This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering

Static Fault Localization in Model Transformations

Loli Burgueño, Javier Troya, Manuel Wimmer and Antonio Vallecillo

Abstract—As the complexity of model transformations grows, there is an increasing need to count on methods, mechanisms and tools for checking their correctness, i.e., the alignment between specifications and implementations. In this paper we present a light-weight and static approach for locating the faulty rules in model transformations, based on matching functions that automatically establish these alignments using the metamodel footprints, i.e., the metamodel elements used. The approach is implemented for the combination of Tracts and ATL, both residing in the Eclipse Modeling Framework, and is supported by the corresponding toolkit. An evaluation discussing the accuracy and the limitations of the approach is also provided. Furthermore, we identify the kinds of transformations which are most suitable for validation with the proposed approach and use mutation techniques to evaluate its effectiveness.

Index Terms—Model transformation, transformation testing, model alignment

1 Introduction

Model transformations are key elements of Model-driven Engineering (MDE) [1]. They allow querying, synthesizing, and transforming models into other models or into code, and are essential for building systems in MDE. In this context, the quality of the resulting systems is therefore highly influenced by the quality of the model transformations employed to produce them. However, users of transformations have to deal with the problem that transformations are difficult to debug and test for correctness [2]. In fact, as the size and complexity of model transformations grow, manual debugging is no longer possible, and there is an increasing need to count on methods, mechanisms and tools for testing their correctness [2], [3].

In general, debugging is readily classified into three parts: the identification of the existence of a problem, the localization of the fault, and the actual correction of the problem [4].

In this paper, the existence of a problem is detected by the misalignment between the model transformation specification and its implementation. The former specifies the *contract* that determines the expected behavior of the transformation and the context in which such a behavior needs to be guaranteed, while the latter provides the actual behavior of the transformation. If the transformation does not behave as expected, a violation of the contract occurs.

- *Loli Burgueño and Antonio Vallecillo are with the Universidad de Málaga, Dept. Lenguajes y Ciencias de la Computación, Bulevar Louis Pasteur, 35, 29071, Malaga, Spain*
  E-mail: {loli,av}@lcc.uma.es
- *Javier Troya and Manuel Wimmer are with the Vienna University of Technology, Business Informatics Group, Vienna, Austria*
  E-mail: {troya,wimmer}@big.tuwien.ac.at

Here we use Tracts [5] for the specification of model transformations, which are a particular kind of *model transformation contracts* [6], [7] especially suitable for specifying model transformation in a modular and tractable manner. Tracts counts on tool support for checking, in a black-box manner, that a given implementation behaves as expected—i.e., it respects the Tracts constraints [8].

Once a problem has been found (i.e., a constraint has been violated), we need to locate the fault [9]. One of the major shortcomings of model transformation specification approaches based on contracts is the lack of traceability links between specifications and implementations. In the case a constraint is not fulfilled, the elements involved in the constraint evaluation could provide valuable information to the transformation engineer, but the links to the transformation implementation are not available.

Based on first ideas which we outlined in previous work [10], this paper presents a solution to this problem. It uses a white-box and static analysis to find the location of the model transformation rules that may have caused the faulty behavior. It provides the first step of an iterative approach to model transformation testing, which aims at locating faults as early as possible in the development process. Although this step cannot fully prove correctness, it can be useful for identifying many bugs in a very early stage and in a quick and cost-effective manner [11]. It can also deal with industrial-size transformations without having to reduce them or to abstract away any of their structural or behavioral properties, and it can represent a very valuable first step before diving into more expensive and complex tests (such as model checking, formal validation, dynamic tests, etc. [12]–[17]) which represent numerous challenges, mainly because of their inherent computational complexity [2], [6].

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering

An evaluation discussing the accuracy and the limitations of the approach is also provided. The evaluation has been conducted on a number of transformations with the goal of quantitatively assessing the correctness (Are the alignments correct?), completeness (Are there any missed alignments?) and usefulness (How useful is the resulting information to the developer for locating the faults?) of the techniques. Furthermore, we also identify the kinds of transformations which are most suitable for validation with the proposed approach, and provide a test to automatically check this a-priori. Finally, we use mutation techniques to evaluate its effectiveness.

This paper is organized as follows. After this introduction, Section 2 briefly presents the background to our work and the technologies we make use of. Then, Section 3 introduces the proposed approach and Section 4 discusses how we have implemented it in the case of ATLAS Transformation Language (ATL) [18] model transformations. Section 5 is devoted to the evaluation of our proposal, and to analyze its advantages and limitations. Finally, Section 6 presents related work before Section 7 concludes and outlines future research lines.

2 Background

This section explains the prerequisites for the rest of the paper, namely how to specify, implement, and test model transformations with a combination of Tracts and ATL.

2.1 Specifying Transformations with Tracts

Tracts were introduced in [5] as a specification and black-box testing mechanism for model transformations. They provide modular pieces of specification, each one focusing on a particular transformation scenario. Thus each model transformation can be specified by means of a set of Tracts, each one covering a specific use case—which is defined in terms of particular input and output models and how they should be related by the transformation. In this way, Tracts allow partitioning the full input space of the transformation into smaller, more focused behavioral units, and to define specific tests for them. Commonly, what developers are expected to do with Tracts is to identify the scenarios of interest (each one defined by a Tract) and check whether the transformation behaves as expected in these scenarios.

In a nutshell, a Tract defines a set of constraints on the *source* and *target* metamodels, a set of *source-target* constraints, and a *test suite*, i.e., a collection of source models. The constraints serve as "contracts" (in the sense of contract-based design [19]) for the transformation in some particular scenarios, and are expressed by means of OCL invariants. They provide the *specification* of the transformation. Test suites models are pre-defined input sets of different sorts,

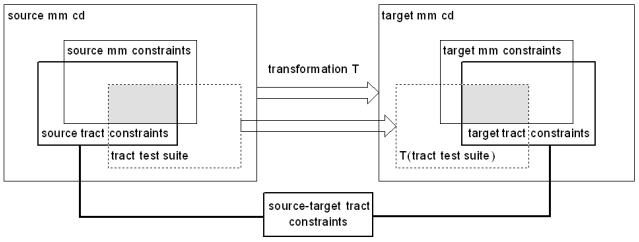

Fig. 1. Building Blocks of a Tract.

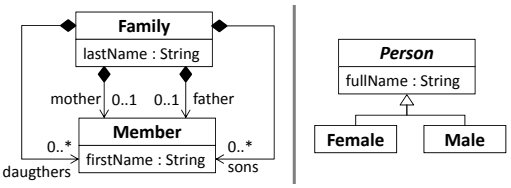

Fig. 2. The Family and Person metamodels.

to exercise an implementation of the transformation (they may not only be positive test models, satisfying the source constraints, but also negative test models, used to know how the transformation behaves with them).

Fig. 1 depicts the main components of the Tracts approach: the source and target metamodels, the transformation T under test, and the transformation contract, which consists of a Tract test suite and a set of Tract constraints. In total, five different kinds of constraints are present: the source and target models are restricted by general constraints added to the language definition, and the Tract imposes additional source, target, and source-target Tract constraints for a given transformation. In the drawing, mm stands for metamodel, and cd is a short for class diagram.

If we assume a source model M being an element of the test suite and satisfying the source metamodel and the source Tract constraints given, the Tract essentially requires the result $T(M)$ of applying transformation T to satisfy the target metamodel and the target Tract constraints, and the tuple $< M, T(M) >$ to satisfy the source-target Tract constraints.

To demonstrate how to use Tracts, we introduce the simple transformation example *Families2Persons* (the complete example is available from our project website [8]). The source and target metamodels of this transformation are shown in Fig. 2.

For this example, one Tract (Listing 1) is developed to consider only those families which have exactly four members (mother, father, daughter, son). The first constraint states that all families in the source model have exactly one daughter and one son. The second and third constraints state that all mothers and daughters are transformed into female persons. Constraint C_4 mandates that all fathers and sons should be transformed into male persons. Constraints C_5 and C_6 state, respectively, that all female and male objects in

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering

the target model come from the corresponding object in the source model. Then, C_7 checks that the size of the source and target models correspond. Finally, C_8 checks that all names in the target model are neither the empty String nor undefined. Note that although some of the constraints could have been written using similar expressions (e.g., C_2, C_3, C_4), we decided to express them using different styles for illustration purposes, and also to be able to differentiate them in our analyses.

Listing 1. Tracts for the *Families2Persons* case study.

```
-- C1: SRC_oneDaughterOneSon
Family.allInstances->forAll(f|f.daughters->size=1 and f.
    sons->size=1)

-- C2: SRC_TRG_Mother2Female
Family.allInstances->forAll(fam|Female.allInstances->
    exists(f|fam.mother.firstName.concat('␣').concat(fam
    .lastName)=f.fullName))

-- C3: SRC_TRG_Daughter2Female
Family.allInstances->forAll(fam|Female.allInstances->
    exists(f|fam.daughters->exists(d|d.firstName.concat(
    '␣').concat(fam.lastName)=f.fullName)))

-- C4: SRC_TRG_FatherSon2Male
Family.allInstances->forAll(fam|Male.allInstances->exists
    (m| fam.father.firstName.concat('␣').concat(fam.
    lastName)=m.fullName xor fam.sons->exists(s|m.
    firstName.concat('␣').concat(fam.lastName)=s.
    fullName))

-- C5: SRC_TRG_Female2MotherDaughter
Female.allInstances->forAll(f|Family.allInstances->exists
    (fam|fam.mother.firstName.concat('␣').concat(fam.
    lastName)=f.fullName xor fam.daughters->exists(d|d.
    firstName.concat('␣').concat(fam.lastName)=f.
    fullName)))

-- C6: SRC_TRG_Male2FatherSon -- analogous to C5

-- C7: SRC_TRG_MemberSize_EQ_PersonSize
Member.allInstances->size=Person.allInstances->size

-- C8: TRG_PersonHasName
Person.allInstances->forAll(p|p.fullName <> '' and not p.
    fullName.oclIsUndefined())
```

Concerning the kinds of constraints defined, C_1 represents a pre-condition for the transformation, $C_2 - C_7$ define constraints on the relationships between the source and target models, i.e., constraints that should be ensured by the transformation, and finally, C_8 represents a post-condition for the transformation. Note that this approach is independent from the model transformation language and platform finally used to implement and execute the transformation.

