The Journal of Systems & Software 168 (2020) 110675

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

SEET: Symbolic Execution of ETL Transformations

Banafsheh Azizi, Bahman Zamani *, Shekoufeh Kolahdouz-Rahimi

MDSE Research Group, Department of Software Engineering, University of Isfahan, Isfahan, Iran

ARTICLE INFO

Article history:
Received 6 June 2019
Received in revised form 18 May 2020
Accepted 28 May 2020
Available online 1 June 2020

Keywords:
Model-Driven Engineering (MDE)
Epsilon Transformation Language (ETL)
Verification of model transformations
Metamodel footprint
Symbolic execution

ABSTRACT

Model transformations are known as the main pillar of model-driven approaches. A model transformation is a program, written in a transformation language, to convert a model into another model or code. Similar to any other program, model transformations need to be verified. The problem is that some transformation errors, e.g., logical errors, can only be detected via execution. Our focus in this research is on the Epsilon Transformation Language (ETL), one of the most extensively used model transformation languages. Lack of approaches to detecting logical errors in ETL transformations is a gap which needs to be addressed.

In this paper, we present an approach to symbolic execution of ETL transformations and detecting logical errors. The approach uses a constraint solver to assess the satisfiability of a path condition and generates a symbolic metamodel footprint which can be used to detect errors. The approach is corroborated by a tool that is integrated with Eclipse. To evaluate the approach, the precision and recall are calculated for two well-known case studies. The scalability is evaluated via nine experiments. The usefulness and usability aspects are evaluated in a subjective manner. The results show the improvement in the field of verifying ETL transformations.

1. Introduction

Model-Driven Engineering (MDE) has emerged as a promising paradigm of software engineering to provide better productivity and higher quality by raising the level of abstraction from code to model (Fondement and Silaghi, 2004). Hence, central to the entire discipline of MDE is the concept of *model* (Schmidt, 2006). The main goal of MDE is that, instead of constructing code, developers create models and write transformations that transform these models to other models or text/code (Brambilla et al., 2017). Model transformations are the cornerstone of MDE. A model transformation is a program that is written using a transformation language. The Epsilon Transformation Language (ETL) (Kolovos et al., 2008) is one of the commonly-used model transformation languages in academia and industry (Jakumeit et al., 2014). ETL is essential to a wide range of transformations since it can be seamlessly integrated with other model management tasks (Kolovos et al., 2008). Since the correctness of the transformation programs affects the quality of the final product, their verification deserves further attention. In the current study, the idea of Boehm (1984) is followed and we maintain that verification refers to "building the things right". With respect to the definition, testing is a verification technique (Ab. Rahim and Whittle, 2015). Consequently, owing to the widespread use of ETL in the model-driven community, it is impossible to overlook the verification of ETL transformations.

Recently, considerable literature has emerged around the theme of verification of model transformations (Ab. Rahim and Whittle, 2015). All the same, there remains a paucity of evidence on verifying ETL transformations. Several attempts have been made to detect errors in the ETL specification (Lano et al., 2015; Wei and Kolovos, 2014; Popoola et al., 2016; García-Domínguez et al., 2011; Azizi et al., 2017). However, some of these studies are limited by generating repetitive test cases (Popoola et al., 2016), input dependency, i.e., providing an answer to an individual input (García-Domínguez et al., 2011), and a semi-automated tool (Lano et al., 2015; Popoola et al., 2016). Some failed to draw on reasoning about the output of the transformation (Wei and Kolovos, 2014) or failed to pay heed to the most imperative part of ETL (Lano et al., 2015; Azizi et al., 2017). Hence, recent research on verifying ETL transformations has highlighted the need for an input-independent tool which, by considering both declarative and imperative parts of the definition of the transformation, can reason about the output.

This paper adopts a novel approach to verifying ETL transformations and detecting their logical errors. A logical error is a mistake in a program that results in an incorrect output or unexpected behavior. They may cause a program to crash at runtime or produce the wrong output without any error message

* Correspondence to: Department of Software Engineering, Faculty of Computer Engineering, University of Isfahan, Hezar-Jerib Ave., Isfahan, Iran.
E-mail addresses: b.azizi.ac@gmail.com (B. Azizi), zamani@eng.ui.ac.ir (B. Zamani), sh.rahimi@eng.ui.ac.ir (S. Kolahdouz-Rahimi).

https://doi.org/10.1016/j.jss.2020.110675
0164-1212/

or crash (Logic Error Definition, 2018). For example, assigning the wrong value to the property of an output element or returning 'false' instead of 'true' in the return statement may produce unwanted results. Symbolic execution has been successful in resolving similar issues in verifying programs, outside the model-driven paradigm, inasmuch as it is used in big companies including Microsoft (Godefroid et al., 2012). We hold that this technique will prove beneficial in addressing similar issues in verifying ETL transformations.

The approach taken in this study is a mixed approach based on symbolic execution and constraint solving. The approach works in both automatic and interactive modes. In the automatic mode, all executable paths of the transformation are traversed. In the interactive mode, by interacting with a human tester, one path of a transformation is selected and executed symbolically. In both modes, the path condition is assessed eagerly to check its satisfiability. This is where a constraint solver can play a part. The engine gradually generates a subset of symbolic target metamodel which in this paper is called the Symbolic Metamodel Footprint (SMF). At the end of the transformation, a test model is generated corresponding to each satisfiable path condition. Finally, the tester must ensure the correctness of the transformation by comparing the results with the expected ones.

In order to provide concrete support for the proposed approach, a tool is developed as an Eclipse plugin. The tool is called SEET, which stands for "Symbolic Execution of ETL Transformations". To evaluate the correctness and completeness of the proposed approach, several mutation operators are defined for ETL and applied to two case studies, and the precision and recall are calculated on the mutants. Our evaluation experiments have shown good results in both respects. To investigate the scalability of the approach, the time of symbolic execution of nine transformation cases has been measured. The results showed that the execution times are increased in a rational way in terms of the increased size of the input metamodel, and the number of lines of code and path conditions. Thus, SEET proved scalable. Finally, the usefulness and usability aspects are evaluated by the survey. The results indicated that SEET is useful and usable.

The main contributions of this study are as follows.

- The present study partially fills a gap in the literature by making an original contribution to the symbolic execution of ETL transformations. It considers both declarative and imperative parts of ETL transformations. The approach works in both automatic and interactive modes, which are discussed in Section 3. The interactive mode relies on putting the human in the loop during the symbolic execution. The approach could be relevant to other technology spaces and is considered as a case study on how symbolic execution can be implemented to verify model transformations. While we have developed the approach for ETL, the same approach can be adopted to implement symbolic executor for other rule-based model transformations, e.g., ATL.
- Introducing the idea of Symbolic Metamodel Footprint (SMF), which is a novel idea even among other studies on the symbolic execution of transformations aside from ETL transformations. SMF is discussed in Sections 3.1 and 3.5.
- A set of mutation operators to evaluate the detection of logical errors in ETL transformations, which is elaborated in Section 5.1.1. A dataset of ETL mutants is available online.
- The open-source SEET tool implementing the proposed approach which is available online.

This paper is organized as follows. Section 2 is dedicated to the background information, which provides an opportunity to advance the understanding of the context of the method. Section 3 gives an account of the approach presented in this study. Section 4 concerns the implemented tool. Section 5 deals with the evaluation. Section 6 provides a brief overview of the recent history by discussing the merits of the proposed approach against other related studies. Finally, Section 7 draws some conclusions.

2. Background

In this section, we first delineate MDE and model transformations. Then, we describe the Epsilon framework, particularly ETL and we introduce a running example. Next, the Epsilon Haetae tool (Wei, 2016) is presented. Eventually, the symbolic execution technique is explained.

2.1. Overview of model-driven engineering and model transformations

One of the contemporary software development methodologies is MDE. One of the use cases of MDE is to generate programs from models automatically (Brambilla et al., 2017). With this purpose in mind, a set of transformations are performed. Typically, a set of model-to-model transformations are applied, and at the end, a model-to-text transformation is carried out. Thus, in this paradigm, models along with transformations result in software. Model to model transformations, which have been viewed as the mapping between input and output elements, are vital in MDE. This kind of transformation converts a source model conforming to a source metamodel (a model that describes a set of models) to a target model, which conforms to a target metamodel. Although model transformations are executed on models, they are specified at the level of metamodels (Brambilla et al., 2017). To entirely exploit the power of a model transformation, its representation as a model is needed. A Higher-Order Transformation (HOT) is a model transformation where at least one of its input or output models is a transformation model (Tisi et al., 2009).

Many model transformation languages have been presented for defining and executing model transformations. A transformation language can be declarative, imperative or hybrid (Sendall and Kozaczynski, 2003). Declarative languages focus on what should be transformed by defining a relation between the source and the target models. Imperative languages focus on how a target model is built from a source model (Mens and Van Gorp, 2006). Hybrid languages, such as ETL and Atlas Transformation Language (ATL) (Jouault et al., 2008), combine these two techniques (Czarnecki and Helsen, 2006).

2.2. The Epsilon framework

Epsilon is a platform for performing model management tasks such as model transformation, code generation, model comparison, model validation, model merging, and model refactoring (Kolovos et al., 2017). It supports these tasks via consistent and task-specific languages, such as ETL and Epsilon Comparison Language (ECL) (Kolovos et al., 2006a). Epsilon is an Eclipse-based project and in accordance with the list of Epsilon users (Epsilon-Users, 2017), it is widely used not only in academia but also in the industry. The Epsilon Object Language (EOL) (Kolovos et al., 2006b), the Epsilon core language, provides a set of reusable model management tasks for implementing task-specific languages. It can be applied as a general-purpose and independent language for model management tasks. EOL programs are organized into modules. An EOL module includes a body and operations. The body consists of several statements which are evaluated in the execution time of the module. The definition for each EOL operation includes a context, name, parameter(s), and return type. EOL modules are capable of importing other EOL modules and accessing their operations (Kolovos et al., 2017).

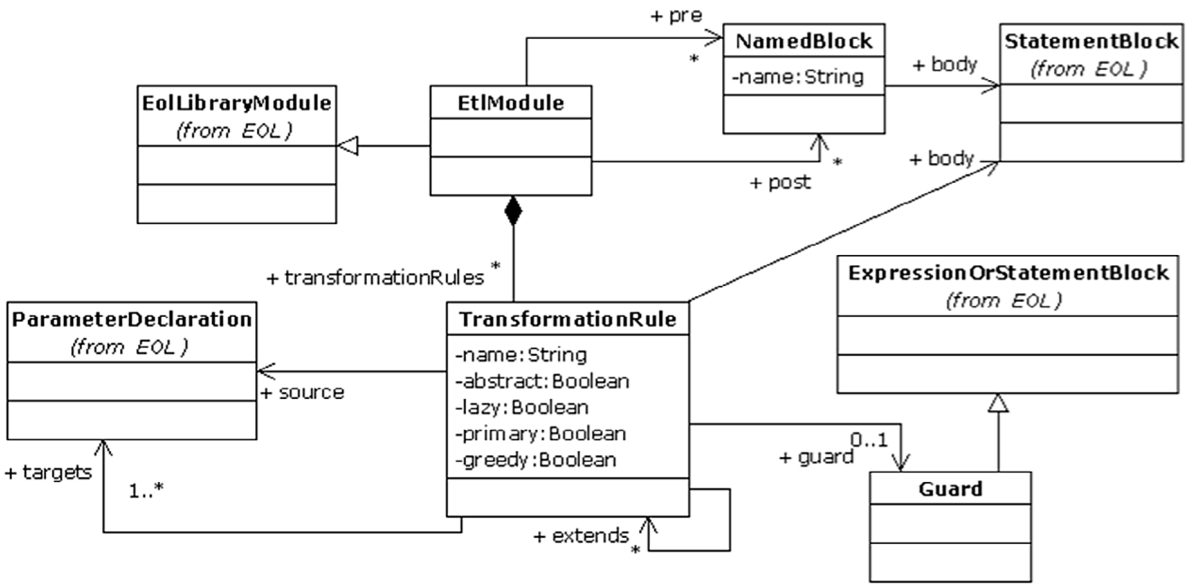


Fig. 1. The ETL metamodel (Kolovos et al., 2017).

2.2.1. Epsilon Transformation Language (ETL)

ETL is the model-to-model transformation language of Epsilon. It can transform one or more source models to one or more target models. These models can conform to different metamodels. ETL is a hybrid language, which offers both declarative and imperative capabilities. The declarative part consists of transformation rules. ETL is aided by EOL for the imperative part. The imperative part offers facilities to handle complex transformations. Moreover, in contrast to many transformation languages, ETL provides the ability to define whether the source and target models can be read or stored. ETL can be seamlessly integrated with other model management tasks in the Epsilon family without extra model loading and storing. Moreover, since Epsilon languages have been built atop EOL, EOL operations can be reused in this workflow. Therefore, with the advent of ETL, it is possible to apply a workflow of model management tasks with reusability and low diversity (Kolovos et al., 2008).

As indicated in Fig. 1, an ETL program is structured as an *ETLModule*. An *ETLModule* consists of several *TransformationRules*, EOL operations, and optionally pre/post blocks. A TransformationRule has a unique name in the context of ETLModule and also one source parameter, one or more target parameters, a guard (which is an EOL ExpressionOrStatementBlock), and finally *Block* of EOL statements as its body. A TransformationRule can extend other TransformationRules and it can be defined as abstract, lazy, primary, or greedy (Kolovos et al., 2008).

As can be seen in Fig. 1, the structure of all types of rule declaration, pre/post blocks, guards, and source/target elements of the rules belong to the declarative part of ETL. All statements and expressions as well as operations belong to the imperative part of ETL transformations, which are coming from EOL.

An ETL module can import other ETL modules. In this case, all rules and pre/post blocks of those modules are inherited. When an ETL module is executed, first, pre-blocks are executed in order. Next, non-lazy and non-abstract rules are executed on the source elements which are of the type of the source parameter of the rule and satisfy its guard and the guard of all rules that this rule extends. When a rule is executed, target elements are created by instantiating the target parameters of the rule. Then, their features are populated during the execution of the body of the rule. After all rules are executed, post-blocks are executed in order (Kolovos et al., 2008).

ETL provides the *equivalent()* and *equivalents()* operations. When the *equivalents()* operation is applied on a set of elements, first of all, the types of those elements are found. Then, rules whose source element is of the mentioned type are invoked and elements with the types of their target element(s) are returned. In a similar way, when an *equivalent()* operation is called on an element, the first element which has been produced by the equivalents operation is returned (Kolovos et al., 2008).

Another important feature of ETL is the SpecialAssignment operator. It is equal to the non-special assignment in which an equivalent operation is applied to its right-hand side (RHS) (Kolovos et al., 2008). By way of illustration, "e.source ::= t.parent;"[1] is equal to "e.source := t.parent.equivalent();".

2.2.2. Running example

As a running example, we have selected the Families2Persons[2] case study. The reason for this selection is that its domain concepts are easily understandable. Also, it has an appropriate number of paths.

Fig. 2 shows both source and target metamodels of the Families2Persons transformation. The source metamodel (Families) has a class named Family. This class has a lastName attribute and can have a mother, a father, some daughters or some sons of type Member. Any Member class has a firstName. The target metamodel (Persons) has a root class named Person, which has a fullName and can be a Male or a Female.

Listing 1 shows the ETL program of the Families2Persons transformation. The Member2Male rule (line 5) is responsible for creating a Male from a Member and the Member2Female rule (line14) creates a Female from a Member. Both rules have a method call expression in their guard (lines 9, 18) which calls the isFemale operation (line 23). This operation returns true if the gender of the Member is Female, i.e., Member is a mother or a daughter of the Family as it checks method call expressions such as "self.familyMother.isDefined()" and "self.familyDaughter.isDefined()" (lines 24, 27). Otherwise, it returns false. The Member2Male rule is executed in the event that

[1] https://www.eclipse.org/epsilon/examples/index.php?example=org.eclipse.epsilon.examples.tree2graph

[2] https://github.com/jreimone/refactory

negation of the return value of the isFemale operation is true and Member2Female rule is executed in the case that the return value of the isFemale operation is true. In the body of either of these rules, there exists an assignment statement (lines 11, 20). On the left-hand side (LHS) of this assignment statement, there exists a property call expression "t.fullName". In this property call expression "t" is a name expression and "fullName" is its property name. In fact, this assignment statement initializes the fullName

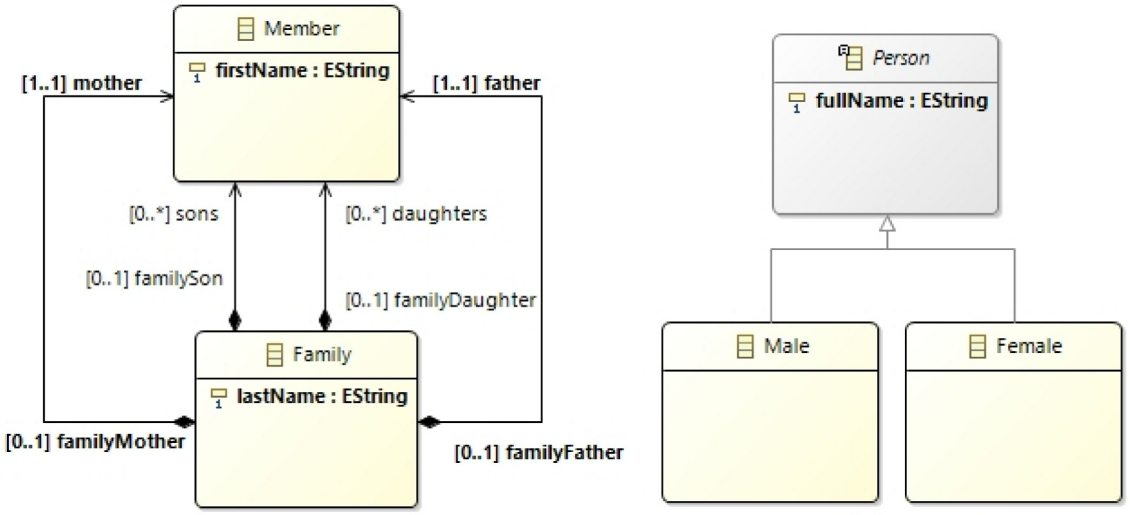
```
1 pre {
2    "Running ETL".println();
3 }
4
5 rule Member2Male
6    transform s : Families!Member
7    to t : Persons!Male {
8
9    guard : not s.isFemale()
10
11   t.fullName = s.firstName + " " + s.familyName();
12 }
13
14 rule Member2Female
15    transform s : Families!Member
16    to t : Persons!Female {
17
18    guard : s.isFemale()
19
20    t.fullName = s.firstName + " " + s.familyName();
21 }
22
23 operation Families!Member isFemale() : Boolean {
24    if (self.familyMother.isDefined()) {
25       return true;
26    } else {
27       if (self.familyDaughter.isDefined()) {
28          return true;
29       } else {
30          return false;
31       }
32    }
33 }
34
35 operation Families!Member familyName() : String {
36    if (self.familyFather.isDefined()) {
37       return self.familyFather.lastName;
38    } else {
39       if (self.familyMother.isDefined()) {
40          return self.familyMother.lastName;
41       } else {
42             if (self.familySon.isDefined()) {
43                return self.familySon.lastName;
44             } else {
45                   return self.familyDaughter.lastName;
46             }
47       }
48    }
49 }
```

Listing 1. The Families2Persons transformation in ETL

```
1 pre {
2    "Running ETL".println();
3    var a : new Persons!Male; --E1
4 }
5
6 rule Family2Female
7    transform s : Families!Family                --E2
8    to t : Persons!Female {
9 }
10
11 @lazy --E3
12 rule Member2Female --E4
13    transform s : Families!Member
14    to t : Persons!Male , t1: Persons!Female { --E5
15
16    guard : not s.isFemale() --E6                                    --E10
17                             --E7
18    t.fullName = "WRONG!"; --E8
19    t.fullName = "WRONG!"; --E9
20 }
21
22 rule Member2Female
23    transform s : Families!Member
24    to t : Persons!Male { --E11
25
26    guard : not s.isFemale() --E12
27                   --E13      --E14          --E15          --E16
28    t.fullName = "WRONG" + "WRONG" + s.familyName() + "WRONG";
29 }
30
31 operation Families!Member isFemale() : Boolean { --E17
32    return true; --E18
33    if (not self.familyMother.isDefined()) { --E19
34       return false; --E20
35    } else {
36       if (self.familySon.isDefined()) { --E21
37          return true;                                              --E22
38       } else {
39          return false;
40       }
41    }
42 }
43
44 operation Families!Member familyName() : String {
45    if (not self.familyFather.isDefined()) { --E23
46       return self.familyFather.lastName; --E24
47    } else {
48       if (self.familyMother.isDefined()) { --E25
49          return self.familyFather.lastName; --E26
50       } else {
51          if (self.familySon.isUndefined()) { --E27
52             return self.familySon.lastName;
53          } else {
54             return self.familyDaughter.lastName;                   --E28
55          }
56       }
57    }
58 }
```

Listing 2. The faulty Families2Persons transformation in ETL. Logical errors are specified according to their categories: Added modified deleted

attribute of the Male class (in the case that the Member2Male rule is executed) or the Female class (in the case that Member2Female rule is executed). The value of this attribute is a plus operator expression (s.firstName + " " + s.familyName()). The familyName operation returns the lastName value of the corresponding Fam-



a) The Families metamodel b) The Persons metamodel

Fig. 2. Families and Persons metamodels.[3]

ily depending on which of familyMother, familyFather, familyDaughter, and familySon is defined for the Member. This running example will be used in Section 3 in order to help to understand the proposed approach.

