

# Solving the Families to Persons Case using EVL+Strace

Leila Samimi-Dehkordi  
samimi@eng.ui.ac.ir

Bahman Zamani  
zamani@eng.ui.ac.ir

Shekoufeh Kolahtouz-Rahimi  
sh.rahimi@eng.ui.ac.ir

MDSE research group  
Department of Software Engineering  
Faculty of Computer Engineering  
University of Isfahan, Iran

## Abstract

Benchmark is the subject of bidirectional transformation case study for the Transformation Tool Contest 2017. The example is a well-known model-to-model transformation from the ATL transformation Zoo named "Families to Persons". This paper presents a solution to provide the inter-model consistency using the Epsilon Validation Language (EVL) and domain-specific traceability techniques. We call this approach EVL+Strace.

## 1 Introduction

Bidirectional transformations (Bx) are used to restore the consistency when both source and target models are allowed to be modified, but they must remain consistent [1]. A Bx provides two directions from source to target (forward) and from target to source (backward); however, in most approaches, both transformation directions cannot be executed simultaneously [2]. In other words, at each time, only one of the models will be made consistent with the other. Besides that, in most cases, there is more than one way to resolve the inconsistencies. The "Families to Persons" case study [3] is an example of these cases. In this paper, a novel Bx approach called EVL+Strace solves the case study<sup>1</sup>. It provides an interactive bidirectional transformation that can execute both directions at the same time and propose more than one fixing ways to restore the consistency. EVL+Strace uses the Epsilon Validation Language (EVL) [4], which expresses constraints between heterogeneous models and evaluates them to resolve the occurred violations. The approach should check if any manual update (deletion, value modification, element relocation, and addition) has occurred in the source or target models. To recognize the type of updates, it is required to store the past information of source and target model in a correspondence (trace) model. This trace model conforms to a metamodel, that we believe it should be specific to the domains of source and target metamodels [5]. EVL+Strace applies EVL on the case-specific trace metamodel to provide a solution for Bx. The rest of the paper is structured as follows. Section 2 describes the EVL+Strace approach. Section 3 presents how the approach solves the case. Section 4 studies the evaluation of the proposed solution. Section 5 concludes the paper.

## 2 EVL+Strace

The EVL+Strace approach defines EVL *modules*. Modules consist of a set of invariants (*constraints*) grouping in the *context*. An EVL constraint contains two main parts including *check* and *fix* blocks. In the check block,

---

*Copyright © by the paper's authors. Copying permitted for private and academic purposes.*

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at <http://ceur-ws.org>

<sup>1</sup>The code of the case solution and the test files are accessible at <http://lsamimi.ir/EVLStrace.htm>

a condition is specified that must be true. If it is evaluated to be false, the fix block triggers some statements to resolve the violation. The Epsilon Object Language (EOL) [6] specifies the checking expressions and fixing statements. The defined constraints in EVL+Strace are applied on the elements of three metamodels including source, case-specific trace, and target. The case-specific trace metamodel defines strongly typed links between source and target meta-elements. The approach code is verbose, and designing a trace metamodel for each transformation case study is time consuming. Therefore, to automatically produce the trace metamodel and generate main parts of code, we implement a tool called MoDEBiTE<sup>2</sup>. EVL+Strace can detect independent updates on the source and target models by checking the information of trace model.

Figure 1 illustrates an example of three consistent models. There are two blue stars and a red square in the source model and there exist a blue triangle and a red circle in the consistent target model. Consequently, for each source or target element, there is a *link end* (TriangleTargetEnd object for the target triangle element) in the trace model. A trace link end keeps the real typed features, the same as features in the corresponding source/target element. There are also a Star2Triangle trace link to relate the star source ends to the triangle target end and a Square2CircleTraceLink link for connecting the square source end with a circle target end.

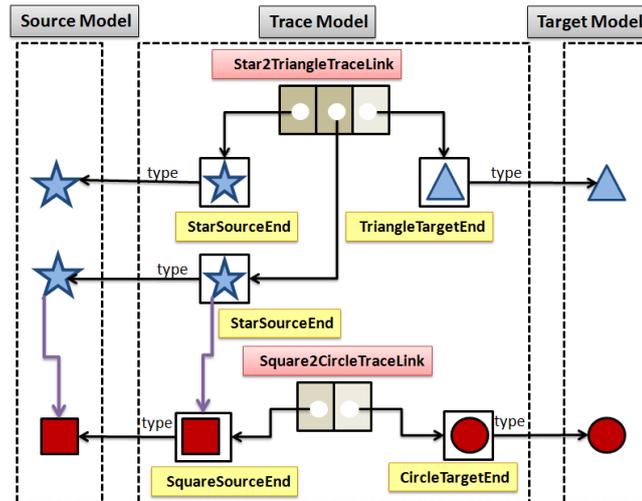


Figure 1: Example of consistent models

Figure 2 presents examples of possible atomic updates on source and target models. If the blue star is deleted manually from the source model, the type reference of the star trace link end to the source element will become empty (from the technical view it will convert to an empty EObject). This case of deletion is depicted in Figure 2a. To restore the consistency, it is required to delete the second star from the source model, the triangle from the target model, and the corresponding trace link and link ends from the trace model.

Figure 2b outlines addition of a green circle in the target model. Elements that are not referred by any trace link end are detected as new inserted ones. As depicted, no trace link ends refer to the green star; therefore, EVL+Strace detects it as a new inserted element. To make models consistent, a green square should be inserted in the source model. Moreover, a Square2Circle trace link and corresponding trace link ends with green values must be added to the trace model.