2.2 Implementing Transformations with ATL

Given this specification, a model transformation language may be selected to implement the transformation. The ATLAS Transformation Language (ATL) [18] is a common choice. ATL is designed as a hybrid model transformation language containing a mixture of declarative and imperative constructs for defining uni-directional transformations. An ATL transformation is mainly composed by a set of *rules*. A rule describes how a subset of the target model should be generated from a subset of the source model. A rule consists of an *input* pattern (henceforth also referred as left-hand side)—having an optional *filter* condition—which is matched on the source model and an *output* pattern (henceforth also referred as right-hand side) which produces certain elements in the target model for each match of the input pattern. OCL expressions are used to calculate the values of target elements' features, in the so-called *bindings*. Given the metamodels in Fig. 2, a possible implementation in ATL is shown in Listing 2.

It comprises two helper functions (whose definition is not shown in the listing) and two rules. One of the helpers is used to decide whether a member is female or not, and the second one is used to compute the family name of a family member. Then, the first rule, R1, transforms male members (note the use of the helper isFemale() to filter the corresponding source objects) into male persons and computes their fullName attribute. Rule R2 is analogous, but for female family members.

Listing 2. *Families2Persons* ATL Transformation.

```
module Families2Persons;
create OUT: Persons from IN: Families;

helper context Families!Member def:isFemale:Boolean=...
helper context Families!Member def:familyName:String=...

rule Member2Male { -- R1
 from
  s: Families!Member (not s.isFemale)
 to
  t: Persons!Male(fullName<-s.firstName+'␣'+s.familyName)
}
rule Member2Female { -- R2
 from
  s: Families!Member (s.isFemale)
 to
  t:Persons!Female(fullName<-s.firstName+s.familyName)
}
```

2.3 Testing Transformations with Tracts

By running the transformation implementation for each model of the test suite and checking the target as well as source-target constraints for the resulting input model and output model pairs, the validation of the transformation with respect to the constraints is achieved. The output of this validation phase is a test report documenting each constraint validation for the given input and output model pairs. An example report for the *Families2Persons* example for an input test model called src_model001 produced by the *TractsTool* [8], [20] is shown in Listing 3. Such a model is composed of 1250 model elements (250 families, each one with one father, one mother, one son and one daughter), and was generated by an ASSL [21] procedure (cf. [8]).

Listing 3. Test result for the *Families2Persons* example.

```
----------------------------
-- Results for src_model001
----------------------------
C1: SRC_oneDaughterOneSon: OK
...
```

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering


```
C4: SRC_TRG_FatherSon2Male: KO
    Instances of src_model001 violating the constraint
    Set(Member001, Member002, ...)
...
```

In order to fix the transformation implementation to fulfill all constraints, the alignments between the transformation rules and the constraints are crucial in order to track the actual faults in the transformation rules from the observed constraints violations. While for the given example this may be achieved by just looking at the constraints and the rules (actually $R2$ misses the white space in the String concatenation), for larger examples automation support is essential due to the complexity of model transformations. Even in this example the alignment between the rules and the constraints is not trivial, and this is precisely where our proposed approach comes into play.

3 Matching Constraints and Rules

In this section, we introduce our approach to locate faults in model transformations.

3.1 Motivation and Challenges

As we have seen in the previous section, Tracts have allowed us to define constraints for specifying transformations, while ATL uses rules to express model transformation implementations. Having independent artifacts for the specification and implementation of model transformations permits choosing which formalism to use for each level. However, the following questions cannot be answered without a thorough analysis of both artifact types:

- Which transformation rule(s) implement(s) which constraint(s)?
- Are all constraints covered by the transformation rules?
- Are all transformation rules covered by the constraints?

In order to establish the relation between the constraints and the rules that might make them fail, two approaches can be followed: *dynamic* or *static*.

Dynamic approaches are based on a concrete model transformation execution over a model or set of models. The procedure consists of tracking the transformation process and storing information about each executed step and the specific instances. Once the transformation has finished and the failures and the objects that caused them are known, it is necessary to go backwards over the trace information stored during the transformation execution to find the errors. In these approaches, an input model needs to be available to execute the transformation, and the environment where the transformation is to be executed must be provided too.

Static approaches, in turn, do not make use of executions. They obtain the relation between the constraints and the rules by means of an algorithm. The only inputs for this process are the transformation implementation and the specification constraints.

Dynamic approaches normally give more precise results, although, as mentioned before, they are dependent on the particular input model and transformation execution, while static ones can compute more general alignments. In this paper, we target the challenge of finding "guilty" transformation rules following a static approach. Since there is no direct relation between the rules and the constraints (constraints are created independently of any transformation implementation), our work computes for each pair (constraint, rule) the probability that the constraint failure comes from the rule making use of the common denominator that both have: the structural elements belonging to the metamodels.

It can also be considered a white-box approach, because it takes into account the internal structure and details of the tract constraints and of the transformation implementation.

3.2 Methodological Approach

Given a set of OCL constraints (from the Tracts) and a set of ATL rules, Fig. 3 summarizes the commonalities between them (in the figure, relationships $\ll$c2$\gg$ and $\ll$u$\gg$ stand, respectively, for "conforms to" and "uses"). There is also a direct relation between the ATL and the OCL metamodels, because the former embeds the latter. This may simplify the alignments between ATL and OCL, although it is also true that the OCL constraints and the ATL rules are written differently. First, the former impose conditions on the relationship between the source and target models, while the latter describe how the target model should be built from the elements of the source model. Second, specifications and implementations are normally written by different people, at different times, with different goals in mind, and using different styles (e.g., they may use different navigation paths to refer to the same elements, because the starting contexts are not the same, or use different OCL operators for querying elements). Finally, there are slight differences between OCL and ATL, e.g., ATL introduces additional operations which are of particular interest for transformations which are not available in OCL. In any case, the OCL constraints and the ATL rules make use of the same source and target metamodels. As we have seen for the *Families2Persons* example, the same types and features are used in the specification and in the implementation of the transformation. Thus, we use these commonalities to indirectly match the constraints and the rules by matching their footprints concerning the source and target metamodels used.

Our approach focuses on the construction and interpretation of the so-called *matching tables* with the alignments we have discussed before. Thus, our approach builds on the following steps:

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering

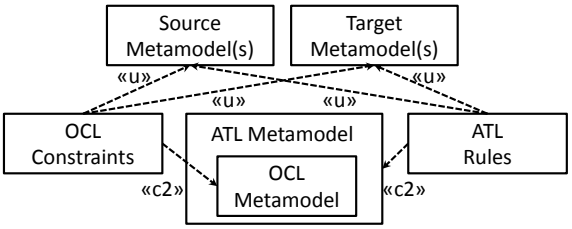

Fig. 3. Heterogeneities and Commonalities between Constraints and Rules.

1) **Footprint Extraction**. The *structural elements* (henceforth referred to as *footprints* or *types*) of both model transformation and constraints are extracted, as explained later in Section 3.3.
2) **Footprint Matching**. The footprints extracted in the previous step are compared for each rule and constraint.
3) **Matching Tables Calculation**. The percentage of types overlapping, so-called *alignment*, for each transformation rule and constraint is calculated. This information is used to produce the matching tables (Section 3.4).
4) **Matching Tables Interpretation**. The resulting tables are analyzed for identifying the guilty rules for each constraint. Guidelines for this analysis, exemplified with a case study, are described in Section 3.5.

3.3 Footprint Extraction

Now we present how we extract footprints from OCL constraints and ATL rules.

3.3.1 Constraints

There are several possibilities for the footprints extraction of OCL constraints. For example, we could take into consideration all types that appear in the OCL expressions, just because they are mentioned. We could even assign weights to these types according to their number of occurrences in the constraints, giving less importance (a lower value) to those that appear less often. However, in order to isolate the information which is really relevant for our purposes, it is important to distinguish between two different kinds of elements that appear in the OCL expressions: those that we want the constraint to refer to, and those which are used for navigation purposes only. Since metamodels are graphs, OCL expressions are heavily dependent on their *contexts* (i.e., the starting class) [22] and also on the path used to navigate to the final type, which is precisely the one we want the constraint to refer to. Thus we need to isolate the target features of the constraint from the ones used to reach it. This is why we only consider as relevant the last elements of the OCL expressions. For example, if we have Family.mother.firstName, then we will only consider mother.firstName whose types are Member and Member.firstName. To consider operations on collections, we take into account only the types inside the body of the deepest (in the sense of nesting) iterators (*forAll*, *exists*, etc.), to extract just the relevant types and not those used for navigation purposes only.

Similarly, primitive types and constants are not considered. Types like *Integer* or *Boolean*, or constants like *true* or *false* can appear frequently, but this does not mean that each appearance provides relevant information for locating a fault. On the contrary, taking them into consideration only introduces more confusion, when precisely our goal is to isolate those elements that are more relevant for locating the faults.

TABLE 1
Footprints for the *Families2Persons* example.

Constraint	Considered Types
C1	Member, Family, Family.daughters, Family.sons
C2	Member, Family, Female, Member.firstName, Family.lastName, Family.fullName
C3	Member, Family, Female, Member.firstName, Family.daughters, Family.lastName, M.firstName, Female.fullName
C4	Member, Family, Male, Member.firstName, Family.lastName, Male.fullName, Family.father, Family.sons
C5	Member, Family, Family.mother, Family.lastName, Female.fullName, Family.daughters, M.firstName
C6	Member, Family, Family.father, Family.firstName, Family.lastName, Male.fullName, Family.sons
C7	Member, Person
C8	Person, Person.fullName

Rule	Considered Types
R1	Member, Family, Male, Family.firstName, Male.fullName
R2	Member, Family, Female, Family.firstName, Female.fullName

3.3.2 Rules

In this paper we deal with ATL as proof of concept, although any transformation language based on rules and that uses OCL could be used. For each rule, we obtain the footprints in the left-hand side, right-hand side and imperative part, and build all navigation paths. Then, as in the OCL constraints, we only consider the last part of these paths. Regarding helpers, they can appear in any part of a navigation path. For this reason, when there is a helper in a path, we simply obtain the type it returns. If it is a collection type, we obtain the type of the collection.

We apply the same approach for calls of ATL (unique) lazy rules and called rules. In these cases, we return the type of the first element created by these rules (since this is what ATL actually returns).

With all this, the footprints extracted for the *Families2Persons* example presented in Section 2 are shown in Table 1, for each rule and constraint.