2.2.3. Logical errors captured by SEET

Logical errors can be divided into two categories. First, those which cause programs to crash, i.e., produce "runtime errors", second, those which cause "undesirable output". Table 1 shows a list of logical errors in ETL transformations which are covered by the SEET tool. It is noteworthy that some of these errors may not be a logical error under all circumstances. This is to mention that we avoid considering extra If statement, Switch statement, loop, or pre/post block since they do not have a logical error in the case that their body is empty. Three transformation examples are shown in Listing 2, Listing 3, and Listing 4, with a total of 48 errors. These listings are using to show concrete examples of each row of Table 1. Please note that, each example should be considered in isolation, to avoid the side effects.

2.3. Epsilon Haetae

Epsilon Haetae (Wei, 2016) is a tool for static analysis of the languages of the Epsilon platform. For static analysis, first, it transforms the Homogeneous Abstract Syntax Tree of a program, obtained with ANTLR, into a Heterogeneous Abstract Syntax Tree. Then, it applies variable and type resolution algorithms to derive a Heterogeneous Abstract Syntax Graph (Wei and Kolovos, 2014). It also supports code to model transformations for EOL, ETL, and EVL. As well, the EOL transformation model can be obtained by this tool programmatically.

2.4. Testing with symbolic execution

Symbolic execution of a program represents various classes of concrete executions. In this type of execution, inputs have symbolic values, each of which covers multiple possible values, instead of concrete ones. It results in a set of path conditions on input variables of the program. A path condition shows the navigation of a conditional branch statement. Path conditions are symbolic which means that they provide various concrete executions any of which has a bearing on an instance of variables of conditions. Since detecting logical errors requires the execution of the program and concrete execution is not cost-effective, symbolic execution offers an effective way of executing the program (King, 1976).

3. SEET: Symbolic execution of ETL transformations

In this section, we present the proposed approach to the verification of ETL transformations via symbolic execution. First, we elaborate on what we mean by symbolic execution of a model transformation. Then, the overall process and its steps are presented.

3.1. Symbolic execution of a transformation

Symbolic execution is a well-established approach to finding logical errors in programs. Due to the shortcomings of current tools on verification of ETL transformations, particularly the lack of automatic tool for finding logical errors in both declarative and imperative parts of ETL transformations, and the power of symbolic execution technique, we based this study on symbolic execution. Symbolic execution of ETL transformations was achieved by adapting the procedure used by King (1976).

To explain the symbolic execution of a transformation, consider Fig. 3 which compares the concrete and symbolic execution of programs versus transformations. In the concrete execution of programs, both inputs and outputs are values. In the process of symbolic execution, the level of abstraction is increased by giving symbolic values to the program, and producing symbolic expressions as the output. Similarly, in the concrete execution of transformations, the source and target metamodels along with the source model are given to the transformation, and then the transformation generates a target model as output. For symbolic execution of model transformations, only the source and target metamodels are required as inputs (no input model is required). The output for this process is a subset of symbolic target metamodel which we call a Symbolic Metamodel Footprint (SMF). The SMF is a subset of the target metamodel whose attributes have symbolic values. Specifically, in an SMF, some classes, references, and attributes are not generated at all because they do

[3] https://www.eclipse.org/atl/documentation/old/ATLUseCase_Families2Persons.pdf

Table 1
List of logical errors covered by SEET.

ETL element	Logical error	Example
Rule	Extra	Listing 2, E2
	Missing	Listing 2, E10
	Wrong name	Listing 2, E4
	Wrong type	Listing 2, E3
	Wrong inpattern element	Listing 4, E5
	Extra outpattern element	Listing 2, E5
	Missing outpattern element	Listing 4, E1
	Wrong outpattern element	Listing 2, E11
	Extra guard	Listing 3, E5
	Missing guard	Listing 2, E6
	Wrong Guard	Listing 2, E12
Assignment Statement (contains special assignment statement)	Extra	Listing 2, E9
	Missing	Listing 2, E8
	Wrong RHS	Listing 2, E7
	Wrong LHS	Listing 3, E8
Operation	Missing context	Listing 2, E17
Return Statement	Extra	Listing 2, E18
	Missing	Listing 2, E24
	Wrong	Listing 2, E20
Non-local variable declaration expression	Extra	Listing 2, E1
	Missing	Listing 4, E3
New expression	Wrong element name	Listing 4, E2
If Statement	Missing If block	Listing 2, E22
	Wrong condition	Listing 2, E23
	Missing Else block	Listing 2, E28
Plus Operator Expression	Extra operand	Listing 2, E16
	Missing operand	Listing 2, E15
	Wrong operand	Listing 2, E13
Property Call Expression	Missing property	Listing 2, E25
	Wrong property	Listing 2, E26
	Wrong target	Listing 3, E6
Method Call Expression	Wrong target	Listing 2, E21
	Wrong method	Listing 2, E27
	Wrong argument	Listing 3, E3
Primitive Expression	Extra	Listing 2, E16
	Missing	Listing 3, E7
	Wrong value	Listing 2, E14
Name Expression	Wrong name	Listing 3, E6
Expression statement	Missing	Listing 3, E1
	Extra	Listing 4, E4
	Wrong expression	Listing 3, E3
Loop	Missing	Listing 3, E4
	Wrong condition	Listing 3, E2
Operator Expression	Extra	Listing 2, E19
	Missing	Listing 4, E6
Switch Statement	Missing	Listing 4, E8
	Extra case	Listing 4, E11
	Missing case	Listing 4, E9
	Wrong case	Listing 4, E10
	Extra default	Listing 4, E12
	Missing default	Listing 4, E7

not appear in the transformation. In such a model, there exists at most one class for each of the target metamodel classes. The references conform to the target metamodel (they are not exactly the same as metamodel relations, e.g., inheritance relation does not exist) and the attributes have symbolic values, which are made meaningful based on the values assigned to them in the transformation.

3.2. The proposed approach

The overall process of the SEET approach is shown in Fig. 4. In step 1, a model of the ETL transformation, conforming to the ETL metamodel, is extracted from the given ETL program using Haetae. In step 2, the CFG generator reads the obtained model of the ETL transformation to produce the equivalent CFG. In step 3,

```
1 rule Competition2Application
2    transform c : Competition!Competition
3    to a : TVApp!Application, v : TVApp!Vote {
4
5    a.name = c.name + " Application";
6    v.name = "Who will win the " + c.name + "?";
7    a.contents.add(v); --E1
8
9    for (g in c.groups) {
10       v.contents.add(g.equivalent());
11       for (comp in c.groups) { --E2
12          v.contents.add(g.equivalent()); --E3
13       }
14   }                                          --E4
15 }
16
17 rule Competitor2Choice
18    transform co : Competition!Competitor
19    to ch : TVApp!Choice {
20
21    guard : co.name = 'initial' --E5
22
23    co.name = co.name; --E6
24 }
25
26 rule Group2Label
27    transform g : Competition!Group
28    to l : TVApp!Label {
29                --E7
30    g.name = "Group " + g.name; --E8
31 }
```

Listing 3. The faulty Competition2TVApp transformation in ETL. Logical errors are specified according to their categories: Added modified deleted

the symbolic execution engine takes the generated CFG along with the ETL transformation model, the source metamodel and the target metamodels as input to navigate the transformation model. During its operation, the engine interacts with the symbol table and uses the traceability links. The symbol table contains information about the identifiers that are defined in the transformation. The traceability links establish an explicit relationship between the source and target parameters of each rule. By symbolically executing each statement, the SMF is gradually generated. When a condition is encountered, the process goes to step 4, in which the path condition generator calculates two path conditions. The first path condition (PC1) is obtained by logical conjunction of the new condition and the current path condition. The second (PC2) is obtained by logical conjunction of the negation of the new condition and the current path condition. Then, in step 5, the path conditions are given to the path condition evaluator. When one of these path conditions cannot be met, the execution will be continued by the satisfiable path condition. If both path conditions are met, in the interactive mode, interaction with the tester is required as indicated in step 6 "preparation". For this purpose, after preparing the corresponding part of the ETL transformation model, the tool interacts with the tester and the execution will be continued by the engine. In case of the automatic mode, the engine considers both paths. When the transformation statements are exhausted, the process goes to step 7 whereby the Conformance checker checks whether the SMF conforms to the target metamodel. In case the generated SMF does not conform to the target metamodel, since the SMF is not usable, it is discarded. Otherwise, provided that path conditions are not empty, SEET will provide two different results based on the execution mode. In the interactive mode, the test model that satisfies the path condition is shown to the tester, while in the automatic mode the mentioned output is provided for all executable paths.

```
1 rule A2BC
2    transform s : Input!A
3    to t1 : Output!B, t2 : Output!C { --E1
4    t1.setDesignPromo(s);
5    var out : OutPut!F = new Output!D; --E2
6    var out1 : new Output!I; --E3
7 }
8
9 rule Mapa2View
10   transform m:Input!Mapa
11   to v:Output!Layout, t2:Output!Row {
12   v.setDesignMapa(m);
13   v.filas.add(t2); --E4
14 }
15
16 rule E2F
17   transform s : Input!G --E5
18   to t : Output!F {
19
20   guard : not (s.name = 'initial') --E6
21
22   t.name = s.name;
23 }
24
25 operation Output!Layout setDesignPromo(f:Input!A){
26   switch(f.estilo){
27      case 'empresarial':
28         self.background=view!Color#gray;
29      default:                                  --E8
30         self.background=view!Color#white; --E7
31   }
32   return null;
33 }
34
35 operation Output!Layout setDesignMapa(f:Input!Mapa){
36   switch(f.estilo){
37      case 'empresarial':
38         self.background=view!Color#gray; --E9
39      case 'wrong'; --E10
40         self.background=view!Color#blue;
41      case 'extra':
42         self.background=view!Color#red;  --E11
43      default:
44         self.background=view!Color#blue; --E12
45   }
46   return null;
47 }
```

Listing 4. The sample of faulty transformation in ETL. Logical errors are specified according to their categories: Added modified deleted

Eventually, the tester ensures the correctness of the transformation by considering the SMF or giving the expected SMF to SEET so that in step 8, its symbolic oracle specifies whether the transformation is implemented correctly. The subsequent sections shed light on each part of the process.

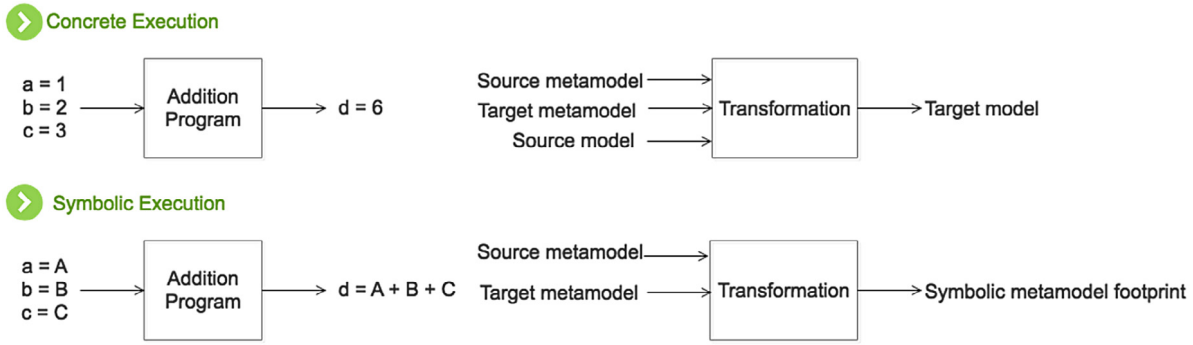


Fig. 3. Concrete and symbolic execution of programs vs. transformations.

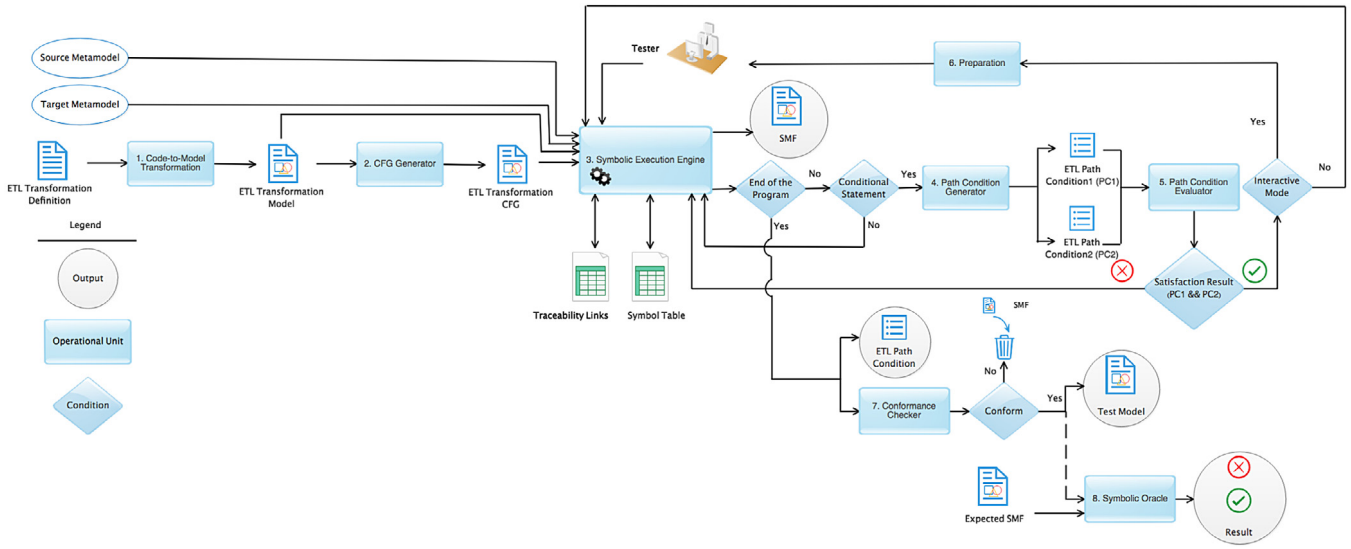


Fig. 4. The SEET approach for symbolic execution of ETL transformations.

3.3. Step 1: Obtaining ETL transformation model from ETL transformation definition

To navigate different execution paths, the transformation must be parsed. ETL code is transformed into a model to facilitate symbolic execution at a high level of abstraction. This model serves as a heterogeneous Abstract Syntax Tree (AST) which conforms to the ETL metamodel. One of the capabilities of the Epsilon Haetae tool is to generate the ETL transformation model from its code. For this reason, in our approach, the Haetae tool is integrated into the SEET plugin to accomplish the task of Step 1.

3.4. Step 2: CFG generator

In the CFG generator, the CFG of the ETL transformation is created from the ETL transformation model. It should be noted that the CFG generator of ETL can be used as a separate unit from symbolic execution. At a high level, Listing 5 describes the algorithm for generating CFG of ETL model transformations. The metamodel of the CFG is depicted in Fig. 5. A CFG contains several nodes. Each Node has a name, which is an integer number except the end node. Nodes have a visited Boolean attribute, which shows whether this node has been visited. Each node can have several incoming and outgoing references. The first node has no incomings, and the end node does not have any outgoing references. Nodes have an *etlelement* reference which refers to an ETL element, particularly, the related statement.

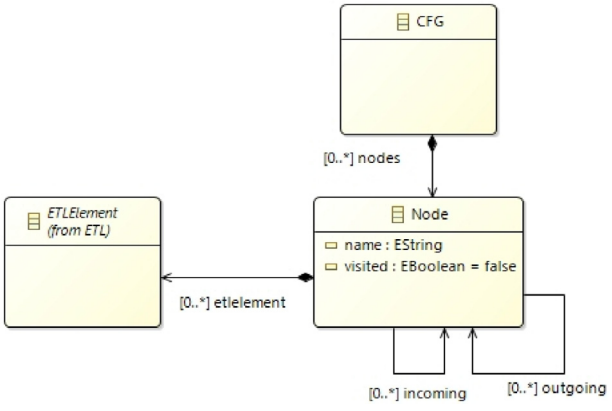


Fig. 5. The CFG metamodel of the ETL transformation.

Fig. 6 shows the CFG of the Families2Persons transformation in Listing 1. Specifically, the statement number of nodes is equal to the line number of the transformation in Listing 1. In the beginning, the nodes of pre-block statements are created sequentially. If several pre-blocks are available, the nodes of pre-blocks statements will be created in the same order they have appeared in the ETL model (lines 2–4 of Listing 5). In Fig. 6, first, the node for the statement of the pre-block is created. Then, the nodes of the rules are generated in the same order as they are specified in the transformation (lines 10–11 of Listing 5). In the

Fig. 6. The CFG of the Families2Persons transformation in ETL.

absence of lazy rules, rules are executed in two passes: (1) Rule matching and (2) Rule application. The rule matching phase is to specify whether a rule is executable. This phase checks if at least one instance of the source parameter of the rule exists. Additionally, in case the rule has a guard, it checks whether the guard is satisfiable. Following that, in the rule matching phase the node is created for the entry condition of the rule, i.e., at least one instance of the source parameter of the rule must exist. If the entry condition is not satisfied, we go to the matching phase of another rule. Otherwise, since a guard exists for the Member2Male rule (line 9 of Listing 1), a node for this guard is created. Importantly, for the operation call of the guard, the operation definition along with all its statements appears in the flow graph. There are some edges between return statements of the operation (stmt25, stmt28, stmt30) and the guard node. If the guard is not satisfied, we go to the matching phase of the Member2Female rule; otherwise, the node is generated to create traceability link. Afterward, the Member2Female matching phase is started. The same procedure is performed for this rule. After that, we go through the rule application phase. In this phase, statements of the body of the executable rules are executed, so their related nodes are created. First, a node is created to check whether the rule exists in the traceability links, i.e., the rule is executable. Then, the node for the assignment statement on line 11 of Listing 1 is created. In this assignment statement, the familyName() operation call exists. To that end, the definition of this operation appears in the flow graph. Some edges between its return statements and the assignment statement are created to return the value. Besides, as this assignment statement is the last statement in the body of the rule, there is an edge from this assignment statement to the node for check traceability of this rule. Then, the flow graph goes through the Member2Female rule application phase similar to the Member2Male rule. Following that, the nodes of post-blocks are created (lines 12–14 of Listing 5). Finally, the end node is created (line 15 of Listing 5). Since there are no post blocks in the Families2Persons transformation, the end node appears.

