Noname manuscript No.
(will be inserted by the editor)

Testing Operational Transformations in Model-Driven Engineering

Andrea Ciancone · Antonio Filieri · Raffaela Mirandola .

Received: date / Accepted: date

Abstract Model-driven development is gaining importance in software engineering practice. This increasing usage asks for a new generation of testing tools to verify correctness and suitability of model transformations. This paper presents a novel approach to unit testing QVT Operational (QVTO) transformations, which overcomes limitations of currently available tools. Our proposal, called MANTra (Model trANsformation Testing), allows software developers to design test cases directly within the QVTO language and verify them without moving from the transformation environment. MANTra is also available as an Eclipse feature that can be easily integrated into established development practice.

1 Introduction

In the last years, Model Driven Engineering (MDE) is emerging as a paradigm for the design of complex software systems, which encourages the realization of new software systems by the use of a model-centric approach [10]. Models may be used at different abstraction levels. At the requirements stage, for example, they can help to identify possible missing parts or conflicts. At design time, they may be used to analyze the effects and trade-offs of different architectural choices before starting an implementation. They may also be used at run time to support continuous monitoring of compliance of the running system with respect to the desired model.

The most striking aspect of models in software engineering, as opposed to models in other traditional engineering fields, is that models and final artifacts are both *software*. This is why *model transformations* may be conceived to support the transition from model to system. Such transformations may be more or less automatic, but in any case they may be stated as precisely defined software manipulation actions, rather than informal design steps.

However, model transformations, as any other pieces of software, can be inconsistent and produce undesirable results in certain conditions. Consequently, it is useful to check their quality using verification techniques [13]. Models transformation testing is one of the adopted technique. The model transformation testing faces all the challenges of the code testing [14]. It is based on the definition of test cases requiring input models and an oracle able to identify the correctness of the output models.

Currently, the black-box testing of model transformation is the most diffused testing approach. The adoption of black-box testing requires to focus on the suitability of generated test models and to design an oracle for the output models, while the model transformation content itself is not inspected (see Section 2 for details).

This approach presents several disadvantages since it requires to manage complete models whose generation involves an high human effort. Besides, the size of the output models has a strong impact on the test case execution time and on the effort for the oracle definition. Finally, the definition of the test cases is usually done with different languages with respect to the ones used for the specification of model transformations. In this way, a gap between the model transformation de-

Andrea Ciancone, Antonio Filieri, and Raffaela Mirandola
Politecnico di Milano
Dipartimento di Elettronica e Informazione
P.zza Leonardo da Vinci 32, Milan, Italy
E-mail: andrea.ciancone@mail.polimi.it
E-mail: {filieri,mirandola}@elet.polimi.it

velopment and the model transformation testing is created with respect to the required skills and tools.

It is our claim that MDE tools and techniques can be used also to perform model transformation testing, allowing both a better exploitation of the potential of the MDE paradigm itself and a more efficient verification process.

To this end we propose in this paper a new testing approach and tool [5], called *MANTra: Model trANsformation Testing*, able to deal with model transformation written in QVT Operational (QVTO), which is a language belonging to the Query/View/Transformation (QVT) standard [12] created by the Object Management Group (OMG) in 2008. MANTra allows software developers to design test cases directly within the QVTO language and verify them without moving from the transformation environment. In this way a white-box testing becomes feasible and unit testing appears as a convenient testing approach.

In order to be more easily adopted in current MDE practice, MANTra is also distributed as an automatically installable feature for the Eclipse IDE [9], which straightforwardly integrates with the modeling tools provided by the suite. Nevertheless, the proposed testing paradigm is applicable with all the transformation engines compliant with the specification of QVTO 2.1 (or superior) [12].

This paper is organized as follows. Section 2 describes the genesis of this work within the context of the European project Q-ImPrESS [7] and reviews related works. Section 3 presents the MANTra approach for the QVTO-based transformations testing and then Section 4 its specific use within the Eclipse IDE. Section 5 shows through a simple working example the practical aspects of unit testing using the MANTra tool. In Section 6 more details about the internal behavior of MANTra are given. Section 7 shortly describes the validation we have performed within the Q-ImPrESS project and proposes some best practices to make the testing process more effective. Finally, section 8 concludes the paper with the description of the limitations of the proposed approach and the planned future work.

2 Motivation

The approach presented in this paper stems from our experience in the European project Q-ImPrESS [7]. Q-ImPrESS aims at building a framework for service orientation of critical systems. Such a framework is deeply founded on model transformations, which allow the automatic filling of the gap between design and analysis models. Hence model transformations, being in the loop of critical software development, require strong validation and verification, to different extents.

Thanks to the adoption of QVTO, we have been able to exploit syntax check and completion feature of the Eclipse Modeling Framework (EMF) [1]. Then we faced three different problems:

1. Input domain coverage;
2. Transformation verification;
3. Mathematical validation of analysis results.

Point 1 was faced by developing an ad-hoc generator of input instances that can be guided by the user. In particular, industrial partners in the Q-ImPrESS consortium defined the typologies of input models they regarded as most critical or significant. This practice, as well as random coverage of the meta-model, is guiding the ongoing testing of the framework.

Point 3's success was somehow settled on transformation correctness, even though it provides no guarantees about the absence of pathological input cases for the transformation itself.

What was really missing is point 2. QVT, indeed, is a quite recent standard and it lacks practices and success stories both in programming and testing the transformations. Before defining our test strategy, we explored a number of more or less mature verification approaches and tools to figure out a proper testing plan for our needs. In the remainder of this section, we shortly summarize the works related to this topic and outline the main limitations of the philosophy underlying these methods.

2.1 Related Work

Fleurey et al. [8] proposed a methodology to automatically generate input test cases for a transformation by looking at its code. Generation is mainly driven by elements' domain boundaries and meta-model constraints. The proposed methodology is implemented for the Tefkat framework [19].

More recently on the same line, Sen et al. [18] presented Cartier, a tool for the automatic test-case generation in MDE. Cartier combines knowledge coming from different sources (meta-models, meta-model constraints, and transformation pre-conditions) in a common model. It adds a set of constraints that produced test cases must satisfy. It also exploits Alloy [6] to produce instance models compliant with specifications and constraints.

Lin et al. [14] focus on models comparison as a means for transformation testing. They discuss which properties have to be compared, how to represent models at different level of abstraction, which are the most

effective comparison algorithms and how to explicitly represent model differences. Then they propose a framework [15] that allows the testing of generic model transformations providing a transformation executor and a comparator for produced and expected models.

Mcgill et al. define Jemtte [17], a product minded to be an extension of the JUnit testing framework including model transformation. It facilitates the definition of simple Java test cases for models represented in XML by exploiting assertions over XPath expressions. Each assertion is able to check the presence of a certain element, to compare a returned value with the expected one and to determine the equivalence between sets of elements.