The example of value modification is presented in Figure 2c. In this case, the color value of the source square is changed from red to yellow. This modification is detected by comparing the value of the corresponding square source end to the color value of the square. When the comparison demonstrates the inequality, the value modification is recognized. To resolve this violation, it is required to update the color value of the target circle and corresponding trace link ends.

Figure 2d presents the element relocation. The reference to the source square is moved from lower star to the upper one. When the relocation is detected, the corresponding reference in the trace model should be changed. The relocation may result in other update operations such as deleting an old element and inserting a new one.

<sup>2</sup>The tool can be downloaded from the MoDEBiTE link in <http://mdse.bahmanzamani.com/tools/>

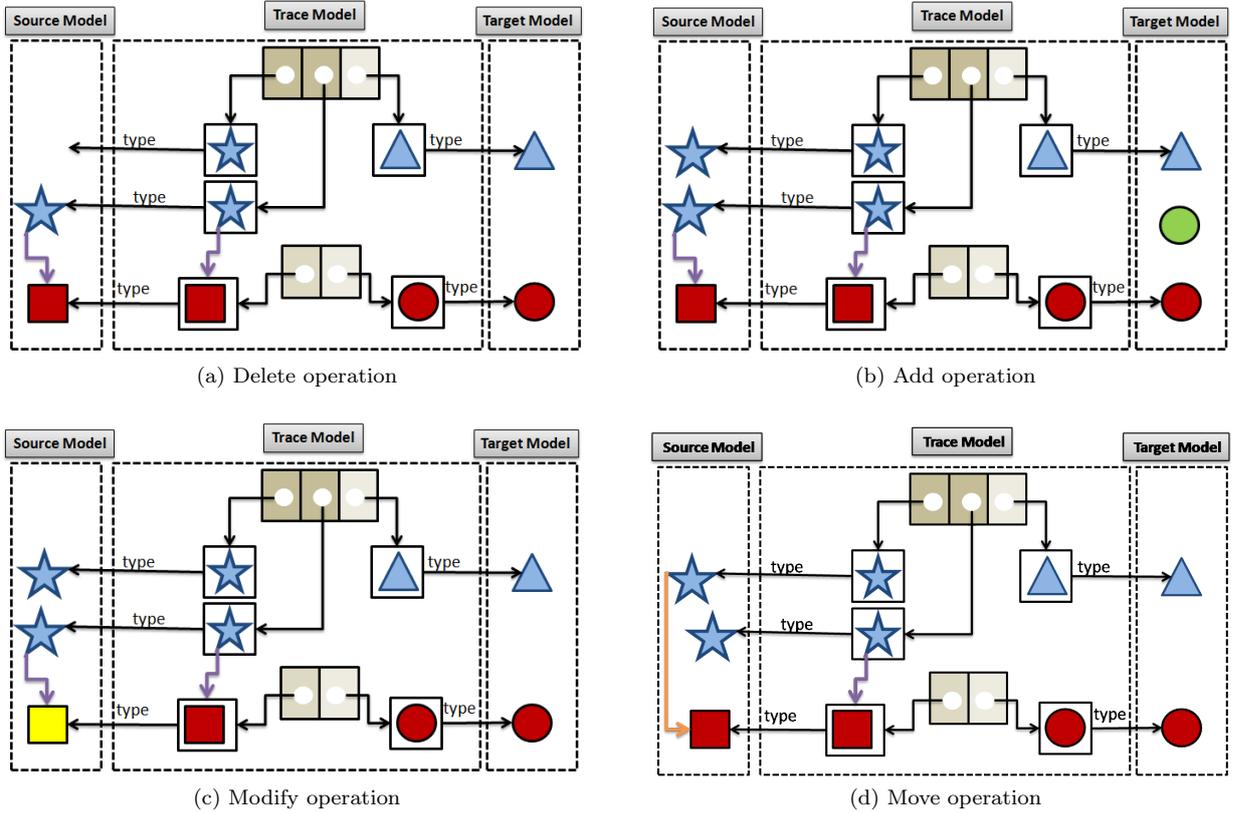


Figure 2: Abstract picture of atomic update operations on source and target models

### 3 Solution

In this section, we show how EVL+Strace works using the Families2Persons trace metamodel. Figure 3 shows the case-specific trace metamodel, which is generated automatically by MoDEBiTE. The root of the metamodel is `TraceModel`. It has two kinds of trace links that are `Reg2RegTraceLink` and `FamilyMember2PersonsTraceLink`. The former connects `FamilyRegisterSourceEnd` to `PersonRegisterTargetEnd`. The latter links `FamilySourceEnd` and `FamilyMemberSourceEnd` to `PersonTargetEnd`. The instance object of `FamilySourceEnd` keeps information of the corresponding `Family` object in the source model such as the value of the `name` and the references to the `MemberSourceEnd` objects. To access the corresponding source/target object, the trace link end has a reference type. For instance, `FamilySourceEnd` defines a `familySourceEndType` reference, which refers to the `Family` object in the source model. Note that, the relation between trace links and trace link ends is a bidirectional reference; therefore, accessing the link (end) is possible from the link end (link).

To modularize the EVL+Strace code, some *EOL* operations are defined for checking or fixing various types of updates. The MoDEBiTE tool automatically generates these operations. The EVL module of EVL+Strace consists of *pre* block, deletion constraints, modification and relocation constraints, and addition ones. The *pre* block sets the `useXmiIds` feature of three resources (models) to `true`. Through this setting, all created objects in three models have their own *xmi:ids*. The Bx code deals with objects by means of these ids.