3.4 Footprint Matching and Matching Tables

A tabular representation (called *matching tables*) is used to depict the alignment between constraints

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering

and rules. We apply three different matching functions to automatically obtain the values for filling the tabular representations. Each function provides a certain viewpoint on the alignment. This allows us to interpret the results and provides an answer to the questions presented in Section 3.1.

In these tables, rows represent constraints and columns represent rules. Each cell links a constraint and a rule with a specific value between 0 and 1. Let C_i be the set of types extracted from constraint i and R_j from rule j. Let $|\cdot|$ represent the size of a set.

3.4.1 Matching Tables: Three Different Viewpoints

The *constraint coverage* (CC) metric focuses on constraints. This metric measures the coverage for constraint i by a given rule j. For this metric, the value for the cell $[i, j]$ is given by the following formula.

$$CC_{i,j} = \frac{|C_i \cap R_j|}{|C_i|} \quad (1)$$

Since the denominator is the number of types in C_i, the result is relative to constraint i and we interpret this value for rule traceability, i.e., to find the rules related to the given constraint. This is, if a constraint fails, the CC table tells us which rule or rules are more likely to have caused the faulty behavior (i.e., be "guilty"). Thus, the CC table is to be consulted by rows.

The *rule coverage* (RC) metric focuses on rules. This metric calculates the coverage for rule j by a given constraint i. We use the RC table to express constraint traceability, i.e., to find the constraints more closely related to a given rule, and therefore it is to be read by columns. The metric is calculated as follows.

$$RC_{i,j} = \frac{|C_i \cap R_j|}{|R_j|} \quad (2)$$

The last metric is relative to both constraints and rules, so the RCR table can be consulted by rows and by columns. Thus, it provides information about the relatedness of both rules and constraints, without defining a direction for interpreting the values. The *relatedness* of *constraints* and *rules* (RCR) metric is computed as follows.

$$RCR_{i,j} = \frac{|C_i \cap R_j|}{|C_i \cup R_j|} \quad (3)$$

The overlap between the elements extracted from the constraints and the rules gives rise to five different cases which are reflected by the previous metrics. They are depicted in Fig. 4 using Venn diagrams.

In case (a), each type present in the constraint is contained in the set of types in the rule: $C_i \subseteq R_j$. Consequently, the value for the CC metric is 1, meaning that the constraint is fully covered by the rule. The other metrics have a value lower than 1.

In case (b), all the types in the rule are contained in the types of the constraint, $R_j \subseteq C_i$. RC metric is 1.

In case (c), C_i and R_j are disjoint sets. Thus, the three metrics are 0, which means that the given constraint and the given rule are completely independent.

In case (d), each metric will have a value between 0 and 1. The specific value depends on the size of the sets and on the number of common types. Thus, the bigger the common part for C_i is, the closer to 1 the value for metric CC will be. Similarly for R_j and metric RC. Regarding the RCR metric, its value only depends on the size of the common part (for a specific size of the footprints); the bigger it is, the closer to 1 the value will be.

In case (e), both constraints and rules have the same types set, so all metrics are 1.

Considering type inheritance. In the three formulas presented above, we consider the intersection $C_i \cap R_j$ as the common types present in constraint C_i and rule R_j. But we should also take inheritance into account. Its consideration is important because some OCL operators used in the Tract constraints and in the ATL rules (such as allInstances) retrieve all instances of a certain class, as well as the instances of all its subclasses, and therefore we can have types in a constraint and in a rule that are not directly related (since they are not the same type), but are related via inheritance (when one type is a sub/super-type of the other). Thus, the fault may be due to a problem not only in a class but also in any of its superclasses. To take this into consideration, we assign a weight to the parent classes, given by the number of its structural features (attributes and references) divided by the number of features of the child class — both sets comprise the class's own features as well as the inherited features from all its superclasses. Thus, the more similar the parent and the child are, the closer to 1 the weight is. Similarly, if the child class incorporates many new features w.r.t. the parent class, the weight assigned to the parent will be closer to 0.

Setting a threshold value. Before going any further, let us explain the need for setting a threshold for cell values in the matching tables. Such a threshold is meant to establish a boundary under which alignments are ignored. It is needed to be able to disregard those situations where a constraint and a rule are minimally related, and thus should not be considered as relevant for locating the fault. Moreover, if a value in a cell is below the threshold in table RCR, then

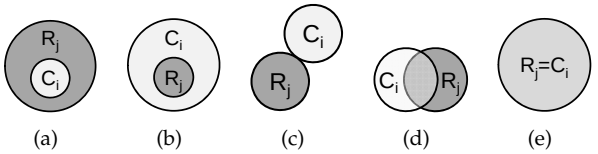

Fig. 4. Possible overlaps for C_i and R_j.

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering

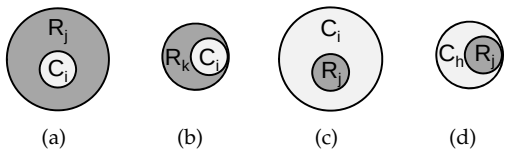

Fig. 5. Situations with differently sized rule/constraint footprints.

the value in the equivalent cells in the other two tables must be disregarded too, even if their value is above the threshold, to avoid considering irrelevant information.

Fig. 5 helps explain this situation. Assume that the types extracted from constraint C_i are a subset of the types extracted from rule R_j, as shown in Fig. 5(a). In this case, the CC metric for this pair is 1. However, since the set of common types is very small in comparison with the size of the set of rule types, the RCR metric is also very small. Despite there being some common types in the rule and in the constraint, it does not mean that, in this case, the rule is covering the constraint. In most cases where the set of common types is much smaller than the set of rule types (even if the CC metric is 1), it is normally because our metamodels are small and the same type may be present in several rules and constraints, and not because there is a relevant relationship. In such cases, when a value is lower than the threshold, we consider that it is not relevant and therefore we do not take it into account. In Fig. 5(b), all the types in the set C_i are also a subset of the types in rule R_k. The difference lies in the fact that the ratio $|Ci|/|R_k|$ is higher and thus, a relevant value. This means that it is more likely that the rule R_k is implementing the use case that constraint C_i is specifying and, therefore, we should consider the alignment between them as being relevant for our purposes. In fact, in order for the constraint to be properly covered, there should exist a rule that covers the constraint with a large portion. In such a case, the RCR metric would be higher than the threshold and the CC metric shall be considered. Similarly for metric RC, let us suppose that rule R_j is completely covered by constraint C_i, as in Fig. 5(c). In this case, the RC metric is 1, since all the types of R_j are included in C_i. However, as very few types of C_i are present in R_j, the RCR metric is very small, so the RC metric should not be taken into account. There should exist, consequently, a constraint that has a larger portion of its types in common with R_j. Fig. 5(d) shows an example where the value of RCR is above the threshold and, thus, metric RC is considered.

In summary, this threshold is needed to eliminate the consideration of matches with very low probability, which only cause interferences when looking for the rules that cause the fault. The need for this

TABLE 2
Families2Persons matching tables.

	CC		RC		RCR	
	R1	R2	R1	R2	R1	R2
C1	0.50	0.50	0.50	0.50	0.33	0.33
C2	0.33	0.66	0.50	1.00	0.25	0.66
C3	0.33	0.66	0.50	1.00	0.25	0.66
C4	0.66	0.33	1.00	0.50	0.66	0.25
C5	0.33	0.66	0.50	1.00	0.25	0.66
C6	0.66	0.33	1.00	0.50	0.66	0.25
C7	0.50	0.50	0.50	0.50	0.33	0.33
C8	0.00	0.00	0.00	0.00	0.33	0.33

threshold is based on our experiments with the tables. The current value for the threshold is 0.1. This means that, at least, 10% of the types that appear in a rule must be present in a constraint in order to consider the CC metric between both. Similarly, at least 10% of the types in a constraint must be covered by a rule in order to take their RC metric into account. This value has proved to be the most effective threshold for obtaining the highest recall and precision in all the case studies we have analyzed. Research is currently in progress to provide a theoretical justification for such a value. In any case, this value is currently a configuration parameter in our toolkit to allow easy tuning.

Example. Table 1 shows the metrics computed for the *Families2Persons* example, presented in Section 2. Note that, for a small example like this, the metrics provide information that can be easily interpreted by just looking at the constraints and the rules. The second and third columns express the constraint coverage, the fourth and fifth ones the rule coverage, and the sixth and seventh ones the relatedness.

3.4.2 *Matching Tables for* UML2ER

The *Families2Persons* case study presented so far is a rather small example, although sufficient for demonstrating the basic process of computing the different metrics. Let us analyze here a bigger transformation, namely the *UML2ER* project, from the structural modeling domain. It generates Entity Relationship (ER) Diagrams from UML Class Diagrams.

We have extended the metamodels for the *UML2ER* case study presented in [23]. They are illustrated in Fig. 6, and the Tracts we have defined for it are shown in Listing 4. Please note the black triangle symbol used in the Listing for the sake of brevity. It is used for marking the place in a constraint (triangle down) that may be extended by another constraint (triangle up). For instance, constraint C2 is extending constraint C1.

Listing 4. Tracts for the *UML2ER* case study.

```
-- C1: SRC_TRG_Package2ERModel
Package.allInstances->forAll(p|ERModel.allInstances
->one(e|p.name=e.name [▼]))

-- C2: C1 + Class2EntityType + Nesting
C1[▲] and p.ownedElements-> forAll(class|e.entities
->one(entity|entity.name=class.name [▼]))

-- C3: C2 + Property2Feature + Nesting
```

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering

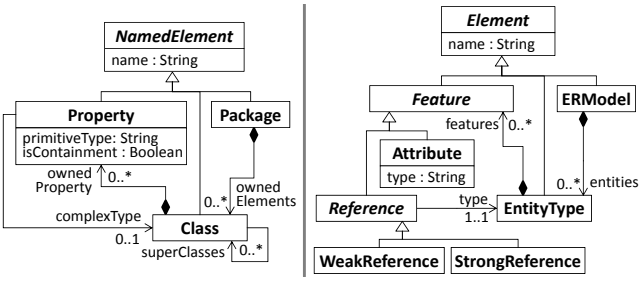

Fig. 6. The UML and ER metamodels.

```
C2[▲] and class.ownedProperty->forAll(p|entity.
features->forAll(f|f.name=p.name))

-- C4: SRC_TRG_NamedElement2Element
NamedElement.allInstances->size=Element.allInstances->
size

-- C5: SRC_TRG_Package2ERModel
Package.allInstances->size=ERModel.allInstances->size

-- C6: SRC_TRG_Class2EntityType
Class.allInstances->size=EntityType.allInstances->size

-- C7: SRC_TRG_Property2Feature
Property.allInstances->size=Feature.allInstances->size

-- C8: C2 + Property2Attribute + Nesting
C2[▲] and class.ownedProperty->forAll(p|p.primitiveType
<> null implies entity.features->select(f|
f.oclIsTypeOf(Attribute))->one(f|f.name=p.name))

-- C9: C2 + Property2WeakReference + Nesting
C2[▲] and class.ownedProperty->forAll(p|p.complexType
<> null implies entity.features->select(f|f.oclIsTypeOf(
Reference))->one(f|f.name=p.name and p.isContainment
implies f.oclIsTypeOf(WeakReference)))

-- C10: C2 + Property2StrongReference + Nesting
C2[▲] and class.ownedProperty->forAll(p|p.complexType
<> null implies entity.features->select(f|f.oclIsTypeOf(
Reference))->one(f|f.name=p.name and not p.isContainment
implies f.oclIsTypeOf(StrongReference)))
```

The transformation (shown in Listing 5) contains eight rules, where three of them are abstract. There is a large number of inheritance relationships between the rules: $R8, R7 < R6; R6, R5 < R4; R4, R3, R2 < R1$.

