MANTra: Towards Model Transformation Testing

Andrea Ciancone Antonio Filieri Raffaela Mirandola
Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza L. Da Vinci 32, 20133 Milan - Italy
andrea.ciancone@mail.polimi.it, {filieri,mirandola}@elet.polimi.it

***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.**

I. 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 [1]. 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 [2]. Models transformation testing is one of the adopted technique. The model transformation testing faces all the challenges of the code testing [3]. 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 II 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 development 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 [4], 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 [5] 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.

This paper is organized as follows. Section II describes the genesis of this work within the context of the European project Q-ImPrESS [6] and reviews related works. Section III presents the MANTra approach for the QVTO-based transformations testing, while Section IV shows through a simple case study the practical aspects of unit testing using the MANTra tool. Section V shortly describes the validation we have performed within the Q-ImPrESS project. Finally, section VI concludes the paper with the description of the limitations of the proposed approach and the planned future work.

II. Motivation

The approach presented in this paper stems from our experience in the European project Q-ImPrESS [6]. Q-ImPrESS aims at building a framework for service orientation of critical systems. Such a framework is deeply founded on model transformations, which allow to automatically fill 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 were able to exploit syntax check and completion feature of the Eclipse Modeling Framework (EMF) [7]. 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 is 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. First of all, QVT is a quite recent standard. It lacks the development of practice both in programming and testing the transformations.

A. Related Work

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.

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 was implemented for the Tefkat framework [9].

More recently on the same line, Sen et al. [10] presented Cartier, a tool for 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 and then it exploits Alloy [11] to produce instance models compliant with specifications and constraints.

Lin et al. [3] 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 proposed a framework [12] that allows the testing of generic model transformations providing a transformation executor and a comparator for produced and expected models.

Mcgill et al. introduced Jemtte [13], 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. Any of these assertions 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.

B. 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 is true because of 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 [14].
- 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 needed by the testing process, and for such a dependency could require configuration effort, could lead to costs, and could limit the portability to other OSs or softwares.

Other more general limitations come from oracle definition and execution context profiling. The oracle issue [14] 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, among which the better adaptability of assertions to describe complex patterns (instead of simple values) and, then, problem of confluence of transformation results (different, correct, outputs could come out from different transformation runs). 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, there is not so much to expect from automatic tools on 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 paying on some other fronts.

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:

- exploit model transformation code to improve testing

effectiveness,
- execute and test parts of the transformation in isolation, reducing testing's complexity,
- remove dependencies of tests on external tool and resource (even input files).

III. 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 oracles, invoke transformations and inspect test results, all within QVTO.

By exploiting QVTO expressiveness, definition of the input models is not an hard issue, as it will be shown later in Section IV. Our oracle is based on assertion check 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 [5]. 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

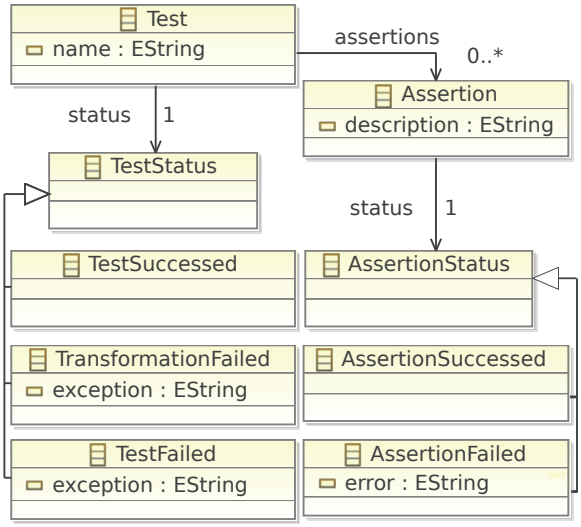

Fig. 1. Testing report meta-model defined in ecore diagram representation

QVTO. Its main strengths are:

- 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.
- 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.
- 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 [5]. The test cases are completely self-contained and it is not necessary to provide input models files of any format.
- 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.
- 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.
- 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.

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.

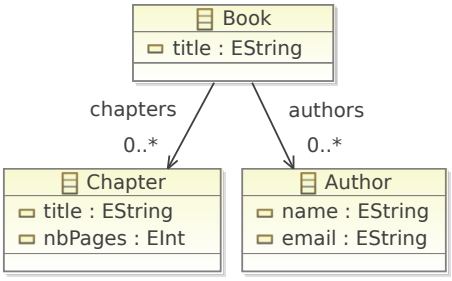

Fig. 2. Books meta-model defined in ecore diagram representation

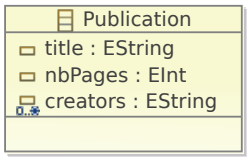

Fig. 3. Publications meta-model defined in ecore diagram representation

IV. CASE STUDY

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

Books metamodel, which is shown in Figure 2, 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.

