Proceedings of the 10th International Workshop
Transformation Tool Contest

TTC 2017

Antonio Garcia-Dominguez
Georg Hinkel
Filip Křikava

Preface

The aim of the Transformation Tool Contest (TTC) series is to compare the expressiveness, the usability, and the performance of transformation tools along a number of selected case studies. A deeper understanding of the relative merits of different tool features will help to further improve transformation tools and to indicate open problems.

This contest was the tenth of its kind. For the fifth time, the contest was part of the Software Technologies: Applications and Foundations (STAF) federation of conferences. Teams from the major international players in transformation tool development have participated in an online setting as well as in a face-to-face workshop.

In order to facilitate the comparison of transformation tools, our programme committee selected three challenging cases via single blind reviews for which there were together ten solutions.

These proceedings comprise descriptions of the case study and all of the accepted solutions. In addition to the solution descriptions contained in these proceedings, the implementation of each solution (tool, project files, documentation) is made available in public version control repositories. Some solution are also available via the SHARE platform (http://share20.eu).

TTC 2017 involved open (i.e., non anonymous) peer reviews in a first round. The purpose of this round of reviewing was that the participants gained as much insight into the competitors' solutions as possible and also to identify potential problems. At the workshop, the solutions were presented. The expert audience judged the solutions along a number of case-specific categories, and prizes were awarded to the highest scoring solutions in each category. Finally, the solutions appearing in these proceedings were selected by our programme committee via single blind reviews. The full results of the contest are published on our website[1].

Besides the presentations of the submitted solutions, the workshop also comprised a live contest. That contest involved creating a solution for transformation reuse in the presence of multiple inheritance and redefinitions. The live contest was announced to all STAF attendees and participants were given four days to design, implement and test their solutions. The contest organisers thank all authors for submitting cases and solutions, the contest participants, the STAF local organisation team, the STAF general chair Gabriele Taentzer, and the program committee for their support.

21th July, 2017
Marburg, Germany

Antonio Garcia-Dominguez
Georg Hinkel
Filip Křikava

[1] `http://www.transformation-tool-contest.eu/`

Organisation

The Transformation Tool Contest has been organized by Philipps-Universität Marburg, Germany.

Program Committee

Olivier Barais	University of Rennes 1, France
Philippe Collet	Université Nice Sophia-Antipolis, France
Coen De Roover	Vrije Universiteit Brussel, Belgium
Antonio Garcia-Dominguez	Aston University, United Kingdom
Jeff Gray	University of Alabama, United States
Tassilo Horn	SHD, Germany
Akos Horvath	Budapest University of Technology and Economics, Hungary
Christian Krause	SAP Innovation Center, Germany
Filip Křikava	Czech Technical University, Czech Republic
Sonja Schimmler	Bundeswehr University Munich, Germany
Arend Rensink	University of Twente, The Netherlands
Louis Rose	University of York, United Kingdom
Bernhard Schatz	Technische Universitat Munchen, Germany
Massimo Tisi	Ecole des Mines de Nantes, France
Tijs van der Storm	Centrum Wiskunde & Informatica, The Netherlands
Pieter Van Gorp	Eindhoven University of Technology, The Netherlands
Gergely Varro	Technische Universitat Darmstadt, Germany
Bernhard Westfechtel	University of Bayreuth, Germany

Table of Contents

Part I.

Smart Grid Case

The TTC 2017 Outage System Case for Incremental Model Views

Georg Hinkel
FZI Research Center of Information Technologies
Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany
hinkel@fzi.de

Abstract

To cope with the increased complexity, physical systems are more and more supported by software systems consisting of multiple subsystems. Usually, each of the subsystems uses standards relevant to the subsystem for interoperability with other tools. Thus, one faces the problem that information about the system as a whole is distributed across multiple models. To solve this problem, model views can be introduced to combine these models and extract application-specific knowledge. As an example, the smart grid is a cyber-physical system where one is interested to detect, manage and prevent system outages. The information necessary to do this is split among the standards IEC 61970/61968, IEC 61850 and IEC 62056. This paper presents a benchmark case and evaluation framework for joining information spread across multiple models into a single view, based on a model-based outage management system for smart grids. Because cyber-physical systems often require very fast response times to changes of underlying models, the benchmark focuses especially on the incremental computation of model views.