Listing 5. *UML2ER* ATL Transformation.

```
module UML2ER;
create OUT : ER from IN : SimpleUML;

abstract rule NamedElement{ --R1
  from s : SimpleUML!NamedElement
  to   t : ER!Element(name <- s.name)
}
rule Package extends NamedElement{ --R2
  from s: SimpleUML!Package
  to   t: ER!ERModel(entities<-s.ownedElements)
}
rule Class extends NamedElement{ --R3
  from s: SimpleUML!Class
  to t: ER!EntityType(features<-s.ownedProperties)
}
abstract rule Property extends NamedElement{ --R4
  from s: SimpleUML!Property
  to   t: ER!Feature ()
}
rule Attributes extends Property{ --R5
  from s: SimpleUML!Property(
          not s.primitiveType.oclIsUndefined())
  to   t: ER!Attribute (type <- s.primitiveType)
}
abstract rule References extends Property{ --R6
  from s: SimpleUML!Property(
          not s.complexType.oclIsUndefined())
  to   t: ER!Reference (type <- s.complexType)
}
rule WeakReferences extends References{ --R7
  from s: SimpleUML!Property (not s.isContainment)
  to   t: ER!WeakReference
}
rule StrongReferences extends References{ --R8
  from s: SimpleUML!Property (s.isContainment)
  to   t: ER!StrongReference }
```

Tables 3 to 5 illustrate the corresponding matching tables for the transformation and the given Tracts (please ignore for the moment the square brackets enclosing some numbers). Those cells without a number indicate there is no alignment between the constraint and the rule. The following subsection explains how the information in these matching tables is to be interpreted.

TABLE 3
Matching table using CC metric.

	R1	R2	R3	R4	R5	R6	R7	R8
C1	0.25	0.5						
C2	0.2	0.6	0.4			[0.4]		
C3	0.25		0.25	0.5	0.38	0.38	0.38	0.38
C4	1.0	1.0	0.75	0.5	0.5	1	0.5	0.5
C5	0.5	1.0						
C6	0.5	0.5	1.0			[1.0]		
C7	0.5		0.5	1.0	0.75	0.75	0.75	0.75
C8	0.25		0.17	0.33	0.5	[0.25]	[0.25]	[0.25]
C9	0.28		0.22	0.22	[0.17]	0.44	0.33	[0.22]
C10	0.28		0.22	0.22	[0.17]	0.44	[0.33]	0.22

TABLE 4
Matching table using RC metric.

	R1	R2	R3	R4	R5	R6	R7	R8
C1	0.25	0.33						
C2	0.25	0.5	0.5			[0.33]		
C3	0.25		0.25	1.0	0.38	0.25	0.5	0.75
C4	0.5	0.33	0.38	0.5	0.25	0.33	0.33	0.5
C5	0.25	0.33						
C6	0.25	0.33	0.5			[0.33]		
C7	0.25		0.25	1.0	0.38	0.25	0.5	0.75
C8	0.38		0.25	1.0	0.75	[0.25]	[0.5]	[0.75]
C9	0.63		0.5	1.0	[0.38]	0.67	1.0	[1.0]
C10	0.63		0.5	1.0	[0.38]	0.67	[1.0]	1.0

TABLE 5
Matching table using RCR metric.

	R1	R2	R3	R4	R5	R6	R7	R8
C1	0.13	0.25						
C2	0.11	0.38	0.29			[0.22]		
C3	0.13		0.14	0.5	0.21	0.17	0.25	0.3
C4	0.5	0.25	0.25	0.25	0.17	0.25	0.2	0.25
C5	0.17	0.33						
C6	0.17	0.33	0.5			[0.33]		
C7	0.17		0.2	1.0	0.3	0.21	0.38	0.5
C8	0.15		0.11	0.33	0.43	[0.14]	[0.19]	[0.21]
C9	0.19		0.18	0.22	[0.13]	0.36	0.33	[0.2]
C10	0.19		0.18	0.22	[0.13]	0.36	[0.3]	0.22

3.5 UML2ER case study: Pragmatics

Recall that the purpose of the matching tables is to help find the rule(s) that caused the fault when a constraint is not satisfied. To show how these rules

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering

are located, let us suppose that we have executed the *UML2ER* transformation for a certain input model and checked the satisfaction of the constraints, something that can be done with our *TractsTool* [8] quite straightforwardly. Let us assume the outcome given by the tool is that constraint $C7$ is not satisfied. In Table 3 we can see that there is a complete coverage of $C7$ by rule $R4$ (as mentioned in Section 3.4, this table is to be consulted by rows). Consequently, it is very likely that the constraint fails due to this rule, so we should start by checking $R4$. Nevertheless, it does not always mean that $R4$ is the guilty rule. In fact, there are other candidate rules (all of them except $R2$, since the value in cell $C7/R2$ is 0) that could be the cause of the fault. Among them, $R5$, $R6$, $R7$ and $R8$ have the same CC value for constraint $C7$. In order to establish a priority order among these rules, we need to have a look at the RCR metric in Table 5. The higher the number in a cell in this table, the higher the priority for the rule to be guilty. The same thing occurs with $R1$ and $R3$, so we need to check their RCR metric for constraint $C7$. After checking both tables, the error tracking process for constraint $C7$ should follow the sequence of rules: $R4$, $R8$, $R7$, $R5$, $R6$, $R3$, $R1$.

Metric RC can be used to check whether the constraints offer a full coverage for the complete transformation or not. RC tables (e.g., Table 4) are to be consulted by columns: if all the values in a column are 0 or close to 0, it is very likely that the rule represented by such a column is not covered by any constraint. The RC metric is also useful for identifying the constraint that is more probably aligned with a certain rule. As with the CC metric, the higher the value in a cell is, the more likely the constraint represented by such a cell will cover the rule. When there is a draw in this table within a column, the corresponding cell in the RCR table should then be consulted.

3.6 Putting the Approach into Context

Once we have an approach to automatically locate the rules of a model transformation that may be the cause of a faulty behavior, it is very important to clarify how and where this approach fits in the overall process of a model transformation development [3]. As in the construction of any other software artifact, we should start with its specification. We believe that specifications should be defined in a modular and iterative manner: it is impossible to build the specifications of any complex system or artifact, and assume they will be complete, accurate and correct without testing them too. This is why in our context we use Tracts, because they allow the software engineer to focus on particular scenarios of special interest, and then build the specifications modularly and progressively.

In turn, the implementation of the model transformation can be built at the same time as the Tracts, or once they have been developed, depending on whether we want the specifications to guide the implementation or just to document its expected behavior. Although in theory the former approach is better, in practice implementations are developed at the same time as specifications (or even before), by different teams, and with different usages in mind (mostly analysis in the case of specifications, and execution in the case of implementations). This is particularly true in the case of specification methods that use precise and formal notations, and require specialized skills.

Once the specifications and the implementation are in place, the debugging process starts [9]. In our view, formal specifications and implementation should be debugged at the same time, assuming that both are complex artifacts and therefore potential subject to errors. The first step would be to discard as soon as possible all small mistakes (in one or the other) in a quick and cost-effective manner, something that can be done with the aid of the appropriate tools [11], before diving into more expensive and complex tests (such as model checking, formal validation, dynamic tests, etc.). And this is precisely where our approach represents a valuable asset.

The first step is to check, using the a-priori applicability test (Section 5.4), if our approach will work with the transformation. In the case it is amenable to be analyzed with it, it is a matter of building the matching tables with our toolkit.

The next step is to execute the transformation with the input models provided by the Tract test suites, using the TractsTool environment. In case a constraint is not fulfilled, our tool will provide the list of ATL rules that may have caused the faulty behavior, ordered according to the chances they have of being blamed. The developer can then look for errors on these rules, until one that can explain the constraint violation is found. But it may also be the case that the specifications are wrong, as it is often the case when they have not been tested before (cf. [24]). In any case, what we have now is a tool that is able to uncover, in a quick and easy manner, many of the errors that happen during the early stage of the testing process, and to help locate the rules that cause the faults.

This process will continue until the transformation works, respecting all the Tracts defined for it, which means that the implementation works for (at least) all the constraints and conditions that specify (at this level) its behavior. Now will be the moment to start going through a more detailed and thorough testing phase, that will help uncover more subtle errors in the transformation—but at a most expensive cost, both time and resource-wise.

4 Implementation

In order to extract the footprints of constraints and rules, as well as to build the matching tables, having

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering

automation support is essential because this is a rather complex and error-prone task, especially in the case of large model transformations.

4.1 Footprint extraction from OCL constraints

The first step is to extract the footprints for each OCL constraint. This is achieved by using the API of the USE (UML based Specification Environment) tool [25].

Firstly, we translate the input and output metamodels to the USE representation by means of a model-to-text transformation. As both the Ecore and the USE meta-metamodels are similar, the translation is straightforward. The relevant differences between both languages are the requirement that all relationships must be bidirectional in USE, and its lack of packages. Furthermore, USE only accepts one metamodel and one model, so we have to merge the input and output metamodels. This limitation implies the need to modify the name of each class and association in order to guarantee unique names. We have done so by adding a prefix to the name of the element: `src_` if it belongs to the source metamodel, and `trg_` if it belongs to the target metamodel.