*Deletion* constraints are defined in the context of the *SourceEnds* or *TargetEnds*. They check if a source/target element has been removed and fix the violation by deleting the corresponding *TraceLinkEnd* instance. In the *fix* block, the owner link of that link end is notified from this deletion and another constraint in the context of that *TraceLink* is called. The called constraint deletes all link end objects and their corresponding source/target elements that the *TraceLink* is referred to. *Addition* constraints are specified in the context of source/ target meta-classes. A new element is an object that has no fingerprint in the trace model. In other words, if there is no trace link end referring to that object, it is detected as a new element. While an addition in source (target) is recognized, the *fix* block should first add corresponding object in the target (source). Then, it adds corresponding trace link ends for new inserted elements in the trace. At last, it links them by a typed trace link.

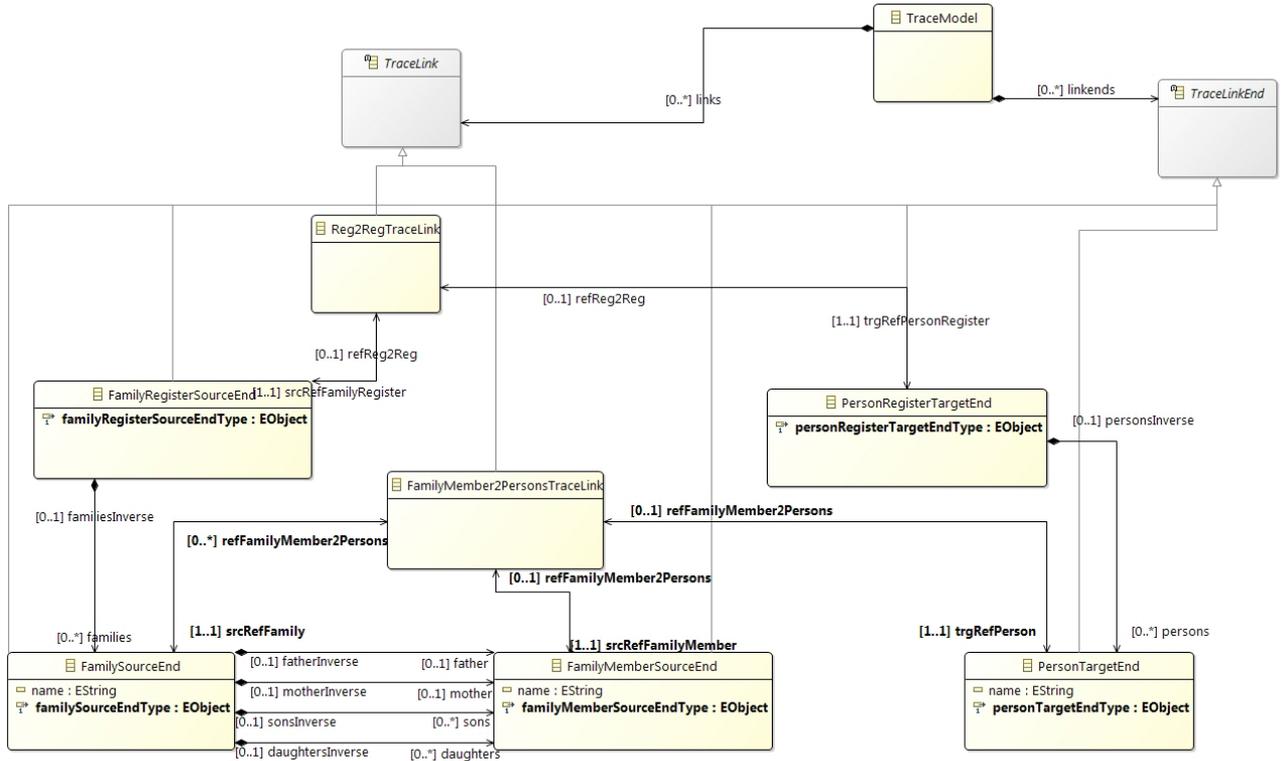


Figure 3: The case-specific trace metamodel for Families to Persons case study

*Modification and relocation* constraints check and fix modifications of the attribute values. They are specified in the context of the *SourceEnds* or *TargetEnds* like deletion constraints. When a modification is recognized by a checking operation, the propagate operation should be called to fix the violation (for instance, modifying the name of **Family** or **Member** objects results in changing the name of corresponding **Person(s)**). Developer may want to delete some objects and add new ones when a modification occurs (modifying the last name of a **Person** object is an example of this situation). Each reference in source/ target model has an equivalent reference in the trace model (if that reference is related to the transformation scenario). For instance, the **father**, **mother**, **sons**, **daughters** references are defined in the trace metamodel. If a reference of trace refers to a trace link end, and its equivalent reference in the source (target) refers to the object that is not corresponding to the mentioned trace link end, an element relocation is detected. Based on the relocation, a fixing strategy should be defined to restore the consistency.