1 Introduction

The complexity of todays systems, for example cyber-physical systems, makes it inevitable to divide the system into multiple subsystems that operate in different domains. In many of these domains, standards exist that the respective subsystem has to comply with or for which many tools can be reused.

For example, the smart grid is a cyber-physical system that spans the physical structures of the electricity network and the system of software systems that monitor, control, and repair the system in case of outages [1]. Currently, many heterogeneous systems and standards have to interoperate to achieve the desired reliability, stability, and efficiency of the electricity network.

Because each of these standards describe different aspects of the system, models according to these standards have to be combined if multiple aspects are required to gain some insights about the system. Applying model-driven engineering, model views are a tool to extract information from multiple models without confronting the user with unnecessary information for a particular purpose, for example analysis.

In the area of smart grids, an additional challenge is the size of the models and the frequency of changes. In combination, this means that very large amounts of data have to be processed in a very short amount of time. However, the changes usually only affect small parts of the model, which is why an incremental view computation appears beneficial.

In: A. Garcia-Dominguez, F. Krikava and G. Hinkel (eds.): Proceedings of the 10th Transformation Tool Contest, Marburg, Germany, 21-07-2017, published at http://ceur-ws.org

In this paper, we propose a benchmark based on selected model views created by Mittelbach and Burger [1], [2] for a model-based outage management system for smart grids. Section 2 introduces the case in a bit more detail, in particular the involved standards. Section 3 defines two tasks of the benchmark. Section 4 introduces the benchmark framework. Section 5 explains how solutions to the benchmark are to be evaluated.

2 Case Description

In the area of smart grids, the relevant standards are IEC 61970/61968, IEC 61850 and IEC 62056. These standards are briefly described below (taken from [1]):

IEC 61970/61968 The IEC 61970 standard defines the *Common Information Model (CIM)*, which is used to describe the physical components, measurement data, control and protection elements, and the SCADA system. It is defined in UML notation. The IEC 61968 standard is an extension of the CIM for the distribution network [3]. It is also called *distributed CIM (DCIM)*

IEC 61850 This is a series of standards for substations with the purpose of supporting interoperability of intelligent electronic devices (IED) in substation automation systems. It defines the *Abstract Communication Service Interface* with a mapping to concrete communication protocols, the XML-based *Substation Configuration Description Language (SCL)*, and the *Logical Node (LN)* model, which describes power system functions [4].

IEC 62056 *COSEM (Companion Specification for Energy Metering)* is the international standard for data exchange for meter reading, tariff and load control in the domain of electricity metering. It works together with the *Device Language Message Specification (DLMS)*. Together, they provide a communication profile to transport data from metering equipment to the metering system and to define a data model and communication protocols for data exchange [5].

While these standards are useful in their domain, one has to combine the information represented by these standards to detect and prevent outage situations. Mittelbach and Burger presented a model-based outage system that synchronizes models of these standards and consists of a set of 15 views to help operators to manage outage situations [1], [2].

3 Tasks

In the scope of the proposed benchmark, we focus on two model views contained in the model-based outage management system. A rather simple view is created to detect outages, while a second slightly more complex view supports the prevention of outages.

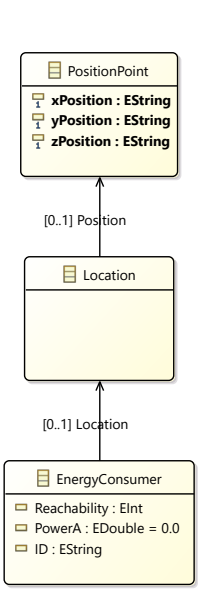
For both tasks, we present the original implementation of the view in ModelJoin [6], a language to specify both the view type and the view definition in a single specification through a language inspired by SQL joins. Currently, an idiomatic QVT-O model transformation is generated from this specification. Due to space limitations, we do not show the generated transformation, but it is available in the benchmark resources for reference.

