Revisiting Fault Localization Techniques for Model Transformations: Towards A Hybrid Approach

Paula Muñoz*, **Javier Troya***, **Manuel Wimmer**†, **and Gerti Kappel**§
*ITIS Software, Universidad de Málaga, Spain
†CDL-MINT, Johannes Kepler University Linz, Austria
§TU Wien, Austria

ABSTRACT The correctness of software built through model transformations highly depends on the correctness of these transformations. Different approaches have been proposed to ensure the correctness of model transformations by checking if pairs of input-output models satisfy a set of contracts. If a contract is not satisfied, at least one transformation rule must contain a bug. Localizing the rules that contain bugs is key for repairing the model transformation. Among others, Spectrum-Based Fault Localization (SBFL) is a dynamic technique to locate the faulty component of a software, and it has already been applied in the context of model transformations considering the rules as the components. As a result, this technique proposes an order (a so-called suspiciousness ranking) in which the rules should be inspected in order to locate the bug. However, SBFL relies on so-called suspiciousness formulae that were created in different domains, so none of them offers a perfect behavior in the context of model transformations. Indeed, some of the rankings for model transformations present many ties, so the tester is uncertain as of which rule to inspect first in the ties. In this paper, we explore how SBFL can be combined with static information in a hybrid approach in order to improve the results obtained from SBFL, specially in the case of ties in the rankings. Our evaluation shows the potential of the hybrid approach to improve previous SBFL results for model transformations.

KEYWORDS Model transformations, Testing, Debugging, Fault localization.

1. Introduction

In Model-Driven Engineering (MDE) (Brambilla et al. 2017), models are considered as first-class artifacts to represent complex systems from various viewpoints and abstraction levels utilizing appropriate modeling languages. Consequently, model transformations are seen as the cornerstone of MDE (Czarnecki & Helsen 2006; Lúcio et al. 2014; Heckel & Taentzer 2020). They represent the core technique for manipulating and creating models and provide an excellent trade-off between theoretical foundations and applicability to different problem domains (Lúcio et al. 2014; Heckel & Taentzer 2020). Existing model transformation languages provide dedicated language concepts, such as transformation rules, to realize model transformations often in a declarative way. In this context, rules are matching input elements from the source models and either produce parts of new target models (so-called out-place model transformations) or drive the evolution, i.e., modification of the source models (so-called in-place model transformations) (Czarnecki & Helsen 2006).

As model transformations are also the core mechanism to translate models to implementations realized in code—potentially applying several transformation steps, the correctness of software systems realized with MDE approaches highly depends on the correctness of the utilized model transformations. Therefore, it is critical to test and debug model transformations in the context of MDE (Troya et al. 2022) as it is the case with compilers for programming languages (Chen et al. 2020). However, several challenges are associated with testing and debugging model transformations as they are specific and complex software systems (Baudry et al. 2006).

JOT reference format:
Paula Muñoz, Javier Troya, Manuel Wimmer, and Gerti Kappel. *Revisiting Fault Localization Techniques for Model Transformations: Towards A Hybrid Approach*. Journal of Object Technology. Vol. 21, No. 4, 2022. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2022.21.4.a7

The correctness of a model transformation can be checked, for instance, against a set of assertions (also called *contracts*) (Gogolla & Vallecillo 2011; Vallecillo & Gogolla 2012) that the pairs of $< input, output >$ models must satisfy. These contracts act as oracle. Thus, if one of the associated contracts of a model transformation fails, a bug must be present in the model transformation implementation. It is common to represent these contracts in the Object Constraint Language (OCL) (Object Management Group 2014). OCL has become a key component of many MDE techniques, and it provides high expressiveness. Regarding the definition of OCL contracts, it can be observed that the conditions of some of them are included in other contracts for the same model transformation (Burgueño et al. 2015). However, the impact this inclusion of conditions in different contracts has in testing and debugging model transformations has not been studied.

When an OCL contract, that we assume is correct, fails for a model transformation, we need to locate the bugs that caused it fail. Burgueño et al. (Burgueño et al. 2015) proposed a static fault localization approach based on the extraction of metamodel footprints from the contracts and the transformation implementation for locating the bug. In contrast, Spectrum-Based Fault Localization (SBFL) for model transformations (Troya et al. 2018; Li et al. 2020; Du et al. 2020) is a dynamic approach that requires input models and to execute the model transformation for locating the bug. Both approaches rank the transformation rules according to the likelihood that they contain the bug. The tester should follow the rank order when inspecting the rules for finding the bug in the most efficient way.

A limitation of both approaches is that it is quite likely that more than one rule is ranked on the same position, i.e., there is a tie in the rank (Wong et al. 2016), even for the first position. Thus, the tester is uncertain as of which rule to inspect first. In fact, the number of ties can be quite large in some cases, and there is not a clear pattern for the number of ties as shown by previous work (Troya et al. 2018).

The goal of this paper is three-fold. First, we aim to study whether the way of implementing a model transformation is related to the number of ties obtained in the ranks. In particular, we aim at studying how the inheritance relationships among transformation rules affect the ties. Second, we aim at breaking the ties computed by pure SBFL approaches by applying the static information in addition for the ranking process leading to a hybrid framework, i.e., a combination of static and dynamic information is used for the ranking process. Finally, we reason about whether the way of specifying the contracts (Gogolla & Vallecillo 2011; Vallecillo & Gogolla 2012) has an influence on how fast the tester can locate the bugs following the rules' ranks. In particular, we study if the order in which the contracts are checked against the model transformation influences the debugging process. These three aspects can be summarized in the research question: *To what extend can static information enhance the effectiveness of SBFL techniques for locating faults in model transformations?* In order to answer it, we analyse and compare against the results of a previous SBFL approach for model transformations (Troya et al. 2018), whose replication package is available online.

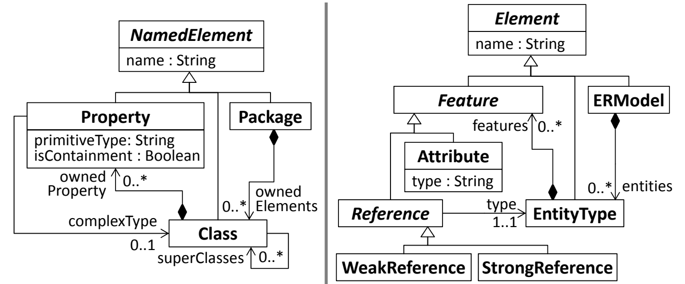

Figure 1 UML and ER metamodels for the UML2ER transformation (taken from (Burgueño et al. 2015))

The remainder of this paper is organized as follows. Section 2 provides the running example and the background of this work, i.e., model transformation and fault localization, as well as discusses open challenges in this field. Section 3 presents a hybrid approach which combines SFBL and static information as well as allows to improve the contract specifications by proposing a new language for building networks of OCL contracts. Section 4 evaluates the hybrid approach while Section 5 discusses related work. Finally, Section 6 concludes the paper with an outlook on future research lines.

2. Background and Running Example

This section explains the background of this work. First, we introduce a running example for this paper which has been used also in previous works on model transformation research (Wimmer et al. 2012; Burgueño et al. 2015; Troya et al. 2018). Then, we summarize two selected approaches for dynamic and static fault localization which we will utilize later on in the hybrid approach, and finally, discuss open challenges of these existing fault localization approaches to motivate the need for a hybrid approach.

2.1. Running Example

As running example we select a model transformation taken from the structural modeling domain: *UML2ER*. The transformation produces Entity Relationship (ER) diagrams from reading UML Class Diagrams as input. This transformation is originally proposed in (Wimmer et al. 2012), and we have considered the extended version proposed in (Burgueño et al. 2015). The model transformation implementation in ATL is shown in Listing 1, and the source and target metamodels are displayed in Figure 1. The aspect to highlight in this model transformation for the context of this work is the extensive use of rule inheritance, also since the metamodels involved in the transformation are utilizing inheritance relationships. If $Ri < Rj$ means that Ri inherits from Rj, then we have $R8, R7 < R6$; $R6, R5 < R4$; $R4, R3, R2 < R1$. As we will discuss later, the presence of inheritance relationships may worsen the results of SBFL techniques.