An example of the EVL+Strace constraint is presented in Listing 1. This constraint checks if the **name** of the **FamilyMemberSourceEnd** object is modified (`self.nameIsModified()`). The `nameIsModified()` operation compare the names of **self** object and its correspondence (**FamilyMember** object in source). If the mentioned values are not equal, then it returns **true**. When the check expression (negation of the `nameIsModified()` operation) becomes **false**, EVL shows the message to the user in the validation view. By right clicking on the appeared *message*, the *title* of the *fix* block is presented to the user. When the user clicks it, the statements of the fix block (here `self.namePropagates()`) are executed. (to see more examples refer to Appendix A).

```

1 context Families2Persons!FamilyMemberSourceEnd{
2   guard: not self.isRemoved() and not self.refFamilyMember2Persons.endTypeIsRemoved()
3   constraint nameIsModified{
4     check: not self.nameIsModified()
5     message: 'name of '+self+' is modified'
6     fix{
7       title:'Propagate the modification'
8       do{ self.namePropagates();}
9     }}}

```

Listing 1: nameIsModified constraint in the context of FamilyMemberSourceEnd

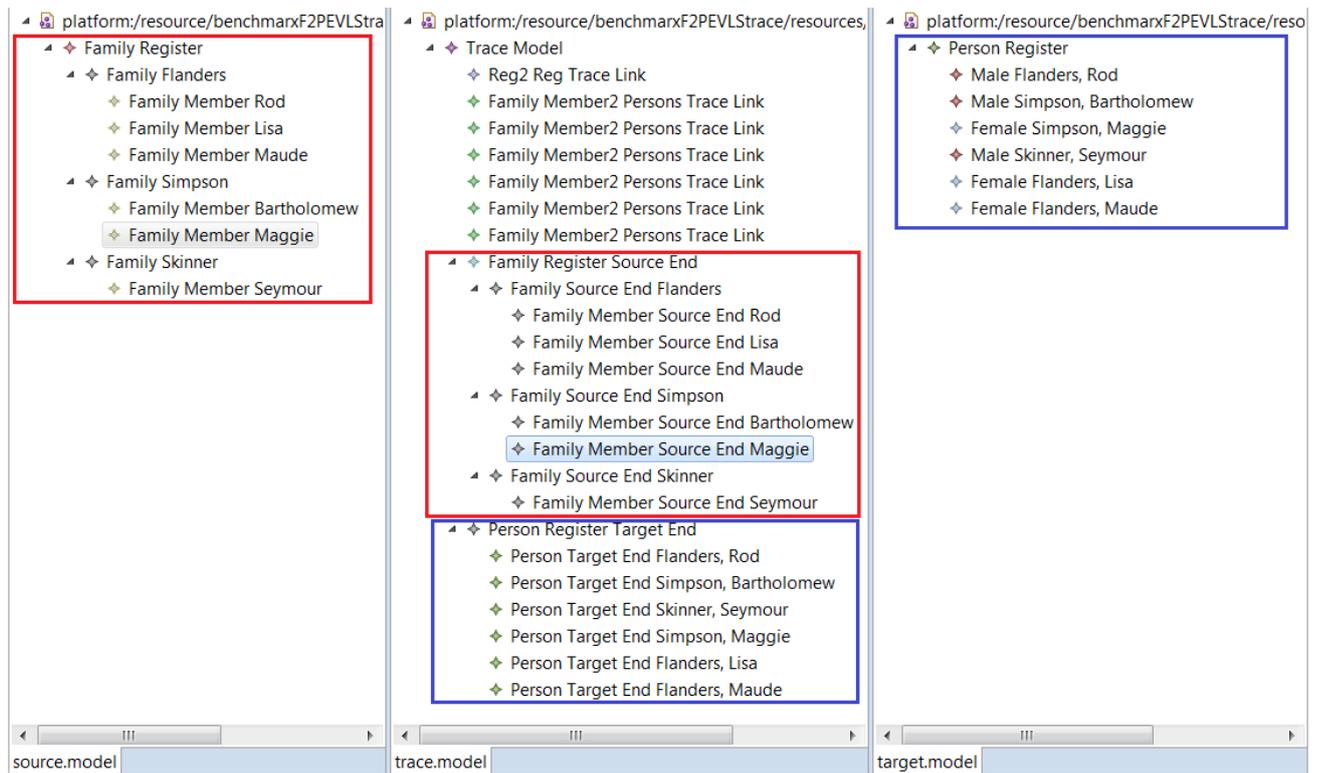


Figure 4: Example of consistent models for Families to Persons case study

Figure 4 presents an example of three consistent models for the Families to Persons case study. For each **Family** element (*Flanders*, *Simpson* and *Skinner*) in the source, there exists a corresponding **FamilySourceEnd** object in the trace model. Likewise, for each member in the source, there is a member in the trace. All source end objects in the trace keep the information (the value of the **name** attributes) of elements of the source model. Note the relation between **FamilySourceEnd** elements and **FamilyMemberSourceEnd** objects, which are the same as the relation between families and members in the source (e.g. the *Skinner* family has a father *Seymour* both in the source and trace models). From the target side, for each **Person** (**Male** or **Female**) object, there is an element of the **PersonTargetEnd** type, which has the same name. A **FamilyMember2PersonsTraceLink** link connects each family member linkend to a person linkend. Since there is one family register and one person register in the trace model, there exists a single **Reg2RegTracelink** object to connect them.

### 3.1 Interactive versus Automatic

EVL+Strace implements an interactive transformation system. In special cases, where there is no conflict between the manual changes on the source and target models, it is possible to specify the constraint in order to be executed automatically. Benefiting from the EVL syntax, Figure 5 shows how to rewrite the constraints to make the auto-fix possible. In automatic EVL+Strace the shape of code is changed, in which fix blocks are removed and their statements are shifted to the check block.