3.1 Task 1: A view to detect outages

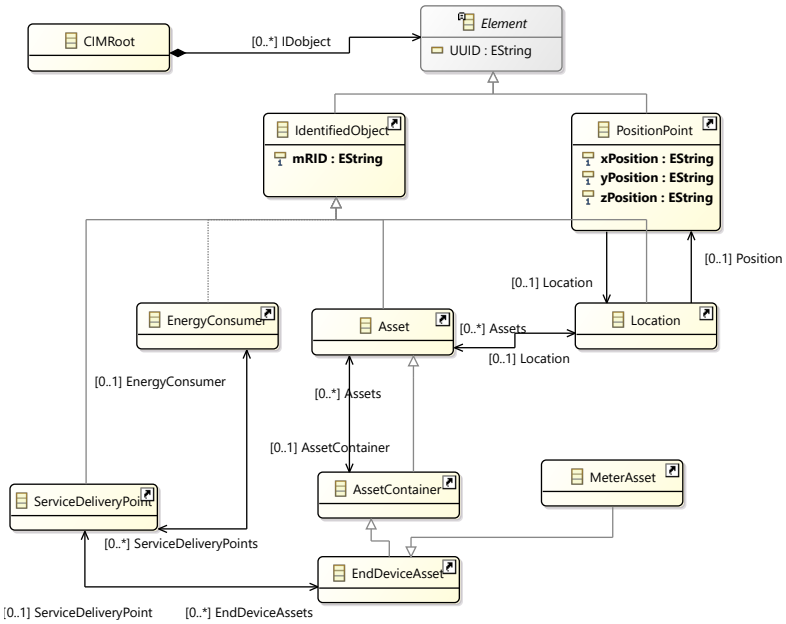
To detect an outage, we use the fact that a smart meter cannot send any data when it is cut off power supply. If this happens, the system can try to reach the meter but will receive a connection failure notification. This is used to detect outages without relying on customer feedback.

The information that a connection to a smart meter is lost is depicted in the CIM model. The relevant excerpt for this task is depicted in Figure 1b. It has to be matched with the corresponding physical devices in the COSEM model where its location is stored. The latter is depicted in Figure 1c.

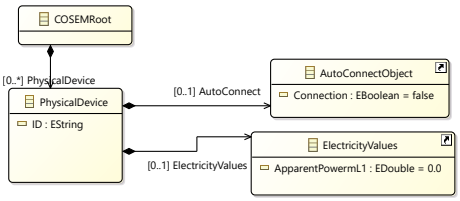
An implementation in ModelJoin is depicted in Listing 1.