Fig. 7 shows CFG examples of some features of an ETL transformation, which have not been used in our running example. Fig. 7a concerns For and While loops in ETL. Note that the execution of a switch statement does not go to the next case after a successful one (Fig. 7b). It is worth mentioning that, the execution of matching and application phases of rules in ETL varies depending on the presence or absence of lazy rules and a call for equivalent(s). When we have a call for equivalent(s) operation, in the absence of lazy rules, first, all rules are matched in the order in which they appear. Then, application phases of rules are performed in order until an equivalent(s) is encountered. At this time, the application phases of applicable rules are executed (Fig. 7c). Finally, the application phases of the remaining rules are carried out. In the presence of lazy rules, by reaching each non-lazy rule, its matching and application phases are executed in one pass when an equivalent(s) operation call is encountered. At this time, applicable rules (including lazy and non-lazy rules) are invoked, which have not already been executed. First, the matching phases of all these rules are carried out. Then, their application phases are carried out one by one. After the invocation, execution is continued normally, i.e., the matching and application phases of remaining non-lazy rules are executed in one pass (Fig. 7d). The explanation about the call for equivalent(s) also holds for the special assignment statement.

3.5. Step 3: Symbolic execution engine

The symbolic execution algorithm is provided in Listing 6. In this listing, IPE stands for In-Pattern Element and OPE stands for Out-Pattern Element. For the symbolic execution of the ETL transformation model, at the onset, symbolic execution engine defines several data structures as follows.

Algorithm 1: Control Flow Graph of ETL Transformations

```
SelectedRules: Sequence
Let Rule-app(tr) be SelectedRules.add(tr); Create StringExpression with "Create Traceability link for tr.name" value; CreateNode
(StringExpression, false, tr);
1 Create CFG EPackage
2 forAll pre in preBlock
3         forAll s in statements of pre
4                 CreateNode (s, false, null)
5 if lazy rule exists
6         forAll tr in TransformationRules
7                 CreateNode (tr, true, tr)
8         forAll tr in TransformationRules which not exist in SelectedRules
9                 Rule-app(tr)
10 else forAll tr in non-Lazy TransformationRules
11                CreateNode (tr, true, tr); Rule-app(tr);
12 forAll post in postBlock
13        forAll s in statements of post
14                CreateNode (s, false, null)
15 Create end Node; Set its incoming(s); Set outgoing(s) of previous Nodes to end Node
16 CreateNode(s: ETL!EOLElement, matching: Boolean, rule: ETL!TransformationRule):
17        Create Node, Set its name, etlelement and incoming(s); Set outgoing(s) of previous Nodes to this Node
18        if s has operation
19                CreateNode(operationDefinition, false, null)
20                forAll s in statements of operation
21                        CreateNode(s, false, null)
22                Set incoming(s) of the caller statement to return statements
23                Set outgoin(s) of return statements of the operation to the caller statement
24        if s has equivalent(s) or is of type Special Assignment Statement
25                if lazy rule exists
26                        forAll tr in non-Lazy TransformationRules which not exist in SelectedRules
27                                CreateNode (tr, true, tr); Rule-app(tr)
28                else            forAll tr in applicable rules
29                                Rule-app(tr)
30        switch (Type of s)
31                case "IfStatement":
32                        forAll s in statements of ifBlock
33                                CreateNode(s, false, null)
34                        if has elseBlock
35                                forAll s in statements of elseBlock
36                                        CreateNode(s, false, null)
37                case "ForStatement" or "WhileStatement":
38                        forAll s in statements of loopBlock
39                                CreateNode(s, false, null)
40                        Set incoming of LoopStatement to last statement of loopBlock
41                        Set outgoing of last statement of loopBlock to LoopStatement
42                case "SwitchStatement":
43                        forAll case in cases of SwitchSatement
44                                forAll s in the statements of case
45                                        CreateNode(s, false, null)
46                        forAll s in the statements of default
47                                        CreateNode(s, false, null)
48        if rule <> null
49                if matching and rule has a guard
50                        CreateNode(guard, false, rule)
51                if rule has not guard or the node for a guard is created
52                        Create StringExpression with "Create Traceability link for ruleName" value;
53                        CreateNode(StringExpression, false, null)
54                else if not matching
55                        forAll s in statements of body of the rule
56                                CreateNode(s, false, null)
57                    Set incoming of CheckTraceability to last statement of body
58                    Set outgoing of last statement of body to CheckTraceability
```

Listing 5. Generating CFG for ETL transformations

- Path condition (PC) is of type Sequence
- Symbol table which is implemented as a Stack
- Traceability link which is of type Sequence

The path condition has been considered as logical conjunction of the conditions on the symbolic input model. These conditions

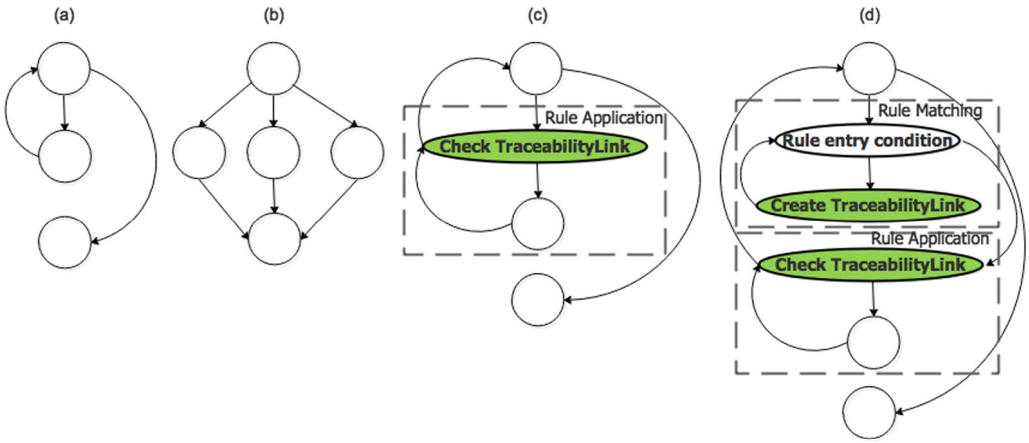


Fig. 7. Some CFG examples for ETL features: (a) while/for statement, (b) switch-case statement, (c) equivalent(s) operation calling in the absence of lazy rules, (d) equivalent(s) operation calling in the presence of lazy rules.

Algorithm 2: Symbolic execution of ETL Transformations

```
PC: Sequence
SymbolTable: Stack
TraceabilityLink: Sequence
Let ExecutionState be SymbolTable, TraceabilityLink, executableRules and SMF
1 Create Package
2 Get mode
3 forAll n in nodes of CFG
4           n.visited = false
5 if nodes[1].ETLElement is of type TransformationRule
6           insert entry condition to PC; Save ExecutionState
7 else      Symbolic Execution (nodes[1].ETLElement)
8 nodes[1].visited = true; Push nodes[1] to DFS;
9 while (DFS is not Empty)
10          Calculate adjacent
11          if node is the last statement in the block      SymbolTable.popScope()
12          if adjacent == null      DFS.pop()
13          else if adjacent == end
14                      if PC is Empty           DFS.popAllElements()
15                      else if mode == Interactive         DFS.popAllElements()
16                      else while PC.last.get('two-paths') <> true    PC.remove(PC.last)
17                           if PC is not Empty
18                                  while DFS.peek.name <> PC.nodeNumber    DFS.pop()
19                                  Create Package; Restore ExecutionState
20          else if adjacent.ETLElement is of type TransformationRule
21                      adjacent.ETLElement.ConditionalStatement(); Save ExecutionState
22          else if adjacent.ETLElement is of type Create TraceabilityLink
23                      forAll target in targets of tr
24                                  if the type of target not exist in TraceabilityLink          Create EClass
25                                  Add {tr.name, tr.source, target} in TraceabilityLink
26                      Add tr in SelectedRules
27          else if adjacent.ETLElement is of type Check TraceabilityLink
28                      if tr exist in executableRules and adjacent.visited == false
39                                  SymbolTable.push(IPE)
30                                  forAll OPE in OutPattern elements of tr
31                                              SymbolTable.push(OPE)
32          else if DFS.peek() is type of TransformationRule
33                      guard = DFS.peek().ETLElement.Condition()
34          else SymbolicExecution (adjacent)
35          adjacent.visited = true; Push adjacent to DFS;
36 Show Result
```

Listing 6. pseudo code of symbolic execution of ETL Transformations

have been encountered along a path of the transformation. If a path is feasible, the path condition is satisfiable.

The symbol table is a data structure, which contains information about variables, including the source and target parameters

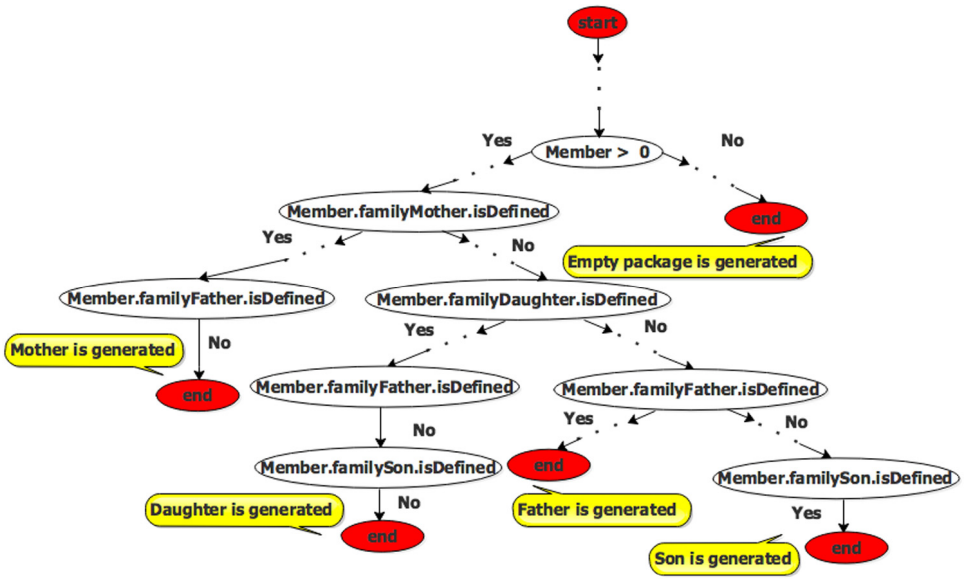


Fig. 8. Satisfiable paths of the Families2Persons transformation.

of the rule as well as variables declared in the body of the rule. When a variable is looked up, it starts from the top of the stack which represents the current scope. Sequentially first the current scope is searched then containing scopes and after all pre-blocks are sequentially searched to find the desired variable. Some scopes in ETL transformations are characterized by the following items.

- Pre-block
- Post-block
- Rule
- If block
- For loop
- While loop
- Operation

Traceability links indicate the relation between the source and the target parameters of the rule. In other words, they are indicative of which target parameter is associated with which source parameter in which ETL rule. To illustrate that, the information of traceability links is used when an *equivalent()* operation is called in the transformation. Each rule with a satisfiable guard (or no guard at all) has a record in the traceability link sequence for each of its target elements. For the running example, {'Member2Male', 'Member', 'Male'} is a record in the traceability link.

The symbol table and the traceability links are presented with 2-way associations in Fig. 4 because the symbolic execution engine can either insert/update or fetch a record.

To perform the symbolic execution in an automatic manner, SEET must be able to traverse different execution paths. To this end, we take advantage of Depth-First Search (DFS) in order to prepare the explored paths quickly with low memory usage. To traverse the CFG in the DFS manner, consider Fig. 8, which shows the idea behind the symbolic execution engine. By reaching any conditional statement, if it (or its negation) does not already exist in the PC, the condition is pushed to the PC. By reaching any forkable conditional statement (such as "Member $>$ 0"), called two-paths condition, the condition is pushed to PC while its two-paths attribute is set to true. Then, the left path, which always considers the condition to be true, is selected. In the selected path, the engine reaches another forkable condition "Member.familyMother.isDefined". Following the left path, "Member.familyFather.isDefined" is reached. Since according to the previous conditions in this path, the condition proved unsatisfiable (a Member cannot be both mother and father at the same time), it is not a forkable condition. Therefore, the only path which can be traversed is selected by the engine. The execution is continued until reaching the end node. At the end of this path, Mother is generated. It is interesting to note that the engine finishes its work in the interactive mode. However, in the automatic mode, at this time, the engine backtracks while it pops conditions from PC until arriving at the two-paths condition (a condition with true two-paths attribute), "Member.familyMother.isDefined" condition. Then, its two-paths attribute is set to false and the second path of this condition is selected which considers the condition to be unsatisfied. This process goes ahead until all the execution paths are traversed and the PC becomes empty.

According to Listing 6, the symbolic execution engine performs the following actions. In the beginning, a package is created on the SMF (line 1). Henceforth, any elements created on the SMF are contained in this package. Symbolic execution of ETL transformations begins at specifying the symbolic execution mode by the tester. Two modes are available: (1) Automatic and (2) Interactive. Assuming the tester selects the automatic mode, no more questions will be asked. First, all nodes of CFG are initialized to be not visited (lines 3–4). The symbolic execution begins at first node and after that, the adjacent of this node is calculated. The execution continues in the same way until the end node is reached. After that, another path is selected and the ExecutionState is reset until all paths are traversed.

Overall, the symbolic execution engine, when it encounters any non-conditional and non-iterative statements (such as assignment statement, expression statement, etc.), symbolically executes each statement, updates the symbol table in need, makes its *visited* true, and inserts the node number into DFS. It is clear that the PC is not changed in this case. For example, when node number 1 arrives in the CFG of Fig. 6, the node number of stat2 (i.e., 1) is pushed to DFS stack and the visited of the node becomes true. For this statement, the symbol table and the SMF are not changed as shown in Fig. 9a.

The evaluation of the path condition can be performed lazily or eagerly (Baldoni et al., 2018). In the lazy evaluation, the satisfiability of the path condition is asserted at the end of the transformation. In the eager evaluation, the path condition is asserted at each branch. Our approach makes use of eager evaluation.

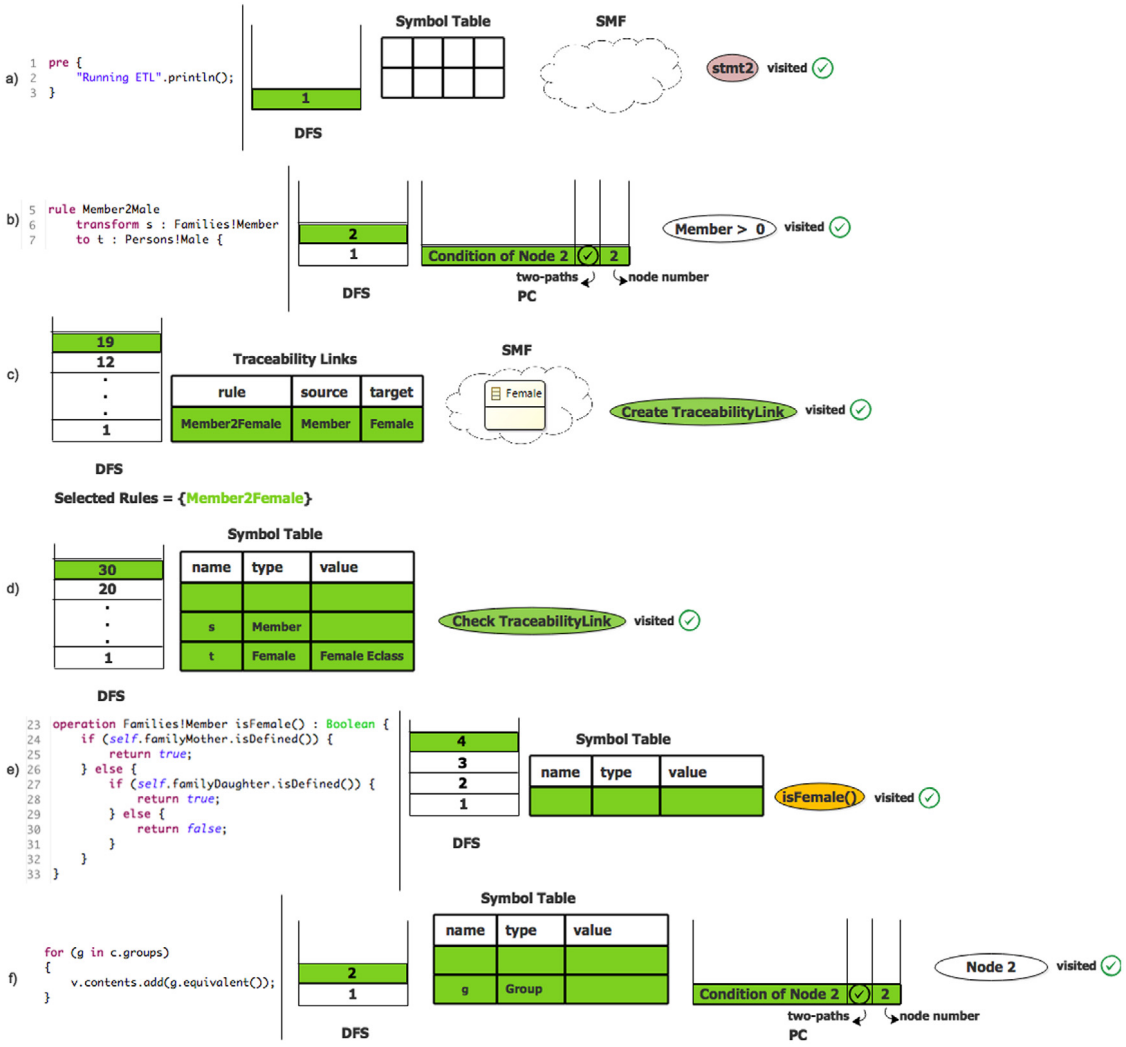


Fig. 9. Symbolic execution of CFG nodes, (a) non-conditional statements, (b) conditional statements, (c) Create traceability link nodes, (d) Check traceability link nodes, (e) operations, (f) iterative statements.

By reaching any conditional statements which can be forked (such as if statements or entry condition of the rule, or guard of rules which do not call any operations), the engine inserts its condition into PC along with assigning true to its two-paths feature that helps once backtracking, inserts the corresponding node number into DFS, and makes its visited true in the CFG. The ExecutionState is also saved. By ExecutionState we mean the traceability link, the symbol table, executable rules, and the DFS stack. For example, confronting the entry condition of the Member2Male rule, a new record (2) is pushed into the DFS stack. The visited attribute of its node is set to true and the condition is pushed into PC while its two-paths attribute is true and the node number is 2 (Fig. 9b).

To evaluate any Create traceability link nodes, traceability links for the corresponding rule are created and its visited attribute is set to true. The node number is pushed into the DFS stack and class(es) of the type of the target(s) of the rule are generated into the corresponding package of the SMF (Fig. 9c).

To evaluate any Check traceability link nodes, whose rule exists in the selected rules, at first, a null record should be pushed into the symbol table to denote a new scope. Then, new records for the source and target(s) of the corresponding rule should be inserted as presented in Fig. 9d.

When it encounters any operation definitions, a null record is inserted in the symbol table, the node number is pushed to DFS and the visited attribute of its node will be set to true (Fig. 9e).

As shown in Fig. 9f, when it encounters any iterative statements (such as For statements or While statements), the engine inserts its condition into PC along with assigning true to its two-paths feature that helps backtracking once, inserts the corresponding node number into DFS and sets the 'visited' attribute to 'true' in the CFG. Assuming it is a For statement, its iterator is inserted into the symbol table. The ExecutionState is also saved. For example, confronting the For statement of Fig. 9f, a new record (2) is pushed into the DFS stack. The condition is pushed into PC while its two-paths attribute is true and the node number is 2. The visited attribute of its node is set to true.