2.2 Black-box Testing in MDE

To the best of our knowledge, most of the sketched approaches try to obtain a wide applicability by considering transformations as black-boxes. Such a strategy was useful in past years because of the absence of a standard for model transformation. Instead of producing approaches tailored on one or another transformation language and engine, most of the testing framework opted to blindly considering only input and output models.

In this way they introduced some intrinsic limitations:

- They have to manage large input and output models. This derives from the need to test the entire transformation as a whole. For large meta-models and complex transformation this operation could be expensive in terms of both test design and execution time. Large model definition is also a typical error-prone procedure when conducted manually, even though it can be supported to a limited extent by automatic tools [2].
- They might require the adoption of special purpose languages, requiring a transformation developer to spend time in acquiring those skills.
- They typically require ad-hoc environments to execute tests. This increases the configuration effort and could lead to costs and portability issues because the testing process does not depend on model transformation only but also on other external resources. These resource are not required by the model transformation but are required by the testing process. This dependency could require configuration effort, lead to costs, and limit the portability to other OSs or software.

Other more general limitations come from the oracle definition step and from the execution context profiling. The oracle issue [2] is a hard and wide problem in the field of testing. In the subfield of testing model transformations, the most common approach is to establish models comparison methodologies or to define assertion-based oracles. While model comparison, in general, is still an open issue, assertion testing is already quite effective. This is due to many different reasons, such as the better adaptability of assertions to describe complex patterns (instead of simple values).

Context profiling is another general issue in testing. It is always hard to figure out how the execution context (related models, needed utilities, representations, and so on) will really look like. Unfortunately, automatic tools cannot provide useful insights or nice solutions to this issue.

Concluding, black-box testing in general has to cope with the above outlined challenges. Each of the tools on the market provides its own way to overcome certain limitations, usually introducing some other drawback.

With respect to existing works, the MANTra approach pays the cost of being focused on QVTO transformations, but benefits from the availability of "white-box insights" in order to be able to:

a. exploit model transformation code to improve testing effectiveness,
b. execute and test parts of the transformation in isolation, reducing testing's complexity,
c. remove dependencies of tests on external tool and resource (even input files).

3 The Approach

The main focus of this paper is on the unit testing of QVTO scripts. The need for unit testing in the Q-ImPrESS project stems from both technical and organization reasons. First of all Q-ImPrESS transformations are often complex and it would be hard and unproductive to test them as a whole. Then, within the project, our aim is to support a test-as-it-goes paradigm, in order to make easier both to define test cases and to provide intermediate results to our partners and co-developers. Finally, we aspire to have a test suite independent from any specific engine and IDE, such that anyone can run his tests everywhere he can run its own QVTO transformations. This goal comes from the fact that different development teams work in different development environments, some of which are proprietary.

Our idea is to define a methodology to test QVTO transformations using only QVTO itself. Hence we need to define a way to design input models, define ora-

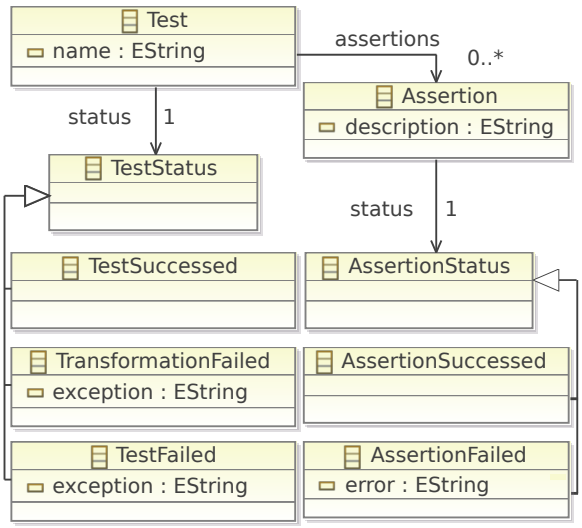

Fig. 1 Testing report meta-model defined in ecore diagram representation

cles, invoke transformations and inspect test results, all within QVTO.

By exploiting QVTO expressiveness, the definition of the input models is not an hard issue, as it will be shown later in Section 5. Our oracle is based on assertion checking on the output models. Assertions have to be defined for each test exploiting QVTO capabilities, enhanced by an ad-hoc defined small library that makes the assertion mechanism more usable. To invoke transformations under test from inside the test script we exploited the reuse by composition and by extension features of QVTO [12]. The last ones allow also overriding original mapping operations to perform specific experiments, such as what-if analysis of possible alternative improvements of the operation without modifying the original code.

Reuse features allow the unit testing in isolation of small portion of the transformation script. Each test case defines a partial input model composed by a few elements, and then applies a partial transformation on it. Finally, the transformation outcome, which contains the output model elements, is inspected using assertions.

Testing reports are provided in two ways. The basic one makes use of QVTO's logging features and provides textual outputs easy to be captured on the fly by a monitor process. The second one is instead more structured and provides a report model that can be easily used to produce human-readable reports or models in any useful form for further automatic evaluation of the test outcomes. The *Test* meta-model is shown in Figure 1. Every test is reported with an exit status, among *success*, *transformation failure* or *test failure*, and a set of assertions, as defined by the user, each with a *success* or *failed* evaluation.

Summarizing, MANTra is a white-box QVTO unit testing tool that allows writing fine-grained test cases in QVTO for QVTO. Its main strengths are:

1. Neither extra skills nor extra tools are required for a QVTO developer to test his own products. This reduces both training time and tools expenses. MANTra does not need any other external software than developer's preferred QVTO IDE.
2. Reduced context reproduction burden, because a developer is able to test the real transformation in its real running environment, but focusing on verification of only some of its parts per time.
3. MANTra test cases can be reproduced on every QVTO-compliant engine due to the fact that the definition methodology is fully compliant with the official QVTO standard [12]. The test cases are completely self-contained and it is not necessary to provide input models files of any format.
4. Test case complexity is completely up to the developer, which is no longer forced to build complete, often large, input models to test even a single mapping operation. He can focus on small partial input models in order to test specific parts of the transformation.
5. MANTra can take the most important aspects of any QVTO IDE, like syntax check and completion, in order to make the developer more comfortable in writing his test cases, but also to reduce the possibility of coding errors typical of hybrid approaches like OCL assertions embedded in Java code.
6. MANTra is designed to make unit testing of QVTO easier and faster, and it can be adopted in test-driven development processes of QVTO transformations.