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

--R1
abstract rule NamedElement{
```

```
  from s : SimpleUML!NamedElement
  to   t : ER!Element(name <- s.name) }

 --R2
rule Package extends NamedElement{
  from  s: SimpleUML!Package
  to    t: ER!ERModel (
            entities <- s.ownedElements) }

--R3
rule Class extends NamedElement{
  from s: SimpleUML!Class
  to   t: ER!EntityType (
          features <- s.ownedProperties) }

--R4
abstract rule Property extends NamedElement{
  from s: SimpleUML!Property
  to   t: ER!Feature () }

--R5
rule Attributes extends Property{
  from s: SimpleUML!Property (not s.primitiveType.
                              oclIsUndefined())
  to   t: ER!Attribute (
          type <- s.primitiveType) }

--R6
abstract rule References extends Property{
  from s: SimpleUML!Property (not s.complexType.
                              oclIsUndefined())
  to   t: ER!Reference (
          type <- s.complexType) }

--R7
rule WeakReferences extends References{
  from s: SimpleUML!Property (not s.isContainment)
  to   t: ER!WeakReference }

--R8
rule StrongReferences extends References{
  from s: SimpleUML!Property (s.isContainment)
  to   t: ER!StrongReference }
```

Listing 1 *UML2ER* ATL Transformation (taken from (Burgueño et al. 2015))

As mentioned before, since the oracle for checking the correctness of a model transformation is a set of OCL contracts, we need these contracts as mandatory input for checking the transformation. Listing 2 displays several contracts for the UML2ER model transformation to give some concrete contract examples. Please note that we use the notation proposed in (Burgueño et al. 2015): a black triangle symbol is introduced for reasons of brevity. It is used for defining the position within a constraint (see triangle down symbol) which can be extended by another constraint (which is represented with a triangle up symbol). In our example, constraint C1 is extended constraint C2. Please note that this notation was informally introduced for readability purposes. We will pick this notation up for this paper and frame it in dedicated language support in Section 3.

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

-- C8: C2 + Property2Attribute + Nesting
C2[▲] and class.ownedProperty->forAll(p|p.primitiveType.
    oclIsUndefined() 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.
    oclIsUndefined() 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.
    oclIsUndefined() implies entity.features->select(f|f.
    oclIsTypeOf(Reference))->one(f|f.name=p.name and not p
    .isContainment implies f.oclIsTypeOf(StrongReference))
    )
...
```

Listing 2 Tracts for the *UML2ER* model transformation (excerpt taken from (Burgueño et al. 2015))

There is support for checking the satisfaction of such sets of OCL contracts in the pairs of <input,output> models after a model transformation has been executed. One example is the *TractsTool* (Atenea Research Group 2013), and another is the *OCL Classic SDK*[1] which is part of the *Eclipse Modeling Tools*. With the help of these tools, failing contracts are reported which are the input for the subsequent fault localization process.

2.2. Fault Localization Approaches for Model Transformations

Our work builds on two existing fault localization approaches for model transformations. As categorized in a recent survey on model transformation testing and debugging (Troya et al. 2022), these are *spectrum-based fault localization* (Troya et al. 2018) and *footprint-based fault localization* (Burgueño et al. 2015). While the former is a dynamic approach, i.e., the model transformation needs to be executed, the latter is static, meaning that the model transformation is not executed, but the transformation definition is directly analyzed. We introduce both of them in the following.

2.2.1. Spectrum-Based Fault Localization. As explained in (Wong et al. 2016), spectrum-based fault localization (SBFL) is a technique to estimate the likelihood of each component of a program, e.g., statement, method, etc., of containing bugs. For this purpose, SBFL uses the results of test cases and their associated code coverage information. This results in the creation of a program *spectrum* that details the execution information of the program from a certain perspective, such as branch or statement coverage (Harrold et al. 2000).

Troya et al. presented an approach to apply SBFL to model transformations (Troya et al. 2018)—other authors also focused on this technique recently (Li et al. 2020; Du et al. 2020). As explained in (Troya et al. 2018), the approach requires test cases that comprise input models for exercising the model transformation. Table 1 depicts an illustrative example showing how SBFL is applied in the model transformation of our running example. In the first column, the table shows the 8 transformation rules of Listing 1, where a bug has been introduced in rule R2. The introduced bug makes the second contract in Listing 2 fail. The table shows 10 test cases for checking the correctness of constraint C2: $<tc_{02}, tc_{12}, \ldots, tc_{92}>$—each OCL contract

[1] https://eclipse.org/modeling/mdt/downloads/?project=ocl

Table 1 Tarantula (Jones & Harrold 2005) suspiciousness values for the *UML2ER* model transformation when contract C2 fails

T. Rule	tc_{02}	tc_{12}	tc_{22}	tc_{32}	tc_{42}	tc_{52}	tc_{62}	tc_{72}	tc_{82}	tc_{92}	N_{CF}	N_{UF}	N_{CS}	N_{US}	N_C	N_U	Susp	Rank
R1	•	•	•	•	•	•	•	•	•	•	9	0	1	0	10	0	0.5	3
R2 (**BUG**)	•		•	•	•	•	•	•	•	•	9	0	0	1	9	1	1	1
R3	•	•	•	•	•		•		•	•	7	2	1	0	8	2	0.44	7
R4	•	•	•	•	•	•	•	•	•	•	9	0	1	0	10	0	0.5	3
R5	•	•		•	•	•	•	•	•	•	8	1	1	0	9	1	0.47	6
R6	•	•	•	•	•	•	•	•	•	•	9	0	1	0	10	0	0.5	3
R7	•						•				2	7	0	1	2	8	1	1
R8	•	•			•		•	•	•		5	4	1	0	6	4	0.36	8
Test Result	F	S	F	F	F	F	F	F	F	F								

that fails for at least one test case must be analyzed individually. A cell is marked with "•" if the transformation rule of the column has been exercised by the test case of the row. This evaluation process produces a so-called *coverage matrix* (Abreu et al. 2007). The final row contains the *error vector* detailing the result (either successful ("S") or failed ("F")) for each test case. In our example, we can see that only one test case tc_{12} is successful and all the others are failing. Therefore, since there is at least one test case where C2 fails, C2 is violated.

Based on the coverage matrix and error vector, a multitude of different SBFL formulae have been presented in order to rank the program statements, i.e., in our context transformation rules, regarding their *suspiciousness*, i.e., the probability of being faulty (Troya et al. 2018). The suspiciousness value is always in the range [0,1]. It is assumed that low values indicate a low probability of containing a fault and high values indicate a high probability. The suspiciousness value for each transformation rule in Table 1 is displayed in the "Susp" column, which in this case has been computed by the well-known fault localization technique *Tarantula* (Jones & Harrold 2005). The suspiciousness value of each rule, according to this technique, is computed as $(N_{CF}/N_F)/(N_{CF}/N_F + N_{CS}/N_S)$, where N_{CF} is the number of failing test cases that cover the rule, N_F is the total number of failing test cases, N_{CS} is the number of successful test cases that cover the rule, and N_S is the total number of successful test cases—for more information the interested reader is referred to (Wong et al. 2016; Troya et al. 2018) for detailed explanations. This yields to a ranking according to each rule's suspiciousness value, as shown in the last column, where top-ranked transformation rules have a higher probability to hold the fault. In the example, the rule that contains the bug is ranked first, but tied with R7.

In order to evaluate how effective the produced rankings are, the EXAM score is proposed (Yu et al. 2008; Xie et al. 2013). It computes the percentage of statements in a program that has to be examined until the first faulty statement is reached, i.e., in our context,

$$EXAM_{Score} = \frac{Number\ of\ rules\ examined}{Total\ number\ of\ rules}$$

An important aspect to consider is that suspiciousness techniques can provide the same value for different elements. Thus, these elements are tied at the same position in the ranking, e.g., rules R2 and R7 in Table 1. These ties need to be broken in order to decide which rule to inspect first, but no technique has been proposed to break the ties in the context of model transformations yet. Thus, in order to measure the effectiveness when there are ties, the faulty rule is inspected first in the best-case scenario. In contrast, in the worst-case scenario it is inspected last. There is also the average-case scenario, which considers both the best and worst cases. In our example, assuming that the rule R2 is examined in second place (worst-case scenario), the EXAM score of *Tarantula* would be $\frac{2}{8} = 0.25$, i.e., 25% of the rules have to be inspected in order to locate the bug.

As we can see by the formula, the possible values for the EXAM score depend on the number of rules of the model transformation under test, which goes in the denominator of the formula. In our example, the best EXAM score we could expect is $\frac{1}{8} = 0.125$, which is the case in which the buggy rule is examined first, i.e., the tie is broken in favor of the buggy rule in our example.

2.2.2. Footprint-Based Fault Localization. The work by Burgueño et al. (Burgueño et al. 2015) focuses on a static approach for locating buggy rules in model transformations. OCL contracts (Gogolla & Vallecillo 2011; Vallecillo & Gogolla 2012), such as those presented in Listing 2, are also used here as oracle. By extracting the *footprints*, i.e., metamodel elements, used in the contracts and in the model transformation rules, matching functions are constructed to automatically generate alignments between model transformation specifications (OCL contracts) and implementations (transformation rules). Such alignments are key ingredients for further interpreting the test results, i.e., the result of the contracts

Table 2 Matching tables for contract C2 in the *UML2ER* model transformation, taken from (Burgueño et al. 2015).

	R1	R2	R3	R4	R5	R6	R7	R8
CC	0.2	0.6	0.4	0	0	0.4	0	0
RC	0.25	0.5	0.5	0	0	0.33	0	0
RCR	0.11	0.38	0.29	0	0	0.22	0	0

evaluation. As example of footprints, the ones extracted from contract C2 in Listing 2, we have `[NamedElement, Package, Package.name, Package.ownedElements, Class, Class.name, Element, ERModel, ERModel.name, ERModel.entities, EntityType, EntityType.name]`. Regarding rule `Package` (R2) in Listing 1, its footprints are `[Package, Package.ownedProperties, ERModel, ERModel.entities]`.