Publication meta-model's (Figure 3) 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:

- A *Book* element is mapped onto a *Publication* element.
- A *Publication*'s *title* is obtained from the *title* of the correspondent *Book*.
- A *Publication*'s *creators* is obtained from the composition of *authors*'s *name* and *email* attributes of the corresponding *Book*.
- 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
            ->forAll(nbPages > 0)) then {
        nbPages :=
            self.chapters.nbPages->sum();
    }endif;
}
query Author::toCreator() : String {
    return self.name +'<'+ self.email +'>';
}
```

Testing cases import *qvtoTesting.Test* library and the model transformation to be tested. Then, all the involved meta-models, 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 *qvtoTesting.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 *qvtoTesting.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;

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 a test is getting executed. In case an error occurs, the

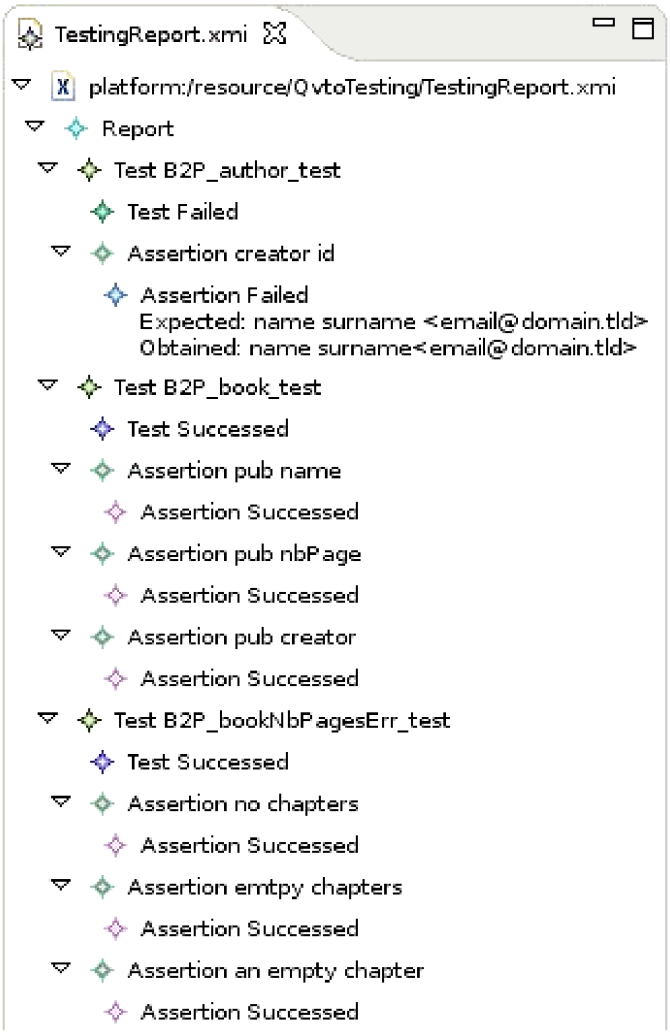

Fig. 4. Testing report model of the case study

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 metamodel (Figure 1). The output model of the test described in this section is shown in Figure 4. 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.

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. Every QVTO transformation is executed in two steps. In the first step, the transformation is performed and all the statements are executed except for the assignments that involve *late resolution* [5]. In the second step, the transformation is finalized by performing all the late resolution assignments.

All the tests so far described are executed in the first step of the QVTO workflow, thus assignments involving late resolution are not yet performed. MANTra provides the possibility to execute testing in two steps as described in the following example:

```
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;
}
```

The developer can postpone assertion check of the second step by means of the function *isFirstStep*. The execution of the second step of the transformation workflow is notified with a message on the console.

```
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 test every construct of QVTO.

V. Validation

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

In subsection V-A a short outline of the Q-ImPrESS project is given, recalling some quantitative measure of the transformations tested via MANTra. In the following subsection V-B we will report the results of the evaluation.

