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 Kolahdouz-Rahimi
sh.rahimi@eng.ui.ac.ir

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

Abstract

Benchmarx 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 is bidirectional in the sense that the source can be transformed into the target (forward direction) and vice versa (backward direction); 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 [1]. It supports an interactive bidirectional transformation that can execute both directions at the same time. It provides multiple ways to restore 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 (element deletion, relocation, and addition and attribute value modification) 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 models 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. Garcia-Dominguez, G. Hinkel, and Filip Křikava (eds.): Proceedings of the 10th Transformation Tool Contest, Marburg, Germany, 21-07-2017, published at http://ceur-ws.org

[1] 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.

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 source, trace, and target models for the Families to Persons case study. It presents examples of possible atomic updates on source and target models, including addition (number 1), deletion (number 2), relocation (number 3), and attribute value modification (number 4). As it is demonstrated, for each source or target object, there is an element in the trace that referenced to the object. Since there are not any element referring to `m4:MemberFamily`, EVL+Strace will recognizes it as a new inserted element. If the manual deletion (shown as number 2 in Figure 1) is performed by user, i.e., `p3:Male` is removed from the target model, the reference of `pT3:PersonTargetEnd` will become empty; therefore, EVL+Strace recognizes a deletion.

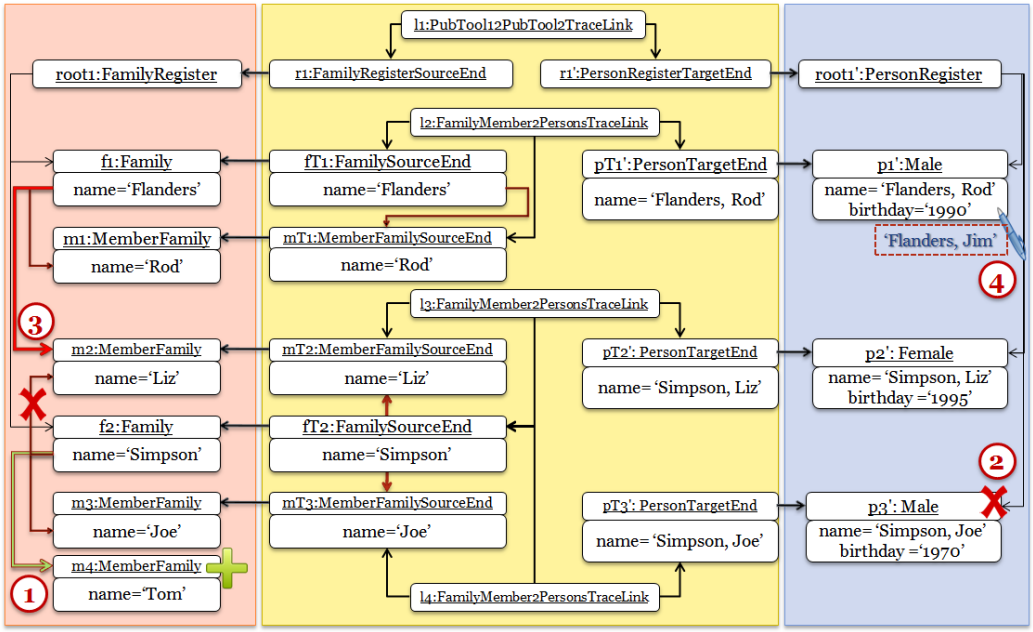

Figure 1: Example of consistent models

Label 3 in Figure 1 presents the element relocation. The reference from `f2:Family` to `m2:MemberFamily` is deleted and a new reference from `f1:Family` to `m2:MemberFamily` is added. In other words, The member `Liz` is moved from `Simpson` family to `Flanders` family. Since each reference in the source has a corresponding reference in the target, EVL+Strace can identify this relocation. As it is shown, there is no reference between `fT1:FamilySourceEnd` and `mT2:MemberFamilySourceEnd` and the reference between `fT2:FamilySourceEnd` and `mT2:MemberFamilySourceEnd` has no correspondence in the source. Therefore, an element relocation will be detected. An example of value modification is presented in Label 4 of Figure 1. In this case, the name of `p1´` is updated to `'Flanders, Jim'`. This modification is detected by comparing the name of `pT1:PersonTargetEnd` with the name of `p1:Male`. When the comparison demonstrates the inequality, the value modification is recognized. Note that, since the `birthday` attribute is not participated in the transformation, the `PersonTargetEnd` class does not contain this field.