In summary, the input for this approach is the set of OCL contracts that have failed as a result of a testing process (using for instance the tools mentioned in Section 2.1). The first step is to extract the footprints of the model transformation specification and implementation, i.e., of every failing OCL contract and for every transformation rule. Then, the extracted footprints are compared for each contract and rule combination. The percentage of footprint overlaps is analyzed in order to build the so-called matching tables. There are three types of matching tables, as explained later—please refer to (Burgueño et al. 2015) for a deeper explanation. Finally, these matching tables must be interpreted and the final outcome is a ranking of the rules for each failing OCL contract. The order of the rules indicates the likelihood of containing the fault. Like in SBFL, the effectiveness of these rankings can be computing using the EXAM score as explained before.

Three different matching tables are constructed after the footprints are extracted, each one providing a certain viewpoint on the alignment (Burgueño et al. 2015):

– The *constraint coverage* (CC) metric focuses on constraints, so tables obtained for it are to be consulted by rows.
– The *rule coverage* (RC) metric focuses on rules, so tables obtained for it are to be read by columns.
– The *relatedness* of *constraints* and *rules* (RCR) metric is related to both constraints and rules, so the table can be consulted by rows and columns.

Table 2 unites the values of the three matching tables for contract C2 of our running example. Each row displays the results of a different matching table for this contract. Without going into much detail (the interested reader is referred to (Burgueño et al. 2015) for details), we can see that contract C2 must be related to R2 according to the three rows, since cell `[C2,R2]` contains the highest value of the footprints alignment (in the case of the RC formula, it is tied with `[C2,R3]`).

2.3. Open Challenges

Troya et al. (Troya et al. 2018) compared the two approaches previously explained. For this, they used the same four model transformations, the same mutants and the same set of contracts in order to check which of the two approaches offers a better EXAM score. Since the results in (Burgueño et al. 2015) of the matching tables are not expressed in terms of suspiciousness rankings and EXAM scores, Troya et al. (Troya et al. 2018) adapted the results to be expressed in these terms. They discovered that both approaches result in ties among several rules in the different suspiciousness rankings.

The inheritance relationships among model transformation rules is one of the reasons for having ties in the rankings. Imagine we have, for instance, $R3 < R2 < R1$ (notation explained in Section 2.1) in a model transformation and rule $R3$ is executed. This means that, implicitly, $R2$ and $R1$ are executed too. If we assume to have an error in one of the three mentioned rules, it is not possible to differentiate between them based on the suspiciousness rankings as all three rules will have the same suspiciousness value. The presence of ties in the rankings is problematic, and it yields bad results for the EXAM score in the worst-case scenario, and different numbers in this score for the best- and worst-case scenarios. Ideally, the EXAM score in the best- and worst-case scenarios should be the same, meaning there are no ties. To the best of our knowledge, no technique has been studied in the literature to break ties in SBFL for model transformations. This leads us to the first challenge.

Challenge 1: Breaking ties in SBFL rankings. In fact, having a look at the suspiciousness rankings of the four best techniques for SBFL (namely *Kulcynski2, Mountford, Ochiai* and *Zoltar* as reported in (Troya et al. 2018)), we can see that all of them have ties for our running example composed by 18 mutants applied to 14 contracts, making 252 scenarios (every contract is checked against every mutant), in which we find 42, 47, 42, and 39 ties respectively.

Another aspect that has not been considered in approaches for SBFL or those based on footprints is the order in which failing OCL contracts are inspected in order to repair the model transformation. As we have seen in our running example (cf. Section 2.1), the OCL contracts defined for the UML2ER model transformation present some dependencies among them. For instance, if C1 is failing, also many other contracts such as C2 and C3 are failing. This brings us to the second challenge.

Challenge 2: Establishing an order in failing OCL contracts to inspect. We hypothesize that the order in which OCL contracts are inspected and the corresponding rules corrected has an influence in the overall effort for model transformation repair. Checking the most general OCL contracts first, considering the dependency on the other contracts, may help pinpoint the errors in the rules more precisely.

3. A Hybrid Fault Localization Framework for Model Transformations

We now describe two novel solutions to help in the model transformation debugging process. First, we introduce an advanced contract language which provides more structure during the definition and execution of contracts. Second, we provide a hybrid framework which combines the usage of dynamic and static information for fault localization in model transformations.

3.1. Overview

In Figure 2, we give an overview of the approach proposed in this paper. The approach extends existing work on spectrum-based fault localization for model transformations (Troya et al. 2018) by providing two main extensions (green colored components in the figure). First, the model transformation specification is no longer just a set of OCL contracts, but a layered network of contracts which allows a more detailed analysis during the debugging process by ranking the failing contracts themselves. Second, the rankings of suspicious rules produced by SBFL for model transformation are post-processed by a dedicated component which analyses static information of the model transformation specification and implementation (the footprints approach described in Section 2.2.2 and other static information). Based on this information, re-rankings can be performed and, in particular, ties, i.e., suspicious rules having the same ranks, may be resolved by prioritizing a particular rule from the ties. The following two subsections will present both extensions in more detail.

3.2. Extended Contract Language

In this subsection, we introduce an extended contract language for model transformation specification which adds additional structures to OCL-based constraints. With the help of this language, improved contracts can be defined which serve as input for the fault localization process as shown on the left-hand side of Figure 2. In particular, the language is inspired by best practices for software testing such as the prepare-do-check approaches[2] which separate different computations needed for testing a certain contract into different phases. Furthermore, the language is providing two additional features for defining OCL contract networks. First, it allows to define dependencies between constraints, i.e., a constraint has as precondition the satisfaction of other constraints in order to be evaluated. Second, the contracts may define concrete extension points, i.e., hooks, in order to allow their extension by other contracts. The metamodel of the new language for constraints is shown in Figure 3. As can be seen in the figure, the contract concept is still the central one, but now we allow in addition to have source and target queries to collect elements needed for formulating the condition to be check in dedicated steps. Furthermore, we have the hook concept introduced explicitly as well as the dependencies between the contract, for the last point see the reflexive relationship.

In addition to writing contracts in a more structured and concise way, the new language also contributes to the separation of different concerns which are for instance tested at once as done in Listing 1. By having the dependencies and hooks for the contracts, these concerns can be more easily separated. The benefit is that if a contract fails, the cause of having the failing contract is more clear as each contract is now only considering one particular concern. Furthermore, the language allows to provide for more guidance by first executing contracts which are concerned with the first levels of the containment hierarchy of a model, e.g., check first that a package is correctly transformed into a model before one reasons about the classes contained in a package, or that the elements are already available before their container correspondence is validated. This also helps to speed-up the testing processes as we aim to find a failing contract fast and do not have to consider depending ones at all. Furthermore, the new contracts also allow for more efficient execution as results between the contracts can be easily cached. In the following, we demonstrate how the contracts from Listing 1 are now defined and improved in the extended contract language as shown in Listing 3.

```
-- C1
-- SRC Elements
var packages = Package.allInstances;
-- TRG Elements
var models = ERModel.allInstances;
-- CHECK
packages ->forAll(p|models->one(m|p.name=m.name H1));

-- C2_1 depends on C1
-- SRC Elements
var classes = packages.ownedElements -> flatten();
-- TRG Elements
var entities = models.entities -> flatten();
-- CHECK
classes ->forAll(c|entitites->one(e|c.name=e.name));

-- C2_2 depends on C2_1
-- CHECK extends C1.H1
and p.ownedElements -> forAll(c|m.entities -> one(e|e.name=
     c.name H2)

-- C3_1 depends on C2_2
-- SRC Elements
var properties = classes.ownedProperty -> flatten();
-- TRG Elements
var features = entities.features -> flatten();
-- CHECK
properties -> forAll(p|features->one(f|p.name=f.name));

-- C3_2 depends on C3_1
-- CHECK extends C2_2.H2
and c.ownedProperty -> forAll(p|e.features -> one(f|f.name=
     p.name ))
```

Listing 3 Tracts for the *UML2ER* model transformation

```
-- C2 footprints
[NamedElement, NamedElement.name, Package, Package.name,
     Package.ownedElements, Class, Class.name, Element,
     Element.name, ERModel, ERModel.name, ERModel.entities,
      EntityType, EntityType.name]

-- C2_1 footprints
[NamedElement, NamedElement.name, Class, Class.name,
     Element, Element.name, EntityType, EntityType.name]

-- C2_2 footprints
[NamedElement, NamedElement.name, Package, Package.
     ownedElements, Class, Class.name, Element, Element.
     name, ERModel, ERModel.entities, EntityType,
     EntityType.name]
```

Listing 4 Footprints for Extended Conctract Language

If we revisit the contracts specified in Listing 1, there are several possibilities to provide improved specifications. Let us start with contract C1. In Listing 3, C1 is defined as in Listing 2, however, we make use of the preparation source and target queries before we check the condition which should hold between the source and target elements. For C2, we first of all split the contract into the two main parts: (i) checking for name equivalences of classes and entities, and (ii) checking the containment relationship, i.e., the elements have to be in the respective containers. For the first part, we make use of constraint C2_1 which depends

[2] Unit testing best practices supported by C# https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices.