Once both metamodels have been merged into a single file, we add to it the OCL expressions that compose the constraints and load the file into USE. For every OCL expression, USE builds a parse tree representing each subexpression with an explicit node which also provides the return type for each subexpression. To take advantage of this, we have built a small program that uses the aforementioned API. This API allows navigation through the parse tree and extracts the relevant information about the footprints, as explained in Section 3.3.

4.2 Footprint extraction from ATL rules

Apart from the extraction of the footprints for some ATL-specific elements, such as input pattern or output pattern, which is quite straightforward, the extraction of footprints from ATL transformations is conceptually mostly analogous to the OCL constraints, since it mainly affects the OCL expressions of the ATL rules. However, in practice the footprint extraction process for ATL rules is rather challenging because the USE tool cannot be used directly. First, there are some subtle but significant differences between the versions of OCL used by USE and by ATL concerning predefined types and operations. Second, the OCL expressions allow to reference variables which are bound by the rules. Thus we decided to implement a *Higher-Order Transformation* (HOT)[1] [26], [27] for extracting the footprints from the ATL rules, because the cost of building and maintaining two individual tools (one for ATL and one for OCL) was less than for developing one common tool. The precise footprint extraction process for ATL rules is described in detail in [8], [28].

1. According to [26], "*a HOT is a model transformation such that its input and/or output models are themselves transformation models*".

4.3 Matching Function

Once we have the types and features used in the constraints and the rules, we apply the matching functions to obtain the measures explained in Section 3.4.

These functions have been implemented in Java and each one calculates the values of the corresponding matching table. The output of the computation is represented in a *csv* (*comma-separated value*) format, so that it can be read by spreadsheet-based applications.

5 Evaluation

In this section, we discuss the accuracy and limitations of our approach, and introduce a method to check if a transformation is amenable to be used with it, based on the concept of *footprint similarity matrix*. To evaluate the accuracy of our approach we performed a case study [29] by following the guidelines for conducting empirical explanatory case studies by Roneson and Hörst [30]. In particular, we report on applying our approach to detect the alignments between Tracts and ATL transformations for four different transformation projects. In addition, we also present the results of a controlled experiment for locating faults in faulty transformations by applying mutations to the four different transformation projects.

5.1 Research Questions

The study was performed to quantitatively assess the completeness, correctness, and usefulness of our approach when applied to a real-world scenario. More specifically, we aimed to answer the following research questions (RQs):

1) *RQ1—Correctness*: Are the detected alignments between constraints and rules correct in the sense that all reported alignments are representing real alignments? If our approach reports incorrect alignments, what is the reason for this?
2) *RQ2—Completeness*: Are the detected alignments complete in the sense that all expected alignments are correctly detected? If the set of detected alignments is incomplete, what is the reason for missed alignments?
3) *RQ3—Usefulness*: In those cases where more than one alignment is reported for a constraint or a rule, are the correctly identified alignments outperforming the falsely identified alignments in terms of the calculated similarity value? We provide this additional question, because the first two questions only consider the evaluation of alignments as true/false, but they do not take the weights of the alignments into account.

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering

5.2 Case Study Design

Before we present the results of our case study, we elaborate on its design.

5.2.1 Requirements

As appropriate inputs we require transformation projects that consist of a set of constraints and a set of rules. We also need the source and target metamodels in order to extract the footprints of constraints and rules. Apart from these artifacts, we further require the alignments between the constraints and the rules given by transformation engineers; otherwise, we would not be able to compare the results obtained by our approach with the expected correct set of alignments. To accomplish an appropriate coverage of different scenarios, the transformations should comprise different intrinsic properties, e.g., having different design complexity measures.

5.2.2 Setup

We analyzed the alignments between transformation requirements and implementations in four different real-world transformation projects.

First, and as already presented in Section 3.4, we selected the transformation project dealing with the generation of Entity Relationship (ER) Diagrams from UML Class Diagram Models (*UML2ER* for short).

Second, we selected a transformation project that deals with behavioral models. Models conforming to CPL (Call Processing Language) [31] are transformed into models conforming to SPL (Session Processing Language) [32]. The *CPL2SPL* transformation [33] is a relatively complex example available from the ATL zoo (http://www.eclipse.org/atl/atlTransformations).

Third, we considered a model transformation project that does not operate on modeling languages but rather on markup languages. More specifically, we considered the *BT2DB* transformation of BibTeX documents into DocBook documents, also available from the ATL zoo. BibTeXML is an XML-based format for the BibTeX bibliographic tool. DocBook, in turn, is an XML-based format for document composition.

Finally, we experimented with a very large transformation called Ecore2Maude (or *E2M* for short) which is used by a tool called e-Motions [34]. It converts models conforming to the Ecore metamodel into models that conform to the Maude [35] metamodel, in order to apply some formal reasoning on them afterwards.

Tables 6 and 7 summarize the main size metrics for the ATL transformations and the corresponding metamodels.

We developed the Tracts for the given transformations. Constraints were written by a member of our team who knows OCL but who was unaware of the ATL implementations. They have been written based on the textual specification of the transformations. For example, the *UML2ER* case study comprises 10 constraints (previously shown in Listing 4) of two different kinds: one for comparing the number of instances of certain source and target classes, and one for checking equivalent elements based on containment relationships and value correspondences. There are 16 constraints in the *CPL2SPL* case study, checking that the proper object types in SPL are created from specific object types in CPL. Furthermore, they check that the number of objects in the target model is correct, and that the URIs are correctly created. The 16 constraints in the *BT2DB* case study make sure that the proper book is created for the different possible entries in BibTeX, and that all entries are properly transformed. Finally, for the *E2M* case study, three kinds of constraints have been developed, to check that the number of elements in the output model is correct, that the *Operation* entities in the output model have been created from the appropriate input elements, and that from each *Class* entity, the corresponding *Sort* has been created in the target model.

The input data including the Tracts constraints, the ATL transformations, the alignments between them, the results and the accuracy of these four projects (and several others) are available on our project's website [8].

TABLE 6
Transformation Metrics Overview.

Metric	*UML2ER*	*CPL2SPL*	*BT2DB*	*Ecore2Maude*
ATL LoC	77	348	286	1397
#Elements	86	497	449	2403
#Links	201	1114	1052	5270
#Rules	8	15	9	40
#Helpers	0	6	4	40
#Bindings	5	73	25	329

TABLE 7
Metamodel Metrics Overview.

Metric	*UML*	*ER*	*CPL*	*SPL*	*BT*	*DB*	*Ecore*	*Maude*
#Class	4	8	31	77	21	8	18	45
#Atts	3	1	42	33	10	1	31	17
#Refs	4	2	16	62	2	5	34	46
#Inhs	3	6	32	76	31	4	16	38

5.2.3 Measures

To assess the accuracy of our approach, we compute the *precision* and *recall* measures originally defined in the area of information retrieval [36]. In the context of our study, precision denotes the fraction of *correctly detected* alignments among the set of *all detected* alignments (i.e., how many detected alignments are in fact correct). Recall indicates the fraction of *correctly detected* alignments among the set of *all actually occurring* alignments (i.e., how many alignments have not been missed). These two measures may also be thought of

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering

TABLE 8
Expected alignments for the *UML2ER* transformation
("×" means direct relation, "(×)" means relation via inheritance).

	R1	R2	R3	R4	R5	R6	R7	R8
C1	(×)	×						
C2	(×)	×	×					
C3	×		×	×	(×)	(×)	(×)	(×)
C4	×	(×)	(×)	(×)	(×)	(×)	(×)	(×)
C5	(×)	×						
C6	(×)	×	×					
C7	(×)		×	×	(×)	(×)	(×)	(×)
C8	(×)		×	(×)	×			
C9	(×)		×	(×)		(×)	×	
C10	(×)		×	(×)		(×)		×

as probabilities: the precision denotes the probability that a detected alignment is correct and the recall is the probability that an actually occurring alignment is detected. Thus, both values range from 0 to 1.

Precision is used to answer RQ1 and recall to answer RQ2. There is a natural trade-off between precision and recall. Thus, these two metrics may be further combined inside the so-called *f-measure* to avoid having only isolated views on both aspects [36]. To answer RQ3, we use the *utility-average* metric, which serves to reason about the relative difference between false positives and true positives for one row (in the CC and RCR tables) or for one column (in the RC and RCR tables).

To check whether or not our approach is accurate for a given model transformation and a given set of constraints, we have manually obtained the alignments between rules and constraints, reflected in a table called *expected alignment table*. An example is shown in Table 8 for the UML2ER transformation. There is a cross mark, ×, in the cells where there is a direct alignment between constraints and rules, and a cross mark in brackets, (×), when the alignment is due to inheritance relationships between meta-classes or transformation rules (cf. Section 3.4). The value of empty cells is 0.

For computing precision and recall, we extract the true-positive values (TPs), false-positive values (FPs) and false-negative values (FNs), with the help of the expected alignment table. A cell contains a TP when (i) its value is above the threshold, (ii) there is an alignment in the expected alignment table, and (iii) the alignment is also identified in the RCR table for the same cell (in the case of CC and RC tables, see Section 3.4). There is an FP when our approach identifies that there is an alignment (CC/RC and RCR cell values above the threshold), but the expected alignment table does not indicate so. Finally, there is an FN between a constraint and a rule when our approach identifies that there is no alignment between them and there is a mark in the equivalent cell in the expected alignment table.

From the TP, FP and FN values we compute the *precision*, *recall* and *f-measure* metrics as follows:

$$precision = \frac{TP}{TP + FP} \tag{4}$$

$$recall = \frac{TP}{TP + FN} \tag{5}$$

$$f\text{-}measure = 2 \times \frac{precision \times recall}{precision + recall} \tag{6}$$

The utility-average metric permits reasoning about the relative value difference between FPs and TPs. For example, if there are five alignments in a row in the CC table and four of them are falsely created (which means that there is only one TP and four FPs), but the TP has the highest value, then the four FPs are disregarded because the TP is the first one checked. We have calculated this metric by rows for the CC metric and by columns for the RC metric. The result is the mean of the values obtained in each row/column. As for the RCR metric, since it can be consulted by columns or by rows, we have considered both situations. The utility-average metric, *UAM*, is computed as follows.

$$UAM = \frac{\sum_{i=1}^{n} u_i}{n} \tag{7}$$

where $u_i = 1$ if there are neither FNs nor FPs in the row/column, or there are no FNs and the value of all FPs is less than the value of the TPs; $u_i = 1 - \frac{|F|}{|F|+|TP|}$ if there are no FNs but there are FPs which are bigger than or equal to at least one of the TPs in the row/colum (in the formula, TP is the set of all true positives in the row/column, and $F = \{x \in \text{FP} \mid \exists y \in \text{TP with } x >= y\}$); finally, $u_i = 0$ if there are FNs in the row/column.