(a) The Viewtype for the Outage Detection Task



(b) Excerpt from the CIM metamodel relevant for task 1



(c) Excerpt from the COSEM metamodel relevant for task 1

Figure 1: Metamodels in task 1

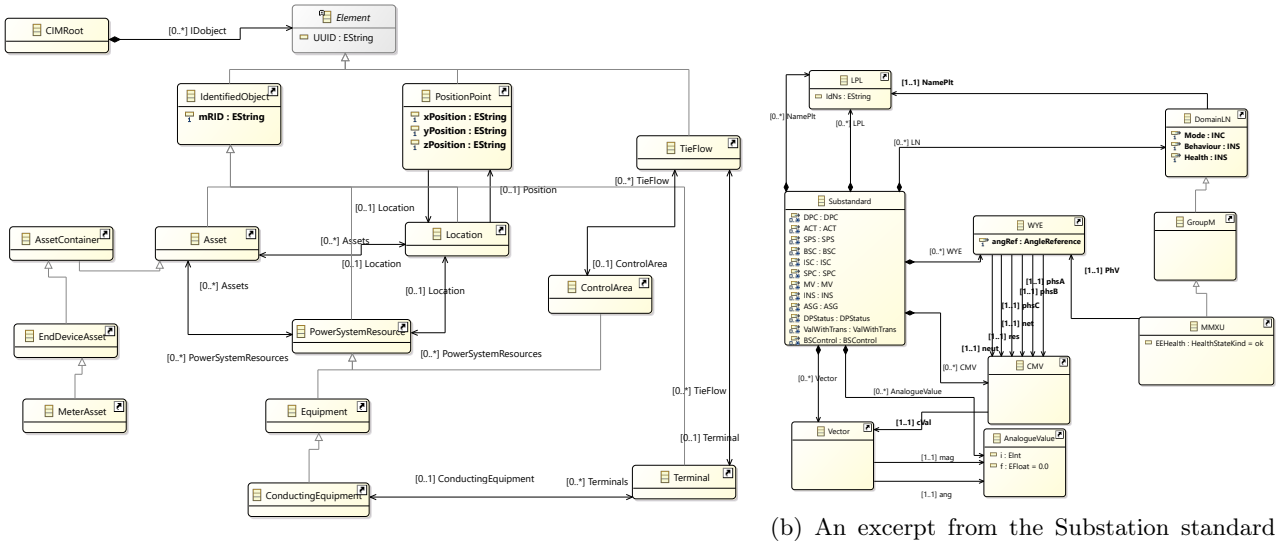
```
1  theta join CIM.IEC61968.Metering.MeterAsset with COSEM.PhysicalDevice where "CIM.IEC61968.Metering.MeterAsset.mRID␣=␣COSEM.
       PhysicalDevice.ID" as jointarget.EnergyConsumer {
2     keep calculated attribute "COSEM.PhysicalDevice.AutoConnect.Connection" as EnergyConsumer.Reachability:Integer
3     keep calculated attribute "COSEM.PhysicalDevice.ElectricityValues.ApparentPowermL1" as EnergyConsumer.PowerA:Double
4     keep calculated attribute "CIM.IEC61968.Metering.MeterAsset.ServiceDeliveryPoint.EnergyConsumer.mRID" as EnergyConsumer.ID:
          String
5     keep calculated attribute "if␣CIM.IEC61968.Metering.MeterAsset.ServiceDeliveryPoint.EnergyConsumer->oclIsKindOf(CIM.IEC61970.
          LoadModel.ConformLoad)␣then␣CIM.IEC61968.Metering.MeterAsset.ServiceDeliveryPoint.EnergyConsumer.ConformLoadGroup.
          SubLoadArea.LoadArea.ControlArea.mRID␣else␣CIM.IEC61968.Metering.MeterAsset.ServiceDeliveryPoint.EnergyConsumer.
          NonConformLoadGroup.SubLoadArea.LoadArea.ControlArea.mRID␣endif" as Consumer.ControlAreaID:String
6     keep outgoing CIM.IEC61968.Assets.Asset.Location as type jointarget.Location {
7        keep outgoing CIM.IEC61968.Common.Location.Position as type jointarget.PositionPoint {
8           keep attributes CIM.IEC61968.Common.PositionPoint.xPosition,
9           CIM.IEC61968.Common.PositionPoint.yPosition,
10          CIM.IEC61968.Common.PositionPoint.zPosition
11       }
12    }
13 }
```

Listing 1: Task 1 realized in ModelJoin

Based on the view specification in Listing 1, the view type in Figure 1a is generated.

A correct solution should match the meter assets in the CIM model with the physical devices in the COSEM model. For each of these matches, the result model should contain a model element that conforms to the generated view type class `EnergyConsumer`. For this element, a range of (partially derived) attributes shall be copied and the reference to the location of meter asset and physical device should be saved.

This reference to location and position point shall respect referential integrity. This means, if two meter assets in the CIM model reference the same location, their joins in the view should also reference the same `Location` element in the view.