4 Integration with the Eclipse IDE

Besides the manual inclusion of the resources presented in Section 3, the MANTra testing framework is available as an Eclipse feature that comes with two plugins embedding the result meta-model as well as the testing libraries. Such a feature can be installed in a fully automatic way as an additional component for the model transformation environment. More specifically, MANTra is integrable by means of an Eclipse update site, with the following advantages:

i. Simple one-click installation for Eclipse 3.6+ with Modeling tools. The additional plugin comes with automatic satisfaction of dependencies, quickly en-

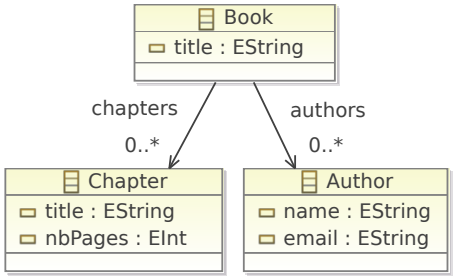

Fig. 3 Books meta-model defined in ecore diagram representation

abling any established modeling environment to support its application.

ii. Required testing resources are accessible from all the model transformation projects, without the need of error-prone replications or manual inclusions.

iii. The update site allows for continuous updates that are automatically reflected in all the transformation projects.

On the other hand, this is an additional feature only for Eclipse, while the basic MANTra framework can be applied on any QVTO engine compliant with specification OMG 1.0 [12].
Eclipse updatesite panel during MANTra installation is show in Figure 2. The installation instructions and requirements are detailed in the documentation available at MANtra site [5].

In the next section the MANTra tool is described, with the support of a working example, in order to show how it works in practice.

5 Working Example

In the following, we provide an overview on how the MANTra tool acts. The use of MANTra requires only a QVTO engine compliant with specifications in OMG 1.0 [12]. The most general way to use it consists in the inclusions of the MANTra components (matamodel and libraries) in the QVTO transformation project. This suffices for the testing procedure, with no need for environment changes. In the following we assume that MANTra's resources are located in directory *qvtoTesting*, reachable from the project main folder. At the end of the section we will show how the workflow can be eased for the Eclipse IDE.

Let us introduce a simple application case. The goal is to define a transformation from models of the Books metamodel to models of the Publications metamodel.

Books metamodel, which is shown in Figure 3, is composed by three elements. A *Book* element has a *title* attribute, a set of *Chapter* and a set of *Author* elements, both of them can be empty. A *Chapter* has a *title* attribute, and a *nbPages* attribute representing its number of pages. Finally, an *Author* has *name* and *email* attributes.

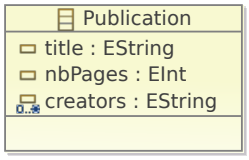

Fig. 4 Publications meta-model defined in ecore diagram representation

Publication meta-model's (described in Figure 4) instances are more general and simpler then Book's ones. Each of them describes a publication by means of a single element called *Publication*. It has three attributes: *title*, *nbPages*, representing the number of pages, and *creators*, representing the set of its authors.

The mapping from Books meta-model to Publications meta-model is informally defined by the following rules *r1* through *r4*:

r1 A *Book* element is mapped onto a *Publication* element.

r2 A *Publication*'s *title* is obtained from the *title* of the correspondent *Book*.

r3 A *Publication*'s *creators* is obtained from the composition of *authors*'s *name* and *email* attributes of the corresponding *Book*.

r4 A *Publication*'s *nbPages* is obtained from the sum of the *chapters*'s *nbPages* of the corresponding *Book*. A zero value for *nbPages* can occur in two cases: the *Book* does not have chapters or every *Book Chapter* does not have *nbPages* attribute.

A QVTO script that performs the described model transformation is described below.

```
modeltype BOOK uses 'file://book.ecore';
modeltype PUB uses 'file://pub.ecore';

transformation Book2Publication
    (in book:BOOK, out pub:PUB);
main() {
    book.objects()[Book]
        ->map toPublication();
}
mapping Book::toPublication () :
        Publication {
    title := self.title;
    creators := self.authors->toCreator();
    nbPages := 0;
    if(self.chapters
```

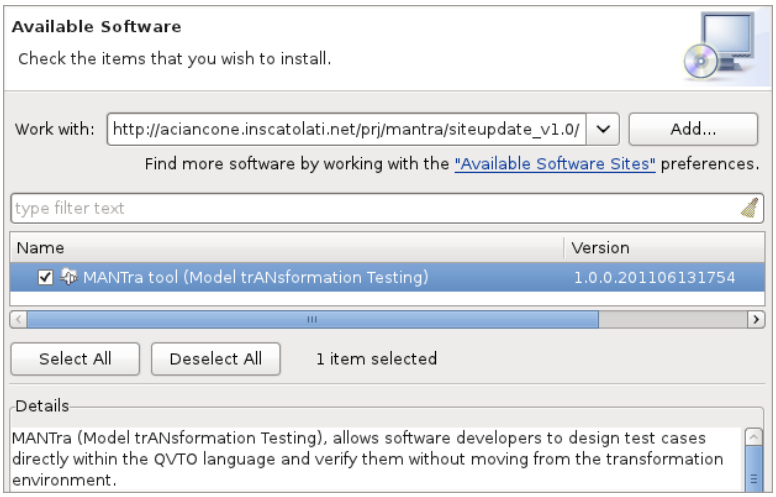

Fig. 2 Eclipse updatesite panel showing the MANTra framework feature during installation

```
            ->forAll(nbPages > 0)) then {
        nbPages :=
            self.chapters.nbPages->sum();
    }endif;
}
query Author::toCreator() : String {
    return self.name +'<'+ self.email +'>';
}
```

Testing cases import *Test* library and the model transformation to be tested. Then, the involved metamodels, the test case signature and the real testing code are defined.

The first example test checks that the query *Author::toCreator()* effectively returns the expected *creator* string from a *Author* element.

```
import qvtoTesting.Test;
import Book2Publication;
modeltype BOOK uses 'file://book.ecore';
modeltype PUB uses 'file://pub.ecore';

transformation B2P_author_test()
    extends Test, Book2Publication;
main() {
    var creator := object Author {
        name := 'name surname';
        email := 'email@domain.tld';
    }.toCreator();
    assertEquals('creator id',
        'name surname <email@domain.tld>',
        creator);
}
```

The test signature declares the test case as a transformation extending *Test* and *Book2Publication* (which is the model transformation under test). *Test* provides some basic functionalities needed for the testing. Extending *Book2Publication* makes the test case aware of all the contents of the transformation.

Test's body builds up an *Author* element to be passed to *Author::toCreator()*. The outcoming value is compared with the expected string by means of the *assertEquals()* function, provided by *Test* library as part of a large set of assertion constructs (e.g. *assertTrue()*).