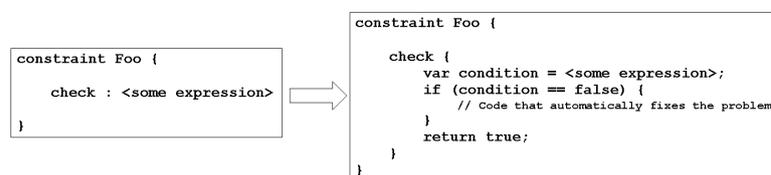


Figure 5: Interactive fix (left) versus automatic fix (right)

When the number of violations in interactive case is enormous, the user must spend extra effort to select from the alternatives. However, being interactive can be beneficial in *check-only mode*. In this case, the user may only want to know which constraints are broken, but it is not needed to enforce the consistency. EVL+Strace

does not need to specify the *execution mode*. The order of selecting the violated messages, which must be fixed, is important in some cases. Therefore, the approach handles execution order by defining some *lazy* constraints, which is required to be called from other constraints.

## 4 Evaluation

To test the solution, we use *EUnit* and *Workflow* tools [7] of the Epsilon framework. It is required to change the code of EVL+Strace to have automatic behavior. In this case, multiple deletions get the approach into trouble, while the interactive approach can pass this case. To have an automatic transformation, a *Configuration* metamodel is introduced to preserve the `preferExistingToNewFamily` and `preferParentToChild` values. From the Bx tool architecture variability view [8], the proposed approach is an *incremental corr-based* Bx tool. We use some update examples defined in `FamilyHelper.eol` and `PersonHelper.eol` files to provide test cases. Table 1 presents the results of testing EVL+Strace.

Table 1: The result of test case correctness

#	direction	Policy	Change Type	Test Case Name	Result
1	fwd	fixed	-	testInitialiseSynchronisation	expected pass
2	fwd	fixed	attribute	testFamilyNameChangeOfEmpty	expected pass
3	fwd	fixed	add	testCreateFamily	expected pass
4	fwd	fixed	add	testCreateFamilyMember	expected pass
5	fwd	fixed	add	testNewFamilyWithMultiMembers	expected pass
6	fwd	fixed	add	testNewDuplicateFamilyNames	expected pass
7	fwd	fixed	add	testDuplicateFamilyMemberNames	expected pass
8	bwd	runtime	add ( $e \wedge p$ )	testCreateMalePersonAsSon	expected pass
9	bwd	runtime	add ( $e \wedge p$ )	testCreateMembersInExistingFamilyAsParents	expected pass
10	bwd	runtime	add ( $e \wedge \neg p$ )	testCreateMalePersonAsSon	expected pass
11	bwd	runtime	add ( $e \wedge \neg p$ )	testCreateMembersInExistingFamilyAsParents	expected pass
12	bwd	runtime	add ( $e \wedge p$ )	testCreateDuplicateMembersInExistingFamilyAsChildren	expected pass
13	bwd	runtime	add ( $\neg e \wedge p$ )	testCreateMalePersonAsParent	expected pass
14	bwd	runtime	add ( $\neg e \wedge p$ )	testCreateMembersInNewFamilyAsParents	expected pass
15	bwd	runtime	add ( $\neg e \wedge p$ )	testCreateDuplicateMembersInNewFamilyAsParents	expected pass
16	bwd	runtime	add ( $\neg e \wedge \neg p$ )	testCreateMalePersonAsSon	expected pass
17	bwd	runtime	add ( $\neg e \wedge \neg p$ )	testCreateFamilyMembersInNewFamilyAsChildren	expected pass
18	bwd	runtime	add ( $\neg e \wedge \neg p$ )	testCreateDuplicateFamilyMembersInNewFamilyAsChildren	expected pass
19	fwd	fixed	add	testIncrementalInserts	expected pass*
20	fwd	runtime	del	testIncrementalDeletions	expected pass*
21	fwd	fixed	attribute	testIncrementalRename	expected pass*
22	fwd	fixed	move	testIncrementalMove	expected pass*
23	fwd	fixed	add+del	testIncrementalMixed	expected pass*
24	fwd	fixed	move	testIncrementalMoveRoleChange	expected pass*
25	fwd	fixed	-	testStability	expected pass
26	fwd	fixed	-	testHippocraticness	expected pass
27	bwd	fixed	add	testIncrementalInsertsFixedConfig	expected pass
28	bwd	runtime	add	testIncrementalInsertsDynamicConfig	expected pass
29	bwd	runtime	del	testIncrementalDeletions	failure
30	bwd	runtime	attribute	testIncrementalRenamingDynamic	expected pass
31	bwd	runtime	del+add	testIncrementalMixedDynamic	failure
32	bwd	runtime	add	testIncrementalOperational	expected pass
33	bwd	runtime	-	testStability	expected pass
34	bwd	runtime	-	testHippocraticness	expected pass

From all 34 test cases, automatic EVL+Strace approach has 32 expected pass and two failures. The date value in set birthday operations (defined in `PersonHelper.eol`) is specified by the `cal.getTime()` statement that returns the date, time (with millisecond), and time zone. The millisecond and time zone are specified based on the current case of the system. Since the expected target models in our test cases are not actively created, the generated and expected target models are only different in two values (millisecond and time zone). In other words, the birthday values of the generated and expected target models are the same in the first parts. Therefore, some results in Table 1 are determined by star(\*) which show this case.

## 5 Conclusion

This paper presents a bidirectional model-to-model transformation solution to the TTC 2017 Families to Persons case study. The proposed solution is based on a novel approach named EVL+Strace, which uses the EVL language (one of the Epsilon family languages) and case-specific trace metamodel. The trace metamodel (correspondence metamodel) is specific to the domains of the Families and Persons case studies. The approach defines constraints to check user updates with the use of EVL. This language enables us to fix the violations if an inconsistency is recognized in constraints. It is possible to program more than one fixing ways, and interactively ask user to restore the consistency. To test the solution, we change the constraints to fix the violations automatically. The evaluation presents that from all 34 test cases, automatic EVL+Strace has 32 expected pass and two failures.