In the interactive mode, one path of the transformation is traversed and the tester determines the satisfiability of conditions (or guards). Admittedly, the tester is not asked to specify the satisfiability of conditions if the satisfaction result can be inferred from the current path condition. In other words, as depicted in Fig. 10, when a condition is encountered, two path conditions are considered: One by logical conjunction of the condition and PC, and another by logical conjunction of the negation of the condition and PC. Assuming that both PCs are satisfiable (Fig. 10-1), the tester is asked to determine satisfiability of the reached

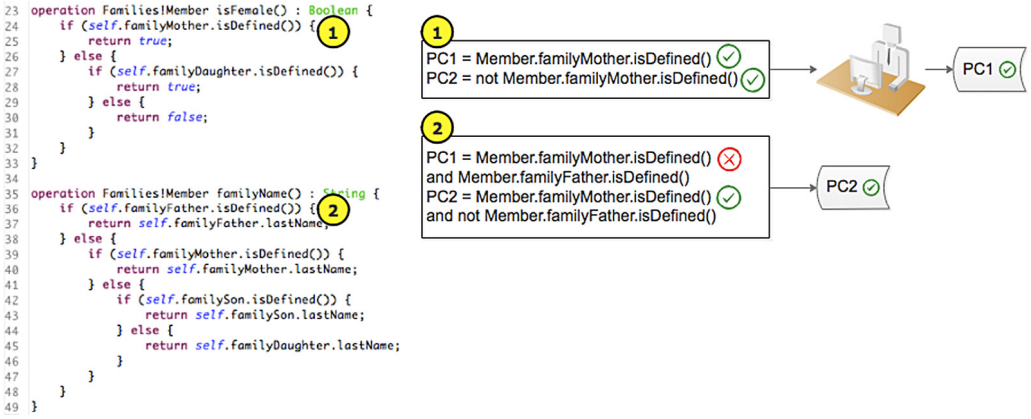


Fig. 10. Encountering condition in the path in the interactive mode.

condition. Then, symbolic execution is continued with the satisfiable PC. If one of these PCs is satisfiable (Fig. 10-2), from the path condition evaluator viewpoint, the symbolic execution of the transformation is continued with the satisfiable PC. With this in mind, henceforth, when a condition/guard is considered, we avoid repeating the explanation.

First, the tester specifies which rule guards are satisfiable. Considering the running example of Listing 1, first, the guard of the Member2Male rule in line 9 is considered. Since in this guard, there exists an operation call, the operation statements are symbolically executed according to CFG. Wherever a condition exists in these statements, the tester is asked about the mentioned condition. Herein the first condition is in line 24 and the tester is asked about the condition.

Assuming the tester considers the "Member.familyMother.isDefined()" condition to be satisfiable, the tester specifies that the Member is the mother of the family, the symbolic execution engine reaches the return statement.

Then, in the Create traceability link node, classes of the type of the target parameters of rules with satisfiable guard are created in the output only on the condition that the class of that type has never been created in the output. The variables of the source parameter and target parameters of the rule are stored in the symbol table. Traceability links are also created.

Then, it is the turn of the guard of the Member2Female rule. The guard of this rule is a negation of the guard of the Member2Male rule which was asked earlier. Thus, satisfiability of this guard is inferred without interaction with the tester.

Next, in the Check traceability node assuming the rule exists in the selected rules, a new record is added to the symbol table including the name and the type of the source parameter of the rule. For any target parameters of the rule, a record is inserted into the symbol table. Next, statements of the body of the rule are executed symbolically.

There is one assignment statement in line 20 of Listing 1, which calls the familyName operation on its RHS. The symbolic execution engine executes the statements of the operation. In the first line of the operation, it confronts the "self.familyFather.isDefined()" condition. This condition or negation of this condition has never been asked. However, since considering this condition as satisfiable causes the generated path condition to be unsatisfiable according to the path condition evaluation unit, the symbolic execution engine considers this condition unsatisfiable, and symbolic execution goes ahead. Then, the symbolic execution engine goes through the else block and arrives at the "self.familyMother.isDefined()" condition. As the tester has been asked about the condition earlier, the previous result is used and the tester is not asked again about the condition. Since the condition is specified as satisfiable, the symbolic execution engine arrives at the return statement and returns the symbolic value. The result of the assignment statement is applied to the SMF.

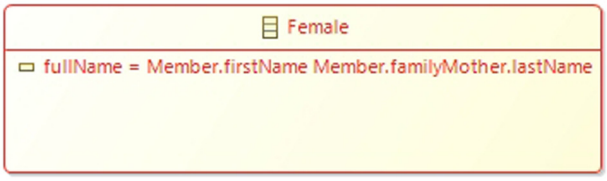


Fig. 11. The SMF of the Families2Persons transformation when member is mother.

Afterward, by achieving the Check traceability link of this rule again, records inserted for variables of the rule are popped out from the symbol table. This is done for every non-lazy rule. Eventually, statements of post-blocks are executed.

At the end of the symbolic execution, the result is shown to the tester. The result involves the path condition, the time of symbolic execution (it does not include the interaction time) and a test model that if given to the transformation, the specified path is traced. Furthermore, the SMF of Fig. 11 is generated.

To demonstrate that, we provide the symbolic execution tree in Figs. 12 and 13. The figures show executable paths of the transformation along with the way in which the SMF is shaped, assuming there exists an instance of Member class. At any states, in the upper half of the state, PC is a path condition, S points to the symbol table, and the Target package maintains the SMF while the lower half of the state indicates the next statement must be executed. It is interesting to note that the tool does not go through unsatisfiable paths of the transformation shown in the figures.

Symbolic execution of statements

Listing 7 contains the algorithm for the symbolic execution of the statements. In Table 2, we elaborate on the symbolic execution of the statements based on the type of the statement. In addition, the example of each statement, its corresponding SMF, and PC are also provided. About For statement and While statement, note that in most cases, the number of times that a loop is executed does not affect the SMF. Hence, the body of the loop is symbolically executed at most once. More specifically, when the iterator of the loop is defined on a model element, such as a reference, the number of loop executions does not affect the result. The reason is that the SMF is the part of the target metamodel. Thus, there is one reference with that name and the mentioned reference, which presents several instances

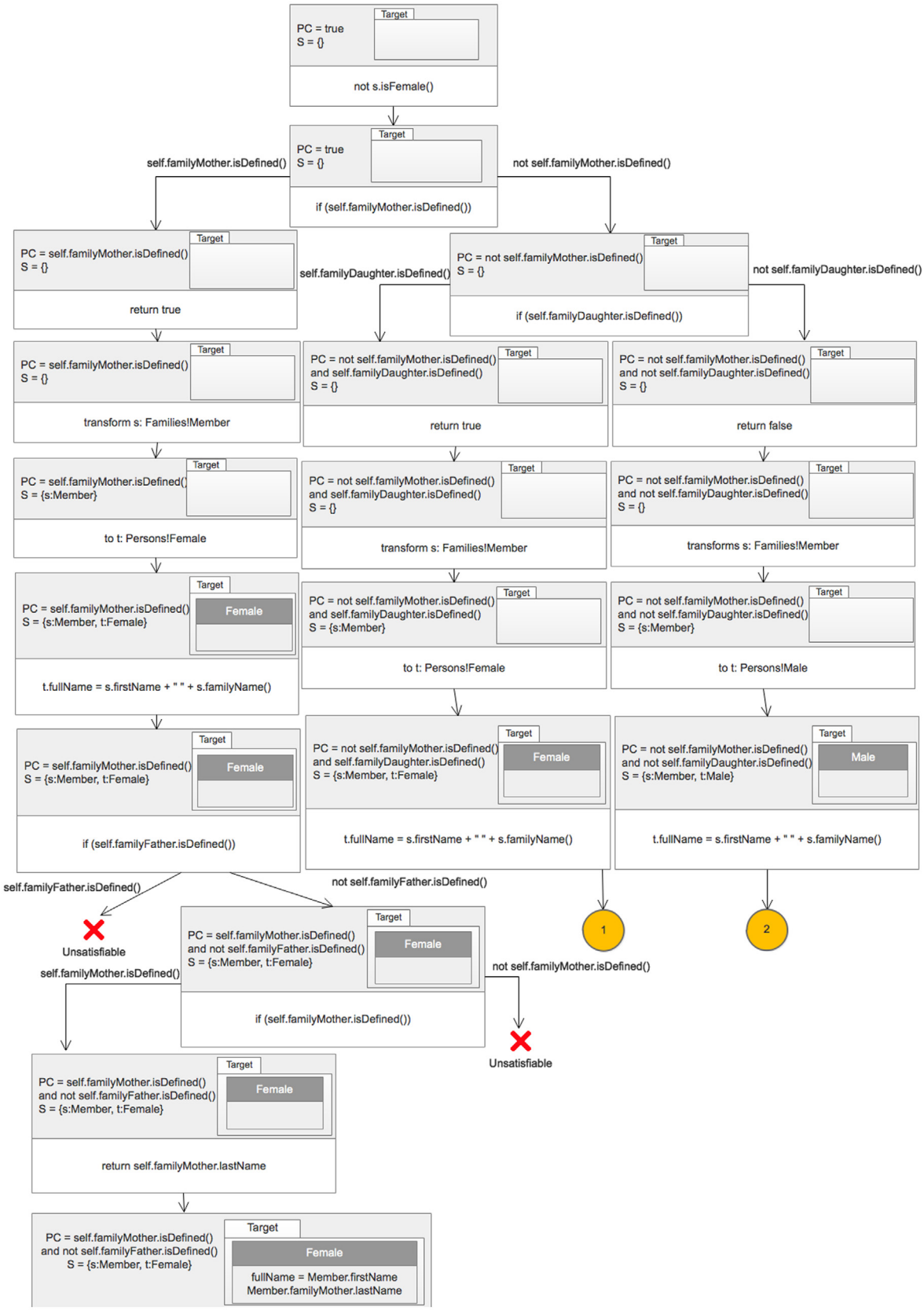


Fig. 12. The symbolic execution tree of the Families2Persons transformation (part 1).

of it, manipulated in the body once, and in the second iteration, the result does not change. The number of the loop executions can affect the SMF under special circumstances in which the local variable, which is affected in the loop body, is used for a model element after the loop. We ignore it to avoid getting into infinite loops.

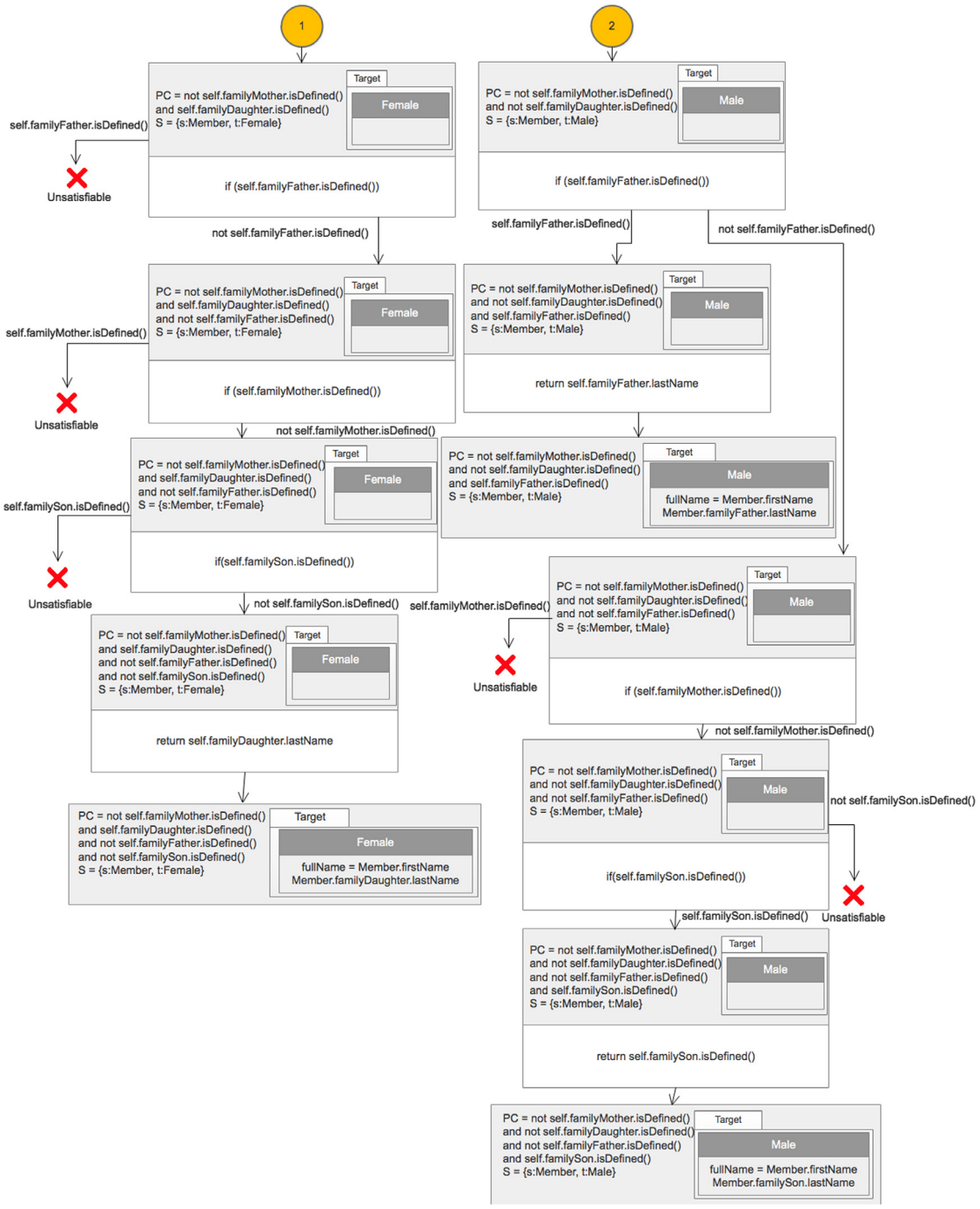


Fig. 13. The symbolic execution tree of the Families2Persons transformation (part 2).

After describing the SEET process, it proves beneficial to list the features of ETL transformations that are covered by the tool. These features are shown in Table 3.

Given the shown cases, although SEET fails to cover all features of ETL transformations, it appropriately covers useful statements of ETL transformations. At the same time, the tool can engage with single input metamodels and single output metamodels, as we take only one input metamodel and one output metamodel from the tester. For future work, the remaining features of ETL syntax will be supported in further development. This approach is premised on the assumption that there is at most one instance of input metamodel classes. Since ETL does not support in-place transformations, SEET does not engage with this kind of transformation either.

3.6. Step 4: Path condition generation

When a condition is encountered, the path condition generation produces two path conditions. One is obtained by applying logical conjunction between the recent condition and the current path condition. Another is computed by applying logical conjunction between the negation of the recent condition and the path condition. For example, consider Fig. 10, in which the engine reaches the point of "self.familyFather.isDefined()" condition. Since the current path condition until this point is "not

Algorithm 3: Symbolic execution of statements

```
Let SBP() be symbolic execution of statements of the block and then SymbolTable.popScope()
SymbolicExecution (s):
1 switch (Type of s)
2       case "AssignmentStatement":
3               switch (lhs of s)
4                       case "PropertyCallExpression":
5                               if s initializes an Attribute of OPE
6                                       if the attribute not exists     Create EAttribute
7                                       else    Create EnumLiteral
8                               else    Create EReference
9                       case "NameExpression":          Update SymbolTable
10                      case “VariableDeclarationExpression”:
11                              if s.rhs is Type of NewExpression Create EClass
12                              SymbolTable.push(newRecord)
13      case "SpecialAssignmentStatement":
14              if  s.rhs is of Type "PropertyCallExpression" and not s.lhs is of Type "PropertyCallExpression"
15                      Update SymbolTable
16              else    Create EReference
17      case "If Statement":
18              if  s.condition is satisfiable          insert condition to PC; SBP()
19              else    insert (not condition) to PC; SBP()
20      case "ExpressionStatement":     SymbolicExecution (s.expression)
21      case "VariableDeclarationExpression":
22              if has new or s is of Type "ModelElementType"           Create EClass
23              SymbolTable.push(newRecord)
24      case "PlusOperatorExpression":  return SymbolicExecution (s.rhs) + SymbolicExecution (s.lhs)
25      case "PropertyCallExpression":
26              if there exists an attribute in s       return type of target.SymbolicExecution (s.property)
27              else    return type of reference
28      case "PrimitiveExpression":     return s.value
29      case "MethodCallExpression":
30              if s.method is operation        SBP()
31              else    switch (s.method.name)
32                              case "add":             Create EReference
33                              case "firstToLowerCase" or "firstToUpperCase" or “toLowerCase” or “toUpperCase”:
34                                      return SymbolicExecution(target of s) + "." + s.method.name + “()”
35                              case "first":           return SymbolicExecution(target of s)
36                              case “equivalents”:
37                                      return types of targets of rules which type of IPE = type of target of method
38                              case "equivalent":
39                                      return type of the first target of rules which type of IPE = type of target of method
40      case "NameExpression":
41              if s refer to self      return context
42              else    return s.name
43      case "ReturnStatement":         SymbolicExecution (s.expression)
44      case "ForStatement":
45              if s.condition is satisfiable
46                      SymbolTable.push (newrecord) //counter
47                      insert condition to PC; SBP()
48      case "WhileStatement":
49              if s.condition is satisfiable           insert condition to PC; SBP()
```

Listing 7. pseudo code of symbolic execution of statements

Member.familyMother.isDefined()”, the following path conditions are generated:

- Member.familyFather.isDefined() and not Member.familyMother.isDefined()
- Not Member.familyFather.isDefined() and not Member.familyMother.isDefined()

At the end of the program, for the convenience of the user, the obtained ETL path condition from the symbolic execution of the transformation is simplified. The simplification includes applying formula (1) (Boolean Idempotent law) and formula (2) (Double Negation law).

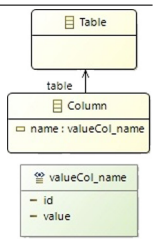
$$a(\wedge a)^K = a \quad k \geq 0 \quad \& \quad k \in \mathbb{Z} \tag{1}$$

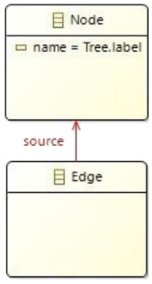
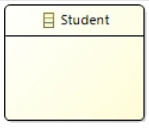
$$(\neg)^k a = \begin{cases} a & k = 2k' \quad \& \quad k' \in \mathbb{Z} \\ \neg a & k = 2k' + 1 \quad \& \quad k' \in \mathbb{Z} \end{cases} \tag{2}$$

In an attempt to simplify the path condition, according to formula (1), if a variable is repeated within the expression comprising logical conjunction, it can be effectively simplified to itself (Boolean Algebra, 2018). This means that when the same conditions are logically conjunct, they may be simplified to one of the conditions. For this purpose in mind, the algorithm prevents adding the repetitive PC. Moreover, regarding formula (2) (Boolean Algebra, 2018), if a condition is negated even times, it may be simplified as the condition. When a condition is negated odd times, it may be simplified as the negation of the condition.

Table 2
Symbolic execution of statements with corresponding example, SMF, and PC.