The second example test case checks the *Book::toPublication* mapping function.

```
import qvtoTesting.Test;
import Book2Publication;
modeltype BOOK uses 'file://book.ecore';
modeltype PUB uses 'file://pub.ecore';

transformation B2P_book_test()
    extends Test, Book2Publication;
main() {
    var pub := object Book {
        title := 'bookTitle';
        chapters += object Chapter {
            nbPages := 12 };
        chapters += object Chapter {
            nbPages := 30 };
        authors += object Author {};
    }.map toPublication();
    assertEquals('pub name',
        'bookTitle', pub.title);
    assertEquals('pub nbPage',
        42,    pub.nbPages);
    assertEquals('pub creator',
        Bag{'stub'}, pub.creators);
}
-- stub function
query Author::toCreator() : String {
```

```
    return 'stub';
}
```

The structure is essentially the same as in the first example, except for two points. A *Book* element is passed to a mapping function rather than a query, and a stub query function is defined in order to isolate the function to test. The procedure can be generalized introducing as many stubs for helper or mapping operations as desired.

The third example tests three boundary values for the mapping of the *nbPages* attribute.

```
import qvtoTesting.Test;
import Book2Publication;
modeltype BOOK uses 'file://book.ecore';
modeltype PUB uses 'file://pub.ecore';

transformation B2P_bookNbPagesErr_test()
    extends Test, Book2Publication;
main() {
    assertEquals('no chapters',
        0, object Book {
        }.map toPublication().nbPages);
    assertEquals('emtpy chapters',
        0, object Book {
            chapters += object Chapter {};
            chapters += object Chapter {};
        }.map toPublication().nbPages);
    assertEquals('an empty chapter',
        0, object Book {
            chapters += object Chapter {};
            chapters += object Chapter {
                nbPages := 1
            };
        }.map toPublication().nbPages);
}
```

This example shows how to test several input cases in a single test case. In general, test scripts can be as flexible as any transformation under testing, by exploiting the whole QVTO language expressiveness.

Test execution is done via the *tests suite* library, that can be accessed by importing *TestsSuite*.

The following code allows the execution of the three tests previously described.

```
import qvtoTesting.TestsSuite;
modeltype testReport
    uses 'http://QvtoTests/1.0';

import B2P_book_test;
import B2P_author_test;
import B2P_bookNbPagesErr_test;
```

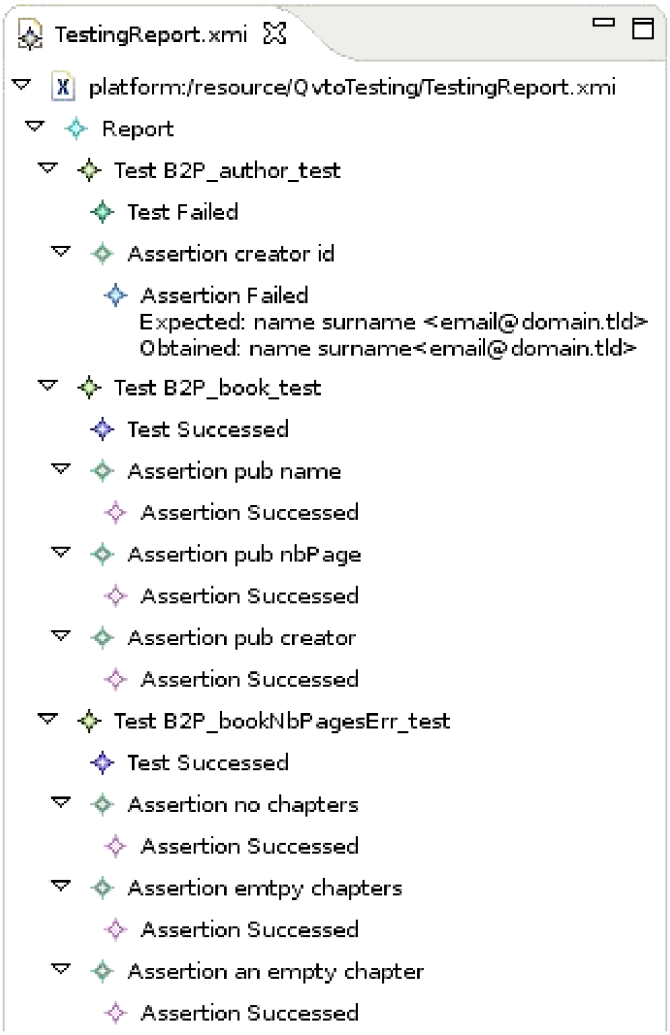

Fig. 5 Testing report model of the example

```
transformation
    B2P_tests(out report: testReport);
main() {
    addTest(new B2P_book_test());
    addTest(new B2P_author_test());
    addTest(new B2P_bookNbPagesErr_test());
    runTests();
}
```

The signature of the previous listing declares to provide an output model. This last contains the report of the test suite run. Test cases can be added to the test suite by means of the *addTest()* function. Then, the *runTests()* function launches test cases one-by-one.

Test cases are executed in isolation to avoid undesired interferences such as test suite execution interruption whenever a test case generate an exception. MANTra assertion prints messages on the standard output. Thus, during the execution of a test suite the developer can see status update messages on the console

as soon as they get available. As an example, running the test suite presented in this section it comes out the following console output:

```
Start tests...
* run test B2P_book_test() @4963ea
* run test B2P_author_test() @174f876
  assertEquals[FAILED] creator id
  Expected: name surname <email@domain.tld>
  Obtained: name surname<email@domain.tld>
* run test B2P_bookNbPagesErr_test() @52f00
End tests.
```

When everything goes right, a notification message informs that the test has been executed. In case an error occurs, the system provides detailed information on it. In the example, the error message explains that the *assertEquals* with message *"creator id"* has failed because expected and obtained values are not equal.

The output of test execution is an instance of the result meta-model (Figure 1). The output model of the test described in this section is shown in Figure 5. Besides the structured result model, as already said, MANTra provides also a textual output, which can be used directly by developers for a quick look at testing results. This output can also be nested in an higher level tool to provide structured report, to support automatic testing tools or to just get stored in the company's knowledge base.

Eclipse. The integration with the Eclipse IDE, achieved through the update site installation, provides a simplified way to use MANTra. First of all the scope of its metamodel and libraries is automatically extended to all QVTO projects with no need of further inclusions. The second main advantage of Eclipse integration comes from its embedded continuous update mechanism, which ensures the availability of the last version of MANTra as soon as it is released.

Test code keeps the same shape as shown before but for a slight variation to the import statements as follows:

```
import transforms.Test;
import transforms.TestsSuite;
```

6 MANTra Internals

In this section deeper notions concerning MANTra's internal behavior are provided. Namely, in the first paragraph we will show how the proposed tool can exploit the two phases of the QVTO transformation process [12] to make more expressive tests. In the second paragraph we present the assertion API provided by MANTra.