5.3 Results

We now present the results of applying our approach to the four different model transformation projects. A summary of these results is shown in Table 9. Detailed results can be found on our project's website [8]. In the matching tables (e.g., Tables 3 to 5), TPs are shown in normal font, FPs within square brackets, and FNs within curly brackets. These values are obtained by comparing the *expected alignment* tables for the four projects, with the matching tables obtained by our approach.

As shown in Table 9, the values obtained for the precision, recall and f-measure metrics are quite acceptable in three of the projects: *UML2ER*, *CPL2SPL* and *Ecore2Maude*. With these accuracy results, we can conclude that our approach works well with these projects, since the alignments found statically are quite reliable. Recall is acceptable in all projects, because the number of FNs is low. However, the number of FPs is very high in the *BT2DB* project,

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering

TABLE 9
Accuracy of case studies.

Metric	UML2ER	CPL2SPL	BT2DB	Ecore2Maude
TPs	46	37	29	11
FPs	9	9	85	3
TNs	-	1	3	-
Precision	0.84	0.80	0.25	0.79
Recall	1.00	0.97	0.91	1.00
F-measure	0.91	0.88	0.40	0.88
Utility average	0.80	0.81	0.60	0.94

TABLE 10
Similarity Matrix for the Rules in *UML2ER*.

	R8	R7	R6	R5	R4	R3	R2	R1
R1	0	0	0	0	0	0	0	1
R2	0	0	0.2	0	0	0.25	1	
R3	0	0	0.25	0	0.2	1		
R4	0.33	0.25	0.14	0.2	1			
R5	0.2	0.17	0.11	1				
R6	0.14	0.13	1					
R7	0.25	1						
R8	1							

resulting in a poor precision (0.25). The reasons for this low performance are discussed in next.

5.4 A-priori Applicability Test

After carefully studying the model transformation that scored a low precision of our approach, we discovered that the footprints of its rules were very similar, i.e., they shared many types and features. This led us to introduce a new measure, based on the concept of *footprint similarity matrix* for model transformation rules. A similarity matrix gives us an indication of how rules are related with each other, i.e., the factor of common types they share. The similarity matrix for the *UML2ER* case study is shown in Table 10.

Evidently, similarity matrixes are symmetric. To compute the aggregated application indicators, we extract the mean and the standard deviation of the rule similarities. The lower both values are (especially the mean), the fewer types and features the rules have in common, and thus, the higher is the chance for a successful application of our approach.

For the matrix shown in Table 10, both metrics have a value of 0.1. This means that rules are *separated enough*, and thus our approach works well because there is no confusion possible when establishing the alignments between the constraints and the rules.

However, the similarity matrix for the *BT2DB* transformation shows quite different values. The mean is 0.41 and the standard deviation is 0.24. Consequently, it is difficult to distinguish among them when looking for the "guilty rule", and this results in the occurrence of many false positives in the matching tables. If we look at the ATL transformation, we find the explanation for such a high value. Since the target metamodel is rather small, many rules create objects of the same target types. For example, 8 rules out of 9 create Paragraph elements, and 33% of the rules contain a TitleEntry element in their input part.

We have automated the process for obtaining the types of any ATL transformation, as well as the computation of the similarity matrixes. With this, we have obtained the similarity matrixes for the transformations in the ATL zoo, in order to investigate the applicability of our approach. Out of the 41 model transformations studied, the mean and standard deviation turned out to be below 0.15 in 21 of them, which means that our approach is perfectly fit for use with around half of the transformations. A summary of these results is available in a technical report [28], while all similarity matrixes obtained, as well as the software that computes them, are available on our project's website [8]. The threshold that we used for the mean and the standard deviation of the similarity matrix, 0.15, is to ensure that precision is above 0.8.

It is important to note that this fitness test ensures good results (since the transformation rules are separated enough to be distinguishable by our proposed approach), but it may be that the fitness test scores low and still our approach works well because of the way in which the constraints are written. In any case, there is no guarantee that our approach is fit for use when the applicability test provides results below 0.15.

We also discovered that the number of rules in the transformations has no impact in the applicability of our approach. In fact, the number of rules used in the set of transformations studied ranged from 3 up to 40. As an example, the similarity matrix of a small transformation (*PetriNet2PathExp*, 3 rules) gave bad results, while the one obtained from the largest transformation (*Ecore2Maude*, 40 rules) gave good results. Contrarily, we obtained adverse results for another large transformation (*R2ML2XML*, 55 rules), while we got good results for small transformations (such as *PetriNet2Grafcet*, 5 rules). We have applied the *Pearson correlation coefficient*, a measure of the linear correlation between two variables, on the results, when the first variable is the number of rules in the transformations and the second is the mean obtained from the similarity matrixes. The obtained value was -0.13, meaning that this dependence is minimal.

5.5 Experimenting with Faulty Transformations

So far, we have illustrated our approach with correct model transformations. However, given that it has been devised to detect errors in faulty transformations, it is essential to test its effectiveness when the transformations are indeed faulty.

Setup. For this reason we have used mutation analysis [37] to systematically inject faults into model transformations [38], and then used our approach to locate the bugs. The purpose of a mutated transformation is to emulate a transformation that contains bugs, and then see if our approach detects them.

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering

TABLE 11
Possible Mutations for ATL Transformations (from [39]).

Concept	Mutation Operators	Concept	Mutation Operators
Matched Rule	Addition Deletion Name Change	Filter	Addition Deletion Condition Change
In/Out Pattern Element	Addition Deletion Type Change Name Change	Binding	Addition Deletion Feature Change Value Change

TABLE 12
Summary of mutations and fault localization results (CPL2SPL project).

Mutation	Constraints Violated	Guilty Rule Located?	Number of Steps
CPL2SPL_1	C1	✓	1
	C2	✓	1
	C3	✗	-
	C11	✓	1
CPL2SPL_2	C4	✓	1
CPL2SPL_3	C5	✓	1
	C6	✓	1
	C14	✓	1
CPL2SPL_4	C12	✓	1
CPL2SPL_5	C15	✓	2
CPL2SPL_6	C5	✓	3
	C13	✓	3
CPL2SPL_7	C10	✓	1

To define the possible mutations of ATL transformations, we use the list of transformation change types presented in [39], which are summarized in Table 11. For more information on the precise mutations and the results obtained for the case studies presented in this paper we kindly refer to [40].

Example. As an example, we have applied the following mutations for the *CPL2SPL* transformation mentioned above:

1) Addition of an OutPatternElement in $R1$, which results in the creation of unexpected additional elements in the target model.
2) Modification of the feature of a binding in $R3$, resulting in incorrectly initialized features in the target model.
3) Modification of the condition of the filter in $R5$, changing the amount of produced target model elements.
4) Modification of a binding and addition of OutPatternElement in $R6$, thus producin more target model elements.
5) Deletion of a binding and an OutPatternElement, along with its binding, in $R8$; emulating the circumstance in which a transformation produces not enough target elements.
6) Addition of a filter in $R9$, making the application of the rule more restricted, thus creating less elements in the target model.
7) Feature modification in a binding and deletion of a binding in $R11$, resulting in wrongly assigned values and missing values in the target model.

Measures For each mutation, we collect: (i) the constraints violated when the mutation is applied; (ii) if the user was able to find the guilty rule using our approach; and (iii) the number of steps needed for finding the guilty rule or not. By number of steps we mean the number of rules that the user needs to check in order to find the one that was mutated (including that one).

Results. The results in Table 12 show that all mutations were detected by our approach for the given example. Each mutation caused one or more constraints to fail, and the guilty rule was correctly identified for all constraints but one (C3). This happened because of false negatives, given that the relation between rule *CPL2SPL_1* and constraint C3 was quite loose. However, the mutation caused several constraints to fail and our approach was able to identify the mutated rule in the rest of the cases, so the guilty rule was eventually identified.

The overall results obtained for all four projects, described in our technical report [40], show similar effectiveness. We injected a total of 21 mutations, causing 48 constraints to fail. All mutants were killed, i.e., all guilty rules were correctly identified by our approach. Only for three constraints that failed we could not identify the rule causing it but, in all cases, these rules caused the violation of several constraints, and the guilty rule was already identified as the one responsible for the violation of a different constraint that failed with the same mutation, such is the case with C3 in *CPL2SPL_1*, so the guilty rule was eventually identified. Regarding how many rules need to be checked before identifying the guilty one, our proposed approach needed an average of 1.78 rules to be checked.

5.6 Threats to Validity

In this subsection, we elaborate on several factors that may jeopardize the validity of our results.

Internal validity—Are there factors which might affect the results of this case study? The quality of the data appearing in the matching tables, as well as the usefulness and accuracy of these, are crucial for the internal validity due to three main factors. First, the Tracts need to be manually defined. If they do not contain valuable restrictions, then the matching tables are not useful. Defining constraints is not a trivial task, and the person responsible for doing so needs to have knowledge of OCL, of the transformation to check, and of what should be checked. Second, the way in which footprints are extracted is crucial

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering

for building the tables. As explained in Section 3.3, there may be very long navigation paths expressed in OCL both in the Tracts and in the rules. From them, we extract the types and features discarding some elements because they are not considered as relevant by giving a higher priority to the results than to the paths used in the computations. Third, in order to study the accuracy of our tables, we have manually defined the expected alignment tables. Should we have failed to properly identify these alignments, the value of precision and recall would have been incorrectly calculated. In any case, they were written by a member of the team and double-checked by another, in order to minimize this risk. We have also made some assumptions in the implementation of our approach. For instance, we have chosen 0.1 as the threshold value for considering alignments relevant, as mentioned in Section 3.4. We also decided not to take constants and primitive types into account (Section 3.3). Although our experiences have shown that these decisions seem to be correct, they need to be further validated with more experiments and case studies. Fourth, different styles of Tracts definition may have an effect on the outcomes. As mentioned in Section 5.2.2, the Tracts constraints were written by a member of our team. Of course, if they had been written by other people, or by the developers of the transformations themselves, the results presented here may have been slightly different. Here we assumed the underlying hypothesis that the constraints and rules are more heterogenous if they are developed by different persons, thus resulting in a more difficult matching problem. Finally, concerning the experiment with faulty transformations, we relied on the state-of-the-art of mutation operators for model transformations, but further operators may be required in the future to deal with more fine-grained OCL expression mutations. Thus, these additional operators may have an impact on the results gained in our experiments.