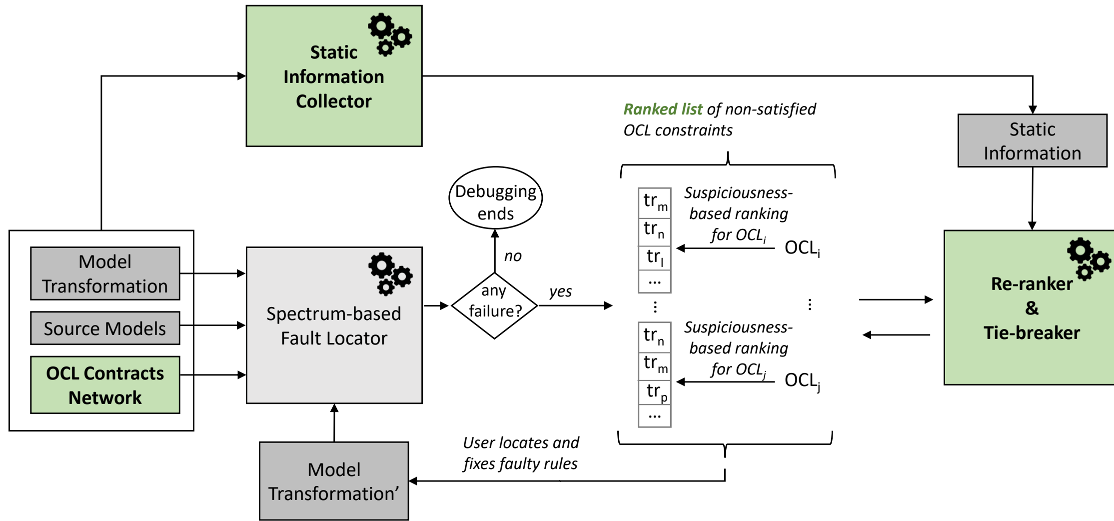

Figure 2 Overview on Hybrid Fault Localization Approach (extension of the version in (Troya et al. 2018), new or adapted components shown in green)

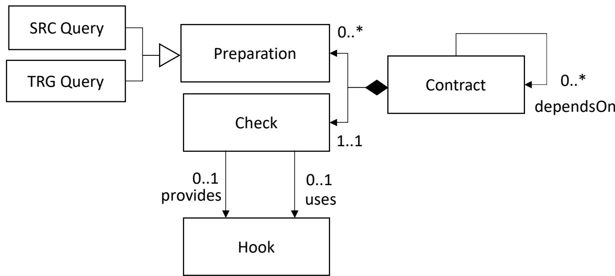

Figure 3 Extended Contract Language for Model Transformations

on C1 and which is making use of the variables defined in C1 to compute the set of classes and entities in the preparation part before we check for the name equivalences. Contract C2_2 is depending on C2_1 and is making use of the hook H1 provided in C1. For C3 of Listing 1, we make the analogous changes to improve its definition by using two subcontracts and making use of the newly introduced contract language.

Furthermore, we would like to discuss the benefits of this extended constraint language. First, the dedicated constraints for checking for existence and for containment in separation result also in more specific footprints, as shown in Listing 4. Second, the reuse of variables between the contracts allow for speed-up of checks as not for every contract all model elements have to be queried again and again. Third, the failing contract can be returned fast, others may be executed in the back to provide further rankings of failing contracts or are skipped based on user preferences.

Finally, to consider the impact of using the preparation/check pattern and the separation of concerns principle for contracts on the extracted footprints used for fault localization, let us perform a short example. Let us focus on the extraction of the footprints out of the original OCL contracts as presented in Section 2 with the footprints concerning the reformulation in the extended contract language. In particular, we focus on the contracts C2 and C3 as they are reformulated by two subcontracts each for checking containment and equivalence in separation. C2_1 is now more related to R3, while C2_2 is more related to R2. Actually, this allows for a better fault localization, as R2 is responsible for the containment and R3 is responsible for generating the equivalent target elements from the source elements.

3.3. Break Ties in Suspiciousness Rankings

As explained in Section 2.2.1, the ranks provided by SBFL techniques[3] can have ties. A tie does not give any hint to the tester as to which transformation rule needs to be inspected first since it may recommend to check more than one rule with the same priority. Out of the 18 SBFL techniques studied in (Troya et al. 2018), we selected the four that provided better results, namely *Kulcynski2, Mountford, Ochiai* and *Zoltar*, for the context of this work.

In this paper, we propose two approaches for breaking the ties and guide the tester to the potentially buggy rule first. The first one involves using the footprints technique (Burgueño et al. 2015) explained in Section 2.2.2. The second one is related to the analysis of the inheritance between the transformation rules, because we noticed that most of the ties involved rules related by inheritance.

3.3.1. Using Footprints to Break Ties

The static information retrieved from the matching tables provides a value between 0 and 1 for every pair of contract and transformation rule, which indicates the likelihood of that constraint detecting a bug produced by the corresponding transformation rule. The higher this value, the greater the likelihood. There are three different calculation methods to obtain the matching-tables values, as presented in (Burgueño et al. 2015).

[3] We use the terms *SBFL techniques* and *suspiciousness formulae* indistinctly

In our approach, we will use these static values provided by the matching tables to break the ties when the suspiciousness formula used gives us the same value for two different transformation rules and study how this affects the final EXAM score.

For example, let us check the example in Table 3. The rows include the N values for all the transformation rules and the suspiciousness values are calculated using the Mountford (Wong et al. 2012) formula. With the result of the suspiciosness formula, we obtain the SBFL rank. The *rule coverage* (RC) is one of the static formulas which relate the corresponding rule with the constraint C1. In this example, we have two ties, one concerning the buggy rule. The suspiciousness value was calculated using the Mountford formula (Wong et al. 2012), but we would still get a tie by using any formula since all the N values, the ones used in all suspiciousness formulas, are all the same.

With this, we obtain a tie in the rank for positions 1 and 4. To solve these ties, we employ the static formula RC. In the highest ranked case, the value of RC for the buggy rule `NamedElement` is greater than the one for the rule `Package`, so we break the tie by assigning a higher ranking to the buggy rule. This way, we solve the tie and obtain a new ranking with the buggy rule first. We do the same for the second tie, obtaining the new ranking.

3.3.2. Using Inheritance Relationships to Break Ties

During our analysis of the obtained data, we realized that most of the ties involve two rules with some inheritance relationship between them. This happens because some part of the inheriting rule is the same as its parents, which means that we are implicitly executing the parent rule, so the N values can be the same.

Our approach consists in checking if there is a tie that involves more than two rules, where at least two of them are related by inheritance. If this is the case, we should break the tie by increasing the rank of the inheritance-related rules so that the tester checks those first.

For example, if we check again Table 3, we notice that the tie in rank one is between two rules related by inheritance: `Package` extends `NamedElement`. If we had the third rule in this tie, meaning that we would have three rules with rank 1, we would solve the tie by promoting `Package` and `NamedElement` to be the ones with rank one and demoting this third rule to the second position.

In the case of these ties with inheritance, the buggy rule may be in the super rule or the sub-rule, but it is needed to recommend the tester that they should check them both. This happens because the parent rule defines some behavior that is inherited by the sub-rule, and at the same time, the sub-rule may refine the behavior defined by the parent. This forces us to have to check both to find the bug.

4. Evaluation

In this section, we evaluate the novel parts introduced in the fault localization pipeline presented in the previous section. For this purpose, we perform an experiment for assessing the impact on the ranking of suspicious rules. Finally, we provide a critical discussion of the results.

4.1. Overview

The approach presented in Section 3.3 aims to resolve the ties when applying SBFL for model transformations. This section evaluates its effectiveness by showing the results we have obtained through the example of the UML2ER transformation, which was already analyzed in the previous work (Troya et al. 2018). In this way, we can compare the results obtained with the prior approach against the approach presented in this paper, demonstrating the effectiveness of the new approach.

The UML2ER example is particularly interesting to test our new process, because it contains many inheritance relationships between its transformation rules. Such relationships generate a significant number of ties in the suspiciousness rankings. This is why, during the analysis, we aimed to create a new version of this model transformation. We redefined the transformation rules by eliminating the inheritance relationships that exist between them. In this way, we re-analyze the existence of ties and re-evaluate the EXAM score for the new implementation.

In addition, to further optimize the use of the static approach, we defined a new method that aims to optimize the score obtained in the matching tables by considering the composition between contracts, e.g., by using hooks, and transformation rules, e.g., by using inheritance relationships.

In the following subsections, we discuss the results of these approaches, analyzing the number of ties we can solve, as well as the effect on the EXAM score, in order to answer the following research question.

4.2. Research Question

One of the challenges we aim to tackle with our proposal is improving the bug localization by complementing the Spectrum-based approach with static information (considering both, footprints and other transformation definitions such as inheritance relationships). With this, the research question that we intend to answer is: *To what extend can static information enhance the effectiveness of SBFL techniques for locating faults in model transformations?*

4.3. Study Setup

In our evaluation, we use the following three methods to define the same problem and analyze the obtained results:

Original. The input material used for the analysis is the same as used in the work of (Troya et al. 2018), enhanced with our approach to break ties. This will allow us to compare our results with the ones obtained as described in Section 2.2.1.

No hierarchy. In order to analyze the effect of using inheritance in the definition of the transformation rules and the existence of ties in the results of the Spectrum-based analysis, we have performed a flattening refactoring of the transformation rules by removing inheritance relationships.

For example, in the following Listing 5, we can see how the `Class` rule, inherited initially from `NamedElement`, is defined without using the latter. We performed equivalent changes for all rules, resulting in a smaller set of rules. The full set of transformations, and the complete analyses we have performed, are available in our project repository (Muñoz et al. 2022).