2-Phase Testing. As a reader with some experience in QVTO transformations has probably noticed, the approach so far presented does not work properly in case the *late* operator is used. The definition of test cases entirely verifiable in the first phase of a QVTO process (as those in Section 5) is recommended whenever possible because it makes tests more compact and readable. Such a kind of tests can be referred to as *single step*. Nonetheless, single step tests are not expressive enough to cover all the possible situations. For example let us assume to add to both the metamodels the attribute *citedBy*, which is a self-relation *1 to many* toward elements of the same type, i.e., each book is cited by other books, and each publication is cited by other publications. An extension of the mapping *toPublication* for such an attribute is the following:

```
mapping Book::toPublication () :
        Publication {
    title := self.title;
    creators := self.authors->toCreator();
\textbf{citedBy := self.citedBy.late resolve();}
    nbPages := 0;
    if(self.chapters
            ->forAll(nbPages > 0)) then {
        nbPages :=
            self.chapters.nbPages->sum();
    }endif;
}
```

In such a case there is no way to define a single step test that is complete for the mapping *toPublication*, because the binding of the *Publication::citedBy* collection element will be available only at the second stage of the transformation, because during the first phase, the transformation is performed and all the statements are executed except for the assignments that involve *late resolution* [12]. Only in the second phase, the transformation is finalized by performing all the late resolution assignments.

To deal with late resolution, MANTra provides the possibility to define testing statements to be executed within the scope of the first or the second phase only. An example of such a feature is shown in the following listing:

```
import qvtoTesting.Test;
import Book2Publication;
modeltype BOOK uses 'file://book.ecore';
modeltype PUB uses 'file://pub.ecore';

transformation B2P_citation_test()
   extends Test, Book2Publication;

property citedPubs: Bag(Publication) = Bag{};
```

```
property citingPub: Publication = null;

main() {
   if(isFirstStep()) then {
       var a := object Book {};
       var b := object Book {};
       citingPub := object Book {
               citedBy := Bag{a, b};
       }.toPublication();
       citedPubs += a.toPublication();
       citedPubs += b.toPublication();
   }else {
       assertEquals('cited by',
       citedPubs,
       citingPub.citedBy->asBag());
   }endif;
}
```

The developer can postpone the assertion check after the first phase by means of check of the boolean function *isFirstStep*. The execution of the second phase of the transformation workflow is notified by a message on the console, as illustrated below:

```
Start tests...
* run test B2P_author_test() @174f876
* second step B2P_author_test() @174f876
End tests.
```

The ability to test both steps of the QVTO workflow makes MANTra able to perform a thorough test of all the QVTO constructs.

Assertions. As already said, MANTra tests are assertion-based. When an assertion fails, the test procedure is interrupted and the specified assertion message is returned. MANTra library provides an expressive API, enabling the developer to write short, readable, and maintainable test cases using a set of convenient assertions. A summary of the API is provided in Table 1. They can be used in any point of test transformations that extends Test library.

Table 1 MANTra assertions API

assertTrue	Asserts that a boolean condition is true.
assertFalse	Asserts that a boolean condition is false.
assertEquals	Asserts that two elements are equal. In case of collections, they have to contain the same elements.
assertSame	Assert that two elements refer to the same element, i.e. it is an identity check.
assertNotSame	The negation of *assertSame*.
assertNull	Assert that an element is null.
assertNotNull	The negation of *assertNull*.
fail	Make the test to fail, providing the specified message.

7 Validation

MANTra is being successfully adopted to test the QVTO transformations for the reliability prediction tool in Q-ImPrESS [7]. Its adoption is changing the development paradigm for those QVTO transformations towards a Test Driven Development (TDD) approach, change also motivated by the increasing complexity of the transformations and the need for dependable code.

In Section 7.1 a short outline of the Q-ImPrESS project is given, recalling some quantitative measures of the transformations tested via MANTra. In Section 7.2 we will report the results of the evaluation and in Section 7.3 we describe the lessons learned during the application of MANTra and some best practices generated by the experience acquired during the work in Q-ImPrESS.

7.1 Q-ImPrESS

Q-ImPrESS is a recently concluded three years project funded by the EU under the 7th Framework Program. It designed and implemented a methodology and a development framework to bring the service-orientation paradigm to advanced industrial domains, such as industrial production control, telecommunication and critical enterprise applications, by guaranteeing end-to-end quality of service. The Q-ImPrESS methodology enables software architects to predict the impact of architectural design decisions on performance, reliability, and maintainability of a service-oriented software system. An overall high level vision of the Q-ImPrESS tool landscape and methodology is illustrated in Figure 7.

The development process in Q-ImPrESS strongly adopts the Model Driven Development paradigm. Goal of the integrated development environment (the Eclipse-based Q-ImPrESS IDE) is to assist software engineers during the development and evolution both in existing as well as in newly started software development projects.

The central element of the Q-ImPrESS-based development process is the Service Architecture Meta-Model (SAMM) [7]), which is a new abstract design model of a software system describing the structure of the system in terms of components, operations, deployment infras-