A. Q-ImPrESS

Q-ImPrESS is a three years project funded by the EU 7th Framework Program. It aims to come out with a methodology and a development framework to bring the service-orientation

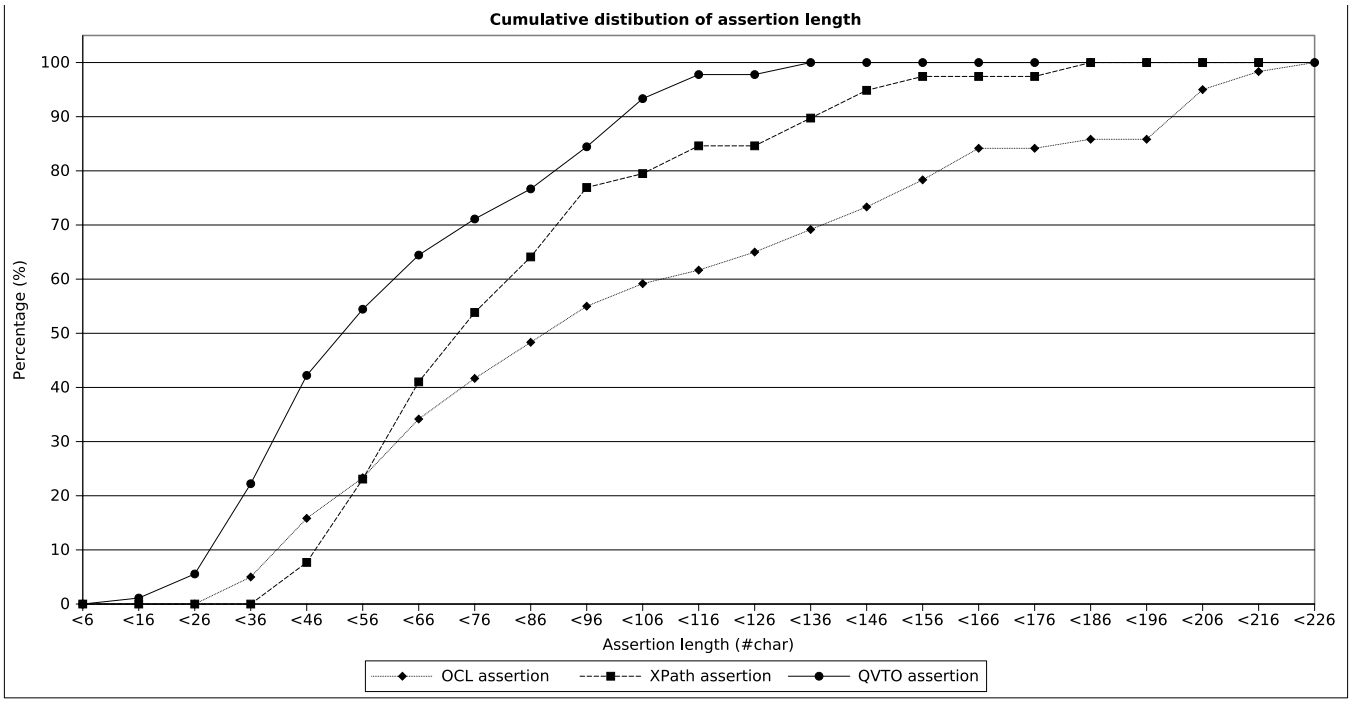

Fig. 5. Cumulative distribution of assertion length

paradigm to advanced industrial domains, such as industrial production control, telecommunication and critical enterprise applications, by guaranteeing end-to-end quality of service. Q-ImPrESS provides support for performance, reliability and maintainability analysis of large software projects, besides design-time decision support tools for goal-driven selection of different possible development alternatives.
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) [6], which is a new abstract design model of a software system describing the structure of the system in terms of components, operations, deployment infrastructure and usage profiling.

Starting from a SAMM instance, separate prediction models can be automatically derived to predict the performance, reliability and maintainability. Specifically, reliability and performance estimation tools exploit model transformation in order to obtain analysis models from design ones. In particular performance is evaluated by means of the Palladio Component Model (PCM) suite [15], while reliability is estimated via a KLAPER-based [16] tool. 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 [6].

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 [17] 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.

B. 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 [13], whose test cases are written as a composition of XPath and Java constructs, and our previous test tool jOMoT (jUnit + OCL Model Testing framework) [17], 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

TABLE I
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 II
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

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 5 shows the cumulative distribution of assertion length for the three tools. 90% of MANTra assertions 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 I. 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 6 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 in the two most common testing scenario, i.e. as a support of TDD directly in