(a) An excerpt from the CIM metamodel relevant for Task 2

(b) An excerpt from the Substation standard relevant for Task 2

Figure 2: Metamodels relevant for Task 2

3.2 Task 2: A view to prevent outages

The analysis algorithms to detect system disturbances proposed in [2] work on *phasor measurement data*: *Their basic concept is to compare the current phasor data of the traveling voltage wave with a historic set of normal phasor data and calculate an equality indicator like a correlation coefficient. This is compared with a certain benchmark. If it lies above, a failure is indicated. To enable this, the following information is necessary: a historic set of normal phasor data of that section, a matrix of the current phasor data and a calculation mechanism to compare the two followed by a comparison mechanism to decide if it is a failure or not.* [1]

Task 2 requires to match elements from all three domain standards. For the COSEM standard, the used metamodel excerpt is very similar to Task 1. The relevant metamodel excerpts for CIM and the substation standard are depicted in Figure 2a and Figure 2b, respectively.

The analysis viewtypes will not provide the analysis result but only the *matrix of phasor data* for the comparison. Six queries were defined in [1] that all have the same structure and provide the three-phase measurements of voltage, frequency, current, active power, reactive power and apparent power. An implementation of this matrix in ModelJoin is depicted in Listing 2.

```
1  theta join CIM.IEC61968.Metering.MeterAsset with substationStandard.LNNodes.LNGroupM.MMXU where "CIM.IEC61968.Metering.MeterAsset.
       mRID␣=␣substationStandard.LNNodes.LNGroupM.MMXU.NamePlt.IdNs" as jointarget.PMUVoltageMeter {
2     keep attributes CIM.IEC61970.Core.IdentifiedObject.mRID
3     keep calculated attribute "substationStandard.LNNodes.LNGroupM.MMXU.PhV.phsA.cVal.mag.f" as PMUVoltageMeter.VoltageAMag:Double
4     keep calculated attribute "substationStandard.LNNodes.LNGroupM.MMXU.PhV.phsA.cVal.ang.f" as PMUVoltageMeter.VoltageAAng:Double
5     keep calculated attribute "substationStandard.LNNodes.LNGroupM.MMXU.PhV.phsB.cVal.mag.f" as PMUVoltageMeter.VoltageBMag:Double
6     keep calculated attribute "substationStandard.LNNodes.LNGroupM.MMXU.PhV.phsB.cVal.ang.f" as PMUVoltageMeter.VoltageBAng:Double
7     keep calculated attribute "substationStandard.LNNodes.LNGroupM.MMXU.PhV.phsC.cVal.mag.f" as PMUVoltageMeter.VoltageCMag:Double
8     keep calculated attribute "substationStandard.LNNodes.LNGroupM.MMXU.PhV.phsC.cVal.ang.f" as PMUVoltageMeter.VoltageCAng:Double
9     keep calculated attribute "substationStandard.LNNodes.LNGroupM.MMXU.PhV.neut.cVal.mag.f" as PMUVoltageMeter.VoltageNeutMag:
          Double
10    keep calculated attribute "substationStandard.LNNodes.LNGroupM.MMXU.PhV.neut.cVal.ang.f" as PMUVoltageMeter.VoltageNeutAng:
          Double
11    keep calculated attribute "substationStandard.LNNodes.LNGroupM.MMXU.PhV.net.cVal.mag.f" as PMUVoltageMeter.VoltageNetMag:Double
12    keep calculated attribute "substationStandard.LNNodes.LNGroupM.MMXU.PhV.net.cVal.ang.f" as PMUVoltageMeter.VoltageNetAng:Double
13    keep calculated attribute "substationStandard.LNNodes.LNGroupM.MMXU.PhV.res.cVal.mag.f" as PMUVoltageMeter.VoltageResMag:Double