Table 3 Mountford (Wong et al. 2012) suspiciousness values for the UML2ER example when C1 fails for Mutant 10 (adapted from (Troya et al. 2018))

T. Rule	N_{CF}	N_{UF}	N_{CS}	N_{US}	N_C	N_U	Susp	Rank	RC	New Rank
WeakReferences	3	97	0	0	3	97	0.0206	6	0.4	6
References	6	94	0	0	6	94	0.0212	8	0.33	8
Attributes	13	87	0	0	13	87	0.0229	4	0.28	5
Class	29	71	0	0	29	71	0.0281	3	0.33	3
Package	100	0	0	0	100	0	1	1	0.67	2
NamedElement (**BUG**)	100	0	0	0	100	0	1	1	1	1
Property	13	87	0	0	13	87	0.0229	4	0.5	4
StrongReferences	4	96	0	0	4	96	0.0208	7	0.4	7

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

rule Class{
  from s: SimpleUML!Class
  to t: ER!EntityType (
     name <- s.name,
     features <- s.ownedProperty )
}
```

Listing 5 *UML2ER* ATL Transformation Flattened

Constraints and rules embedding. As we have mentioned in previous sections, in the UML2ER case study there is a large amount of composition among contracts and among transformation rules.

This means that if the most abstract contract fails, all those that also include the abstract contract as one of their conditions (composed with the *and* operator) will fail too. Therefore, we check the contracts from the less to the most concrete one. In this approach, we do not consider the conditions that are present in a previous contract when we perform the footprints extraction. This means that, for each contract, we only extract the footprints out of those conditions that are not present in any previous contract. The extended contract language presented before in Section 3 favors such analysis.

By doing this, we can optimize the scores obtained in the matching tables, focusing the affinity only on the conditions specific of each contract—let us recall the matching tables are used in this paper to break the ties. For example, let us consider contracts C1 and C2 in Listing 2. In C1, we would consider all their elements for extracting the footprints (i.e., `[NamedElement, NamedElement.name, Package, Package.name, Element, Element.name, ERModel, ERModel.name]`), and in C2, only the elements of the second condition of the contract (i.e., `[NamedElement, NamedElement.name, Package, Package.ownedElements, Class, Class.name, Element, Element.name, ERModel, ERModel.entities, EntityType, EntityType.name]`).

Concerning the rules, the inheritance relationships mean that some elements do not appear explicitly during the static analysis, i.e., the footprints for these elements are not extracted. For example, if we look at the `NamedElement` rule from which all the other rules inherit, it refers to the assignment of element names. Although this transformation rule is executed every time any other rule is executed, none of them reflects the effect they have in the `name` attribute in their footprint. Therefore, in our approach, we include the elements of the super-rule when collecting the elements of a sub-rule, so that the footprints or the former appear also in the footprints of the latter. This will further increase the affinity with the contracts that genuinely check all the elements specified in the transformation rule, even if the inheritance is involved. The embedding for `Class` rule is available in Listing 6, in which we can see how we add the footprints of the super-rule.

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

rule Class extends NamedElement {
  from s: SimpleUML!Class
  to   t: ER!EntityType (
          features <- s.ownedProperties)
}

NamedElement footprints: [NamedElement, NamedElement.name,
     Element, Element.name]

Class footprints: [NamedElement, NamedElement.name, Class,
     Class.name, Class.ownedProperty, Element, Element.name
     , EntityType, EntityType.name, EntityType.features],
```

Listing 6 Embedding for rule `Class`

To evaluate the effectiveness of the approaches applied, we will employ the following metrics:

Number of ties. We count the number of ties present in the Spectrum-based approach data to study the effect of defining the rules with and without inheritance.

Number of ties resolvable by static information. By counting the number of ties we can actually break with static information, we are able to determine the effect of applying the embedding method for footprints extraction and whether it improves error localization. Additionally, we can also determine the effectiveness of the static information for breaking the ties derived from the SBFL methods, and if it can improve the results obtained by such methods.

Number of ties resolvable by inheritance. Similarly to the previous metric, this one allows us to reason about how many of these ties have to do with the inheritance defined between rules.

EXAM score. This value will allow us to determine whether we have improved the rankings we suggest by breaking the ties (explained in Section 2.2). The EXAM score will be evaluated taking into account the three scenarios mentioned in Section 2.2.1: worst-case, best-case, average-case.

4.4. Results

The effect of problem definition on the number of ties. As we mentioned in Section 4.3, we defined three different methods to define the rules and contracts with the intention of reducing the number of ties to optimize the performance of SBFL techniques. In Table 4, we have the number of ties generated for the selected suspiciousness formulas. As we can see in the first row of the table, the *original* approach generates the same number of ties as *the rules and contracts embedding approach*. The *no hierarchy* approach gets rid of all the ties. However, we cannot conclude that this would be true in other cases where, even with the absence of inheritance, there are ties, so we would need to present further research with other examples.

Effectiveness of the tie breaking techniques. With respect to the proposed techniques for breaking the ties when there is one, we can see how the *inheritance approach* is able to solve all the ties. This means that all the ties present in our example were related to the inheritance relationship between rules. It happens because part of the super-rule is executed in the sub-rule, generating similar N values for the Spectrum based analysis, and most of the rules in the *UML2ER* example inherit from each another.

On the other hand, the *static approach* is able to solve at least half of the ties in most cases for both the *original* and *embedding approach*. The results are similar for the CC and RCR formulas for the *original* and *embedding* approach both approaches, having around 50% of the ties solved. However, in the latter approach we get better results for the RC formula, reaching almost 50% for all suspiciousness formulae, in which with the *original* approach, we were only able to solve around 30% of the ties. This means that we can obtain better-optimized matching tables for the RC technique thanks to the *embedding* approach.

The effect of our approach on EXAM scores. First of all, we shall remind that, in the case of ties where the buggy rule is involved, the EXAM score in the best-case scenario assumes the buggy rule is inspected first in the tie, the EXAM score in the worst-case scenario assumes it is inspected last in the tie, and the EXAM score in the average-case scenario gives the average of both.

The purpose of our approach is to break ties between rules with the same suspiciousness values. Regarding the EXAM score, this means that, when the buggy rule is involved in a tie, our approach will be improving the average- and worst-case scenarios when the tie is broken by promoting the buggy rule. The worst-case scenario will not change if the tie is broken by demoting the buggy rule—in this case, the EXAM score in the average-case scenario will be worse. Regarding the EXAM score in the best-case scenario, it will not change if the tie is broken by promoting the buggy rule, while it will be worse if the tie is broken by demoting the buggy rule. Therefore, by analyzing the EXAM scores, we are able to make conclusions about the effectiveness of our approach.

Table 5 shows the results for the worst-case scenario. As mentioned before, we are focusing on the four suspiciousness formulae that gave the best results in (Troya et al. 2018). Let us focus on the mean values in the table. The value of the EXAM score obtained in (Troya et al. 2018) is present in the `[Original,Original]` cells for all four formulae. Then, cells `[Original,Embedding]` display the EXAM score with our embedding implementation for extracting the footprints and without using techniques for breaking the ties. Similarly, cells `[Original,No hierarchy]` display the EXAM score with the implementation without inheritance. In the latter case, please note that EXAM score values are higher. The reason is we have now 5 rules instead of 8, so the best EXAM score is 0.2, while the best EXAM score having 8 rules is 0.125 (the second best EXAM score will also be higher, and so on).

The remaining rows (`Inheritance`, `Static CC`, `Static RC` and `Static RCR`) refer to the technique used to break the ties, as explained above. Having a look at the `Original` column, we can see that most static techniques enhance the bug localization process, since the EXAM score in the worst-case scenario has been reduced (i.e., improved). Thus, by using the same implementation of the *UML2ER* model transformation as in (Troya et al. 2018) as well as the same process for footprints extraction as in (Burgueño et al. 2015), the use of static information has a positive influence when breaking the ties. We have highlighted in bold those cells that have the best values (the lowest EXAM scores). Please note that, in all cases, the information from the three matching tables is better for breaking the ties than breaking the ties by using the information regarding rule inheritance. This suggests we could use the matching tables to break ties in model transformations that do not have inheritance relationships. The results of this table can be complemented with those of Tables 6 and 7 (Appendix A), which contains the EXAM scores in the best- and average-case scenarios. Regarding the average-case scenario, we can see that the matching tables with CC information produce the best results when ties are broken. Finally, the results in the best-case scenario suggest that, by breaking the rules using the rules inheritance information, the buggy rule is demoted the fewest number of times when breaking the ties.

Regarding the consideration of our *embedding* proposal for obtaining the footprints of rules and constraints (as explained in Section 4.3), the results are displayed in the `Embedding` column. We can see that the results are not improved with respect to the previous column except for the RC measure, with a minimal improvement. Plus, still, the RCR measure in the original extraction of the footprints is more effective for breaking the ties than this one in the worst- and average-case scenarios, and the CC measure is even better in the average-case scenario.

Finally, we have displayed the `No hierarchy` column sim-

Table 4 Number of ties for the *UML2ER* case study.