3 Solution

In this section, we show how EVL+Strace works using the Families2Persons trace metamodel. Figure 2 shows the case-specific trace metamodel. 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 `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).

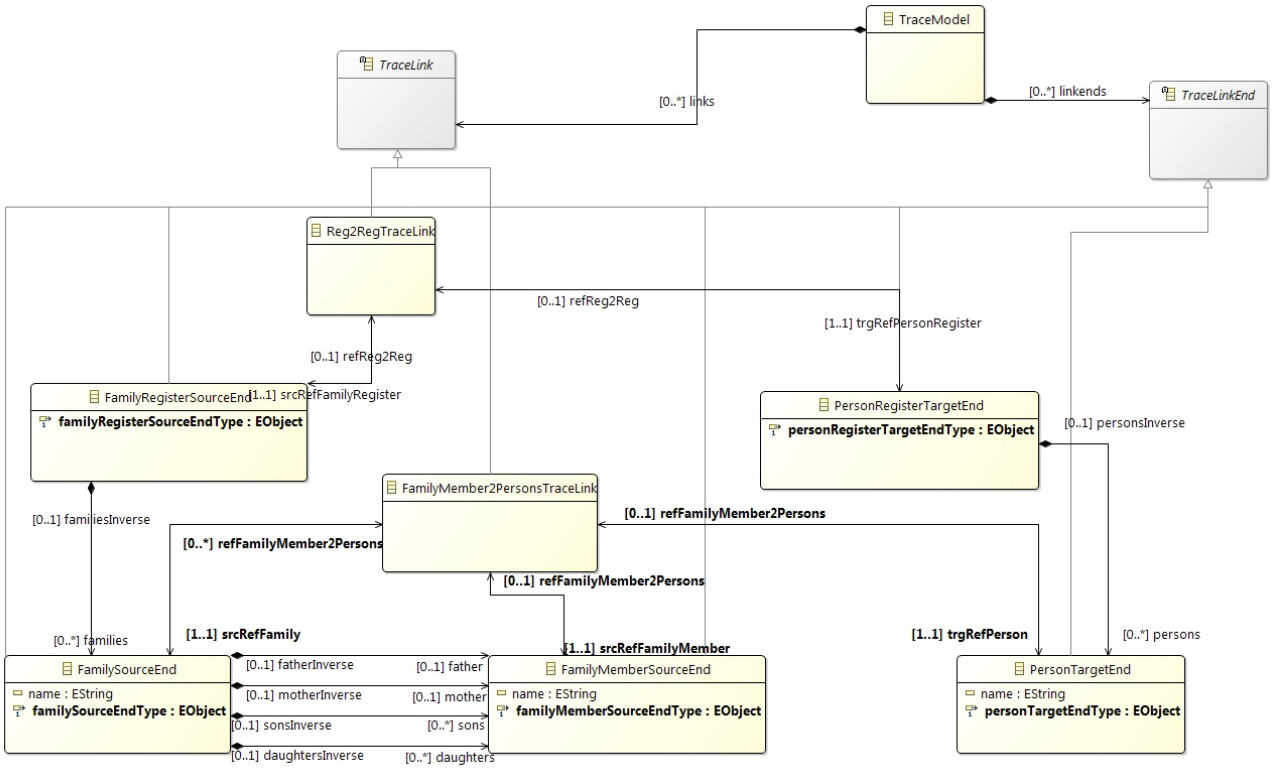

Figure 2: The case-specific trace metamodel for Families to Persons case study

To modularize the EVL+Strace code, some *EOL* operations are defined for checking or fixing various types of updates. The EVL module consists of *pre* block, deletion, modification, relocation, and addition constraints. 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 are referenced by the *TraceLink*. *Addition* constraints are specified in the context of source/ target meta-classes. A new element 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. Finally, it links them by a typed trace link.

