditionally understood. Instead of generating instances for a meta-model we transform (decompose) existing ones into pairs of them. This enables the reuse of existing model synthesis techniques and coverage criteria. Decomposing models into pairs guarantees (1) the composability of the *synthesized* pair, and (2) that their composition will produce a coherent result.

Decomposing models enables the synthesis of data suitable to be composed, and hence suitable to test de composition engine. In this paper we have proposed a complete suite that supports the testing of *CE* through a *DE*.

It is possible for *DE* to be faulty or hard to implement. However, even in that case, we think that it is likely to be less difficult to find faults in it, or at least ensure that it produces a coherent result. Yet, if there are faults in the *DE*, these faults may lay in loss of coverage of the possible composition paths, reducing the possible conclusion from the tests, but should not dim the test results.

References

[1] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," in *FOSE '07: 2007 Future of Software Engineering*, Washington, DC, USA, 2007, pp. 37--54.

[2] P. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J.-M. Jézéquel, "Introducing variability into Aspect-Oriented Modeling approaches," in *In Proceedings of ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 07)*, Nashville, TN, USA, 2007.

[3] R. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh, "Providing Support for Model Composition in Metamodels," in *edoc* Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 253-265.

[4] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave, "Matching and Merging of Statecharts Specifications," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, Washington, DC, USA, 2007, pp. 54--64.

[5] M. D. D. Fabro and P. Valduriez, "Semi-automatic model integration using matching transformations and weaving models," in *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, New York, NY, USA, 2007, pp. 963--970.

[6] F. Fleurey, R. France, B. Baudry, and S. Ghosh, "Kompose : A generic model composition tool," 2008.

[7] S. a. Clarke and E. Baniassad, *Aspect-Oriented Analysis and Design*: Addison-Wesley Professional, 2005.

[8] Y. R. Reddy, S. Ghosh, R. B. France, G. Straw, J. M. Bieman, N. McEachen, E. Song, and G. Georg, "Directives for Composing Aspect-Oriented Design Class Models," in *Transactions on Aspect-Oriented Software Development I*, 2006, pp. 75--105.

[9] J. Whittle, A. Moreira, J. Araújo, P. Jayaraman, A. Elkhodary, and R. Rabbi, "An Expressive Aspect Composition Language for UML State Diagrams," in *Model Driven Engineering Languages and Systems*, 2007, pp. 514--528.

[10] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel, "Weaving Executability into Object-Oriented Meta-languages," in *Model Driven Engineering Languages and Systems*, 2005, pp. 264-278.

[11] C. Herrmann, H. Krahn, B. Rumpe, M. Schindler, and S. V\"olkel, "An Algebraic View on the Semantics of Model Composition," in *Model Driven Architecture- Foundations and Applications*, 2007, pp. 99--113.

[12] K. Ehrig, J. Küster, and G. Taentzer, "Generating instance models from meta models," in *Software and Systems Modeling*.

[13] 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 '06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, Washington, DC, USA, 2006, pp. 85--94.

[14] S. Sen, B. Baudry, and J.-M. Mottu, "On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing," *icst,* vol. 0, pp. 328-337, 2008.

[15] J. Daniel, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.,* vol. 11, pp. 256-290, 2002.

[16] A. B. Phillip, Y. H. Alon, and A. P. Rachel, "A vision for management of complex models," *SIGMOD Rec.,* vol. 29, pp. 55-63, 2000.

[17] F. Fleurey, B. Baudry, P. A. Muller, and Y. Traon, "Qualifying input test data for model transformations," in *Software and Systems Modeling*, 2008.

[18] A. J. Offutt, "A practical system for mutation testing: help for the common programmer," in *Test Conference, 1994. Proceedings., International*, 1994, pp. 824-830.

[19] M. Sabetzadeh and S. Easterbrook, "An Algebraic Framework for Merging Incomplete and Inconsistent Views," in *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering*, Washington, DC, USA, 2005, pp. 306--318.

[20] M. Sabetzadeh and S. Easterbrook, "Analysis of inconsistency in graph-based viewpoints: a category-theoretical approach," in *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, 2003, pp. 12-21.

[21] M. Sabetzadeh and S. Easterbrook, "View merging in the presence of incompleteness and inconsistency," *Requir. Eng.,* vol. 11, pp. 174--193, 2006.

[22] O. Barais, J. Klein, B. Baudry, A. Jackson, and S. Clarke, "Composing Multi-view Aspect Models," in *Composition-Based Software Systems, 2008. ICCBSS 2008. Seventh International Conference on*, 2008, pp. 43-52.

[23] J. M. Küster, "Definition and Validation of Model Transformations," *Software and Systems Modeling,* vol. 5, pp. 233-259, 2006.

[24] M. Lamari, "Towards an Automated Test Generation for the Verification of Model Transformations," in *Symposium on Applied Computing SAC'07*, Seoul, Korea, 2007.

[25] K. Duddy, A. Gerber, M. Lawley, K. Raymond, and J. Steel, "Model Transformation: A declarative, reusable patterns approach," in *EDOC'03 (Entreprise Distributed Object Computing Conference)*, Brisbane, Australia, 2003, pp. 174 - 185.

[26] Y. Lin, J. Zhang, and J. Gray, "A Testing Framework for Model Transformations," in *Model-driven Software Development - Research and Practice in Software Engineering*: Springer, 2005.

[27] J.-M. Mottu, B. Baudry, and Y. Le Traon, "Model transformation testing : oracle issue," in *MoDeVVa workshop colocated with ICST08*, Lillehammer, Norway, 2008.

[28] D. Lopes, S. Hammoudi, J. De Souza, and A. Bontempo, "Metamodel Matching: Experiments and Comparison," in *International Conference on Software Engineering Advances (ICSEA'06)*, Los Alamitos, USA, 2006.

[29] D. S. Kolovos, R. F. Paige, and F. a. C. Polack, "Model Comparison: A Foundation for Model Composition and Model Transformation Testing," in *workshop GaMMa'06*, Shangaï, China, 2006.

[30] Y. Lin, J. Zhang, and J. Gray, "Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development," in *OOPSLA'04*, 2004.

[31] Z. Xing and E. Stroulia, "UMLDiff: an algorithm for object-oriented design differencing," in *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, New York, NY, USA, 2005, pp. 54--65.

[32] A. Andrews, R. France, S. Ghosh, and G. Craig, "Test adequacy criteria for UML design models," *Software Testing, Verification and Reliability,* vol. 13, pp. 95 -127, 2003.

[33] T. T. Dinh-Trong, S. Ghosh, and R. B. France, "A Systematic Approach to Generate Inputs to Test UML Design Models," in *ISSRE '06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, Washington, DC, USA, 2006, pp. 95--104.

[34] T. T. Dinh-Trong, S. Ghosh, R. B. France, and A. A. Andrews, "A Systematic Approach to Testing UML Design Models," in *4th International Workshop on Critical Systems Development Using Modeling Languages (CSDUML)*, Fredrikstad, Norway, 2005.