		Original				Embedding				No hierarchy			
		Mountford	Kulcynski2	Zoltar	Ochiai	Mountford	Kulcynski2	Zoltar	Ochiai	Mountford	Kulcynski2	Zoltar	Ochiai
	Ties	47	42	39	42	47	42	39	42	0	0	0	0
Inheritance	Solved	47	42	39	42	47	42	39	42	-	-	-	-
	Unsolvable	0	0	0	0	0	0	0	0	-	-	-	-
Static	Solved by CC	24	24	21	24	24	22	19	22	-	-	-	-
	Solved by RC	14	12	12	12	21	20	20	20	-	-	-	-
	Solved by RCR	28	16	24	16	21	20	19	20	-	-	-	-
	Unsolvable	11	8	8	8	10	8	8	8	-	-	-	-

ply for completeness purposes. Having no ties in this implementation, the EXAM score in all scenarios is the same in all cases.

4.5. Answering the research question

By performing the analysis of the presented hybrid approach for the *UML2ER* example, we found that the use of static information allows to enhance the rankings provided by SBFL techniques, as we improve the results in the worst- and average-case scenarios. This means that the static information is used to break ties in which the buggy rule is present by promoting the position of the buggy rule in most cases. We are aware of the fact that this hybrid approach also points to the non-buggy rule in some of the cases, leading to a demote of the buggy rule in the suspiciousness ranking, as represented by the EXAM scores in the best-case scenario. Still, we get an improvement in the average EXAM Score, meaning that we get better results than the original approach even with this shortcoming. The best results are obtained using the static CC approach. With it, we get the closest to the best average-case values, and it is the second-best for the best and worst cases. If the average-case values are better, it means that if the tester decides randomly which rule to check first in a tie, it would still get worse results than the ones by the hybrid approach.

Answering our RQ, static information is able to improve the effectiveness of SBFL techniques for locating faults in case there are ties in the SBFL tables, produced by inheritance relationship between rules. In these cases, we are able to improve the average case, which means that we perform better than a random approach for selecting the buggy rule in a tie. Still, there is potential to improve the tie-breaking approach as the best case could not be realized by the presented technique.

However, we have seen that, at least for this case study, the new proposal for footprints extraction both in contracts and rules considering embeddings does not really offer a positive effect in the effectiveness for the suspiciousness rankings.

4.6. Threats to validity

We now discuss threats to validity of our results based on (Wohlin et al. 2012).

Construct validity is concerned with the relationship between theory and what is observed. In our study, we only used one single metric, i.e., the EXAM score, to evaluate the performance of the presented approach in comparison with a previous study (Troya et al. 2018). Of course, other metrics are available (Wong et al. 2016), e.g., T-score (Liu et al. 2006), P-score (Zhang et al. 2009), and N-score (Gong et al. 2012) to mention just a few. We decided to use the EXAM score as it is a standard metric in SBFL research and it is directly applicable in the comparison to previous work which was one of our main goals.

Internal validity is concerned with those factors that might affect the results of our evaluation. In the first place, we have taken as starting point the results of the research work by Troya et al. (Troya et al. 2018). Thus, our results may be biased for this particular set of inputs. Another threat is about the extension of the inputs. We use a limited number of OCL contracts and mutants for one transformation case. In particular, we have used 14 OCL contracts and 18 mutants reused from (Troya et al. 2018) in order to ensure comparison. In addition, the mutants provided by (Troya et al. 2015) are a subset of the operators defined in (Sánchez-Cuadrado et al. 2017) representing semantic faults which may be introduced by transformation developers (Mottu et al. 2006).

Finally, *external validity* is concerned with the question to the extent it is possible to generalize the findings of the evaluation. The first threat is that the results of our experiments have been obtained with only one transformation case, which of course threatens the generalizability of our results. However, we have focused on already well-studied model transformation where rules have inheritance relationships defined. The reason for this is that rule inheritance relationships result in many ties, and our work is precisely about breaking these ties.

Another threat to external validity is that we have considered rules as components as it was done in previous work. Of course, there are also other components which could be considered such as helper functions or even a more fine-grained notion such as lines of code inside a transformation rule. Studying this aspect, we leave as subject for future work as our intention was to compare to existing approaches using rules as components. A final threat, related to the previous one, is that the approach has been applied to model transformations implemented in ATL. The main reason was to use the same technology as the previous works on the topic. We plan to extend the prototype to transformations written in different languages.

Table 5 EXAM score in the worst-case scenario for the *UML2ER* case study.

Technique		Original			Embedding			No hierarchy		
		mdn	mean	sd	mdn	mean	sd	mdn	mean	sd
Mountford	Original	0.250	0.374	0.309	0.250	0.374	0.309	0.200	0.419	0.323
	Inheritance	0.250	0.369	0.308	0.250	0.369	0.308	0.200	0.419	0.323
	Static CC	0.250	**0.336**	0.320	0.188	**0.336**	0.322	0.200	0.419	0.323
	Static RC	0.250	**0.350**	0.310	0.250	**0.340**	0.315	0.200	0.419	0.323
	Static RCR	0.125	**0.331**	0.317	0.250	**0.340**	0.313	0.200	0.419	0.323
Kulcynski2	Original	0.250	0.360	0.307	0.250	0.360	0.307	0.200	0.416	0.324
	Inheritance	0.250	0.360	0.307	0.250	0.360	0.307	0.200	0.416	0.324
	Static CC	0.125	**0.326**	0.317	0.125	**0.329**	0.318	0.200	0.416	0.324
	Static RC	0.250	**0.343**	0.308	0.125	**0.332**	0.312	0.200	0.416	0.324
	Static RCR	0.125	**0.324**	0.314	0.188	**0.332**	0.311	0.200	0.416	0.324
Zoltar	Original	0.250	0.379	0.316	0.250	0.379	0.316	0.200	0.416	0.324
	Inheritance	0.250	0.379	0.316	0.250	0.379	0.316	0.200	0.416	0.324
	Static CC	0.250	**0.354**	0.321	0.250	**0.357**	0.323	0.200	0.416	0.324
	Static RC	0.250	**0.367**	0.312	0.250	**0.356**	0.316	0.200	0.416	0.324
	Static RCR	0.250	**0.350**	0.317	0.250	**0.357**	0.314	0.200	0.416	0.324
Ochiai	Original	0.250	0.363	0.312	0.250	0.363	0.312	0.200	0.416	0.324
	Inheritance	0.250	0.363	0.312	0.250	0.363	0.312	0.200	0.416	0.324
	Static CC	0.125	**0.329**	0.322	0.125	**0.332**	0.323	0.200	0.416	0.324
	Static RC	0.250	**0.346**	0.312	0.125	**0.335**	0.317	0.200	0.416	0.324
	Static RCR	0.125	**0.326**	0.318	0.188	**0.335**	0.316	0.200	0.416	0.324

5. Related Work

There are several works which focus on testing and debugging of model transformations. A detailed survey on the topic has been recently published in (Troya et al. 2022). With respect to the contribution of this paper, we focus on approaches that propose to locate bugs in model transformations in the following discussion on related work.

As explained in Section 2, the work by Troya et al. (Troya et al. 2018) was the first applying SBFL in the context of MTs, obtaining promising results. They applied 18 different techniques for SBFL, and concluded that four of them outperformed the rest. Our approach builds on this work and proposes a methodology to improve the fault localization process. Li et at. (Li et al. 2020) presented another SBFL approach for model transformations. They proposed to use weighted test models as well as weighted rule coverage to improve the performance of SBFL for model transformations. In addition, Du et al. (Du et al. 2020) also considered SBFL for model transformations without having to manually define oracles. Instead, the oracle is generated by applying metamorphic testing techniques (Segura et al. 2016). However, none of these mentioned SBFL approaches deals with breaking the ties in the suspiciousness rankings, which is the focus of our work.

Aranega et al. (Aranega et al. 2009a,b) proposed to locate model transformation faults by inspecting the traces after the model transformation is executed. In this line of work, local and global traces are built during the model transformation execution. In case there is an error in the output model found, the traces are automatically analysed in order to select a set of likely buggy rules. Then, a manual inspection process is required to find the actual buggy rule in the set of the likely buggy rules.

There are additional approaches to locate faults in model transformations using traces, also considering model-to-text transformations. Examples are the works by García et al. (Garcia et al. 2014) and by Dhoolia et al. (Dhoolia et al. 2010). García et al. (Garcia et al. 2014) focus on MOFScript, a model-to-text transformation language, and augment its native trace model with fine-grained traceability between transformation elements and locations in generated text files. Dhoolia et al. (Dhoolia et al. 2010) associates taint marks with input model elements and propagate the taint marks by a dedicated transformation engine to generate a taint log. In the produced log, the taint marks are associated with substrings of the output. Thus, erroneous elements in the output can be projected back to the related input model elements with the help of the taint marks which is important information for the debugging process.

While the approaches described above require to execute

the model transformation under test, there are also works that propose static approaches for fault localization. One example is the work by Burgueño et al. (Burgueño et al. 2015) we have described in Section 2.2.2. In our work we have used this technique as a basis in order to break the ties obtained with SBFL.