Modification 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). The 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). *Relocation* constraints are also defined in the context of the *TraceLinkEnds*. 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* in the *Validation* view, the *title* of the *fix* block is presented to the user. When the user clicks on it, the statements of the fix block (here `self.namePropagates()`) are executed. (For more examples refer to Appendix A)

```
context Families2Persons!FamilyMemberSourceEnd{
  guard: not self.isRemoved() and not self.refFamilyMember2Persons.endTypeIsRemoved()
  constraint nameIsModified{
    check: not self.nameIsModified()
    message: 'name of '+self +' is modified'
    fix{
      title:'Propagate the modification'
      do{ self.namePropagates();}
    }}}
```

Listing 1: nameIsModified constraint in the context of FamilyMemberSourceEnd

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 [2] (please see Appendix A.3). EVL+Strace provides an interactive transformation system. In special cases, when 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. To provide auto-fix constraints, the shape of code should be changed, in which *fix* blocks are removed and their statements are shifted to the *check* block. 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 point of 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.

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 a case-specific trace metamodel. The trace metamodel (correspondence

[2]The tool can be downloaded from the MoDEBiTE link in http://mdse.bahmanzamani.com/tools/

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

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. It is possible to program more than one fixing ways, and interactively ask the 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, EVL+Strace can pass 32 cases.

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, "The Families to Persons Case," in *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences* (A. Garcia-Dominguez, G. Hinkel, and F. Krikava, eds.), CEUR Workshop Proceedings, CEUR-WS.org, July 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 13 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.

```
operation computeFamilyName(x:String):String
{ return x.split(', ').first; }
operation computeMemberName(x:String):String
{ return x.split(', ').second; }
operation computePersonName(x:String,y:String):String
{ return x+', '+y; }
@cached
operation Source!FamilyRegister getTraceLinkEnd()
{
  var seq = Families2Persons!FamilyRegisterSourceEnd.all.
    select(s|not s.familyRegisterSourceEndType.isTypeOf(Families2Persons!EObject));
  for (s in seq)
    if (self.id = s.familyRegisterSourceEndType.id)
      return s;
  return null;
}
```

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.

```
operation Source!FamilyMember isFather():Boolean{ return self.fatherInverse.isDefined();}
operation Source!FamilyMember isMother():Boolean{ return self.motherInverse.isDefined();}
operation Source!FamilyMember isSon():Boolean{ return self.sonsInverse.isDefined();}
operation Source!FamilyMember isDaughter():Boolean{ return self.daughtersInverse.isDefined();}
operation Source!FamilyMember isMale():Boolean{ return self.isFather() or self.isSon();}
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.

```
@cached
operation Source!FamilyMember isNew(): Boolean
{
  var seq = Families2Persons!FamilyMemberSourceEnd.all.
    select(s|not s.familyMemberSourceEndType.isTypeOf(Families2Persons!EObject));
  for (s in seq)
    if (self.id = s.familyMemberSourceEndType.id)
      return false;
  return true;
}
```

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.

```
operation addFamilyMemberSourceEnd(object: Source!FamilyMember ){
  var end = new Families2Persons!FamilyMemberSourceEnd;
  end.familyMemberSourceEndType = object;
  end.name = object.name;
  Families2Persons!FamilyMemberSourceEnd.all.add(end);
  Families2Persons!TraceModel.all.first.linkends.add(end);
  return end;
}
```

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

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

```
operation addReg2RegTraceLink (srcFamilyRegisterSourceEnd: Families2Persons!FamilyRegisterSourceEnd,
                tarPersonRegisterTargetEnd: Families2Persons!PersonRegisterTargetEnd){
  var link = new Families2Persons!Reg2RegTraceLink;
  link.srcRefFamilyRegister = srcFamilyRegisterSourceEnd;
  link.trgRefPersonRegister = tarPersonRegisterTargetEnd;
  Families2Persons!Reg2RegTraceLink.all.add(link);
  Families2Persons!TraceModel.all.first.links.add(link);
  return link;
}
```

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.

```
operation insertFamilyMember (personObject : Target!Person ): Source!FamilyMember{
  var source = new Source!FamilyMember;
  source.name = computeMemberName(personObject.name);
  return source;
}
```

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).