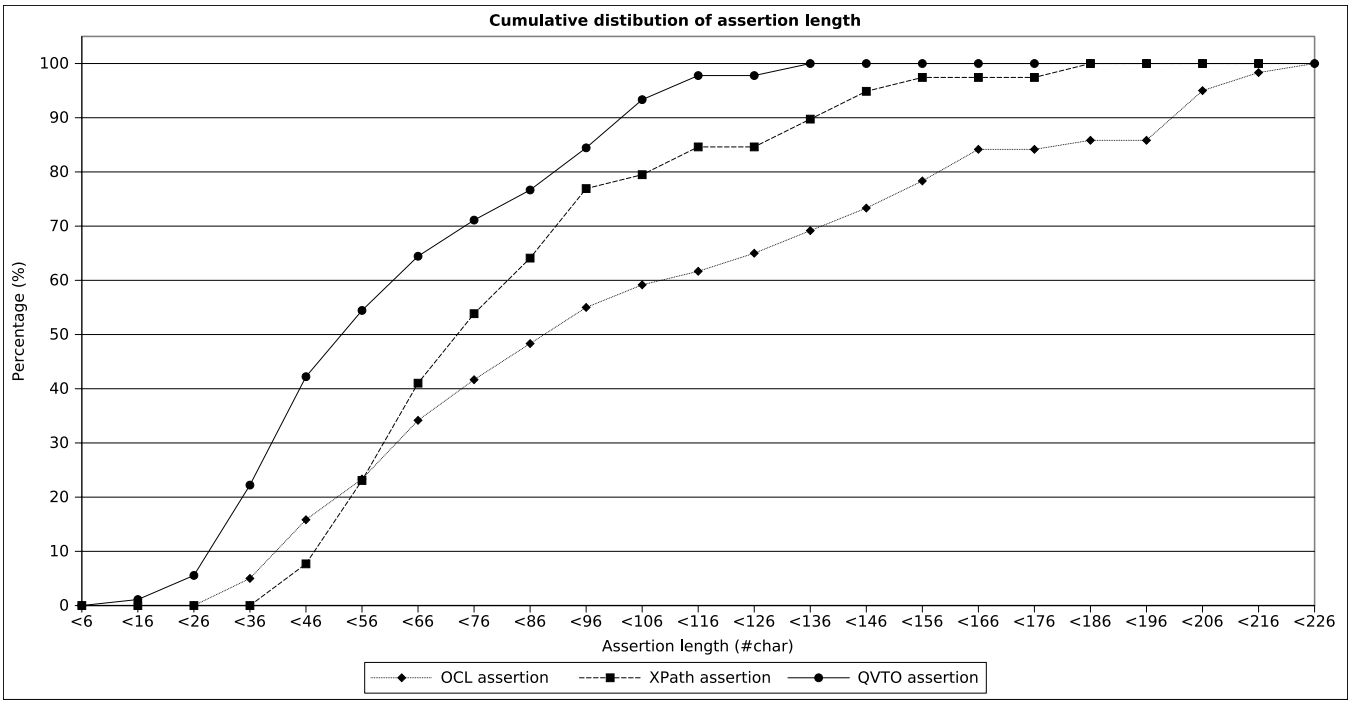

Fig. 6 Cumulative distribution of assertion length

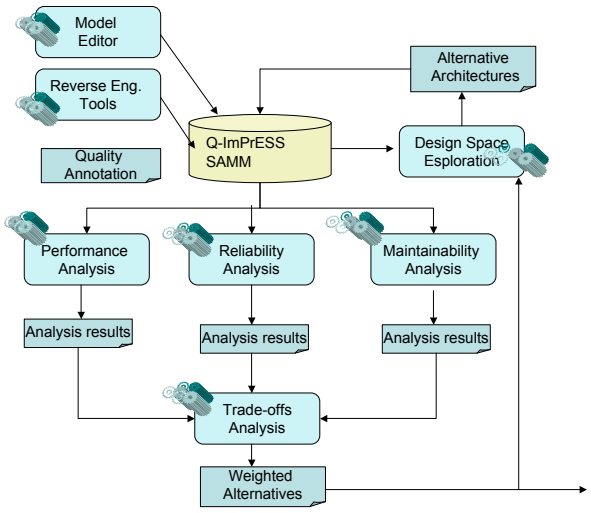

Fig. 7 Q-ImPrESS methodology and tool architecture overview

tructure and usage profiling. SAMM is a metamodel defined via the Eclipse Modeling Framework [1].

A SAMM instance is the core of the process and it provides elements to design the system under development. It can be defined manually by the architect or can be automatically inferred by existing source code by reverse engineering tools. Details can be found at [7]. Besides the design model, Q-ImPrESS projects comes with QoS metamodels including information concerning reliability, performance and maintainability of the system to be, as well as a stochastic characterization of its behavior.

A SAMM instance and the corresponding QoS annotations have to be transformed into analytical models in order to be processable for non-functional properties verification by means of convenient tools, e.g. model-checkers or simulators.

The verification tools applied in Q-ImPrESS are based on the modeling languages KLAPER [11] and the Palladio Component Model (PCM) suite [3]. Both of them are defined by the corresponding EMF metamodels. Hence the core of the verification process lays in the transformation from the design models (SAMM+QoS) to the analysis models (KLAPER+PCM). KLAPER models will then be transformed again in Discrete Time Markov Chain models in order to be analyzed for reliability purposes.

SAMM to PCM and SAMM to KLAPER are the two largest transformations developed in QVTO, and are available under the EPL license terms from the Q-ImPrESS website [7]. Q-ImPrESS also supports automatically generating architectural alternatives, and their evaluation with the tool PerOpteryx [16]. As a results, the architect gets a set of Pareto-optimal alternatives among which he can select the one that satisfies her quality requirements.

The transformation extensively tested via MANTra is the one from SAMM to KLAPER. It operates on five input models, and produces one output model. The

transformation script is composed of 65 mapping functions and 17 queries, plus 15 conditional mappings including *when* filters, disjuncts mapping and *if* statements.

Previously, we developed an automated testing tool [4] based on JUnit to test SAMM to PCM. Test cases were written in OCL and the evaluation framework was implemented in Java. The SAMM to PCM transformation has the same five input models as the SAMM to KLAPER one, and a comparable complexity. In particular the transformation is composed by 76 mapping functions and 27 queries, plus 32 conditional mappings, including, even in this case, *when* filters, disjuncts mapping and *if* statements. In the following we present the results obtained with MANTra and we compare them with some other existing tools.

7.2 Evaluation result

MANTra has been effective in testing Q-ImPrESS model transformations. Our experience did not reveal any unbearable limitation for large scale applicability. Even if the tool is quite easy to use, at the very beginning of the testing procedure it took some time to figure out how to identify significant test cases, due to the lack of established practices in the area.

Besides qualitatively evaluate the MANTra approach easier to be used when compared with our previous experience in Q-ImPrESS, we try here to propose a quantitative evaluation of its effectiveness. We compared MANTra with Jemtte [17], whose test cases are written as a composition of XPath and Java constructs, and our previous test tool jOMoT (jUnit + OCL Model Testing framework) [4], developed in Java, integrated in JUnit, with assertion in OCL.

The evaluation aims at comparing testing tools with respect to (1) complexity of assertions and (2) test execution performance. The first metric is used as an index of how much burden is required to manually write test cases. We have chosen assertion length to compare assertion complexity. Effective burden depends heavily on a number of un-quantifiable parameters such as developer experience, availability of special purpose constructs and so on. Concerning performance, we measured test execution time.

All tests have been applied in analogous external conditions: similar transformation complexity, same developer, same training time for the developer and same execution environment.