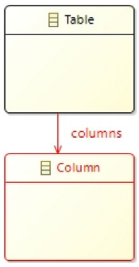
Statement	Description	Example	SMF/PC

Assignment statement	(a) Initializes an attribute of OPEs (line 5 of Listing 7): if the attribute has never been created in that class in the SMF, the attribute is created. Then, the symbolic value of RHS is calculated and assigned to the value of the attribute. If the attribute has been already initialized and the new value is not equal to the old one, Enum is created. Provided that Enum has been already created and the new value does not exist in the Enum, an EnumLiteral is added to Enum.
	(b) Initializes a reference of OPE (line 8 of Listing 7): if the reference has never been created, a new reference is created with that name. The type for the reference is defined based on the type of RHS.
	(c) Initializes a name expression (line 9 of Listing 7): the value of the variable of LHS is updated in the symbol table.
	(d) Initializes a variable declaration expression (line 10 of Listing 7): if RHS is a new expression a class of type elementName is created. A record with the name of the variable is inserted to the symbol table.
		rule MultiValuedAttribute2Table
		transform a : OO!Attribute
		to t : DB!Table, pkCol : DB!Column,
		valueCol : DB!Column,
		fkCol : DB!Column {
		pkCol.name = "id";
		pkCol.table = t;
		valueCol.name = "value";
		valueCol.table = t;
		}[a]
			

Special assignment statement	(a) If LHS is not property call expression, the values of the record in the symbol table whose name is equal to LHS will be updated. The update value is set to the first target parameter of the first rule, the type of whose source parameter is equal to the type of the reference of RHS (lines 14–15 of Listing 7). (b) If the property of LHS is not included in the references of the class, the reference with the name of the property is created. Moreover, the type of the first target parameter of rules, in which the type of source parameter is included in the set of the type of the reference of RHS and classes whose supertypes contain the type of RHS, is added to the type of this reference. The results of primary rules are preceded (line 16 of Listing 7).	e.source ::= t.parent;[b]	
If statement	If the condition does not exist in the PC, (a) For a satisfiable condition, it is added to the PC. Then, the statements of ifBlock are executed when the engine reaches their corresponding nodes in the CFG. (b) According to an unsatisfiable condition, its negation is inserted into the PC. Provided that the condition contains else part, statements of elseBlock are executed when the engine reaches their corresponding nodes in the CFG. Finally, the records of this scope are popped from the symbol table (lines 17–19 of Listing 7).	Lines 24–26 of Listing 1	PC = Member.familyMother.is Defined()
Expression statement	Its expression is symbolically executed (line 20 of Listing 7).	Line 2 of Listing 1	–
Variable declaration expression	(a) If there exists a *new* operator in the variable declaration or the declaration is of type ModelElementType, when there is not a record in the symbol table or traceability link whose type is equal to elementName of the expression, a class of type elementName is created in the output. A record with the name of that variable and of type elementName is inserted into the symbol table (lines 21–23 of Listing 7)	(a) var student : new Student;	
	(b) Otherwise, a record with the name and type of that variable is inserted into the symbol table.	(b) var b: String;	–
Plus operator expression	The results of calculating the two sides of the operator are added together (line 24 of Listing 7).	s.firstName + " str"	Member.firstName str
Property call expression	(a) Providing that the called property is an attribute or is a reference which is contained in another property call expression whose property is an attribute, the symbolic execution of these expressions adhere to the pattern as follows. The pattern is equal to "the type of the target" followed by "the symbolic expression of the property", separated by a dot symbol (line 26 of Listing 7).	"s.firstName" in line 11 of Listing 1	"Member.firstName"
	(b) Otherwise, the type of the reference is returned (line 27 of Listing 7).		
Primitive expression	Their values are returned (line 28 of Listing 7)	2	2
Method call expression	(a) Operation: When the engine arrives its statements, they are executed symbolically. In the end, the records of this scope are popped from the symbol table (line 30 of Listing 7).	"s.isFemale()" on line 9 of Listing 1	True (or False)

(*continued on next page*)

Table 2 (*continued*).

Statement	Description	Example	SMF/PC
	(b) Add(): As the method usually is applied to the property call expression, first the type of the target of the property call expression is obtained. If the class did not have a reference with the name of the called property, a reference is created in the output. As a final stage, types of "add" method arguments are assigned to this reference (Line 32 of Listing 7).	t.columns.add(pk) [a]	
	(c) firstToLowerCase() or firstToUpperCase() or toLowerCase() or toUpperCase(): Since it is not clear if the first letter of symbolic expression gets lowered, after the target of the method call expression is symbolically executed, ".method name()" is concatenated at the end of its string (lines 33–24 of Listing 7).	"self.name.firstToLowerCase()"[a]	Table.name. firstToLower Case()
	(d) First(): Its target is symbolically executed, if it is of collection type, the first element is returned, otherwise, first() is ignored because this method does not have any meaning in the abstract view (line 35 of Listing 7).	"parentTable.primaryKeys.first()"[a]	Column
	(e) Equivalents(): The types of the target parameters of the rules are returned, in which the type of its source parameter is the same as the type of the called reference. The results of primary rules are preceded (line 36–37 of Listing 7).	g.equivalents()	{Label}
	(f) Equivalent(): The first element of symbolic execution of equivalents method is returned (lines 38–39 of Listing 7).	g.equivalent()[c]	Label [class]
Name expression	(a) If it refers to *self* in an operation, the context of the related operation is returned (lines 41 of Listing 7).	Line 37 of Listing 1 (self)	Member
	(b) Otherwise, the name of the name expression is returned (line 42 of Listing 7).	Line 11 of Listing 1 (t)	t
Return statement	Its expression is symbolically executed (line 43 of Listing 7)	Lines 25, 37 of Listing 1	–

For statement	For a satisfiable condition, properties of the counter, including its name and its type, are inserted into the symbol table. The condition is inserted into the PC, if it does not exist. Then, by traversing next nodes in the CFG, the statements of the body of ForBlock are executed. In the end, the records of this scope are popped from the symbol table (lines 44–47 of Listing 7).	for (g in c.groups) {
		//Something
		}
			PC = [PC and] Competition .groups. allInstances() -> size() <> 0

While statement	If the condition does not exist in the PC, the condition is added to PC. In case the condition is satisfiable, by traversing next nodes in the CFG, statements of the body of WhileBlock are executed. In the end, the records of this scope are popped from the symbol table (lines 48–49 of Listing 7).	while (c.groups <> null)
		{
		//Something
		}
			PC = [PC and] Competition .groups .allInstances() -> size() <> 0

Switch statement	By traversing adjacent nodes in the CFG, the satisfiable case is selected. The corresponding condition is inserted into the PC. Next, the statements of the satisfiable case are executed. Finally, the records of this scope are popped from the symbol table.	switch (g.type){ case gui!Attribute#Linear: tipo = "lineal"; default: tipo = "tabla"; }[d]	PC = [PC and] Gui.type == "Linear"

[a] https://www.eclipse.org/epsilon/examples/index.php?example=org.eclipse.epsilon.examples.oo2db
[b] https://www.eclipse.org/epsilon/examples/index.php?example=org.eclipse.epsilon.examples.tree2graph
[c] https://www.eclipse.org/epsilon/examples/index.php?example=org.eclipse.epsilon.examples.mddtif
[d] https://github.com/phillipus85/ETLMetricsDataset

3.7. Step 5: Path condition evaluation

To evaluate the path condition, as shown in Fig. 14, first the path condition is given to the containment preservation checker. Assuming there exists one instance of any input metamodel classes, this unit assesses whether any instance can only be in at most one containment reference. Also, since our assumption is that there exists one instance of any classes, in case

Table 3
ETL features covered and uncovered by SEET.

Feature		Support	Feature		Support
Several target parameters in the rule		✓	Annotations		✓
rule	No-annotation	✓	Variable declaration statement		✓
	Primary	✓	Assignment Statement	Variable assignment	✓
	Lazy	✓		Native object property assignment	–
	Abstract	–		Model element property assignment	✓
	Extend	–		Special Assignment Statement	✓
	Greedy	–	If Statement, Else part		✓
Guard		✓	Switch Statement		✓
Pre block, Post block		✓	While Statement		✓
Transaction Statement		–	For Statement		✓
Operation	Simple	✓	Break, BreakAll, Continue		–
	Polymorphic	–	Throw Statement		–

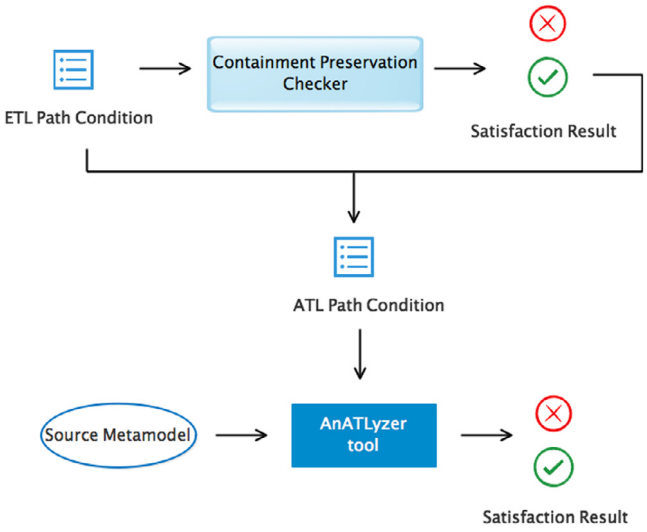


Fig. 14. The path condition evaluation unit.

the program is ended, the unit checks whether a class, which has at least one containment reference attends one containment reference. For example, in Fig. 13, it is not possible to set "self.familySon.isDefined()" to false, on the grounds that the previous conditions in this path show that this Member is not father, mother, and daughter and the Member has a reference of type containment. If the unit concludes that the path condition is not satisfiable, the symbolic execution comes to an end and the SMF is removed. In contrast, when the unit approves that the path condition is satisfiable, the ETL path condition is transformed to its equivalent ATL path condition. After that, the ATL path condition is transformed to an AnATLyzer-based ATL path condition. Then, it is given to the AnATLyzer tool (Cuadrado et al., 2017) along with the source metamodel as inputs. The AnATLyzer tool examines the satisfiability of the given path condition on the source metamodel.

To specify whether a path condition is satisfiable, the constraint solver is applied. Nowadays, several constraint solvers have been offered. Each has its own pros and cons. In particular, the UMLtoCSP (Cabot et al., 2007) tool fails to support EMF models and three-valued logic. A case in point is the EMFtoCSP (González et al., 2012) solver. Supporting EMF models gives EMFtoCSP a distinct advantage over UMLtoCSP. Though it benefits from supporting EMF models, it suffers from a lack of support for OCL's three-valued logic. By way of illustration, the null and undefined values are not supported by the tool. This is also true for the USE Model Validator (Kuhlmann et al., 2011) plugin. The tool supports three-valued logic. At the same time, the tool fails to support EMF models. Whereas the UML semantics is supported, the UML format is not considered. To make use of it, the metamodel and the OCL expression should be written in the USE format. Consequently, we use the AnATLyzer tool. Although AnATLyzer is a tool for static analysis of ATL transformations, it can also be used to check constraint satisfiability.[4] In this case, it serves as the wrapper of USE Validator. AnATLyzer covers some drawbacks of USE Validator such as not supporting EMF.

Since the AnATLyzer tool has already been applied for the ATL and OCL languages and it fails to support the ETL language, the transformation from ETL to ATL is required such that the part of the ETL model including an expression can be transformed to its equivalent ATL model. However, as the standard ATL metamodel differs from the ATL metamodel used in this tool, the achieved ATL model needs to be given to the ATLModel class of the AnATLyzer tool, and the equivalent ATL model which is informed by AnATLyzer is taken. The model contained an OCL expression which is based on AnATLyzer along with the source metamodel is given to the tool so that the tool specifies its satisfiability. The main demerit of AnATLyzer is that the tool can only evaluate single expressions, that is, operations are not supported and we cannot refer to a variable or a parameter. In other words, conditions/guards of the transformation are not supported directly. This is true when there exists an operation call in the condition or there exists a variable in the condition which refers to another expression. For example, its referredVariable is the source parameter of the rule or it refers to variable declaration out of this expression. To remove the first limitation, the operation is unrolled and its conditions are considered. For example, in order to have an evaluation on "s.isFemale()", the isFemale operation is unrolled and its conditions are considered such that "s.isFemale()" or its negation have not been seen in PC and condition(s) inside the isFemale operation can be seen instead. To overcome the second limitation, the expression is created in the form of the following.

$$Type.allInstances() \rightarrow exists(p|Condition)$$

For example, in order to have an evaluation of conditions such as self.familyMother.isDefined(), the condition is provided in the form of "Member.allInstanses() − > exists(p| not p.familyMother.oclIsUndefined())" for AnATLyzer.

ETL Expression to ATL OCLExpression Transformation

[4] https://github.com/anatlyzer/anatlyzer/wiki/Constraint-satisfiability

Table 4
ETL Expressions to ATL OCL Expressions mapping.

ETL	ATL
PropertyCallExpression	NavigationOrAttributeCallExp
NameExpression	VariableExp
StringExpression	StringExp
IntegerExpression	IntegerExp
MethodCallExpression	OperationCallExpression
NotEqualsOperatorExpression	
EqualsOperatorExpression	
GeaterThanOperatorExpression	OperatorCallExp
LessThanOperatorExpression	
GreaterThanOrEqualToOperatorExpression	
LessThanOrEqualToOperatorExpression	

Table 5
Examples of ETL Expressions and their equivalents in ATL.

ETL	ATL
t.parent.isDefined()	not t.parent.oclIsUndefined()
s.eContainer().isUndefined()	s.refImmediateComposite().oclIsUndefined()
mElem.name <>null	not mElem.name.oclIsUndefined()
mElem.kind = 'initial'	mElem.kind = 'initial'
not a.isMany	not a.isMany
string.length > 2	string.size() > 2
a.asInteger() > = 6	a.toInteger() > = 6

Due to the fact that ETL and ATL languages are both OCL-based, it is possible to transform ETL expressions to ATL OCL expressions. Table 4 shows the mapping we have considered. According to Table 4, any PropertyCallExpression in the ETL transformation is transformed to NavigationOrAttributeCallExp in the ATL transformation. Its property is mapped to the name of NavigationOrAttributeCallExp in ATL. Its target is transformed separately. Eventually, NavigationOrAttributeCallExp is assigned to the appliedProperty of the transformed target. Any NameExpression is mapped to VariableExp. Any StringExpression is transformed to StringExp and its value is assigned to stringSymbol. Any MethodCallExpression is transformed to OperationCallExp. The name of its method is assigned to the name of the operation. Any EqualityOperatorExpressions and ComparisonOperatorExpressions are transformed to OperatorCallExp. Examples of ETL expressions and their equivalents in ATL are shown in Table 5. As can be seen, there is no same mapping between ETL and ATL expressions.

3.8. Step 6: Interaction with the tester

In the interactive mode of the tool, when the execution of the transformation can be forked, i.e., the path condition evaluator cannot determine which paths should be traversed, the symbolic execution engine interacts with the tester to determine the satisfiability of the condition/guard. As we work on the ETL model, in order to show the text of the guard/condition, a model to code transformation is employed from the Haetae tool. To give the tester more control over the execution process, the name of the corresponding rule is shown to the tester. Also, in the condition preparation, if the condition contains "not", i.e., the condition is of type NotOperatorExpression, to avoid confusing the tester, the condition is asked without "not". Moreover, providing that the condition contains the self keyword, to shape the tester's understanding, the type of the related class (the context of the related operation) is asked and the tester specifies the result.

3.9. Step 7: Checking metamodel conformance

The tool checks the SMF to conform to the target metamodel. What we mean by conformance is as follows.

- All classes of the SMF exist in the target metamodel.
- All its attributes and references exist in the attributes and references of the corresponding class in the target metamodel or in its parents in a hierarchy.

To illustrate this, considering the SMF model in Fig. 11 and the target metamodel in Fig. 2b, the Female class exists in the target metamodel and its fullName attribute exists in the parent of the Female class, the Person class, in the target metamodel.

3.10. Test model generation

A test model or test case is a model, which conforms to the source metamodel and satisfies the path condition. At the end of the symbolic execution of a feasible path, the test model is generated by the AnATLyzer tool in some cases. We have customized it in our tool. As we do not allow the tester to select an unsatisfiable path, two scenarios can be considered. Both of them are based upon the PC is satisfiable.

- The corresponding path does not include any conditions. In this case, only one execution path exists and it makes no difference which test model the tester selects. For this reason, SEET does not generate a test model.
- The corresponding path includes conditions. Under this circumstance, the SEET tool generates the test model, which satisfies the path condition in order to help the tester to examine whether the ETL transformation is specified correctly.

3.11. Step 8: Symbolic oracle

To have a conclusion about whether or not the transformation is implemented correctly, the tester can check the SMF. If an error has been shown in the SMF, e.g., SEET generates a Female class while considering the corresponding PC, the tester expects Male, we consider SEET to be able to detect the error and report it. As another capability, if the tester selects more evaluation, SEET serves as an oracle and compares the real SMF with the expected one. This part has been implemented with ECL. For each package generated, the tool checks whether all classes, their attribute, and references are the same. To create an expected SMF, the tester should create a package corresponding to each path condition. In this package, the tester specifies the expected classes and their references and attributes. If attribute values are not constant, the tester should go back to the code and calculate the expected value.

It should be mentioned that expected SMFs (corresponds to each PC) and actual SMFs are connected by numeric IDs. If the tester changes a transformation so that the possible paths are modified, the tester needs to renumber all existing expected SMFs.

4. The SEET tool

The SEET tool is integrated as an Eclipse plugin. Considering the applied approach in the tool as a black-box, the tool takes the ETL transformation definition and source and target metamodels as inputs and generates an SMF, the path condition, the test model (and in case the expected model has been provided, the result of the correctness of the transformation) as outputs. Fig. 15 shows the tool in the form of a black-box.

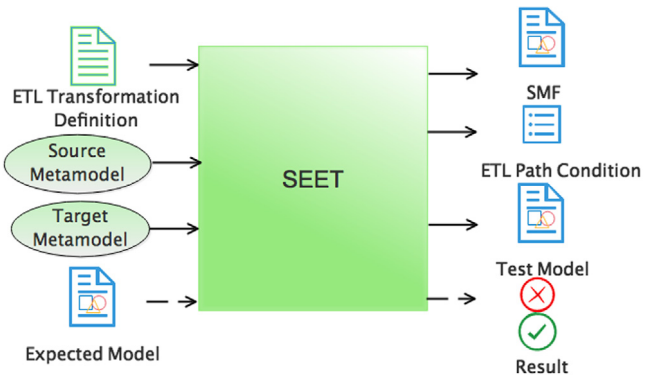


Fig. 15. The SEET approach as a black-box.

The source code, the screencast, update site for Eclipse, and RCP can be found at Github.[5] For usage instructions and screenshots, refer to Appendix A.

5. Evaluation

In this section, the evaluation of correctness, completeness, scalability, usefulness, and usability of the proposed approach are addressed. Correspondingly, the evaluation aims to answer the following questions.

- Correctness: Are the errors detected correctly?
- Completeness: Are all possible errors detectable by this approach? It is worth mentioning that completeness results are based on those features of ETL that are covered by SEET, as shown in Table 3.
- Scalability: Does the increased number of lines of code and path conditions, and the increased size of input metamodel cause a rational increase in the execution time?
- Usefulness: Are users able to detect whether the transformation is faulty using the generated SMF?
- Usability: How easy is using the SEET?

In the following, first, the evaluation criteria are introduced; then, the evaluation results are interpreted. After that, possible threats to the validity of the evaluation result are discussed. Finally, some suggestions for using the tool along with the limitations are discussed. Please note that the evaluation data can be found at Github.[6]

5.1. Evaluation criteria

5.1.1. Correctness and completeness

To assess the correctness and completeness of the method, the precision and recall criteria are used (Schütze et al., 2008). Precision offers a measure of correctness or quality. In our context, it is calculated as the ratio of real errors to all errors detected by SEET. Recall has been regarded as a measure of completeness. In our context, it is calculated as the ratio of errors detected by SEET over all the existing errors.

To perform an evaluation, two well-known case studies were employed: Families2Persons and Competition2TVApp (Kolovos et al., 2007). The Families2Persons case study acted as our running example. Likewise, the Competition2TVApp was chosen to cover further features, which are not found in the Families2Persons case study. Specifically, the Competition2TVApp transformation provides more than one target parameter in a rule, and it also offers (nested) loops. Using the "add" method and the *equivalent()* operation as the parameter of the method are other attractive features of this transformation.

Although there are some studies on generating mutants for ATL transformations (Alhwikem et al., 2016; Troya et al., 2015) or EOL programs (Alhwikem et al., 2016), designing mutation operators for ETL model transformations is neglected by the community. Therefore, first, a set of mutation operators are presented for the mentioned case studies. These mutation operators are achieved systematically. The main distinguishing feature of the proposed operators, compared to other approaches, lies in the fact that they focus on logical errors. The approach proposed by Troya et al. (2015) does not consider many operators, which are covered by SEET mainly relating to the imperative part of ETL (e.g., operators related to if-block, For loops, and operations). In contrast, SEET does not provide the "InPattern element addition" operator because ETL rules cannot have more than one in-pattern element. In other words, having two or more in-pattern element in an ETL rule causes a syntax error, which is not a matter of focus in our approach. It is worth noting that "InPattern element deletion" and "InPattern element name change" operators are also syntax errors. Hence, SEET does not care about these syntax errors.