## References

- [1] K. Czarnecki, J. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger, “Bidirectional Transformations: A Cross-Discipline Perspective,” in *Theory and Practice of Model Transformations*, vol. 5563 of *Lecture Notes in Computer Science*, pp. 260–283, 2009.
- [2] S. Hidaka, M. Tisi, J. Cabot, and Z. Hu, “Feature-based classification of bidirectional transformation approaches,” *Software & Systems Modeling*, vol. 15, no. 3, pp. 907–928, 2016.
- [3] A. Anjorin, T. Buchmann, and B. Westfechtel, “Thefamilies to persons case,” in *Transformation Tool Contest 2017*, CEUR-WS, pp. 15–30, 2017.
- [4] D. Kolovos, R. Paige, and F. Polack, “On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages,” in *Rigorous Methods for Software Construction and Analysis*, vol. 5115 of *Lecture Notes in Computer Science*, pp. 204–218, 2009.
- [5] D. S. Kolovos, R. F. Paige, and F. A. Polac, “On-demand merging of traceability links with models,” in *2nd EC-MDA Workshop on Traceability (July 2006)*, 2006.
- [6] D. S. Kolovos, R. F. Paige, and F. A. Polac, “The Epsilon Object Language (EOL),” in *Model Driven Architecture – Foundations and Applications: Second European Conference, ECMDA-FA 2006.*, pp. 128–142, 2006.
- [7] D. S. Kolovos, L. M. Rose, R. F. Paige, and A. Garcia-Dominguez, *The Epsilon Book*. Eclipse, 2010.
- [8] A. Anjorin, Z. Diskin, F. Jouault, H.-S. Ko, E. Leblebici, and B. Westfechtel, “Benchmarx reloaded: A practical benchmark framework for bidirectional transformations,” in *Proceedings of the 6th International Workshop on Bidirectional Transformations*, CEUR-WS, pp. 15–30, 2017.

## A Appendix: Details of our solution

### A.1 EOL operations

We divide the EOL operations into four groups including *Auxiliary*, *Delete*, *Modify*, and *Add* operations. The last three groups have two types, i.e., *Check* and *Fix*. Auxiliary operations are used by other EOL operations. They compute values and get or find objects. A Delete operation checks if an object is removed, or in some cases, it deletes an object from models. Following that, Modify operations identifies whether an attribute value is modified, or an element is relocated and then propagates the update. Finally, Add operations investigate if a source/target object is manually added, or in some cases, they insert new elements in models, fix the references, and transform the references from one model into another. For this case study, the MoDEBiTE tool generates 18 auxiliary operations. Three of them compute the values of the name attributes for Family, FamilyMember, and Person objects. Five operations are for getting the source/target objects from the trace link ends and five ones are defined to get the corresponding trace link end for an object in the source/target model. Listing 2 presents examples of auxiliary operations for the case study.

```
1 operation computeFamilyName(x:String):String
2 { return x.split(', ').first; }
3 operation computeMemberName(x:String):String
4 { return x.split(', ').second; }
5 operation computePersonName(x:String,y:String):String
```

```

6 { return x+', '+y; }
7 @chached
8 operation Source!FamilyRegister getTraceLinkEnd()
9 {
10     var seq = Families2Persons!FamilyRegisterSourceEnd.all.
11         select(s|not s.familyRegisterSourceEndType.isTypeOf(Families2Persons!EObject));
12     for (s in seq)
13         if (self.id = s.familyRegisterSourceEndType.id)
14             return s;
15     return null;
16 }

```

Listing 2: Example of automatically generated Auxiliary operations

Additionally, we define eight auxiliary operations (Listing 3) to make programming easier such as `isMale()` operation to check if a member is a male person or `getFamily()` to find the Family object from a member object.

```

1 operation Source!FamilyMember isFather():Boolean{ return self.fatherInverse.isDefined();}
2 operation Source!FamilyMember isMother():Boolean{ return self.motherInverse.isDefined();}
3 operation Source!FamilyMember isSon():Boolean{ return self.sonsInverse.isDefined();}
4 operation Source!FamilyMember isDaughter():Boolean{ return self.daughtersInverse.isDefined();}
5 operation Source!FamilyMember isMale():Boolean{ return self.isFather() or self.isSon();}
6 operation Source!FamilyMember isFemale():Boolean{ return self.isMother() or self.isDaughter();}

```

Listing 3: Example of user defined Auxiliary operations

The tool generates five operations from the *Add-Check* category, named `isNew`, to check if a source/target object is new inserted or not. The `isNew` operation for `FamilyMember` object is shown in Listing 4.

```

1 @chached
2 operation Source!FamilyMember isNew(): Boolean
3 {
4     var seq = Families2Persons!FamilyMemberSourceEnd.all.
5         select(s|not s.familyMemberSourceEndType.isTypeOf(Families2Persons!EObject));
6     for (s in seq)
7         if (self.id = s.familyMemberSourceEndType.id)
8             return false;
9     return true;
10 }

```

Listing 4: `isNew` operation for `FamilyMember` objects

There are 63 operations in the *Add-Fix* category:

1. Five operations for adding trace link ends (because there are five different types of trace link ends). An example is shown in Listing 5.

```

1 operation addFamilyMemberSourceEnd(object: Source!FamilyMember ){
2     var end = new Families2Persons!FamilyMemberSourceEnd;
3     end.familyMemberSourceEndType = object;
4     end.name = object.name;
5     Families2Persons!FamilyMemberSourceEnd.all.add(end);
6     Families2Persons!TraceModel.all.first.linkends.add(end);
7     return end;
8 }