Concerning the complexity of assertion definition, Figure 6 shows the cumulative distribution of assertion length for the three tools. 90% of MANTra assertions

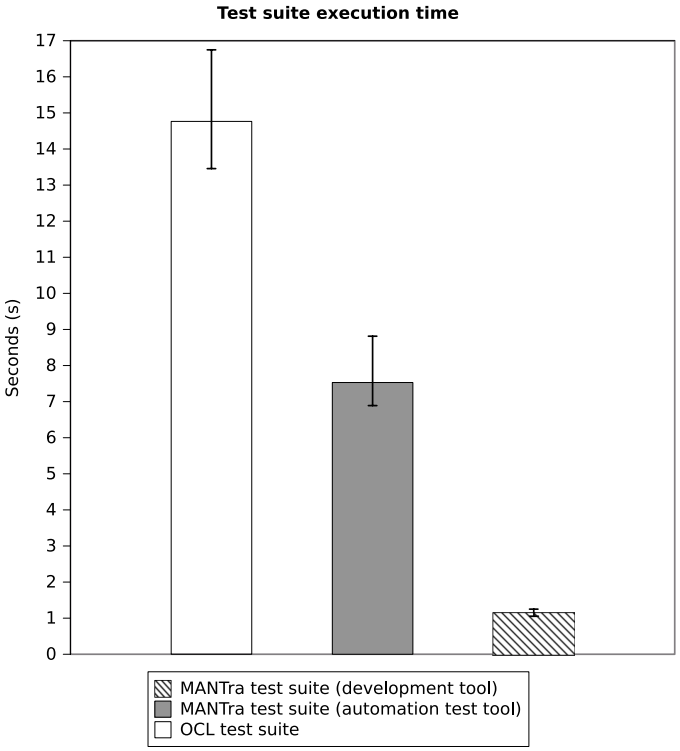

Fig. 8 Tests suite execution time

are less than 100 chars long and none of them exceed 140 chars. While XPath places 90% of assertion under 140 chars and OCL up to 200 chars.

The reduced complexity in writing assertion in QVTO is due to the higher level of abstraction of this language, explicitly designed to deal with model transformations and hence equipped with compact and direct constructs to access transformation's and models' elements.

A numerical summary of assertion lengths measurement is provided in Table 2. Data are related to different real-life projects, that is why the number of assertion checked is different. Standard deviation can be reduced adopting a larger set of samples. It could be interesting as future work to identify different benchmark for model transformation testing in order to automatically produce large sample sets.

Concerning performance, Figure 8 reports the average execution time of the entire test suite for MANTra and jOMoT. We decided to focus the evaluation on jOMoT because it has a more similar workflow, if compared with MANTra, than Jemtte, and jOMoT provides exactly the same features as MANTra. The test suite is composed, globally, by 100 assertions and each time value reported is the average over 30 runs. Results are quite stable, with a low standard deviation. Notice that MANTra can be executed both from the QVTO IDE and as a JUnit instance. These two variants are presented separately and show the efficiency of MANTra

Table 2 Assertions length information

Assertion type	Average (chars)	Standard Deviation (chars)	N. Assertion
QVTO	53	28	90
XPath	73	33	39
OCL	88	55	120

Table 3 Test Suite Execution time

Test Suite	Average (sec)	Standard Deviation (sec)
OCL test suite	14.764	0.809
MANTra (automation test tool)	7.529	0.222
MANTra (development tool)	1.148	0.002

in the two most common testing scenario, i.e. as a support of TDD directly in the development IDE or in an automated testing procedure.

The development environment was standard Eclipse Galileo installation with QVTO 2.0.1 engine, equipped also with JUnit 3.8.2 which was instead used for the automated test. The machine on which all the tests have been run is a 2 GHz Pentium (R) M with 1 Gb RAM.

The speed of MANTra test suite execution within automatic testing tool depends by the fact that it is a single transformation containing all the test cases that has to be compiled after each change. Eclipse provides a really fast access to the QVTO engine and an on-going compilation of the transformation which turn out as an increased execution speed, as evidenced in the graph. The jOMoT suite is an extension of JUnit designed for TDD of QVTO transformations against assertion-based test cases. Both jOMoT and MANTra in automated mode were executed using JUnit bundled with Eclipse.

Table 3 reports basic statistics on the performance data-set. MANTra is faster than jOMoT. Even more, it performs particularly well inside the development tool, proving one more time to be an effective support tool for TDD.

Concluding, MANTra has been proved to be more effective than competitors in supporting test-cases definition, thanks to its high abstraction, and to perform very fast, thanks to the availability of always more efficient QVTO engines.

7.3 Lessons Learned and Best Practices

MANTra was extensively used within the Q-ImPrESS project. Experience led to the elicitation of some advisable coding styles to be applied to make more testable code. As it can be easily expected, most of the common practices for software testing keep being valid also for QVTO, with convenient adaptations. Tough, same peculiar features of QVTO may induce some specific types of errors. The next three practices have been selected as the most commonly appeared in our experience and somehow the most sensitive.

One is better than two. Two phases tests are expressive and captivating because of their ability to ensure fine grained control on the execution test cases. Nevertheless, they are much harder to be read and maintained because of the increased complexity. As an example of this bad practice, let use show how the *B2P_author_test* shown in Section 5 would appear if implemented as a two phase test:

```
import qvtoTesting.Test;
import Book2Publication;
modeltype BOOK uses 'file://book.ecore';
modeltype PUB uses 'file://pub.ecore';

transformation B2P_author_test()
    extends Test, Book2Publication;
property creator:String = null;
main() {
    if(isFirstStep()) then {
        creator := object Author {
            name := 'name surname';
            email := 'email@domain.tld';
        }.toCreator();
    }else {
```

```
        assertEquals('creator id',
        'name surname <email@domain.tld>',
        creator);
    }endif;
}
```

Conclusions come easily to their end.

Transformation parameterization. Analogously to any other languages, QVTO scripts does not retrieve all the needed information from source model. Indeed many aspects of the transformation are tuned by assigning specific values to internal variables, thought as parameters. Common cases of parameterization of the transformation are, for example, default values of attributes or format templates. **non capisco:**It is quick and handy to hard-code parameters directly in a mapping function. On the other hand, this makes test definition and maintenance much more costly because any further change to the hard-coded parameters has to be reflected in test assertions. As for other programming languages, it is a good practice to parametrize both the transformation and the mapping operations, by making parameter dependency visible from their interfaces.

QVTO provides an effective feature to parameterize transformations and, possibly, mappings. Such feature comes from the use of *configuration properties* [12]. Their use is possible for both transformations and MANTra testing suites and make the developer aware about all the dependencies of both of them. Configuration properties' values have to be set before launching transformations or tests, hence the developer is necessarily aware of their values, avoiding dangling references or hard to diagnose behaviors due to some hard-coded "default" value. Also, QVTO configuration properties make both transformations and test suites more flexible and reusable, thus speeding up development process as well as refactoring.

En excerpt of code showing the use of configuration properties follows:

```
configuration property
   defaultEntityName : String;

...

query  getDefaultEntityName ( ) : String {
  var internalDefinition := aName;
  return
     if(defaultEntityName.oclIsUndefined())
     then
        ...
     else
        defaultEntityName
     endif;
}
```

The use of configuration properties is made particularly handy in QVTO IDEs like Eclipse.