Each mutation operator shown in Table 6 must adhere to these conditions: (1) The operator does not cause a compilation error in ETL transformations, which can be detected by the static analyzer of ETL (Haetae), (2) While most of them cause logical errors, some may not cause in all circumstances (e.g., changing the name of the rule with a non-existing one). Consequently, a combination of correct and faulty ETL transformations are defined, (3) It is related to the selected case studies.

Fig. 16 depicts the evaluation process of correctness and completeness. First, mutation operators are applied to the ETL transformation in order to achieve ETL transformation mutants manually. After that, the results from the SEET tool are compared with those from manual testing. In manual testing, input models are generated manually. Then, we check whether there is at least one state leading to incorrect output. Manual testing reports an error if at least the output of one state is not equal to the expected one. Otherwise, it reports no error. If by getting the expected SMF, SEET reports that the transformation has not been written correctly we consider that SEET can detect the error. To assess the evaluation criteria, 159 mutants of the Families2Persons transformation and 75 mutants of the Competition2TVApp transformation are generated. Each of the mutants has been obtained by applying one mutation operator to the original transformation. Mutation operators are shown in Table 6. Assigning a value to the wrong property of the output class, incorrectly writing the class name in the in-pattern or out-pattern of the rule, and incorrectly inserting negation into the Boolean condition are examples of logical errors in the transformations. Each mutation operator may be applied to different lines of the transformation code and generates a different mutant. Since each mutant has a partial difference with the original transformation, it avoids the dependency of errors in mutants that may distort the evaluation.

Relying on the results that are obtained from the SEET tool, true negative, true positive, false negative (SEET proved unable to detect the error), and false positive (SEET detects an error incorrectly), the computations of precision, recall, accuracy, and f-measure are provided according to formula (3) to (6), respectively.

$$precision = \frac{\#TP}{\#TP + \#FP} \tag{3}$$

$$recall = \frac{\#TP}{\#TP + \#FN} \tag{4}$$

[5] http://github.com/BanafshehAzizi/SEET
[6] https://github.com/BanafshehAzizi/SEET/tree/master/evaluation

Table 6
Mutation operators.

Type	Target	
Create	Assignment statement	
	Target parameter of the rule	
	Rule	
	Expression statement	
	Return statement	
	Operand of plus operator expression	
	Guard	
Delete	Rule	
	Assignment statement	
	Guard	
	Operand of plus operator expression	
	If block	
	Else block	
	Return statement	
	Not operator expression	
	Property of property call expression (in cases that it cannot detect with Haetae)	
	For loop	
	Expression statement	
	Context	
Modify	Rule name	
	Parameter of add operator	
	Target of method call expression	
	Target of property call expression	
	Property of property call expression	
	RHS of assignment statement	
	LHS of assignment statement	
	Return statement	
	Operand of plus operator expression	
	Predefined method (modifying isDefined() to isUndefined())	
	Boolean expressions	Adding "not" operator to boolean expression
		"True" to "false" and "false" to "true"
	Content of string expression	
	If condition	
	Type of operation to "cached" operation	
	Source parameter of the rule	
	Target parameter of the rule	

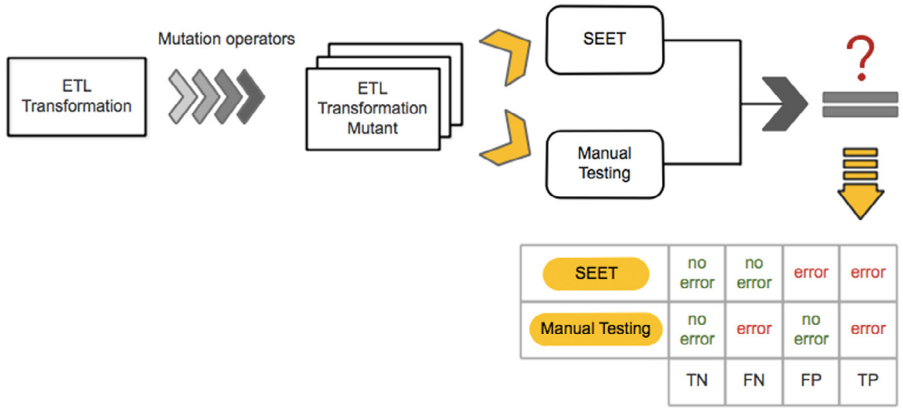


Fig. 16. The evaluation process of correctness and completeness.

Table 7
Specification of case studies used to evaluate the scalability.

Case study	Transformation name	#Classes of source metamodel	#Classes of target metamodel	#Non-blank lines of code	#Rule	#Operation	#Guard	#Condition
1	Tree2Graph	1	3	10	1	0	0	1
2	Competition2TVApp	4	10	23	3	0	0	0
3	Flowchart2HTML	7	59	20	4	0	0	0
4	Families2Persons	2	3	41	2	2	2	5
5	CopyTVApp	10	10	41	7	0	0	0
6	Gui2Html5	10	12	67	8	0	0	1
7	OO2DB-Reduced	16	6	72	4	2	2	1
8	Market2View	6	6	123	5	4	0	16
9	Newsletter2HTML	10	18	245	4	8	0	44

Table 8
Usefulness questionnaire.

Before using the tool					
Question	Trans1		Trans2		Trans3
Which transformations are faulty?					
After using the tool					
Question	Trans1		Trans2		Trans3
Which transformations are faulty?					
Could you do all the steps of using the tool? (Yes/No) If No which steps?					
Question	Very high	High	Medium	Low	Very low
To what extent have you found the automatic mode of the tool to be helpful to detect faulty transformations?					
To what extent have you found the automatic mode of the tool to be helpful to detect correct transformations?					
How helpful is the interactive mode of the tool to detect errors?					
How much does the SEET tool save your time?					
How much does the interactive mode of the tool give you more control over the execution process?					

$$accuracy = \frac{\#TP + \#TN}{\#TP + \#TN + \#FP + \#FN} \quad (5)$$

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

5.1.2. Scalability

In an endeavor to assess the scalability of the method, nine case studies were selected. These case studies are nine ETL transformations which are mainly obtained from Epsilon's git repository.[7] They differ in their application domains, size of metamodels, the number and the type of ETL features. Table 7 summarizes the information about these case studies. The size of metamodels varies from 1 to 16 classes in the input and 3 to 59 classes in the output. From the size of the transformation view, the number of rules is variable between 1 and 8 and the number of lines of code is between 10 and 245. Accordingly, the biggest transformation is 25 times bigger than the smallest. There exist transformations with 0 to 8 operations. In addition, further information such as the number of guards and conditions is illustrated. The number of guards is 0 or 2 and the number of conditions is variable between 0 and 44. In the following, the domain of each transformation is outlined.

- Tree2Graph: It is a simple tree to graph transformation.
- Competition2TVApp: It is applied to build interactive content for digital TV from a set of high-level components.
- Flowchart2HTML: It is defined to obtain HTML models from flowcharts.
- CopyTVApp: It is applied to copy elements of the TVApp metamodel.
- Families2Persons: It acts as our running example and is explained in Section 2.2.2.
- Gui2html5[8]: It transforms some elements of a Gui model to its equivalent in a HTML5.
- OO2DB-Reduced: It is a well-known transformation employed to transform the elements of the OO metamodel to a relational database.
- Market2View[9]: It transforms a marketplace to a design view.
- Newsletter2HTML[10]: It transforms some elements of a Newsletter to their corresponding HTML elements.

5.1.3. Usefulness and usability

To evaluate how useful and usable is the generated SMF to detect logical errors, questionnaires were completed by 12 students familiar with model-driven engineering. Three of the students were ETL experts. It should be noted that in order for the participants to be able to do manual testing, little explanation about ETL transformations was expressed to those who had not worked with ETL before the survey. Three Families2Persons transformations with different logic, including correct and faulty transformations, were recruited. The questions were divided into two groups including before and after using the tool. In the beginning, participants were asked to detect faulty transformations without using any test tool. After that, they did the same task using the SEET tool. Table 8 shows the usefulness questions and Table 9 indicates usability ones.

[7] https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git

[8] https://github.com/phillipus85/ETLMetricsDataset

[9] https://github.com/phillipus85/ETLMetricsDataset

[10] https://github.com/phillipus85/ETLMetricsDataset

Table 9
Usability questionnaire.

Question	Very high	High	Medium	Low	Very low
How user-friendly is the tool?					
How good is the appearance design of the tool?					
After right-clicking on an ETL file, how easy did you find the tool icon from a pop-up menu?					
How easy is to use the tool?					
How learnable is the tool?					
How is your satisfaction with the tool?					

Table 10
The results of evaluation of correctness and completeness.

Transformation	#mutants	TP	TN	FP	FN	Precision	Recall	Accuracy	F-measure
Families2Persons	159	105	33	19	2	0.85	0.98	0.87	0.91
Competition2TVApp	75	54	20	0	1	1	0.98	0.99	0.99
Overall	234	159	53	19	3	0.89	0.98	0.91	0.93

5.2. Results and interpretation

In this section, we elaborate on the results of evaluating the correctness, completeness, and scalability of the proposed approach. For interpreting the results obtained from evaluation, the questions raised at the beginning of Section 6 are answered.

5.2.1. Correctness and completeness

Table 10 shows the results obtained from the SEET tool which comprised true negative, true positive, false negative, and false positive. From these results, of the total of 159 mutants on the Families2Persons transformation, 19 false positive errors are found, 12 of which are related to the presence of dead code which contrary to expectations, SEET has not detected it. Seven of them are related to conjunction a condition to the existence condition/guard. Two false negative errors are detected by SEET two of which are related to adding an extra target parameter to the rule whose type is the same as the existing target parameter. This may be explained by the fact that due to the increase in the level of abstraction, the tool has been unable to encompass this type of error (For more details, refer to Appendix B). Of the total of 75 mutants on the Competition2TVApp transformation, one false negative error was found which is interpreted similar to the Families2Persons transformation. It is due to adding an extra target parameter to the rule whose type is the same as the existing target parameter.

- Correctness: The first question can be answered with respect to the precision value. According to Table 10, from the experiment of 234 mutants, the precision of the Families2Persons transformation is 0.85, and the precision of the Competition2TVApp transformation is 1 and overall is equal to 0.89 which are acceptable.
- Completeness: To answer the second question, the value obtained from computing recall sets in Table 10 is equal to 0.98. Especially, as discussed in Section 6, it is very high and acceptable. This means that SEET covers 0.98 of logical errors in transformations which SEET can cover.

5.2.2. Scalability

Turning now to the scalability, the results can be seen in Table 11. The experiments were run on a desktop computer with an Intel Core i5, 4 GB of RAM, and Windows 8.1 OS. The time of automatic mode with comparison for each case study includes the time taken to compare the SMF and the expected one. The experiment of each case study for calculating the time of automatic mode without/with comparison was conducted ten times, with the results being averaged. In each execution time, the user response time was ignored. The time of generating the visualizations of the test models only was included in the calculations of time with comparison. The number of path conditions shown in Table 11 refers to the number of satisfiable path conditions (reachable execution paths) in the transformation. Simultaneously increasing the number of lines of code and path conditions as well as the size of input metamodel raises the execution time in a linear way. Considering the first two case studies of the table, from the first case study to the second case study, it is apparent that the number of lines of code has been doubled and the number of path conditions has been five times greater, and the size of input metamodel has been four times, in this case, the execution times have been rather tripled. Taking case study 3 and case study 4, for another instance, it can be seen that the number of lines of code has been relatively doubled, the number of path conditions has decreased by a factor of 2, and the size of input metamodel has decreased by a factor of 3.5 in case study 4. In this case, the execution times remain relatively constant. In Table 11 there is a clear increasing trend in execution time when the number of lines of code levels off and there is a sharp increase in the number of path conditions and the size of the input. Case studies 4 and 5 are cases in point. From the data in Table 11, we can see that from case study 6 to 7, while the number of path conditions has declined by a factor of 4, the execution time has increased by factor 1.3 due to the fact that the number of lines of code has a gradual rise and the size of input has increased by a factor of 1.6. Finally, from case study 8 to 9, even though the number of path conditions has declined by a factor of 7, the execution times have been doubled because the number of lines of code and the size of input metamodel has been relatively doubled.

Fig. 17 shows the correlation between symbolic execution time and the number of path conditions, lines of the code, and size of input metamodel. By increasing the number of path conditions, lines of code or the size of input metamodel, the allocated execution time is increased. Therefore, there has been a positive correlation between time and each of these parameters. In some points of Fig. 17, by raising the parameter, the execution time has fallen down or remained the same, suggesting that no single parameter affects the execution time. In Fig. 17, when the number of lines of code remains at 41, the execution time peaks due to a significant rise in the number of path conditions. According to the graph, when the number of path conditions has decreased in case study 9, the execution time rocketed to a record high due to a substantial increase in the number of lines of code.

- Scalability: The third question aimed to determine whether, with increased size and complexity of the transformation, the execution time increased in a rational way. According to the results

Table 11
The result of scalability evaluation.

Case study	Transformation	#Non-blank lines of code	#path conditions	#Classes of source metamodel	Time of automatic mode without comparison (ms)	Time of automatic mode with comparison (ms)
1	Tree2Graph	10	3	1	421	553
2	Flowchart2HTML	20	16	4	1178	1726
3	Competition2TVApp	23	11	7	2726	3247
4	Families2Persons	41	5	2	3104	3418
5	CopyTVApp	41	128	10	11781	18050
6	Gui2Html5	67	67	10	11779	14448
7	OO2DB-Reduced	72	18	16	15345	16076
8	Market2View	123	92	6	72448	77518
9	Newsletter2HTML	245	13	10	130015	143653

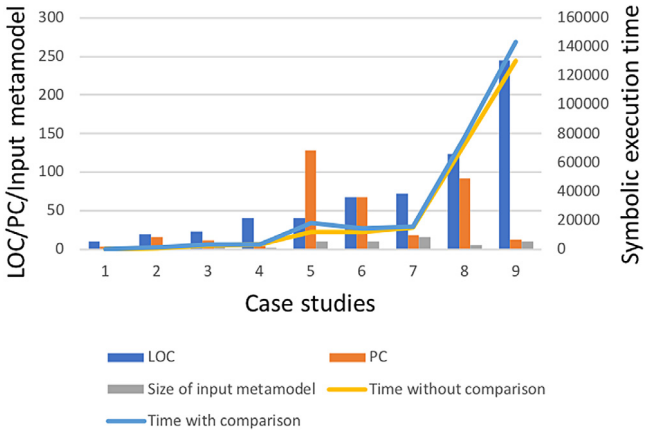


Fig. 17. Correlation between symbolic execution time without/with comparison and (a) lines of code, (b) path conditions, (c) size of input metamodel.

of executing the tool on nine transformations with the divergent domains, size, and complexity, shown in Table 11, their achieved time is linear; consequently, they confirm the hypothesis that SEET is scalable.

5.2.3. Usefulness and usability

Turning to the usefulness and usability, with respect to the question asking to determine faulty transformations before using the tool, seven students could not find out Trans2 is correct, and one could not find out Trans3 is faulty, and one another could not find out Trans1 is faulty. What is interesting in this data is that all the participants were able to correctly identify faulty and sound transformations via SEET. They are including those who were unable to determine sound and faulty transformations in manual testing correctly and lacked good ETL knowledge prior to the survey. It is suggesting that SEET still could be helpful for those who were not so familiar with ETL. One may argue that looking at the transformation before using the SEET tool may have learning effect. Although the participants knew the transformations, we believe that in the case that the participants would not do manual testing before using the SEET tool, still they can detect faulty and correct transformations via SEET. In other words, without knowing how the transformation is written, the participants could detect the correspondence between PCs and SMFs since this checking does not need knowing the logic of the written transformation. As it has been explained previously, e.g., while seven students cannot find out Trans2 is correct in the manual testing, all of them can detect it via SEET. It is shown that they can use the SEET tool, and it is useful for them to detect correct transformation, and manual testing has not learning effect in this case. About faulty transformations, the SEET tool is useful for two students to detect them. The reason behind we have asked participants to examine transformations manually at the beginning is that they can answer whether SEET saves their time or not. For other questions, the results have been shown in Tables 12 and 13. We have quantified options "very low" to "very high" with numbers "1" to "5". The first question is also mapped with numbers 1 to 5. All participants were able to complete all the steps of using the tool. Most of the participants agreed that the automatic mode of the tool is useful to detect correct transformations. Most of the individuals specified that the interactive mode of the tool gives them more control over the execution process. Participants found the tool easy to use.

- Usefulness: To answer the fourth question, according to Table 12, the rate of the usefulness of each question is between %73 and %100, and the total usefulness is %83.
- Usability: To answer the fifth question, according to Table 13, the total usability of the tool is %85.

5.3. Threats to validity

In this section, some threats to the validity of the results are discussed, including internal and external threats.

5.3.1. Threats to internal validity

Considering the evaluation of the correctness and completeness of the approach, there are some threats to the internal validity of the result. First, the number of mutants generated from mutation operators is not the same, because mutation operators are injected into possible lines of code. For example, in the Families2Persons transformation, we might be able to inject the 'modify if condition' operator in several lines of the code as we have the number of if conditions. However, the 'modify target parameter of the rule' operator can only be inserted into two rules. Therefore, we have five mutants from the 'modify if condition' operator; besides we have two mutants from the 'modify target parameter' operator. Second, the precision and recall were calculated by manually extracting the true positives and false positives of mutants. Even though the authors have good knowledge of MDE and model transformations, some TPs or FPs may have been incorrectly identified. In the experiments, we assume that the tester creates the expected SMF correctly; however, if the tester makes a mistake, the results may be changed.

Regarding the evaluation of scalability, the complexity of ETL statements in its evaluation has been overlooked. However, we hold that the complexity of statements can increase only the constant of the time relation, as both complex transformations and simple transformations in the experiment adhere to the linear relation for the execution time. In another view, the number of

Table 12
Usefulness results.

#	Question	Average rating
1	Could you do all the steps of using the tool?	%100
2	To what extent have you found the automatic mode of the tool helpful to detect faulty transformations?	%80
3	To what extent have you found the automatic mode of the tool helpful to ensure that the transformation is correct?	%87
4	How helpful is the interactive mode of the tool to detect errors?	%73
5	How much does the SEET tool save your time?	%78
6	How much does the interactive mode of the tool give you more control over the execution process?	%82
	Total	%83

Table 13
Usability results.

#	Question	Average rating
1	How user-friendly is the tool?	%80
2	How good is the appearance design of the tool?	%85
3	After right-clicking on an ETL file, how easy did you find the tool icon from a pop-up menu?	%98
4	How easy is to use the tool?	%85
5	How learnable is the tool?	%80
6	How is your satisfaction with the tool?	%80
	Total	%85

lines of code, paths, and input classes are not very big. Thus, it is possible that in more complex transformations much higher execution times are achieved.

5.3.2. Threats to external validity

To evaluate correctness and completeness, we have used two case studies whose all ETL features are supported in SEET. For the ETL transformations which some of their features are not supported in SEET, the precision and recall are lower than the obtained values. Therefore, the results cannot be generalized to other case studies. For the evaluation of usefulness, we consider the Families2Persons case study to be understandable for participants as we argue that the tester in our approach also has a good knowledge of the transformation. There are some probabilities that in case the transformation is more complex, the achieved rate for usefulness is different. It may be less than the current rate because of checking the complex SMF or be more than the current rate on the grounds that the participant is more aware of the importance of the method. Since increasing the complexity of the transformation makes the manual testing of the transformation harder, we believe that our approach is still useful in comparison with manual testing. Since the three transformations solve the same problem, their correctness can be deduced by comparing them to each other. It is the case of both manual testing and testing via SEET. It would be better if different transformations are recruited for manual testing and testing via SEET as knowing the transformations and their potential errors before using the tool may prevent to measure the effect of the tool. Moreover, we have recruited 12 students for our evaluation; if the number of participants increases, the achieved rate of usefulness and usability may change. Though the experiment could have been done in a more controlled experiment, the results are informative.

5.4. Discussion

This section seeks to discuss some suggestions for using the SEET tool and express its current limitations.