```
operation Families2Persons!FamilySourceEnd copyFamilySourceEndfather ()
{ var modelObject = self.getEndType();
  if(self.father.isDefined())
    modelObject.setFather(self.father.getEndType());
}//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).

```
operation copyFamilyRegisterSourceEndfamilies2PersonRegisterTargetEndpersons(){
  for(familyRegister in Families2Persons!FamilyRegisterSourceEnd.all)
    for(family in familyRegister.families)
      for(familylink in family.refFamilyMember2Persons)
```

```
      familyRegister.refReg2Reg.trgRefPersonRegister.setPersons(familylink.trgRefPerson);
}
operation copyPersonRegisterTargetEndpersons2FamilyRegisterSourceEndfamilies(){
  for(personRegister in Families2Persons!PersonRegisterTargetEnd.all){
    for(person in personRegister.persons)
      personRegister.refReg2Reg.srcRefFamilyRegister.setFamilies(person.refFamilyMember2Persons.srcRefFamily);
  }
}
```

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.

```
operation Families2Persons!FamilySourceEnd nameIsModified(): Boolean
{
  if(self.name<> self.familySourceEndType.name) return true;
  else return false;
}
```

Listing 10: nameIsModified operation for FamilySourceEnd

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

```
operation Families2Persons!FamilySourceEnd namePropagates(){
  self.name = self.getEndType().name;
  for (tr in self.refFamilyMember2Persons){
    if(not tr.endTypeIsRemoved()){
      tr.trgRefPerson.name = computePersonName(self.name,tr.trgRefPerson.name.split(', ').second);
      var targetObject = Target!Person.all.selectOne(o|o.id = tr.trgRefPerson.personTargetEndType.id);
      targetObject.name = computePersonName(self.name,targetObject.name.split(', ').second);
  }}
  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).

```
import 'atomicOperations.eol';
pre{
  Families2Persons.resource.useXmiIds= true;
  Source.resource.useXmiIds= true;
  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.

```
context Families2Persons!FamilyMemberSourceEnd{
  constraint isRemoved{
    check: not self.isRemoved()
    message: 'The '+self +' has a removed type'
    fix{
      title:'delete the '+self
      do{
        var tracelink = self.refFamilyMember2Persons;
        delete self;
        tracelink.satisfies("srcRefFamilyMemberIsRemoved");
    }}}
}
```

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.

```
context Families2Persons!FamilyMemberSourceEnd{
  guard: not self.isRemoved() and not self.refFamilyMember2Persons.endTypeIsRemoved()
  constraint familyMemberRoleIsRelocated{
    guard: not self.getEndType().getFamily().isNew() and
        self.getEndType().getFamily().getTraceLinkEnd()= self.getFamily()
    check: not ((self.fatherInverse.isDefined() and not self.getEndType().fatherInverse.isDefined())
        or (self.motherInverse.isDefined() and not self.getEndType().motherInverse.isDefined())
        or (self.sonsInverse.isDefined() and not self.getEndType().sonsInverse.isDefined())
        or (self.daughtersInverse.isDefined() and not self.getEndType().daughtersInverse.isDefined())
        or (self.getEndType().getFamily().getTraceLinkEnd()<> self.getFamily()))
    message: self+' role is changed or\n'+self+' family ='+self.getFamily()+
         ' is changed to '+self.getEndType().getFamily()
    fix{
      title: 'Propagate the relocation for '+self
      do{
      var family= self.getEndType().getFamily();
      var person;
      if((self.fatherInverse.isDefined() or self.sonsInverse.isDefined()) and self.getEndType().isFemale()){
        person =insertFemale(family,self.getEndType());
        person.birthday = self.refFamilyMember2Persons.trgRefPerson.getEndType().birthday;
        var personTargetEnd = addPersonTargetEnd(person);
        delete self.refFamilyMember2Persons.trgRefPerson.getEndType();
        delete self.refFamilyMember2Persons.trgRefPerson;
        self.refFamilyMember2Persons.trgRefPerson = personTargetEnd;}
      else
       if((self.motherInverse.isDefined() or self.daughtersInverse.isDefined()) and self.getEndType().isMale()){
        person = insertMale(family,self.getEndType());
        person.birthday = self.refFamilyMember2Persons.trgRefPerson.getEndType().birthday;
        var personTargetEnd = addPersonTargetEnd(person);
        delete self.refFamilyMember2Persons.trgRefPerson.getEndType();
        delete self.refFamilyMember2Persons.trgRefPerson;
        self.refFamilyMember2Persons.trgRefPerson = personTargetEnd;}
      self.refFamilyMember2Persons.srcRefFamily = family.getTraceLinkEnd();
      copySrc2Trg();}
    }}}
```