"Decoupled inheritance". In MDE the decoupling between model structure and mapping operations, defined respectively in the metamodels and the transformation scripts, is a feature. Nevertheless, this could make more probable the arising of the problem of code replication inside mapping functions. Indeed, especially in presence of inheritance in the metamodel, assignment to a target element's attributes often appear, with the same purpose, in more than one mapping operations. To ground this claim, consider for example a variation to the metamodel Books (Fig. 3) consisting in the addition of the two elements *ScientificBook* and *Periodic* extending the element *Book*, and adding each one its specific attributes. From our experience, a common practice is to define a mapping for *ScientificBook* and another for *Periodic*, as in the following:

```
mapping ScientificBook::toPublication () :
        Publication {...}

mapping Periodic::toPublication () :
        Publication {...}
```

The problem in the example is that mapping operations do not reflect the elements' inheritance defined in the metamodel. Thus each mapping has to replicate the assignment of the inherited attributes *title*, *creator* and *nbPages*. Such organization of the mapping operations leads also to the duplication of tests. The situation suffers exactly the same problems test duplication does in every programming language [20]. The coding of QVTO scripts, especially with Eclipse, is particularly prone to duplications because the editor does not distinguish between the attributes properly belonging to an element and the ones it inherited.

A better mappings' structure for the previous case could be the following:

```
mapping Book::toPublication () :
    Publication {--asBefore--}

mapping ScientificBook::toPublication () :
    Publication inherits Book::toPublication
        {...}

mapping Periodic::toPublication () :
    Publication inherits Book::toPublication
        {...}
```

where the sub-mappings *ScientificBook* and *Periodic* manipulate specific attributes only. The derived good

practice that can be recommended is that mapping structure should reflect as much as possible elements hierarchies present in the metamodel. In a straightforward way, tests will comply too and will allow for correct isolation of tested statements as well as for better inspection and maintainability of both transformations and MANTra tests.

8 Conclusions

This paper presented MANTra, a new testing approach for QVTO-based model transformations. The idea underlying MANTra definition is to exploit the potentialities of MDE techniques and tools to deal with the complexity of model transformations testing.

To bring this approach to fruition we developed also a tool using which we analyzed transformations coming from the Q-ImPrESS project. The integration with the Eclipse IDE is also a quick and effective way to bring the application of MANTra into industrial contexts. The practical aspects of the testing tool have been presented in Section 5 together with a small example, which shows how it is possible to write test cases, create a test suite and launch tests to verify the model transformation correctness.

MANTra is designed to make unit testing of QVTO easier and faster. It exploits QVTO features allowing the definition of input models. It also requires that the transformation under test avoids direct access to input and output model elements, that is, reading an element value has to be accomplished through QVTO queries. This is not a limitation at all, but a practice to be kept in mind during transformation coding.

Nevertheless it still requires a broad usability validation. We are planning a training and coding session with several model transformations developers, in order to get a non-biased feedback on how easy to use and appealing the tool is.

MANTra can be extended along several directions. We plan to define how QVTO IDEs could be enhanced to better support MANTra testing development, for example, making it able to provide the execution trace of failed assertions in form of text messages or graphs. This feature is going to be realized by exploiting QVTO trace files, in order to keep everything QVTO compliant. Furthermore, we plan to include our approach within an higher level development tool for automatic generation of QVTO transformations. By integrating MANTra in the automatic code generation chain, it is possible to produce, besides transformations, also proper unit-test suites. All this by exploiting the same structure of the established code generator: MANTra tests are completely defined in QVTO language as well.

Finally, we are also planning to assess the effectiveness of the proposed approach through a comprehensive set of experiments in a real testbed and to perform an extensive study of the test cases development to extend the definition of "best practices" that are specific for this testing approach and tool.

Acknowledgements Work partially supported by the European Union projects Q-ImPrESS (FP7 215013) and SM-Scom (IDEAS 227077).

References

1. EMF: Eclipse Modeling Framework (2nd Edition). Addison-Wesley Longman, Amsterdam (2009)
2. Baudry, B., Dinh-Trong, T., Mottu, J., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Le Traon, Y.: Model transformation testing challenges. In: Proceedings of IMDT workshop in conjunction with ECMDA'06
3. Becker, S., Koziolek, H., Reussner, R.: The palladio component model for model-driven performance prediction. Journal of Systems and Software **82**(1), 3 – 22 (2009). Special Issue: Software Performance - Modeling and Analysis
4. Ciancone, A.: jomot framework
5. Ciancone, A., Filieri, A.: Mantra website. http://aciancone.inscatolati.net/prj/MANTra/
6. Community, A.: The alloy analyzer. http://alloy.mit.edu/
7. Consortium, Q.I.: Project website. http://www.q-impress.eu
8. Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: testing model transformations. In: First International Workshop on Model Design and Validation, pp. 29–40 (2004)
9. Foundation, T.E.: Project website. http://www.eclipse.org
10. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Proc. Future of Software Engineering FOSE '07, pp. 37–54 (2007). DOI 10.1109/FOSE.2007.14
11. Grassi, V., Mirandola, R., Randazzo, E., Sabetta, A.: Klaper: An intermediate language for model-driven predictive analysis of performance and reliability. The Common Component Modeling Example pp. 327–356 (2007)
12. Group, O.M.: Qvt 1.0 specification. http://www.omg.org/spec/QVT/1.0/
13. Harrold, M.J.: Testing: A roadmap. In: In The Future of Software Engineering, pp. 61–72. ACM Press (2000)
14. Lin, Y., Zhang, J., Gray, J.: Model comparison: A key challenge for transformation testing and version control in model driven software development. In: Control in Model Driven Software Development. OOPSLA/GPCE: Best Practices for Model-Driven Software Development, pp. 219–236. Springer (2004)
15. Lin, Y., Zhang, J., Gray, J.: A testing framework for model transformations. In: in Model-Driven Software Development - Research and Practice in Software Engineering, pp. 219–236. Springer (2005)

16. Martens, A., Koziolek, H., Becker, S., Reussner, R.: Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In: Proc. 1st Joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW'10), pp. 105–116. ACM, New York, NY, USA (2010). DOI http://doi.acm.org/10.1145/1712605.1712624
17. Mcgill, M.J., Cheng, B.H.C.: Test-driven development of a model transformation with jemtte (2007)
18. Sen, S., Baudry, B., Mottu, J.M.: On combining multi-formalism knowledge to select models for model transformation testing. In: Software Testing, Verification, and Validation, 1st International Conference on, pp. 328–337 (2008)
19. Wang, J., Kim, S.K., Carrington, D.: Automatic generation of test models for model transformations. In: ASWEC '08: Proceedings of the 19th Australian Conference on Software Engineering, pp. 432–440. IEEE Computer Society, Washington, DC, USA (2008)
20. Zhang, M., Hall, T., Baddoo, N.: Code bad smells: a review of current knowledge. Journal of Software Maintenance and Evolution: Research and Practice **23**(3), 179–202 (2011)