14    keep calculated attribute "substationStandard.LNNodes.LNGroupM.MMXU.PhV.res.cVal.ang.f" as PMUVoltageMeter.VoltageResAng:Double
15    keep supertype CIM.IEC61968.Assets.Asset as type jointarget.Asset {
16       keep outgoing CIM.IEC61968.Assets.Asset.Location as type jointarget.Location {
17          keep outgoing CIM.IEC61968.Common.Location.Position as type jointarget.PositionPoint {
18             keep attributes CIM.IEC61968.Common.PositionPoint.xPosition,
19             CIM.IEC61968.Common.PositionPoint.yPosition,
20             CIM.IEC61968.Common.PositionPoint.zPosition
21          }
22          keep outgoing CIM.IEC61968.Common.Location.PowerSystemResources as type jointarget.PowerSystemResource {
23             keep subtype CIM.IEC61970.Core.ConductingEquipment as type jointarget.ConductingEquipment {
24                keep outgoing CIM.IEC61970.Core.ConductingEquipment.Terminals as type jointarget.Terminal {
25                   keep outgoing CIM.IEC61970.Core.Terminal.TieFlow as type jointarget.TieFlow {
26                      keep outgoing CIM.IEC61970.ControlArea.TieFlow.ControlArea as type jointarget.ControlArea {
```

```
                    keep attributes CIM.IEC61970.Core.IdentifiedObject.mRID
                }
            }
        }
    }
  }
}

theta join CIM.IEC61968.Metering.MeterAsset with COSEM.PhysicalDevice where "CIM.IEC61968.Metering.MeterAsset.mRID = COSEM.
    PhysicalDevice.ID" as jointarget.PrivateMeterVoltage {
  keep attributes COSEM.PhysicalDevice.ID
  keep calculated attribute "COSEM.PhysicalDevice.ElectricityValues.VoltageL1" as PrivateMeterVoltage.VoltageA:Double
  keep calculated attribute "COSEM.PhysicalDevice.ElectricityValues.VoltageL2" as PrivateMeterVoltage.VoltageB:Double
  keep calculated attribute "COSEM.PhysicalDevice.ElectricityValues.VoltageL3" as PrivateMeterVoltage.VoltageC:Double
  keep supertype CIM.IEC61968.Metering.EndDeviceAsset as type jointarget.EndDeviceAsset {
    keep outgoing CIM.IEC61968.Metering.EndDeviceAsset.ServiceDeliveryPoint as type jointarget.ServiceDeliveryPoint {
      keep outgoing CIM.IEC61968.Metering.ServiceDeliveryPoint.EnergyConsumer as type jointarget.EnergyConsumer {
        keep attributes CIM.IEC61970.Core.IdentifiedObject.mRID
        keep subtype CIM.IEC61970.LoadModel.ConformLoad as type jointarget.ConformLoad {
          keep outgoing CIM.IEC61970.LoadModel.ConformLoad.LoadGroup as type jointarget.ConformLoadGroup {
            keep supertype CIM.IEC61970.LoadModel.LoadGroup as type jointarget.LoadGroup {
              keep outgoing CIM.IEC61970.LoadModel.LoadGroup.SubLoadArea as type jointarget.SubLoadArea {
                keep outgoing CIM.IEC61970.LoadModel.SubLoadArea.LoadArea as type jointarget.LoadArea {
                  keep outgoing CIM.IEC61970.LoadModel.EnergyArea.ControlArea as type jointarget.ControlArea
                }
              }
            }
          }
        }
        keep subtype CIM.IEC61970.LoadModel.NonConformLoad as type jointarget.NonConformLoad {
          keep outgoing CIM.IEC61970.LoadModel.NonConformLoad.LoadGroup as type jointarget.NonConformLoadGroup {
            keep supertype CIM.IEC61970.LoadModel.LoadGroup as type jointarget.LoadGroup
          }
        }
      }
    }
  }
}
```