Although the tester can ensure the correctness of the transformation in four arbitrary ways, we have some suggestions about it. For transformations that the tester has a good knowledge of the written transformation, e.g., the tester has written the transformation by himself/herself, it is suggested to use symbolic oracle. For complex transformations, especially in the case that the tester is not completely aware of the written code, it is suggested to check the generated SMF manually. In case the SMF is so complex that it becomes hard to check it manually, it is suggested to use the generated test cases. Interactive mode is suitable when the tester wants to traverse a path of a transformation consciously. In transformations with so many rules, especially with not many conditional statements, this mode is useful to decrease the symbolic execution time.

Some limitations should be noted. First, the whole syntax of ETL transformations is not covered in the tool, logical conjunction/disjunction of conditions/guards, and methods such as ‘select’ and ‘exists’, to name a few. This does not mean that the SEET approach has fundamental limitations, but it shows the cases that have not been implemented in the current tool. For example, the ‘select’ method can be implemented in the same way as if condition. Hence, we left those implementations as future work. Second, the fact that SMF is kind of a metamodel causes the SEET approach not to be able to fundamentally detect some errors when more than one instance of the same class is created by the transformation. This limitation has been addressed in detail in Appendix B. The third limitation concerns the implementation of the SEET tool, whereby it assumes that there exist one source metamodel and one target metamodel. For multiple metamodels in the source or target, the tool should be further developed. Fourth, since SEET uses the Haetae and AnATLyzer tools, their limitations can affect the functionality of our tool. Fifth, the applicability of the interactive mode can be jeopardized by large numbers of conditions, when their satisfiability is asked for each execution of the tool, even when a small part of the transformation has been changed. Sixth, such as other symbolic execution techniques, the automatic mode of the SEET tool does not work for very complex transformations with big state space.

6. Related work

In recent years, there has been an increasing amount of literature on the verification of model transformations (Ab. Rahim and Whittle, 2015). However, the verification of ETL transformations especially in a white-box manner was surprisingly neglected although some studies such as Haetae and UML-RSDS have been undertaken on ETL. In the section, we reviewed related work on verification of model to model transformations. With respect to the problem stated in preceding sections, related work on verification of ETL transformations is provided in Section 6.1, which also considers the language-independent verification techniques applicable to ETL. Due to the fact that the solution set forth in this study is motivated by a symbolic execution, the research on the symbolic execution of model transformations is studied in Section 6.2. Section 6.3 concerns the verification of other rule-based model transformations especially ATL. Approaches to computing the transformation footprint are also considered in Section 6.4, as they have similarities with the SMF generated by SEET. At the end, we summarize the more related work in the form of a table in Section 6.5.

6.1. Verification of ETL transformations

Prior studies that have noted the verification of ETL transformations are as follows. Preliminary work on verification of ETL transformations was undertaken by García-Domínguez et al. (2011), which is supported by the EUnit oracle. Feeding input and output metamodels along with input model and expected output model, EUnit can investigate whether an expected output model is equal to a concrete generated model. However, for judging about the output, it requires inputting test cases, and it does not generate test cases itself. As long as test cases are generated manually, it is a time consuming and boring task. In addition, EUnit misses an automatic tool to generate the oracle programs. Correspondingly, the seminal study aimed at verifying ETL transformations in a white-box manner is the work of Wei and Kolovos (2014) which is supported by the Haetae tool. It is much more concerned with the static analysis of ETL transformations. They looked more at detecting syntax and semantic errors instead of logical errors. In a follow-up study, Lano et al. (2015) used the UML-RSDS tool and created a framework for detecting some syntax and logical errors in transformations, such as ETL transformations. In this study, an ETL specification should be converted into a UML-RSDS program (Lano et al., 2015), which is then verified via the UML-RSDS tool. Their approach suffers from a lack of higher-order transformation from an ETL program to UML-RSDS representation. Therefore, this approach required the tester to do programming to transform an ETL transformation to an intermediate representation whereas, as noted earlier, our approach does not require any programming. On the other hand, it fails to cover the imperative parts of ETL transformations appropriately. For instance, else block or loops have not been considered. In the same vein, by drawing on the concept of black-box testing, EMG (Popoola et al., 2016) came up with a solution to generate test models. It is limited by a lack of a fully automatic tool and not considering the transformation definition. Because in EMG test models are generated randomly, finding the test case which leads to detecting an error may be time consuming. Moreover, the generated test cases may recur or they may navigate the same execution path of the transformation. Another demerit of EMG has to do with writing the code for generating a test case by the tester which commands learning EMG language whilst the tester does not need to do programming in SEET. Recently, we have conducted research on contract verification of ETL transformations (Azizi et al., 2017). In this work, an ETL specification is transformed into a DSLTrans program, which should be symbolically executed using the SyVOLT tool. Therefore, the contract verification of ETL transformation is indirectly conducted. We refer to our previous study (Azizi et al., 2017) as the "indirect approach" since it performs the symbolic execution of ETL transformations indirectly. Providing contracts and an ETL transformation, the indirect approach manages to specify whether contracts are satisfactory without the tester checking. At the same time, it has been mostly restricted to the declarative part. Thus, this indirect approach cannot check statements such as conditional statements and loops, which are supported in the SEET approach.

Given the above cases, in terms of usefulness, due to the fact that SEET works in the automatic mode, it is more useful than current studies for detecting logical errors of ETL transformations in a white-box manner.

6.2. Symbolic execution of model transformations

$TETRA_{BOX}$ (Schönböck et al., 2013) provided white-box testing of model transformations by offering a framework for symbolic execution. It was applied for ATL model transformations, however, the tool support is not available anymore. In case we transform an ETL transformation to its equivalent CFG, this approach can also be used for ETL. One of the demerits of $TETRA_{BOX}$ in comparison with the proposed approach is the lack of an available tool. Moreover, it utilizes the UMLtoCSP tool, which fails to support Ecore-based metamodels while used metamodels in transformations are mainly Ecore-based. In our view, with respect to the publication date of the paper, EMFtoCSP (González et al., 2012) was better to use at that time which supports Ecore-based metamodels, although this tool also has some drawbacks. The SEET approach uses the anATLyzer tool to check constraint satisfiability, which supports Ecore-based metamodels as well as three-valued logic which EMFtoCSP fails to support. Last but not least, $TETRA_{BOX}$ is less-documented and very little is known about $TETRA_{BOX}$, e.g., it is not clear whether it supports imperative parts of transformations or not. Likewise, SyVOLT (Oakes et al., 2018) broke new ground on contract verification of ATL transformations via applying symbolic execution. Its main drawback is the lack of facilities for the imperative part of the transformation. By transforming ETL to DSLTrans, this approach can be used for ETL (Azizi et al., 2017). SymexTRON (Al-Sibahi et al., 2016) applied the symbolic execution technique for verifying high-level transformations. It is implemented for TRON, an imperative language suitable for theoretical development of the analysis method. Since TRON is not a rule-based language, the peculiarities of these kinds of languages were not considered in that work.

6.3. Verification of model transformations

In this section, we introduce some white-box and black-box approaches to verifying model transformations.

6.3.1. White-box

Cuadrado et al. (2017) proposed an approach to the static analysis of ATL model transformations, which is supported by a tool called AnATLyzer. Comparing SEET with AnATLyzer, while the AnATLyzer tool focused on typing and rule errors, SEET addresses logical errors. AnATLyzer generated path conditions backwards from the problematic statement (error location) to the transformation entry points, as they are only interested in one execution path to ensure whether the problematic statement raises an error in execution. In this paper, path conditions are produced in a forward manner to cover all possible execution paths. Troya et al.

Table 14
Comparing SEET with other approaches.

Approach	Symbolic execution technique	White-box verification	Detecting logical errors	Contract verification	Available tool	Test case generation	Ecore support	Documentation	Not need programming	Imperative part support	ETL
EUnit	–	✓	✓	–	✓	–	✓	✓	∼	✓	✓
EMG	–	–	✓	–	✓	✓	✓	✓	–	✓	✓
Haetae	–	✓	–	–	✓	–	✓	✓	✓	✓	✓
UML-RSDS	–	✓	✓	n/a	✓	–	✓	✓	–	∼	✓
Indirect approach	✓	✓	✓	✓	✓	–	✓	✓	✓	∼	✓
SyVOLT	✓	✓	✓	✓	✓	–	✓	✓	✓	–	–
$TETRA_{BOX}$	✓	✓	✓	∼	–	✓	–	∼	✓	n/a	–
SEET	✓	✓	✓	–	✓	✓	✓	✓	✓	✓	✓

Guideline: ✓ full support ∼partial support — not support n/a not available

(2018) adopted an automatic approach to infer likely metamorphic relations for ATL model transformations. Jilani et al. (2014) generated test data by a search-based method, developing the MOTTER tool to test ATL transformations. González and Cabot (2012) offered a white-box testing approach that is based on generating a dependency graph, navigating it and generating test data by EMFtoCSP (González et al., 2012). The approach is not supported by any tool. González and Cabot (2014) have presented an approach to generating test cases based on a deep analysis of the OCL invariants in the input metamodel of the transformation.

Among verification approaches of other transformation languages, some can be applied to the verification of the ETL transformations. However, it is required to tackle the peculiarities of ETL, such as imperative part of the rule, their execution, and the equivalent method.

6.3.2. Black-box

Guerra and Soeken (2015) automatically generated test cases by SAT solver, applying mtUnit to report detected errors. Brottier et al. (2006) provided an approach to generating test models based on the input metamodel and a set of object fragments. Sen et al. (2008) proposed a novel approach to test data selection of model transformations. Almendros-Jiménez and Becerra-Terón (2016) automatically generated random Ecore model for ATL transformations. Sen et al. (2012) simplified the development of effective test models by presenting a semi-automated tool.

Comparing SEET with black-box approaches, since they do not consider the transformation definition, they generate test cases, which navigate the same execution path making the task of finding errors so time consuming.

6.4. Computing transformation footprint

Burgueno et al. (2015) conducted research on fault localization of ATL model transformations. In this approach, footprints or types of in-pattern and out-pattern elements of transformation rules and constraints were extracted. The alignment for each rule and constraint was calculated and matching tables were created. These tables were used to detect faulty rules. Mottu et al. (2012) proposed an approach to generate test cases of model transformations based on knowledge of test cases. Knowledge of test cases was achieved from a static analysis of the transformation and the extracted input metamodel footprint. This kind of footprint consists of input classes and properties of input metamodel used in the transformation. The footprint is a set of *<operation, Feature, Type>*. These footprints are transformed to model fragments, from which the complete usable models are created using the ALLOY solver. They applied the approach to the Kermeta language (Jézéquel et al., 2011). Jeanneret et al. (2011) presented an automatic approach to estimate the static metamodel footprint of operations. It is worth mentioning that researchers have not treated computing target metamodel footprint by executing the transformation.

Considering the approaches dealing with computing transformation footprint, the research in this field mainly has tended to focus on computing input metamodel footprint rather than output metamodel footprint. However, our approach by symbolic execution of a transformation generates a target metamodel footprint. Some approaches focus on fault localization instead of finding errors, which is beyond the scope of this study.

6.5. Synopsis

To the best of our knowledge, symbolic execution of ETL transformations has received scant attention in the literature. It is noteworthy that the approach for symbolic execution of ATL model transformations cannot be applied directly to ETL transformations. ETL has less constrained imperative constructs than ATL, which may make the implementation more difficult, especially the synthesis of OCL code compliant with the standard.

Table 14 compares the SEET approach with the works that are more related to our work. We divide the prior studies into two groups. The first set of studies have something to do with ETL. The set mostly comprised studies whose techniques are not symbolic execution. The only study in the set which supports symbolic execution also neglects the imperative part. The second set, comprising two studies, is in the relationship with symbolic execution. Neither of these studies is implemented for ETL transformations. Hence, there are no studies that we can compare with our approach accurately.

The current study found the approach for symbolically executing the transformation written in ETL. The most interesting finding is that the method whereby we did this research is capable of detecting logical errors in ETL transformations. Another important finding was that it supports imperative part of ETL transformations, as ETL transformations are mainly imperative.

Altogether, there is no in-depth technique for finding logical errors in ETL transformations.

7. Conclusion

In light of the popularity of ETL, it is becoming extremely difficult to ignore its verification. The present research aimed to detect logical errors in a transformation written in this language. This study strengthens the idea that it is possible to apply symbolic execution to model transformations and provides additional evidence when it comes to it. The most obvious finding to surface from this study is that the generated SMF

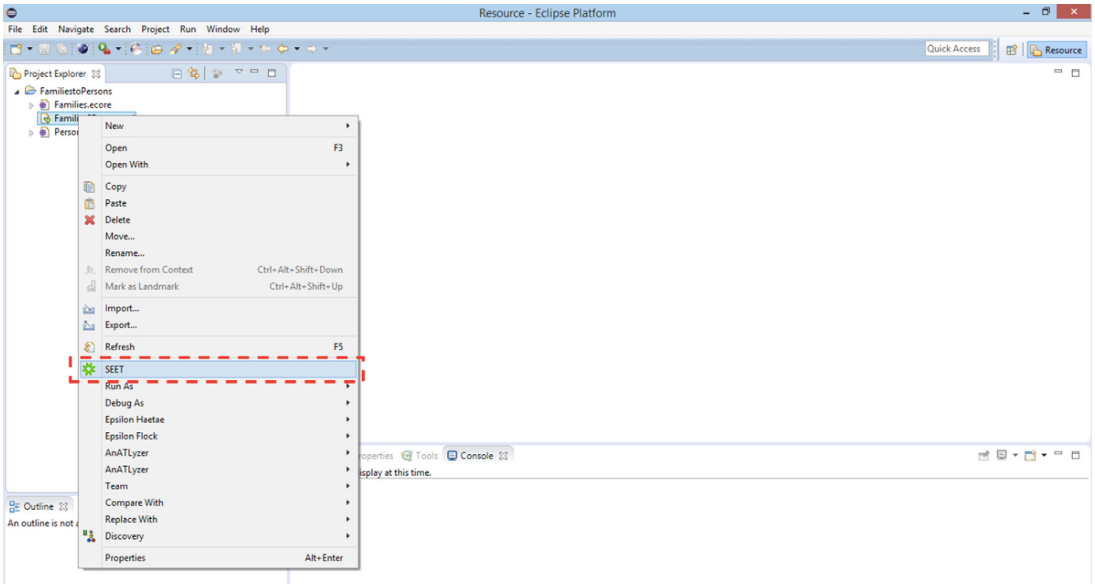


Fig. 18. The execution of the SEET tool.

provides a general schema of the ETL transformation output. The approach has a number of striking features: (1) a deeper focus on the imperative part, which raises the possibility of detecting errors, (2) no need for tester programming, (3) considering the transformation definition, (4) input independence, (5) providing an accessible Eclipse-based plugin namely SEET. Moreover, the approach provides support for generation of ETL CFG as a separate unit. Eventually, the findings from this study make another contribution to the current literature, which is a set of mutation operators to detect logical errors in ETL transformations.

The most important limitation lies in the fact that the current implementation of SEET fails to encompass the entire features of ETL transformations. Further research in this field would be of great help in extrapolating the findings to all ETL transformations. In other words, the generalizability of the results is subject to certain limitations. Overall, these findings suggest several courses of action for applying SEET to all ETL model transformations on the one hand and other model transformations on the other hand. It is suggested that ETL transformation mutants are generated automatically in further studies. Further research should focus on providing support for various metamodels in the input or the output. A further study can make the generated test model acceptable for the EUnit framework in oracle testing. Further studies to consider the concolic testing, i.e., dynamic symbolic execution of ETL transformations will be interesting to be undertaken to compare the results with our approach.

CRediT authorship contribution statement

Banafsheh Azizi: Conceptualization, Methodology, Software, Validation, Investigation, Writing - original draft, Writing - review & editing, Visualization. **Bahman Zamani:** Conceptualization, Methodology, Supervision, Writing - review & editing, Project administration. **Shekoufeh Kolahdouz-Rahimi:** Conceptualization, Supervision, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A

As shown in Fig. 18, by right clicking on an etl file and selecting the SEET option from a popup menu, the tool will be executed. In the beginning, the configuration wizard is shown to the tester. As can be seen in Fig. 19 the tester configures by selecting source and target metamodels. In this case, the symbolic execution of the transformation is started.

Following the configuration, the tester is asked to select a mode. If the tester selects the automatic mode, SEET traverses executable paths of the transformation automatically. Otherwise, in the case that the tester selects the interactive mode, satisfiability of conditions is determined by the tester's control (Fig. 20-1).

Assuming the tester selects an interactive mode, first the tester is asked to specify whether an instance of in-pattern of the rule exists (Fig. 20-2). Moreover, in the event that rules consist of a guard, the tester specifies if guards are satisfiable. Considering also Listing 1, first, the guard of the Member2Male rule in line 9 is considered. Since in this guard there exists the operation call, the operation is unrolling and its statements symbolically execute line by line. Wherever a condition exists in these statements, the mentioned condition is asked from the tester. Herein the first condition is in line 24 and according to Fig. 20-3 the condition is asked from the tester. For the convenience of the tester, instead of the *self* keyword in the transformation definition, its related class namely Member is written in the wizard.

Assuming the tester considered the "Member.familyMother.isDefined()" condition to be satisfiable, to put it simply, the tester specified that the Member is the mother of the family, the symbolic execution engine reaches the return statement. Afterward, the body of rules which their entry condition and their guards are satisfiable symbolically executed. There is one assignment statement in line 20 which the familyName operation calling exists on its RHS. The symbolic execution engine executes statements of the operation. At the first line of the operation, it confronts the "self.familyFather.isDefined()" condition. As the tester specified the Member is a mother of the family, this condition considered as unsatisfiable and the symbolic execution goes ahead without asking.

The symbolic execution engine goes through else block and arrives at the "self.familyMother.isDefined()" condition. As the condition is asked from the tester earlier, the previous result is used

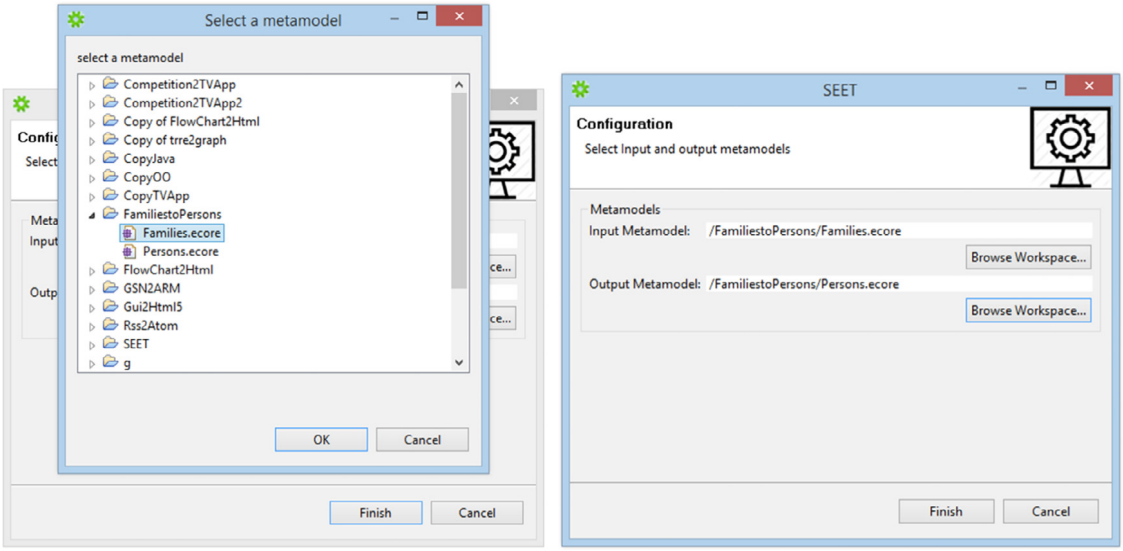


Fig. 19. The configuration.