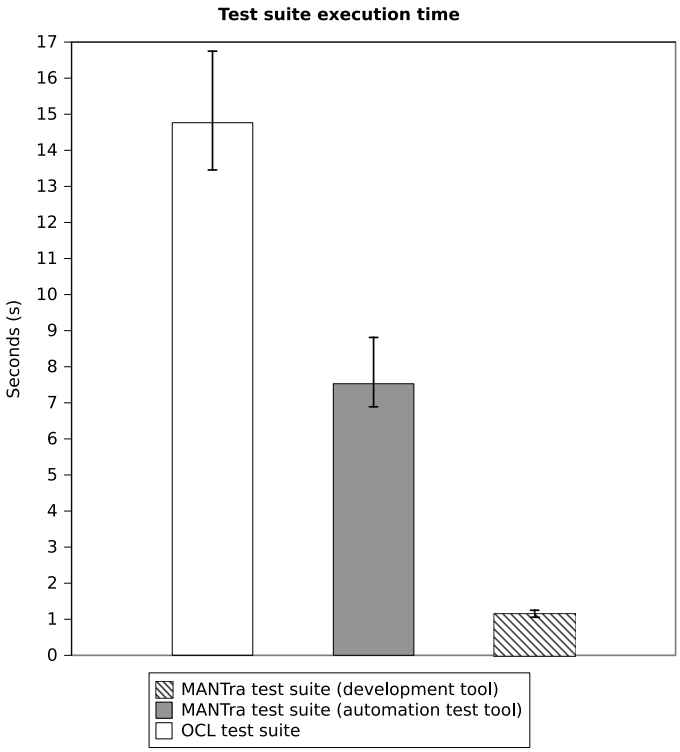

Fig. 6. Tests suite execution time

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 II reports basic statistics on the performance dataset. 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.

VI. 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 practical aspects of the testing tool are presented in Section IV 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 define some kind of ”best practices” that are specific for this testing approach and tool.

ACKNOWLEDGMENTS

Work partially supported by the EU project Q-ImPrESS (FP7 215013).

REFERENCES

[1] R. France and B. Rumpe, “Model-driven development of complex software: A research roadmap,” in *Proc. Future of Software Engineering FOSE ’07*, 23–25 May 2007, pp. 37–54.

[2] M. J. Harrold, “Testing: A roadmap,” in *In The Future of Software Engineering*. ACM Press, 2000, pp. 61–72.

[3] Y. Lin, J. Zhang, and J. Gray, “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*. Springer, 2004, pp. 219–236.

[4] A. Ciancone and A. Filieri, “Mantra website,” http://aciancone.inscatolati.net/prj/MANTra/.

[5] O. M. Group, “Qvt 1.0 specification,” http://www.omg.org/spec/QVT/1.0/.

[6] Q.-I. Consortium, “Project website,” http://www.q-impress.eu.

[7] *EMF: Eclipse Modeling Framework (2nd Edition) (Eclipse)*, 2nd ed. Addison-Wesley Longman, Amsterdam, January 2009.

[8] F. Fleurey, J. Steel, and B. Baudry, “Validation in model-driven engineering: testing model transformations,” in *First International Workshop on Model Design and Validation*, 2004, pp. 29–40.

[9] J. Wang, S.-K. Kim, and D. Carrington, “Automatic generation of test models for model transformations,” in *ASWEC ’08: Proceedings of the 19th Australian Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 432–440.

[10] S. Sen, B. Baudry, and J. M. Mottu, “On combining multi-formalism knowledge to select models for model transformation testing,” in *Software Testing, Verification, and Validation, 1st International Conference on*, 2008, pp. 328–337.

[11] A. Community, “The alloy analyzer,” http://alloy.mit.edu/.

[12] Y. Lin, J. Zhang, and J. Gray, “A testing framework for model transformations,” in *in Model-Driven Software Development - Research and Practice in Software Engineering*. Springer, 2005, pp. 219–236.

[13] M. J. Mcgill and B. H. C. Cheng, “Test-driven development of a model transformation with jemtte,” 2007.

[14] B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon, “Model transformation testing challenges,” in *Proceedings of IMDT workshop in conjunction with ECMDA’06*.

[15] S. Becker, H. Koziolek, and R. Reussner, “The palladio component model for model-driven performance prediction,” *Journal of Systems and Software*, vol. 82, no. 1, pp. 3 – 22, 2009, special Issue: Software Performance - Modeling and Analysis. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0N-4SK62RX-1/2/1cfdee66a9dec4b2685c2d20705a5a2c

[16] V. Grassi, R. Mirandola, E. Randazzo, and A. Sabetta, “Klaper: An intermediate language for model-driven predictive analysis of performance and reliability,” *The Common Component Modeling Example*, pp. 327–356, 2007.

[17] A. Ciancone, “jomot framework,” http://aciancone.inscatolati.net/prj/jOMoT/.

[18] T. E. Foundation, “Project website,” http://www.eclipse.org.

[19] A. Ciancone, “Mapping the service architecture meta-model to the palladio component model,” Master’s thesis, Politecnico di Milano, Dipartimento di Elettronica e Informazione, Piazza L. Da Vinci 32, 20133 Milan, Italy, 2010.