In a different line of research, the work by Sánchez-Cuadrado et al. (Sánchez-Cuadrado et al. 2014; Sánchez-Cuadrado et al. 2015, 2017; Sánchez-Cuadrado et al. 2018; Sánchez-Cuadrado et al. 2018; Sánchez-Cuadrado 2020) present also a method for statically analysing ATL model transformations. However, instead of checking contracts, the goal is to find typing errors and other issues which can be statically found such as conflicting rules. The approach is implemented in the AnATLyzer tool, which also supports test-driven development of ATL model transformations (Sánchez-Cuadrado 2020).

Finally, there is a dedicated work which applies symbolic execution for debugging model transformations (Oakes et al. 2018). The approach is based on the SyVOLT tool for verifying DSLTrans transformations. SyVOLT is able to produce the full state space for a model transformation, i.e., representing all possible executions. Having the full state space allows SyVOLT to prove structural contracts for the model transformation. It not only reports and localizes errors in model transformation implementations, but also in the contracts of model transformations. In our work we assume to localize errors only in the implementation and make use of SBFL based on concrete input models instead of symbolic execution.

To sum up, given the existing work on testing and debugging of model transformations, to the best of our knowledge, we are the first ones who combine static and dynamic fault localization techniques for model transformations in a hybrid framework.

6. Conclusion and Future Research Lines

In this paper, we have presented an enhanced contract language and a hybrid framework which combines dynamic and static fault localization approaches proposed in previous research. A first evaluation shows the benefits of aiming for a hybrid approach especially for breaking ties is rankings of suspicious rules. While these results seem promising for aiming for a hybrid framework, there are still several open topics which are worth to be investigated in fault localization for model transformations. In the following we outline several research directions for future work in this field.

Research Line 1: Locating Bugs inside Rules. The inheritance cases have shown that locating bugs inside rules may be a promising feature. Finding the problem in a more fine-grained manner may be important to reason about if the query or generation parts of the transformation rules are buggy or even further going down to filter expressions or feature assignments of these parts. However, this would also require a more fine-grained tracing of the transformation execution which is currently not available out-of-the-box for most transformation languages and engines.

Research Line 2: Model Footprints. There is the opportunity to compute additional spectrums such as considering the footprints on models, i.e., model coverage of rules and constraints executions. This opportunity may be a compromise between the existing dynamic and static approaches and may provide another dimension to break ties.

Research Line 3: Parallel Contract Evaluation. The extended contract language may be used for providing improved execution modes for checking contracts. One approach may be parallelization as we have several start points, i.e., contracts without preconditions, and then, execute the different independent contracts in parallel as already done for model transformation implementations (Cuadrado et al. 2022).

Research Line 4: Dedicated User Interfaces for Debugging. The hybrid framework provides several information sources which are quite extensive to be explored. In addition, how the transformation rules are actually shown for debugging as well as the constraints may have a huge impact on the efficiency of the debugging process. Thus, we consider research on dedicated user interfaces for debugging of model transformations as an important line of research to better support testers and allow to have an aggregated view on the dynamic and static debugging information.

Verifiability

For the sake of verifiability, all artifacts of the experiments are available in our project repository (Muñoz et al. 2022).

Acknowledgments

This work is partially supported by the European Commission (FEDER) and Junta de Andalucia under project EKIPMENT-PLUS (P18-FR-2895), by the Spanish Government (FEDER/Ministerio de Ciencia e Innovación – Agencia Estatal de Investigación) under project COSCA (PGC2018-094905-B-I00), by the Austrian Science Fund (P 30525-N31), and by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development (CDG).

References

Abreu, R., Zoeteweij, P., & van Gemund, A. J. C. (2007). On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques (TAICPART-MUTATION 2007)* (p. 89-98). doi: 10.1109/TAIC.PART.2007.13

Aranega, V., Mottu, J.-M., Etien, A., & Dekeyser, J. (2009a). Traceability mechanism for error localization in model transformation. In *Proc. of ICSOFT* (pp. 66–73).

Aranega, V., Mottu, J.-M., Etien, A., & Dekeyser, J. (2009b). Using Trace to Situate Errors in Model Transformations. In *Proc. of ICSOFT* (pp. 137–149). doi: 10.1007/978-3-642-20116-5_11

Atenea Research Group. (2013). *TractsTool.* (http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Tracts)

Baudry, B., Dinh-Trong, T., Mottu, J.-M., Simmonds, D., France, R., Ghosh, S., ... Le Traon, Y. (2006). Model Transformation Testing Challenges. In *Proc. of IMDT* (pp. 1–10). Retrieved from https://hal.inria.fr/inria-00542781

Brambilla, M., Cabot, J., & Wimmer, M. (2017). *Model-Driven Software Engineering in Practice (2nd edition)*. Morgan&Claypool. doi: 10.2200/S00441ED1V01Y201208SWE001
Burgueño, L., Troya, J., Wimmer, M., & Vallecillo, A. (2015). Static Fault Localization in Model Transformations. *IEEE Transactions on Software Engineering*, 490–506. doi: 10.1109/TSE.2014.2375201
Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., & Zhang, L. (2020). A Survey of Compiler Testing. *ACM Comput. Surv.*, *53*(1), 4:1–4:36. doi: 10.1145/3363562
Cuadrado, J. S., Burgueño, L., Wimmer, M., & Vallecillo, A. (2022). Efficient Execution of ATL Model Transformations Using Static Analysis and Parallelism. *IEEE Trans. Software Eng.*, *48*(4), 1097–1114. doi: 10.1109/TSE.2020.3011388
Czarnecki, K., & Helsen, S. (2006). Feature-based survey of Model Transformation Approaches. *IBM Systems Journal*, 621–646. doi: 10.1147/sj.453.0621
Dhoolia, P., Mani, S., Sinha, V. S., & Sinha, S. (2010). Debugging Model-Transformation Failures Using Dynamic Tainting. In *Proc. of ECOOP* (pp. 26–51). doi: 10.1007/978-3-642-14107-2_3
Du, K., Jiang, M., Ding, Z., Huang, H., & Shu, T. (2020). Metamorphic testing in fault localization of model transformations. In *Proc. of SOFL+MSVL* (pp. 299–314). doi: 10.1007/978-3-030-41418-4_20
Garcia, J., Azanza, M., Irastorza, A., & Diaz, O. (2014). Testing MOFScript Transformations with HandyMOF. In *Proc. of ICMT* (pp. 42–56). doi: 10.1007/978-3-319-08789-4_4
Gogolla, M., & Vallecillo, A. (2011). *Tract*able Model Transformation Testing. In *Proc. of ECMFA* (pp. 221–235). Springer.
Gong, C., Zheng, Z., Li, W., & Hao, P. (2012). Effects of Class Imbalance in Test Suites: An Empirical Study of Spectrum-Based Fault Localization. In *Proc. of COMPSAC Workshops* (p. 470-475). doi: 10.1109/COMPSACW.2012.89
Harrold, M. J., Rothermel, G., Sayre, K., Wu, R., & Yi, L. (2000). An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, *10*(3), 171–194.
Heckel, R., & Taentzer, G. (2020). *Graph Transformation for Software Engineers - With Applications to Model-Based Development and Domain-Specific Language Engineering*. Springer. doi: 10.1007/978-3-030-43916-3
Jones, J. A., & Harrold, M. J. (2005). Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *Proc. of ASE* (pp. 273–282). ACM. doi: 10.1145/1101908.1101949
Li, P., Jiang, M., & Ding, Z. (2020). Fault Localization With Weighted Test Model in Model Transformations. *IEEE Access*, 14054–14064. doi: 10.1109/ACCESS.2020.2966540
Liu, C., Fei, L., Yan, X., Han, J., & Midkiff, S. P. (2006). Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, *32*(10), 831–848. doi: 10.1109/TSE.2006.105
Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G., … Wimmer, M. (2014). Model Transformation Intents and Their Properties. *Software and Systems Modeling*, 1–35. doi: 10.1007/s10270-014-0429-x
Mottu, J.-M., Baudry, B., & Le Traon, Y. (2006). Mutation analysis testing for model transformations. In *Proc. of ECMFA* (pp. 376–390). doi: 10.1007/11787044_28
Muñoz, P., Troya, J., & Wimmer, M. (2022). *FL4MT repository.* GitHub. Retrieved from https://github.com/paumunoz/FL4MT
Oakes, B. J., Lucio, L., Verbrugge, C., & Vangheluwe, H. (2018). Debugging of Model Transformations and Contracts in SyVOLT. In *Proc. of MODELS Workshops* (pp. 532–537).
Object Management Group. (2014). *Object Constraint Language (OCL) Specification. Version 2.4.* (OMG Document formal/2014-02-03)
Sánchez-Cuadrado, J. (2020). Towards interactive, test-driven development of model transformations. *Journal of Object Technology*, *19*(3), 1–12. doi: 10.5381/jot.2020.19.3.a18
Sánchez-Cuadrado, J., Guerra, E., & de Lara, J. (2015). Quick fixing ATL model transformations. In *Proc. of MODELS* (pp. 146–155). doi: 10.1007/s10270-016-0541-1
Sánchez-Cuadrado, J., Guerra, E., & de Lara, J. (2017). Static analysis of model transformations. *IEEE Transactions on Software Engineering*, 868–897. doi: 10.1109/ISSRE.2014.10
Sánchez-Cuadrado, J., Guerra, E., & de Lara, J. (2018). Quick fixing ATL transformations with speculative analysis. *Software and Systems Modeling*, 779–813. doi: 10.1007/s10270-016-0541-1
Segura, S., Fraser, G., Sánchez, A., & Ruiz-Cortes, A. (2016). A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 805–824. doi: 10.1109/TSE.2016.2532875
Sánchez-Cuadrado, J., Guerra, E., & de Lara, J. (2014). Uncovering Errors in ATL Model Transformations Using Static Analysis and Constraint Solving. In *Proc. of ISSRE* (pp. 34–44).
Sánchez-Cuadrado, J., Guerra, E., & de Lara, J. (2018). AnATLyzer: An Advanced IDE for ATL Model Transformations. In *Proc. of ICSE Companion* (pp. 85–88). doi: 10.1145/3183440.3183479
Troya, J., Bergmayr, A., Burgueno, L., & Wimmer, M. (2015). Towards systematic mutations for and with ATL model transformations. In *Proc. of ICSTW Workshops* (pp. 1–10). doi: 10.1109/ICSTW.2015.7107455
Troya, J., Segura, S., Burgueño, L., & Wimmer, M. (2022). Model transformation testing and debugging: A survey. *ACM Comput. Surv.*. doi: 10.1145/3523056
Troya, J., Segura, S., Parejo, J., & Ruiz-Cortés, A. (2018). Spectrum-based fault localization in model transformations. *ACM Transactions on Software Engineering and Methodology*, 1–50. doi: 10.1145/3241744
Vallecillo, A., & Gogolla, M. (2012). Typing model transformations using tracts. In *Proc. of ICMT* (pp. 56–71). doi: 10.1007/978-3-642-30476-7_4
Wimmer, M., Martínez, S., Jouault, F., & Cabot, J. (2012). A catalogue of refactorings for model-to-model transformations. *Journal of Object Technology*, *11*(2), 2:1–40. doi: 10.5381/jot.2012.11.2.a2