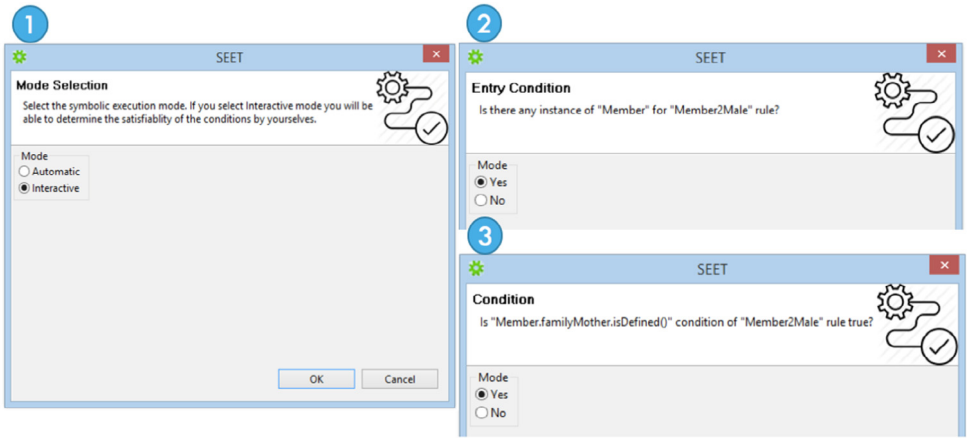


Fig. 20. Interaction with the tester: (1) Selecting execution mode, (2) Asking satisfiability of the entry condition, (3) Asking the satisfiability of the condition.

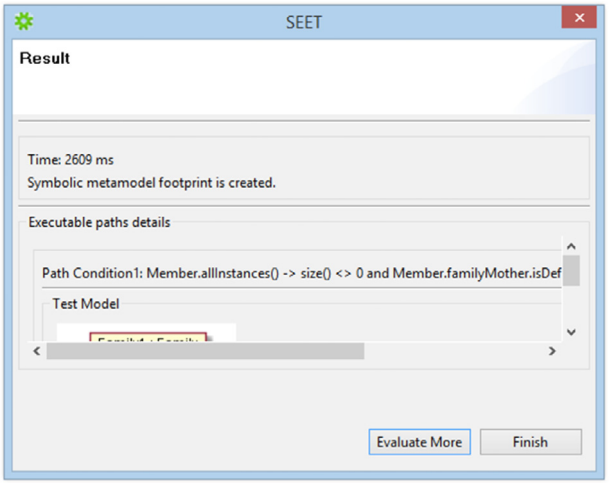


Fig. 21. The result containing the satisfiable path condition and the test model.

and the condition is not asked again from the tester. Symbolic execution engine arrived at the return statement and returns the symbolic value. The result of the assignment statement is applied to the SMF and since there are no other statements in the body of the rule, the symbolic execution of the program is ended and the result is shown to the tester (Fig. 21). The result involves the time of symbolic execution, the satisfiable path condition, and its corresponding test model. The satisfiable path condition is also stored as a text file. Furthermore, the SMF of Fig. 11 is generated. The tester can click on Finish and investigate the SMF such that the final correctness of the ETL transformation is specified. It is also possible that the tester clicks on "Evaluate More" and inputs the expected SMF (Fig. 22) and the tool specifies that whether the transformation is implemented correctly (Fig. 23). In case that the tester would selected automatic mode, the result involves all satisfiable path conditions and their corresponding test models. For each path condition, its corresponding SMF is generated.

Appendix B

In the SMF, there exists at most one class for each of the target metamodel classes. This section aims to elaborate on how the approach handles in case after a transformation, there exist two instances of the same class with different attributes or references.

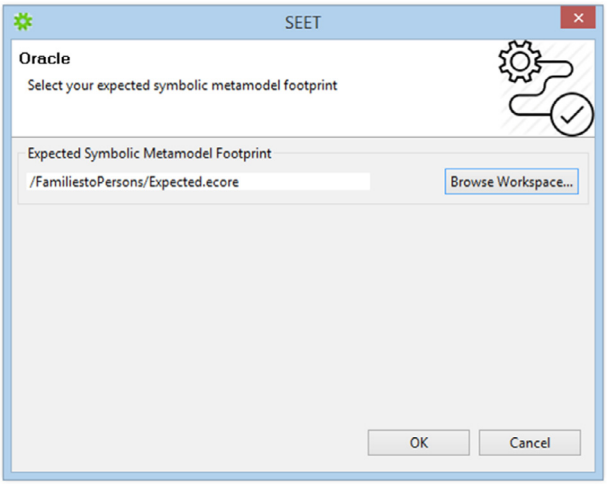


Fig. 22. Selecting the expected SMF.

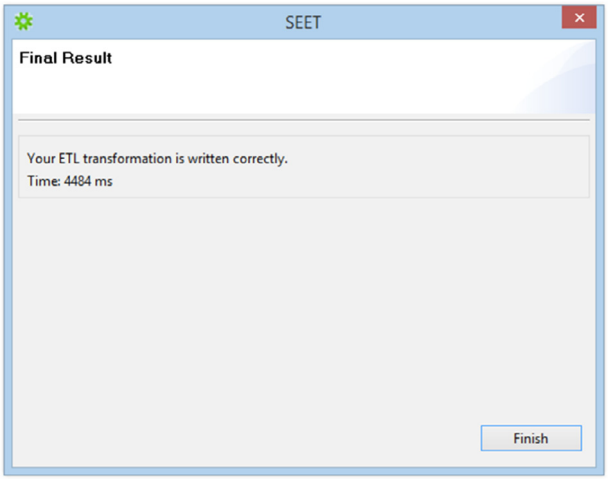


Fig. 23. The final result.

If the attribute values are different, all of them are listed in an enumeration. If there should difference in the references, e.g., one instance should be collected to a specific class, while the other instance not, it is highly dependent on how the transformation program is written. Consider the following examples.

(1) Consider Listing 8. An example of the input model and its corresponding output model in the concrete execution can be seen in Fig. 24. In the input model, we have two instances of the same class with different attribute values. In the output model, we have two instances of Class B with different references, i.e., one has a "ref" reference, and the other one does not.

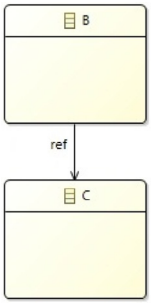


Fig. 25. The SMF of PC1.

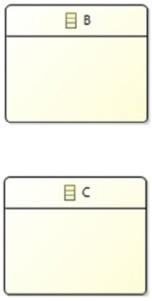


Fig. 26. The SMF of PC2.

Now, we want to see whether or not SEET is capable of covering such transformation. The SMFs, which are generated by the tool and corresponds to each path condition, are shown in Figs. 25 and 26. Considering the aim of the transformation, the tester checks whether each SMF matches the PC.

PC1: A.allInstances().size() <> 0 and A.age >= 10

PC2: A.allInstances().size() <> 0 and not A.age >= 10

PC3: not A.allInstances().size() <> 0

The corresponding SMF of PC3 is empty.

As you can see, SEET completely covers the first example.

(2) Consider Listing 9. An example of the input model and its corresponding output in the concrete execution can be seen in Fig. 27. In the output model, we have two instances of Class B with different references, i.e., one has a "ref" reference, and the other one does not.

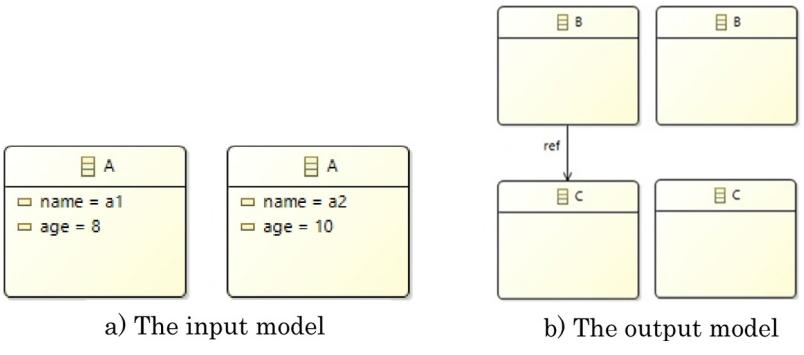


Fig. 24. Input and output models of the A2BC transformation.

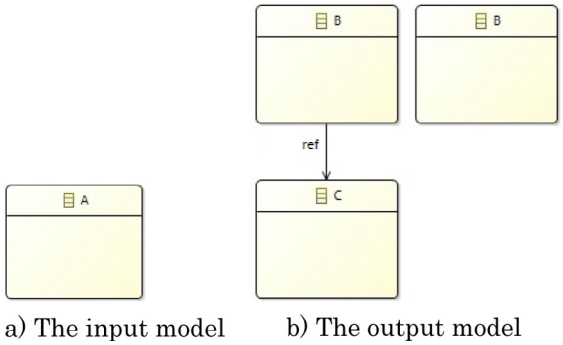


Fig. 27. Input and output models of the A2BBC transformation.

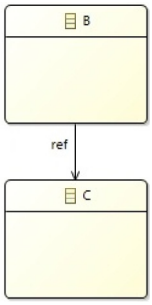


Fig. 28. The SMF of PC1.

```
1 rule A2B
2    transform a:Source!A
3    to b:Target!B, c: Target!C {
4
5    if (a.age >= 10)
6       b.ref = c;
7 }
```

Listing 8. The A2BC transformation in ETL

```
1 rule A2B
2    transform a:Source!A
3    to b1:Target!B, b2:Target!B, c: Target!C {
4
5       b1.ref = c;
6 }
```

Listing 9. The A2BBC transformation in ETL

The SMFs, which are generated by the SEET tool and corresponds to each path condition, are shown in Fig. 28.

PC1: A.allInstances().size() <> 0

PC2: not A.allInstances().size() <> 0

The corresponding SMF of PC2 is empty.

In the second example, the approach has a limitation. This is due to our definition of the SMF. Since in the symbolic execution we have to work in a higher meta level, i.e., this footprint is part of the target metamodel, as in a metamodel we have one class of any type, in the SMF also we have to have one class of any type.

About attributes, the tester sees if there are all attribute values. About references, with respect to the corresponding path condition, the tester can check the correctness.

References

Ab. Rahim, L., Whittle, J., 2015. A survey of approaches for verifying model transformations. Softw. Syst. Model. 14 (2), 1003–1028.

Al-Sibahi, A.S., Dimovski, A.S., Wąsowski, A., 2016. Symbolic execution of high-level transformations. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering. pp. 207–220.

Alhwikem, F.H.M., Paige, R.F., Rose, L.M., Alexander, R.D., 2016. A systematic approach for designing mutation operators for MDE languages. In: Proceedings of the 13th Workshop on Model-Driven Engineering, Verification and Validation co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MODELS. pp. 54–59.

Almendros-Jiménez, J.M., Becerra-Terón, A., 2016. Automatic generation of ecore models for testing ATL transformations. In: International Conference on Model and Data Engineering. MEDI, In: LNCS, vol. 9893, pp. 16–30.

Azizi, B., Zamani, B., Kolahdouz-Rahimi, S., 2017. Contract verification of ETL transformations. In: 2017 7th International Conference on Computer and Knowledge Engineering, ICCKE. pp. 154–160.

Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I., 2018. A survey of symbolic execution techniques. ACM Comput. Surv. 51 (3), 50.

Boehm, B.W., 1984. Verifying and validating software requirements and design specifications. IEEE Softw. 1 (1), 75–88.

Boolean Algebra, 2018. Boolean algebra - 2. laws - boolean algebra tutorial. [Online]. Available: https://ryanstutorials.net/boolean-algebra-tutorial/boolean-algebra-laws.php. (Accessed 02 April 2018).

Brambilla, M., Cabot, J., Wimmer, M., 2017. Model-Driven Software Engineering in Practice, Vol. 3, No. 1, second ed. Morgan & Claypool Publishers.

Brottier, E., Fleurey, F., Steel, J., Baudry, B., Le Traon, Y., 2006. Metamodel-based test generation for model transformations: an algorithm and a tool. In: Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium on. pp. 85–94.

Burgueno, L., Troya, J., Wimmer, M., Vallecillo, A., 2015. Static fault localization in model transformations. IEEE Trans. Softw. Eng. 41 (5), 490–506.

Cabot, J., Clarisó, R., Riera, D., 2007. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering. pp. 547–548.

Cuadrado, J.S., Guerra, E., de Lara, J., 2017. Static analysis of model transformations. IEEE Trans. Softw. Eng. 43 (9), 868–897.

Czarnecki, K., Helsen, S., 2006. Feature-based survey of model transformation approaches. IBM Syst. J. 45 (3), 621–645.

2017. Epsilon - users. [Online]. Available: https://www.eclipse.org/epsilon/users/. (Accessed 23 April 2017).

Fondement, F., Silaghi, R., 2004. Defining model driven engineering processes. In: Third International Workshop in Software Model Engineering. pp. 1–11.

García-Domínguez, A., Kolovos, D.S., Rose, L.M., Paige, R.F., Medina-Bulo, I., 2011. EUnit: A unit testing framework for model management tasks. In: 14th International Conference on Model Driven Engineering Languages and Systems. MODELS, In: LNCS, vol. 6981, pp. 395–409.

Godefroid, P., Levin, M.Y., Molnar, D., 2012. SAGE: whitebox fuzzing for security testing. Queue 10 (1), 20.

González, C.A., Büttner, F., Clarisó, R., Cabot, J., 2012. Emftocsp: A tool for the lightweight verification of emf models. In: 2012 First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches, FormSERA. pp. 44–50.

González, C.A., Cabot, J., 2012. ATLTest: a white-box test generation approach for ATL transformations. In: International Conference on Model Driven Engineering Languages and Systems. MODELS, In: LNCS, vol. 7590, pp. 449–464.

González, C.A., Cabot, J., 2014. Test data generation for model transformations combining partition and constraint analysis. In: International Conference on Theory and Practice of Model Transformations. ICMT, In: LNCS, vol. 8568, pp. 25–41.

Guerra, E., Soeken, M., 2015. Specification-driven model transformation testing. Softw. Syst. Model. 14 (2), 623–644.

Jakumeit, E., et al., 2014. A survey and comparison of transformation tools based on the transformation tool contest. Sci. Comput. Program. 85, 41–99.

Jeanneret, C., Glinz, M., Baudry, B., 2011. Estimating footprints of model operations. In: International Conference on Software engineering, ICSE.

Jézéquel, J.M., Barais, O., Fleurey, F., 2011. Model driven language engineering with Kermeta. In: Generative and Transformational Techniques in Software Engineering III. GTTSE, In: LNCS, vol. 6491, pp. 201–221.

Jilani, A.A., Iqbal, M.Z., Khan, M.U., 2014. A search based test data generation approach for model transformations. In: International Conference on Theory and Practice of Model Transformations. ICMT, In: LNCS, vol. 8568, pp. 17–24.

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., 2008. ATL: A model transformation tool. Sci. Comput. Program. 72 (1–2), 31–39.

King, J.C., 1976. Symbolic execution and program testing. Commun. ACM 19 (7), 385–394.

Kolovos, D.S., Paige, R.F., Polack, F.A.C., 2006. Model comparison: a foundation for model composition and model transformation testing. In: Proceedings of the 2006 international workshop on Global integrated model management. 2006. pp. 13–20.

Kolovos, D.S., Paige, R.F., Polack, F.A.C., 2006b. The epsilon object language (EOL). In: European Conference on Model Driven Architecture-Foundations and Applications. In: LNCS, vol. 4066, pp. 128–142.
Kolovos, D.S., Paige, R.F., Polack, F.A.C., 2008. The epsilon transformation language. In: International Conference on Theory and Practice of Model Transformations. ICMT, In: LNCS, vol. 5063, pp. 46–60.
Kolovos, D.S., Paige, R.F., Rose, L.M., Polack, F.A.C., 2007. Implementing the interactive television applications case study using epsilon. In: Model-Driven Development Tool Implementers Forum.
Kolovos, D., Rose, L., Paige, R., García-Domínguez, A., 2017. The epsilon book. [Online]. Available: https://www.eclipse.org/epsilon/doc/book/.
Kuhlmann, M., Hamann, L., Gogolla, M., 2011. Extensive validation of ocl models by integrating sat solving into use. In: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. TOOLS, In: LNCS, vol. 6705, pp. 290–306.
Lano, K., Clark, T., Kolahdouz-Rahimi, S., 2015. A framework for model transformation verification. Form. Asp. Comput. 27 (1), 193–235.
Logic Error Definition, 2018. [Online]. Available: https://techterms.com/definition/logic_error. (Accessed 28 October 2018).
Mens, T., Van Gorp, P., 2006. A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci. 152, 125–142.
Mottu, J.-M., Sen, S., Tisi, M., Cabot, J., 2012. Static analysis of model transformations for effective test generation. In: 2012 IEEE 23rd International Symposium on Software Reliability Engineering, ISSRE. pp. 291–300.
Oakes, B.J., Troya, J., Lúcio, L., Wimmer, M., 2018. Full contract verification for ATL using symbolic execution. Softw. Syst. Model. 17 (3), 815–849.
Popoola, S., Kolovos, D.S., Rodriguez, H.H., 2016. EMG: A domain-specific transformation language for synthetic model generation. In: International Conference on Theory and Practice of Model Transformations. ICMT, In: LNCS, vol. 9765, pp. 36–51.
Schmidt, D.C., 2006. Model-driven engineering. Comput. Comput. Soc. 39 (2), 25.
Schönböck, J., Kappel, G., Wimmer, M., Kusel, A., Retschitzegger, W., Schwinger, W., 2013. TETRABox-A generic white-box testing framework for model transformations. In: 2013 20th Asia-Pacific Software Engineering Conference, APSEC, Vol. 1. APSEC. pp. 75–82.
Schütze, H., Manning, C.D., Raghavan, P., 2008. Introduction to Information Retrieval, Vol. 39. Cambridge University Press.
Sen, S., Baudry, B., Mottu, J.-M., 2008. On combining multi-formalism knowledge to select models for model transformation testing. In: IEEE International Conference on Software Testing, ICST'08. pp. 328–337.
Sen, S., Mottu, J.-M., Tisi, M., Cabot, J., 2012. Using models of partial knowledge to test model transformations. In: International Conference on Theory and Practice of Model Transformations. ICMT, In: LNCS, vol. 7307, pp. 24–39.
Sendall, S., Kozaczynski, W., 2003. Model transformation: The heart and soul of model-driven software development. IEEE Softw. 20 (5), 42–45.
Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J., 2009. On the use of higher-order model transformations. In: European Conference on Model Driven Architecture-Foundations and Applications. In: LNCS, vol. 5562, pp. 18–33.
Troya, J., Bergmayr, A., Burgueno, L., Wimmer, M., 2015. Towards systematic mutations for and with ATL model transformations. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops, ICSTW. pp. 1–10.
Troya, J., Segura, S., Ruiz-Cortés, A., 2018. Automated inference of likely metamorphic relations for model transformations. J. Syst. Softw. 136, 188–208.
Wei, R., 2016. epsilonlabs/haetae, GitHub. [Online]. Available: https://github.com/epsilonlabs/haetae. (Accessed 25 August 2016).
Wei, R., Kolovos, D.S., 2014. Automated analysis, validation and suboptimal code detection in model management programs. In: BigMDE@ STAF. pp. 48–57.

Banafsheh Azizi received her B.Sc. from the Semnan University, Semnan, Iran, in September 2013, and her M.Sc. from the University of Isfahan, Isfahan, Iran, in January 2018, both in Computer Engineering (Software). Her research interests include Model-Driven Software Engineering (MDSE), Model Transformations, Verification of Model Transformations, and Symbolic Execution. She is a member of the MDSE Research Group at the University of Isfahan.

Bahman Zamani holds a Ph.D. in Computer Science from Concordia University, Montreal, QC, Canada for his work on the pattern language verification. Currently, he is an associate professor in the department of software engineering, University of Isfahan, Isfahan, Iran. His main research interest is Model-Driven Software Engineering (MDSE). He is the founder and director of the MDSE Research Group at the University of Isfahan.

Shekoufeh Kolahdouz-Rahimi is an Assistant Professor in the Software Engineering Department at the University of Isfahan. She is an active member of the Model Driven Software Engineering (MDSE) research group at this University. She has completed her PhD in Computer Science at Kings College London in 2013. Her current research interests include model-driven software engineering and domain-specific languages.