External validity—To what extent is it possible to generalize the findings? As a proof of concept of our approach, we have extracted the matching tables for model transformations written in the ATL language. The metamodel of ATL comprises, amongst others, a package for OCL. Currently, the footprint extraction operates on this representation, and thus, works only for ATL transformations. Nevertheless, it would be possible to reuse parts of the ATL footprint extraction for other rule-based transformation languages that also integrate OCL as a sublanguage. Another threat to external validity would be considering further features of model transformations, such as reflection [41]. Finally, our studies are focussing for out-place transformation scenarios, and thus, additional studies are needed for in-place transformation scenarios. As part of our future work we plan to investigate these issues, and also try to define a minimal set of requirements on the kinds of specification notations and implementation languages which are amenable to be directly addressed by our approach.

6 Related Work

With respect to the contribution of this paper, three threads of related work are discussed: (i) general traceability approaches in software engineering as well as specific approaches for tracking "guilty" transformation rules, i.e., those whose behavior violates the transformation specifications, (ii) approaches for generating test cases for model transformations, and (iii) approaches that build on model footprints as does our approach.

6.1 Tracing Faults in Model Transformations

IEEE [42] defines traceability as the degree to which a relationship between two or more artifacts can be established. Most tracing approaches are dedicated to establishing traceability links between artifacts that are in a predecessor/successor relationship with respect to their creation time in the software development process, e.g., between requirements, features, design, architecture, and code. Our approach for automatically finding the alignments between constraints and transformation rules is in the spirit of traceability rules as presented in [43], [44]. A survey dedicated to traceability in the field of MDE is presented in [45], where the possibilities of using trace links established by model transformations are discussed. However, this survey does not report on tracing approaches between transformation specifications and implementations.

Tracking guilty transformation rules using a dynamic approach, i.e., by executing the model transformation under testing, has been subject to investigations. Hibberd et al. [9] present forensic debugging techniques for model transformations based on the trace information of model transformation executions for determining the relationship between source elements, target elements, and the transformation logic involved. With the help of such trace information, it is possible to answer debugging questions implemented as queries. In [46], we used OCL-based queries for the backwards debugging of model transformations using an explicit runtime model based on the trace model between the source and target models. Aranega et al. [47] present an approach for locating transformations errors by also exploiting the traces between the source and target models. The dynamic approach is also used in [48] to build slices of model transformations and in [49] following a white-box testing approach. A complementary approach to model transformation testing has been proposed by Kessentini et al. [50], using a generic oracle function. The idea of this approach is that the traces between the source and target models of a transformation should be similar to existing example traces. Specifically, the oracle

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering

function checks how large a derivation there is of the generated traces of a model transformation from existing traces in the example base. While all these approaches track transformation rules using specific test input models, our aim is to statically build more general traceability models between transformations' specifications and their implementations (the pros and cons of dynamic vs. static approaches have already been discussed in Section 3.1).

In addition to Tracts, other approaches have been proposed that build on the notion of transformation contracts to specify transformation specifications [24]. While other OCL-based specification approaches, e.g., [51], are obviously supported by the approach presented in this paper, for non OCL-based approaches, e.g., [52], additional transformations for computing the metamodel footprints may be developed or these specifications may be internally translated to OCL to reuse the existing footprint computation. Analogously, if other transformation implementation languages such as RubyTL [53], ETL [54], or QVT [55] need to be supported, additional higher-order transformations like those for ATL, need to be developed.

There are some other transformation testing approaches that directly annotate assertions inside transformation implementations [56], [57]. Thus, these approaches have no need to compute the alignments between the specification and the implementation, as they are already provided by the transformation engineer. However, the specification and implementation of the transformation is intermingled, and thus, specifications are specific to a certain transformation implementation.

There are several approaches that define contracts for model transformations by defining a set of input/output model pairs and employing model comparison techniques to look for differences between the expected output models (provided by the engineer) and the actual outputs of the transformation [58], [59]. In this context, basic support for a failure trace is provided, since the different elements (added, updated, and deleted elements) between an actual target model and an expected target model may be calculated, but the tracing to the corresponding source model elements as well as to the transformation rules is left open.

6.2 Test Generation for Model Transformations

For tracking guilty rules, the availability of appropriate test input models is assumed in our approach. Many research efforts have been investigated in software engineering, in this area including black-box, gray-box and white-box approaches.

Only Küster et al. [60] and Gonzalez & Cabot [61] focus on white-box methods. In the former, the existence of a high-level design of model transformations, consisting of conceptual transformation rules, is assumed. In the latter, a white-box based testing approach for ATL transformations is provided by extracting OCL constraints and using a model finder to compute test input models fulfilling certain path conditions.

Many approaches have been proposed for black-box testing, whereby test source models are generated either on the basis of the source metamodel (e.g. [14], [62], [63]) or on the basis of specified requirements [64], [65]. For the actual test source model generation, most of these approaches rely on constraint satisfaction, e.g., by means of SAT solvers. Furthermore, an approach has been proposed, which allows automatically completing test input models, i.e., the transformation engineer has to specify an intention by a defining model fragment, only, and an algorithm complements this fragment for a valid test input model [66].

In our approach, we provide two ways of defining a test suite in addition to its manual definition. First, a semi-automatic approach is supported by defining model generation script on basis of the declarative ASSL language [21]. Second, we employ the model finder component present in the USE environment [67], [68] which is based on relational logic and KodKod to produce test input models based on the given input metamodels and source constraints by following a similar strategy as presented in [65].

6.3 Model Transformation Footprinting

Recently, some approaches for computing and utilizing model footprints have been presented. In [69], the footprints of model operations are statically computed by introducing the idea of metamodel footprints. We pursue this idea of computing metamodel footprints from transformation specifications and implementations for establishing traceability links instead of reasoning solely on model footprints. Mottu et al. [70] compute the input metamodel footprints for ATL transformations in order to slice the input metamodels as a prerequisite step for computing test input models for the transformations being studied with Alloy. Compared to our work, the work of Mottu et al. is orthogonal in the sense that their approach could complement ours. While we focus on fault localization, Mottu et al. are concerned with test model generation.

7 Conclusions and Future Work

In this paper we have presented a static approach to trace errors in model transformations. Taking as input elements an ATL model transformation and a set of constraints that specify its expected behavior, our approach automatically extracts the footprints of both artifacts and compares transformation rules and constraints one by one, obtaining the overlap of common footprints. Subsequently, it returns three

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering

matching tables where the alignments between rules and constraints are recorded. By using these tables, the transformation engineer is able to trace the rules that can be the cause of broken constraints due to faulty behavior.

Our evaluation shows that the presented approach is expected to be accurate for a large set of model transformations. By using the similarity matrixes, an automated and instant fitness test is available to check a-priori whether the approach will be helpful for a given transformation. Several executables of our approach are available on our website [8].

For future work we aim to explore the usage of similarity matrixes for other use cases, such as to reason about the maintainability of transformations in the case of evolving metamodels or to reason about the completeness of transformations. Furthermore, the footprint extraction for transformation contracts in OCL is currently supported, but other contract languages such as [52] may be employed as well. Similarly, the application of the ideas presented here to other transformation languages which do not use OCL, like graph-based languages (e.g. AGG) or other kinds of languages (e.g. Tefkat) opens the way to further lines of research. Finally, we would like to explore how dynamic approaches could complement our static approach for tracing guilty transformation rules.

ACKNOWLEDGMENTS

This work has been supported by Spanish Project TIN2011-23795, by Austrian Research Promotion Agency (FFG) under grant 832160 and by the EC under ICT Policy Support Programme (grant no. 317859).

REFERENCES

[1] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, ser. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[2] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. L. Traon, and J.-M. Mottu, "Barriers to systematic model transformation testing," *Communications of the ACM*, vol. 53, no. 6, pp. 139–143, 2010.

[3] E. Guerra, J. de Lara, D. S. Kolovos, R. F. Paige, and O. Santos, "Engineering model transformations with transML," *Software and Systems Modeling*, vol. 12, no. 3, pp. 555–577, 2012.

[4] R. L. Sedlmeyer, W. B. Thompson, and P. E. Johnson, "Knowledge-based fault localization in debugging," *Journal of Systems and Software*, vol. 3, no. 4, pp. 301–307, Dec. 1983.

[5] M. Gogolla and A. Vallecillo, "*Tract*able model transformation testing," in *Proc. of ECMFA'11*, ser. LNCS, vol. 6698. Springer, 2011, pp. 221–236.

[6] B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. L. Traon, "Model transformation testing challenges," in *Proc. of IMDD-MDT'06*, 2006.

[7] E. Cariou, R. Marvie, L. Seinturier, and L. Duchien, "OCL for the specification of model transformation contracts," in *Proc. of the OCL and Model Driven Engineering Workshop*, 2004.

[8] L. Burgueño, M. Wimmer, J. Troya, and A. Vallecillo, "Fault Localization in Model Transformations," 2014, http://atenea.lcc.uma.es/index.php/Main_Page/Resources/FaultLocMT.

[9] M. Hibberd, M. Lawley, and K. Raymond, "Forensic debugging of model transformations," in *Proc. of MODELS'07*, ser. LNCS, vol. 4735. Springer, 2007, pp. 589–604.

[10] L. Burgueño, M. Wimmer, and A. Vallecillo, "Towards tracking "guilty" transformation rules: a requirements perspective," in *Proc. of the 1st Workshop on the Analysis of Model Transformations (AMT) @ MoDELS*. ACM DL, 2012, pp. 27–32.

[11] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 240–253, 2006.

[12] L. Baresi, K. Ehrig, and R. Heckel, "Verification of model transformations: A case study with BPEL," in *Proc. of TGC'06*, ser. LNCS, vol. 4661. Springer, 2007, pp. 183–199.

[13] H. Ehrig, K. Ehrig, J. D. Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay, "Termination criteria for model transformation," in *Proc. of FASE'05*, ser. LNCS, vol. 3442. Springer, 2005, pp. 49–63.