```

Listing 5: `addFamilyMemberSourceEnd` operation for adding `MemberSourceEnd` in the trace model

2. Two operations for adding trace links (two types of trace links).

```

1 operation addReg2RegTraceLink (srcFamilyRegisterSourceEnd: Families2Persons!FamilyRegisterSourceEnd,
2     tarPersonRegisterTargetEnd: Families2Persons!PersonRegisterTargetEnd){
3     var link = new Families2Persons!Reg2RegTraceLink;
4     link.srcRefFamilyRegister = srcFamilyRegisterSourceEnd;
5     link.trgRefPersonRegister = tarPersonRegisterTargetEnd;
6     Families2Persons!Reg2RegTraceLink.all.add(link);
7     Families2Persons!TraceModel.all.first.links.add(link);
8     return link;
9 }

```

Listing 6: Operations for adding trace links

3. Six operations for inserting new objects in the source and target models. Listing 7 presents the `insertFamilyMember` operation, which takes a `Person` object and create a `FamilyMember` object in the source.

```

1 operation insertFamilyMember (personObject : Target!Person ): Source!FamilyMember{
2   var source = new Source!FamilyMember;
3   source.name = computeMemberName(personObject.name);
4   return source;
5 }

```

Listing 7: `insertFamilyMember` operation for inserting a member in the source

4. 12 operations for setting source/target references and 12 operations for setting trace references.  
5. 24 operations for transforming from source/target references to trace references, and vice versa (Listing 8).

```

1 operation Families2Persons!FamilySourceEnd copyFamilySourceEndfather ()
2 { var modelObject = self.getEndType();
3   if(self.father.isDefined())
4     modelObject.setFather(self.father.getEndType());
5 }//copy from FamilySourceEnd.father to Source!Family.father

```

Listing 8: `copyFamilySourceEndfather` operation transforms elements from trace to source

6. Two operations for transforming from the families reference of `FamilyRegisterSourceEnd` to the persons reference of `PersonRegisterTargetEnd`, and vice versa (Listing 9).

```

1 operation copyFamilyRegisterSourceEndfamilies2PersonRegisterTargetEndpersons(){
2   for(familyRegister in Families2Persons!FamilyRegisterSourceEnd.all)
3     for(family in familyRegister.families)
4       for(familylink in family.refFamilyMember2Persons)
5         familyRegister.refReg2Reg.trgRefPersonRegister.setPersons(familylink.trgRefPerson);
6 }
7 operation copyPersonRegisterTargetEndpersons2FamilyRegisterSourceEndfamilies(){
8   for(personRegister in Families2Persons!PersonRegisterTargetEnd.all){
9     for(person in personRegister.persons)
10      personRegister.refReg2Reg.srcRefFamilyRegister.setFamilies(person.refFamilyMember2Persons.srcRefFamily);
11   }
12 }

```

Listing 9: Operations for transforming the relations between `SourceEnds` and `TargetEnds`

There are four *Modify-Check* operations for checking if the name attribute is modified or not. One example is presented in Listing 10.

```

1 operation Families2Persons!FamilySourceEnd nameIsModified(): Boolean
2 {
3   if(self.name<> self.familySourceEndType.name) return true;
4   else return false;
5 }

```

Listing 10: `nameIsModified` operation for `FamilySourceEnd`

For the *Modify-Fix* category, three operations are defined. Listing 11 shows one of these operations.

```

1 operation Families2Persons!FamilySourceEnd namePropagates(){
2   self.name = self.getEndType().name;
3   for (tr in self.refFamilyMember2Persons){
4     if(not tr.endTypeIsRemoved()){
5       tr.trgRefPerson.name = computePersonName(self.name,tr.trgRefPerson.name.split(', ').second);
6       var targetObject = Target!Person.all.selectOne(o|o.id = tr.trgRefPerson.personTargetEndType.id);
7       targetObject.name = computePersonName(self.name,targetObject.name.split(', ').second);
8     }}
9   return self.name;}

```

Listing 11: `namePropagates()` operation for `FamilySourceEnd`

The MoDEBiTE tool generates 12 operations for the *Delete-Check* category including five operations for checking removed source/target objects from the context of trace link ends, five operations for checking trace link ends from the context of trace links and two ones for checking removed trace links. It also produces five operations for deleting the source/target objects and corresponding trace link ends.

## A.2 EVL constraints

The pre block sets the xmiId property of resources (Listing 12).

```
1 import 'atomicOperations.eol';
2 pre{
3   Families2Persons.resource.useXmiIds= true;
4   Source.resource.useXmiIds= true;
5   Target.resource.useXmiIds= true;}
```

Listing 12: pre block of the EVL+Strace code

Deletion constraints check if a source/target object is removed and fix the violation. MoDEBiTE generates 10 constraints for checking and fixing deletions. In Listing 13, the isRemoved constraint is defined in the context of FamilyMemberSourceEnd, and check if a FamilyMember object is removed.

```
1 context Families2Persons!FamilyMemberSourceEnd{
2   constraint isRemoved{
3     check: not self.isRemoved()
4     message: 'The '+self+' has a removed type'
5     fix{
6       title:'delete the '+self
7       do{
8         var trancelink = self.refFamilyMember2Persons;
9         delete self;
10        trancelink.satisfies("srcRefFamilyMemberIsRemoved");
11      }}
12 }
```