Listing 2: Three Phase Measurement Matrix Viewtype

"Such analysis viewtypes can be used together with a basic network topology view to calculate the exact location of a failure. Phasors are traveling waves in the system, which means that the failure travels with the wave through the grid. Therefore it is important to find its origin. The topology viewtype includes the length of the transmission line segments. They can be used together with the timestamp of the measured phasor to calculate from where the wave came and where it was when the failure started. This is its origin [7]." [1]

From the ModelJoin view definition in Listing 2, the view type depicted in Figure 3 is generated.

The biggest difference to Task 1 besides the increased complexity is the fact that this view definition keeps subtypes of some model elements. If an energy consumer in a service delivery point is a `ConformLoad`, then the view computation should be different to the case when the energy consumer is a `NonConformLoad`.

4 Benchmark Framework

The benchmark framework is based on the benchmark framework of the TTC 2015 Train Benchmark case [8] and supports a generator of change sequences, automated build and execution of solutions as well the visualization of the results using R. The source code and documentation of the benchmark as well as metamodels, reference solutions in ModelJoin and QVT-O, example models and example change sequences are publicly available online at `http://github.com/georghinkel/ttc2017SmartGrids`.

The benchmark consists of the following phases:

1. **Initialization:** In this phase, solutions may load metamodels and other infrastructure independent of the used models as required. Because time measurements are very hard to measure for this phase, the time measurement is optional.

2. **Loading:** The initial model instances are loaded.

3. **Initial:** An initial view is created.

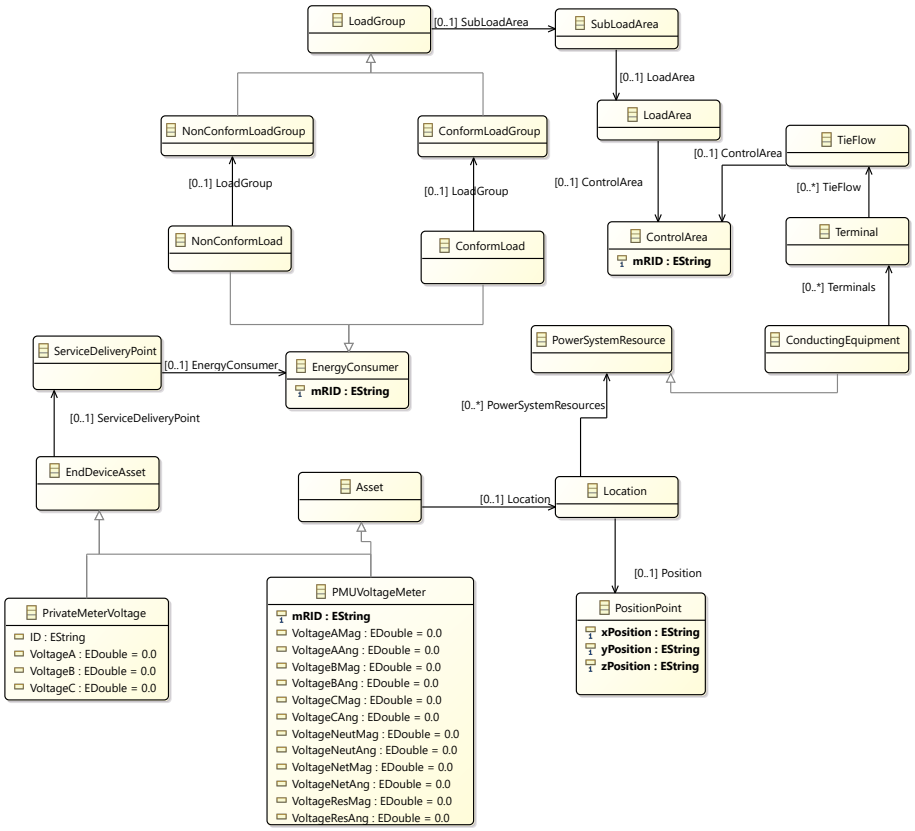


Figure 3: The Voltage Three-Phase Measurement Matrix

4. **Updates:** A sequence of change sequences is applied to the model. Each change sequence consists of several atomic change operations. After each change sequence, the view must be consistent with the changed source models, either by entirely creating the view from scratch or by propagating changes to the view result.

In the following subsections, the change sequences, solution requirements and the benchmark configuration are explained in more detail.

4.1 Change Sequences

To measure the incremental performance of solutions, the benchmark uses generated change sequences. These change sequences are in the `changes` directory of the benchmark resources. Additional change sequences can be generated using a generator contained in the benchmark resources, however the generator is implemented using NMF [9] and thus requires .NET Framework 4.5.1 to be installed.

The changes are available in the form of models. An excerpt of the metamodel is depicted in Figure 4: There are classes for each elementary change operation that distinguish between simple assignments and collection interactions, such as adding or removing single elements or resetting, which means erasing the collection contents. The true metamodel contains concrete classes that distinguish further between the type of feature, whether it is an attribute, association or composition change. In these concrete classes, the added, deleted or assigned items are included[1]. The change metamodel also supports change transactions where a source change implies some other changes, for example setting opposite references.