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., & Regnell, B. (2012). *Experimentation in Software Engineering*. Springer. doi: 10.1007/978-3-642-29044-2

Wong, W. E., Debroy, V., Li, Y., & Gao, R. (2012). Software Fault Localization Using DStar (D*). In *Proc. of SERE* (p. 21-30). doi: 10.1109/SERE.2012.12

Wong, W. E., Gao, R., Li, Y., Abreu, R., & Wotawa, F. (2016). A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering*, *42*(8), 707–740. doi: 10.1109/TSE.2016.2521368

Xie, X., Chen, T. Y., Kuo, F.-C., & Xu, B. (2013). A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-based Fault Localization. *ACM Trans. Softw. Eng. Methodol.*, *22*(4), 31:1–31:40. doi: 10.1145/2522920.2522924

Yu, Y., Jones, J. A., & Harrold, M. J. (2008). An Empirical Study of the Effects of Test-suite Reduction on Fault Localization. In *Proc. of ICSE* (pp. 201–210). doi: 10.1145/1368088.1368116

Zhang, Z., Chan, W., Tse, T., Hu, P., & Wang, X. (2009). Is non-parametric hypothesis testing model robust for statistical fault localization? *Information and Software Technology*, *51*(11), 1573 - 1585. doi: 10.1016/j.infsof.2009.06.013

About the authors

Paula Muñoz is a PhD candidate at the University of Málaga. She graduated in Software Engineering from the University of Málaga in June 2019. Her research focuses on the precise specification and testing of software systems using models. Contact her at paulam@uma.es.

Javier Troya is Associate Professor of Software Engineering at the University of Malaga, Spain. Before, he was a post-doctoral researcher, assistant and associate professor at the University of Seville, Spain (2016-2020), and a post-doctoral researcher in the TU Wien, Austria (2013-2015). His current research interests include Model-based Software Engineering, Software Testing and Uncertainty modeling. Contact him at jtroya@uma.es, or visit http://webpersonal.uma.es/de/jtroya.

Manuel Wimmer is a full professor in, and the head of, the Department of Business Informatics – Software Engineering at the Johannes Kepler University Linz, Austria. His research interests include software engineering, model-driven engineering, and cyber-physical systems. Contact him at manuel.wimmer@jku.at, or visit https://se.jku.at/manuel-wimmer

Gerti Kappel is full professor at the Institute of Information Systems Engineering at TU Wien, chairing the Business Informatics Group. Since the beginning of 2020 she acts as the dean of the Faculty of Informatics at TU Wien. Her current research interests include Model Engineering, Web Engineering, and Process Engineering, with a special emphasis on cyber-physical production systems. You can contact the author at kappel@big.tuwien.ac.at or visit https://www.big.tuwien.ac.at/people/gkappel.

A. EXAM score tables

Table 6 Best EXAM Score for *UML2ER* case study.

Technique		Original			Embedding			No hierarchy		
		mdn	mean	sd	mdn	mean	sd	mdn	mean	sd
Mountford	Original	0.125	**0.310**	0.324	0.125	**0.310**	0.324	0.200	0.419	0.323
	Inheritance	0.125	**0.304**	0.323	0.125	**0.304**	0.323	0.200	0.419	0.323
	Static CC	0.125	0.314	0.322	0.125	0.318	0.324	0.200	0.419	0.323
	Static RC	0.250	0.344	0.313	0.250	0.340	0.315	0.200	0.419	0.323
	Static RCR	0.125	0.328	0.318	0.250	0.340	0.313	0.200	0.419	0.323
Kulcynski2	Original	0.125	**0.301**	0.318	0.125	**0.301**	0.318	0.200	0.416	0.324
	Inheritance	0.125	**0.301**	0.318	0.125	**0.301**	0.318	0.200	0.416	0.324
	Static CC	0.125	0.308	0.319	0.125	0.314	0.319	0.200	0.416	0.324
	Static RC	0.250	0.339	0.309	0.125	0.332	0.312	0.200	0.416	0.324
	Static RCR	0.125	0.322	0.315	0.188	0.332	0.311	0.200	0.416	0.324
Zoltar	Original	0.125	**0.324**	0.326	0.125	**0.324**	0.326	0.200	0.416	0.324
	Inheritance	0.125	**0.324**	0.326	0.125	**0.324**	0.326	0.200	0.416	0.324
	Static CC	0.125	0.336	0.321	0.250	0.342	0.322	0.200	0.416	0.324
	Static RC	0.250	0.363	0.312	0.250	0.356	0.316	0.200	0.416	0.324
	Static RCR	0.250	0.349	0.318	0.250	0.357	0.314	0.200	0.416	0.324
Ochiai	Original	0.125	**0.304**	0.323	0.125	**0.304**	0.323	0.200	0.416	0.324
	Inheritance	0.125	**0.304**	0.323	0.125	**0.304**	0.323	0.200	0.416	0.324
	Static CC	0.125	0.311	0.323	0.125	0.317	0.324	0.200	0.416	0.324
	Static RC	0.250	0.342	0.313	0.125	0.335	0.317	0.200	0.416	0.324
	Static RCR	0.125	0.325	0.319	0.188	0.335	0.316	0.200	0.416	0.324

Table 7 Average Case EXAM Score for *UML2ER* case study.

Technique		Original			Embedding			No hierarchy		
		mdn	mean	sd	mdn	mean	sd	mdn	mean	sd
Mountford	Original	0.188	0.342	0.315	0.188	0.342	0.315	0.200	0.419	0.323
	Inheritance	0.188	0.337	0.314	0.188	0.337	0.314	0.200	0.419	0.323
	Static CC	0.188	**0.325**	0.320	0.156	**0.327**	0.322	0.200	0.419	0.323
	Static RC	0.250	0.347	0.311	0.250	0.340	0.315	0.200	0.419	0.323
	Static RCR	0.125	0.329	0.317	0.250	0.340	0.313	0.200	0.419	0.323
Kulcynski2	Original	0.188	0.331	0.311	0.188	0.331	0.311	0.200	0.416	0.324
	Inheritance	0.188	0.331	0.311	0.188	0.331	0.311	0.200	0.416	0.324
	Static CC	0.125	**0.317**	0.317	0.125	**0.322**	0.318	0.200	0.416	0.324
	Static RC	0.250	0.341	0.308	0.125	0.332	0.312	0.200	0.416	0.324
	Static RCR	0.125	0.323	0.314	0.188	0.332	0.311	0.200	0.416	0.324
Zoltar	Original	0.188	0.351	0.320	0.188	0.351	0.320	0.200	0.416	0.324
	Inheritance	0.188	0.351	0.320	0.188	0.351	0.320	0.200	0.416	0.324
	Static CC	0.188	**0.345**	0.320	0.250	**0.349**	0.321	0.200	0.416	0.324
	Static RC	0.250	0.365	0.312	0.250	0.356	0.316	0.200	0.416	0.324
	Static RCR	0.250	0.349	0.318	0.250	0.357	0.314	0.200	0.416	0.324
Ochiai	Original	0.188	0.333	0.316	0.188	0.333	0.316	0.200	0.416	0.324
	Inheritance	0.188	0.333	0.316	0.188	0.333	0.316	0.200	0.416	0.324
	Static CC	0.125	**0.320**	0.322	0.125	**0.324**	0.323	0.200	0.416	0.324
	Static RC	0.250	0.344	0.312	0.125	0.335	0.317	0.200	0.416	0.324
	Static RCR	0.125	0.326	0.318	0.188	0.335	0.316	0.200	0.416	0.324