Listing 13: isRemoved constraint for FamilyMemberSourceEnd

Modification and relocation constraints check if any attribute value is modified or any element is moved. There are six constraints in this category. Listing 14 demonstrates the code of familyMemberRoleIsRelocated constraint.

```
1 context Families2Persons!FamilyMemberSourceEnd{
2   guard: not self.isRemoved() and not self.refFamilyMember2Persons.endTypeIsRemoved()
3   constraint familyMemberRoleIsRelocated{
4     guard: not self.getEndType().getFamily().isNew() and
5           self.getEndType().getFamily().getTraceLinkEnd()= self.getFamily()
6     check: not ((self.fatherInverse.isDefined() and not self.getEndType().fatherInverse.isDefined())
7               or (self.motherInverse.isDefined() and not self.getEndType().motherInverse.isDefined())
8               or (self.sonsInverse.isDefined() and not self.getEndType().sonsInverse.isDefined())
9               or (self.daughtersInverse.isDefined() and not self.getEndType().daughtersInverse.isDefined())
10              or (self.getEndType().getFamily().getTraceLinkEnd()<> self.getFamily()))
11     message: self+' role is changed or\n'+self+' family =' +self.getFamily()+
12             ' is changed to '+self.getEndType().getFamily()
13     fix{
14       title: 'Propagate the relocation for '+self
15       do{
16         var family= self.getEndType().getFamily();
17         var person;
18         if((self.fatherInverse.isDefined() or self.sonsInverse.isDefined()) and self.getEndType().isFemale()){
19           person =insertFemale(family,self.getEndType());
20           person.birthday = self.refFamilyMember2Persons.trgRefPerson.getEndType().birthday;
21           var personTargetEnd = addPersonTargetEnd(person);
22           delete self.refFamilyMember2Persons.trgRefPerson.getEndType();
23           delete self.refFamilyMember2Persons.trgRefPerson;
24           self.refFamilyMember2Persons.trgRefPerson = personTargetEnd;}
25         else
26         if((self.motherInverse.isDefined() or self.daughtersInverse.isDefined()) and self.getEndType().isMale()){
27           person = insertMale(family,self.getEndType());
28           person.birthday = self.refFamilyMember2Persons.trgRefPerson.getEndType().birthday;
29           var personTargetEnd = addPersonTargetEnd(person);
30           delete self.refFamilyMember2Persons.trgRefPerson.getEndType();
31           delete self.refFamilyMember2Persons.trgRefPerson;
32           self.refFamilyMember2Persons.trgRefPerson = personTargetEnd;}
33         self.refFamilyMember2Persons.srcRefFamily = family.getTraceLinkEnd();
34         copySrc2Trg();}
35     }}}}
```

Listing 14: familyMemberRoleIsRelocated constraint for detecting element relocation

We define 8 constraints for Addition category. Listing 15 represents the code of the familyObjectIsNew constraint. it checks if the family of one member in the source is moved to a new Family object.

```

1 context Source!FamilyMember{
2   constraint familyObjectIsNew{// the generated code of this constraint should be checked
3     guard: not self.isNew()
4     check: not ((self.fatherInverse.isDefined() and self.fatherInverse.isNew()) or
5               (self.motherInverse.isDefined() and self.motherInverse.isNew()) or
6               (self.sonsInverse.isDefined() and self.sonsInverse.isNew()) or
7               (self.daughtersInverse.isDefined() and self.daughtersInverse.isNew()))
8     message: self+' is related to the new family'
9     fix{
10      title: 'Insert the correspondence'
11      do{
12        var familyMemberSourceEnd = self.getTraceLinkEnd();
13        var family;
14        var familyMember2Personslink = self.getTraceLinkEnd().refFamilyMember2Persons;
15        var oldFamilySourceEnd = familyMember2Personslink.srcRefFamily;
16        var oldPerson = familyMember2Personslink.trgRefPerson.getEndType();
17        var person;
18        family = self.getFamily();
19        family.satisfies("isNew");
20        var familySourceEnd = family.getTraceLinkEnd();
21        familyMember2Personslink.srcRefFamily = familySourceEnd;
22        if((oldPerson.isTypeOf(Target!Female) and self.isMale()) or
23          oldPerson.isTypeOf(Target!Male) and self.isFemale())
24        {if(self.isMale()) person = insertMale(family,self);
25         else person = insertFemale(family,self);
26         person.birthday = oldPerson.birthday;
27         delete oldPerson.getTraceLinkEnd();
28         delete oldPerson;
29         var personTargetEnd = addPersonTargetEnd(person);
30         familyMember2Personslink.trgRefPerson = personTargetEnd;}
31        else{
32          oldPerson.name = computePersonName(family.name,self.name);
33          oldPerson.getTraceLinkEnd().name = oldPerson.name;}
34        copySrc2Trg();}
35      }}}

```

Listing 15: familyObjectIsNew constraint

In automatic EVL+Strace the shape of code is changed, in which fix blocks are removed and their statements are shifted to the check block. Listing 16 shows the excerpt code of this transfiguration. To make the code more readable and modular, we define some new operation and put the statements of fix block in them.

```

1 context Target!Female{
2   constraint isNew{
3     check{ var result = not self.isNew();
4     if(not result){
5       if(preferExistingToNewFamily and Source!Family.all.exists(f|f.name = computeFamilyName(self.name))){
6         if(preferParentToChild){self.fixIsNewExistingFamilyParent();}
7         else{self.fixIsNewExistingFamilyDaughter();}}
8     else{
9       if(preferParentToChild){self.fixIsNewNewFamilyParent();}
10      else{self.fixIsNewNewFamilyDaughter();}}
11     return true;
12   }}}

```

Listing 16: isNew constraint in the context of Female