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.

```
context Source!FamilyMember{
  constraint familyObjectIsNew{// the generated code of this constraint should be checked
    guard: not self.isNew()
    check: not ((self.fatherInverse.isDefined() and self.fatherInverse.isNew()) or
    (self.motherInverse.isDefined() and self.motherInverse.isNew()) or
    (self.sonsInverse.isDefined() and self.sonsInverse.isNew()) or
    (self.daughtersInverse.isDefined() and self.daughtersInverse.isNew()))
    message: self+' is related to the new family'
    fix{
      title: 'Insert the correspondence'
      do{
      var familyMemberSourceEnd = self.getTraceLinkEnd();
      var family;
      var familyMember2Personslink = self.getTraceLinkEnd().refFamilyMember2Persons;
      var oldFamilySourceEnd = familyMember2Personslink.srcRefFamily;
      var oldPerson = familyMember2Personslink.trgRefPerson.getEndType();
      var person;
      family = self.getFamily();
      family.satisfies("isNew");
      var familySourceEnd = family.getTraceLinkEnd();
      familyMember2Personslink.srcRefFamily = familySourceEnd;
      if((oldPerson.isTypeOf(Target!Female) and self.isMale()) or
          oldPerson.isTypeOf(Target!Male) and self.isFemale())
      {if(self.isMale()) person = insertMale(family,self);
       else  person = insertFemale(family,self);
       person.birthday = oldPerson.birthday;
```

```
      delete oldPerson.getTraceLinkEnd();
      delete oldPerson;
      var personTargetEnd = addPersonTargetEnd(person);
      familyMember2Personslink.trgRefPerson = personTargetEnd;}
     else{
      oldPerson.name = computePersonName(family.name,self.name);
      oldPerson.getTraceLinkEnd().name = oldPerson.name;}
     copySrc2Trg();}
   }}}
```

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.

```
context Target!Female{
 constraint isNew{
   check{ var result = not self.isNew();
   if(not result){
    if(preferExistingToNewFamily and Source!Family.all.exists(f|f.name = computeFamilyName(self.name))){
     if(preferParentToChild){self.fixIsNewExistingFamilyParent();}
     else{self.fixIsNewExistingFamilyDaughter();}}
    else{
     if(preferParentToChild){self.fixIsNewNewFamilyParent();}
     else{self.fixIsNewNewFamilyDaughter();}}}
   return true;
   }}}
```

Listing 16: isNew constraint in the context of Female

A.3 The MoDEBiTE toolkit

The MoDEBiTE toolkit is developed to produce the artifacts of EVL+Strace transformation, including the specific trace metamodel and the Epsilon code (EOL operations and EVL constraints). To generate the mentioned artifacts, the toolkit asks the developer to design a weaving model conforming to the MoDEBiTE weaving metamodel. For the Families to Persons case study, MoDEBiTE can automatically generate the specific trace metamodel (presented in Figure 2) from the weaving model. As mentioned in Appendix A.1 and A.2, MoDEBiTE can produce main parts of the transformation code. Table 2 presents that how much of code can be generated by MoDEBiTE.

Table 2: The result of code generation

category of operations	generated (#)	refined (#)	user-defined (#)	total(#)
Auxiliary	13	0	8	21
Delete (Check and Fix)	17	0	0	17
Modify (Check and Fix)	6	3	1	7
Add (Check and Fix)	67	1	1	68
category of constraints	generated (#)	refined (#)	user-defined (#)	total(#)
Deletion	10	0	0	10
Modification and Relocation	4	2	2	6
Addition	6	3	2	8

Table 2 shows the categories of operations/constraints in the first column. It presents the number of generated operations/constraints by MoDEBiTE in the second column. The third column demonstrates how much of the generated code is required to be refined. The fourth column identifies the number of operations/constraints that should be defined by the user. In the last column, the total number of operations/constraints is presented.