[14] K. Ehrig, J. M. Küster, and G. Taentzer, "Generating instance models from meta models," *Software and Systems Modeling*, vol. 8, no. 4, pp. 479–500, 2009.

[15] J. M. Küster, "Definition and validation of model transformations," *Software and Systems Modeling*, vol. 5, no. 3, pp. 233–259, 2006.

[16] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara, "Verification and validation of declarative model-to-model transformations through invariants," *Journal of Systems and Software*, vol. 83, no. 2, pp. 283–302, 2010.

[17] K. Anastasakis, B. Bordbar, and J. M. Küster, "Analysis of model transformations via Alloy," in *Proc. of MODEVVA'07*, 2007.

[18] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 31–39, 2008.

[19] B. Meyer, "Applying design by contract," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[20] M. Wimmer and L. Burgueño, "Testing M2T/T2M transformations," in *Proc. of MODELS'13*, ser. LNCS, vol. 8107. Springer, 2013, pp. 203–219.

[21] M. Gogolla, J. Bohling, and M. Richters, "Validating UML and OCL Models in USE by Automatic Snapshot Generation," *Software and Systems Modeling*, vol. 4, no. 4, pp. 386–398, 2005.

[22] J. Cabot and E. Teniente, "Transforming OCL constraints: a context change approach," in *Proc. of SAC'06*. ACM, 2006, pp. 1196–1201.

[23] M. Wimmer, S. Martínez, F. Jouault, and J. Cabot, "A catalogue of refactorings for model-to-model transformations," *Journal of Object Technology*, vol. 11, no. 2, pp. 1–40, 2012.

[24] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, and L. Hamann, "Formal specification and testing of model transformations," in *Formal Methods for Model-Driven Engineering (SFM)*, ser. LNCS, vol. 7320. Springer, 2012, pp. 399–437.

[25] M. Richters and M. Gogolla, "OCL: Syntax, semantics, and tools," in *Object Modeling with the OCL*, ser. LNCS, vol. 2263. Springer, 2002, pp. 42–68.

[26] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, "On the Use of Higher-Order Model Transformations," in *Proc. of ECMDA-FA'09*, 2009.

[27] D. Varró and A. Pataricza, "Generic and meta-transformations for model transformation engineering," in *Proc. of UML'04*, ser. LNCS, vol. 3273. Springer, 2004, pp. 290–304.

[28] J. Troya, L. Burgueño, M. Wimmer, and A. Vallecillo, "Footprints Extraction and Similarity Matrixes in ATL Transformations," Tech. Rep., 2013, http://atenea.lcc.uma.es/Descargas/TechReportExtATL.pdf.

[29] A. S. Lee, "A scientific methodology for MIS case studies," *MIS Quarterly*, vol. 13, no. 1, pp. 33–50, 1989.

[30] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[31] J. Lennox, X. Wu, and H. Schulzrinne, "Call Processing Language (CPL): A language for user control of internet telephony services," 2004, http://www.ietf.org/rfc/rfc3880.txt.

[32] L. Burgy, C. Consel, F. Latry, J. Lawall, N. Palix, and L. Reveillere, "Language technology for internet-telephony service creation," in *Proc. of ICC'06*. IEEE, 2006, pp. 1795–1800.

[33] F. Jouault, J. Bézivin, C. Consel, I. Kurtev, and F. Latry, "Building DSLs with AMMA/ATL, a Case Study on SPL and

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2375201, IEEE Transactions on Software Engineering

CPL Telephony Languages," in *Proc. of ECOOP Workshop on Domain-Specific Program Development*, 2006.

[34] J. E. Rivera, F. Durán, and A. Vallecillo, "A Graphical Approach for Modeling Time-Dependent Behavior of DSLs," in *Proc. of VL/HCC'09*. IEEE, 2009, pp. 51–55.

[35] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude – A High-Performance Logical Framework*, ser. LNCS. Springer, 2007, vol. 4350.

[36] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[37] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[38] J.-M. Mottu, B. Baudry, and Y. L. Traon, "Mutation analysis testing for model transformations," in *Proc. of ECMDA-FA*, ser. LNCS, vol. 4066. Springer, 2006, pp. 376–390.

[39] A. Bergmayr, J. Troya, and M. Wimmer, "From Out-Place Transformation Evolution to In-Place Model Patching," in *Proc. of ASE'14*. ACM, 2014, accepted for publication.

[40] J. Troya, L. Burgueño, M. Wimmer, and A. Vallecillo, "Mutations in ATL Transformations and their Identification with Matching Tables," Tech. Rep., 2014, http://atenea.lcc.uma.es/Descargas/MTB/Mutations/TechReport.pdf.

[41] I. Kurtev, "Application of Reflection in Model Transformation Languages," in *Proc. of ICMT'08*, ser. LNCS, vol. 5063. Springer, 2008, pp. 199–213.

[42] IEEE, *Standard glossary of software engineering terminology*, IEEE Std. 610.12, 1990.

[43] B. Ramesh and V. Dhar, "Supporting systems development by capturing deliberations during requirements engineering," *IEEE Transactions on Software Engineering*, vol. 18, no. 6, pp. 498–510, 1992.

[44] F. A. C. Pinheiro and J. A. Goguen, "An object-oriented tool for tracing requirements," *IEEE Software*, vol. 13, no. 2, pp. 52–64, 1996.

[45] I. Galvão and A. Goknil, "Survey of traceability approaches in model-driven engineering," in *Proc. of EDOC'07*. IEEE, 2007, pp. 313–326.

[46] M. Wimmer, G. Kappel, J. Schönböck, A. Kusel, W. Retschitzegger, and W. Schwinger, "A Petri Net based debugging environment for QVT Relations," in *Proc. of ASE'09*. IEEE, 2009, pp. 3–14.

[47] V. Aranega, J.-M. Mottu, A. Etien, and J.-L. Dekeyser, "Traceability mechanism for error localization in model transformation," in *Proc. of ICSOFT'09*. INSTICC Press, 2009, pp. 66–73.

[48] Z. Ujhelyi, Á. Horváth, and D. Varró, "Dynamic backward slicing of model transformations," in *Proc. of ICST'12*. IEEE, 2012, pp. 1–10.

[49] C. A. González and J. Cabot, "ATLTest: A White-Box Test Generation Approach for ATL Transformations," in *Proc. of MoDELS'12*, ser. LNCS, vol. 7590. Springer, 2012, pp. 449–464.

[50] M. Kessentini, H. A. Sahraoui, and M. Boukadoum, "Example-based model-transformation testing," *Automated Software Engineering*, vol. 18, no. 2, pp. 199–224, 2011.

[51] E. Cariou, N. Belloir, F. Barbier, and N. Djemam, "OCL contracts for the verification of model transformations," *ECEASST*, vol. 24, 2009.

[52] E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger, "Automated verification of model transformations based on visual contracts," *Automated Software Engineering*, vol. 20, no. 1, pp. 5–46, 2013.

[53] J. Sánchez Cuadrado, J. García Molina, and M. Menárguez Tortosa, "RubyTL: A practical, extensible transformation language," in *Proc. of ECMDA-FA'06*, ser. LNCS, vol. 4066. Springer, 2006, pp. 158–172.

[54] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The Epsilon Transformation Language," in *Proc. of ICMT'08*, ser. LNCS. Springer, 2008, vol. 5063, pp. 46–60.

[55] OMG, *Meta Object Facility (MOF) 2.0 Query/View/Transformation. Version 1.1*, Object Management Group, 2011.

[56] P. Giner and V. Pelechano, "Test-driven development of model transformations," in *Proc. of MODELS'09*, ser. LNCS. Springer, 2009, vol. 5795, pp. 748–752.

[57] A. Ciancone, A. Filieri, and R. Mirandola, "MANTra: Towards model transformation testing," in *Proc. of QUATIC'10*. IEEE, 2010, pp. 97–105.

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

[59] A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige, and I. Medina-Bulo, "EUnit: A unit testing framework for model management tasks," in *Proc. of MODELS'11*, ser. LNCS, vol. 6981. Springer, 2011, pp. 395–409.

[60] J. M. Küster and M. Abd-El-Razik, "Validation of Model Transformations: First Experiences Using a White Box Approach," in *Proc. of of MODELS'06 Workshops*, ser. LNCS, vol. 4364. Springer, 2006, pp. 193–204.

[61] C. A. González and J. Cabot, "ATLTest: A White-Box Test Generation Approach for ATL Transformations," in *Proc. of of MODELS'12*, ser. LNCS, vol. 7590. Springer, 2012, pp. 449–464.

[62] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. L. Traon, "Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool," in *Proc. of ISSRE'06*. IEEE, 2006, pp. 85–94.

[63] S. Sen, B. Baudry, and J.-M. Mottu, "Automatic Model Generation Strategies for Model Transformation Testing," in *Proc. of ICMT'09*, ser. LNCS, vol. 5563. Springer, 2009, pp. 148–164.

[64] P. Giner and V. Pelechano, "Test-driven development of model transformations," in *MODELS'09*, ser. LNCS, vol. 5795. Springer, 2009, pp. 748–752.

[65] E. Guerra, "Specification-driven Test Generation for Model Transformations," in *Proc. of ICMT'12*, ser. LNCS, vol. 7307. Springer, 2012, pp. 40–55.

[66] S. Sen, J.-M. Mottu, M. Tisi, and J. Cabot, "Using models of partial knowledge to test model transformations," in *Proc. of ICMT'12*, ser. LNCS, vol. 7307. Springer, 2012, pp. 24–39.

[67] M. Kuhlmann, L. Hamann, and M. Gogolla, "Extensive Validation of OCL Models by Integrating SAT Solving into USE," in *Proc. of TOOLS'11*, ser. LNCS, vol. 6705, 2011, pp. 290–306.

[68] M. Kuhlmann and M. Gogolla, "From UML and OCL to Relational Logic and Back," in *Proc. of MODELS'12*, ser. LNCS, vol. 7590, 2012, pp. 415–431.

[69] C. Jeanneret, M. Glinz, and B. Baudry, "Estimating footprints of model operations," in *Proc. of ICSE'11*. ACM, 2011, pp. 601–610.

[70] J.-M. Mottu, S. Sen, M. Tisi, and J. Cabot, "Static analysis of model transformations for effective test generation," in *Proc. of ISSRE'12*. IEEE, 2012, pp. 291–300.

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.