Unfortunately, the implementation to apply these changes is only available in NMF. Incremental tools are therefore asked to transform the change sequences into their own change representation. To ease the specification of batch solutions, the generator also outputs the models after each change sequence step.

[1]In a composite insertion, the added element is contained, otherwise only referenced.

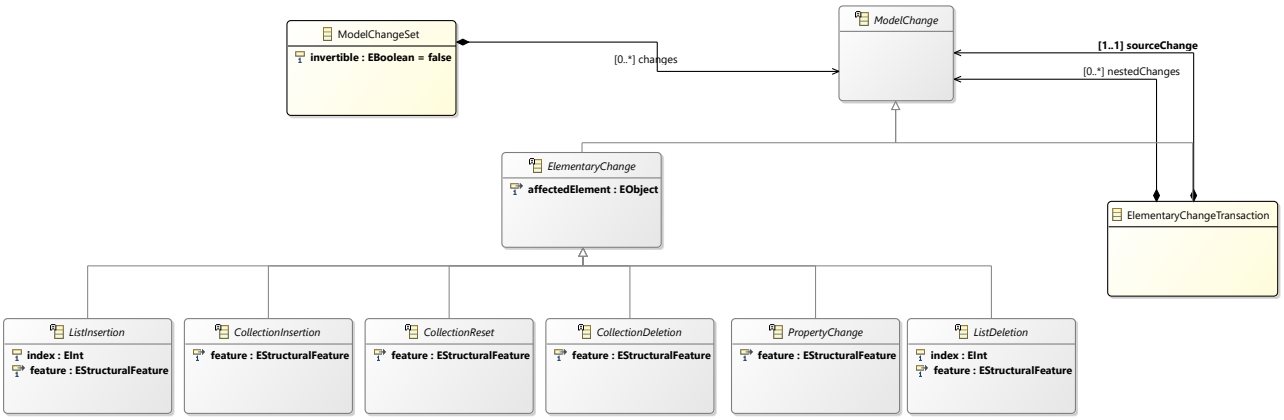


Figure 4: Metamodel of model changes (simplified)

4.2 Solution requirements

The solutions are required to perform the steps of the benchmark and report the following metrics after each step, in case of the update phase after every change sequence.

- **Tool:** The name of the tool.
- **View:** The view that is currently computed
- **ChangeSet:** The name of the change set that is currently run
- **RunIndex:** The run index in case the benchmark is repeated
- **Iteration:** The iteration (only required for the Update phase)
- **PhaseName:** The phase of the benchmark
- **MetricName:** The name of the reported metric
- **MetricValue:** The value of the reported metric

Solutions should report on the runtime of the respective phase in integer nanoseconds (**Time**), the working set in bytes (**Memory**) and the root element count in the created view (**Elements**). The memory measurement is optional. If it is done, it should report on the used memory after the given phase (or iteration of the update phase) is completed. Solutions are allowed to perform a garbage collection before memory measurement that does not have to be taken into account into the times. In the update phase, we are not interested in the time to load models or changes or the perhaps required transformation of changes, but only the pure view update, i.e. either recomputation of the view or propagation of the change.

To enable automatic execution by the benchmark framework, solutions should add a subdirectory to the `solutions` folder of the benchmark with a `solution.ini` file stating how the solution should be built and how it should be run. An example configuration for the ModelJoin solution is depicted in Listing 3. Because the solution contains the already compiled Jar archive, no action is required for build. However, solutions may want to run build tools like maven in this case to ensure the benchmark runs with the latest version.

```
1 [build]
2 default=echo ModelJoin solution is already compiled
3 skipTests=echo The ModelJoin solution is already compiled
4
5 [run]
6 OutageDetection=java -Xmx8G -jar solution.jar -view OutageDetection
7 OutagePrevention=java -Xmx8G -jar solution.jar -view OutagePrevention
```

Listing 3: An example `solution.ini` file

The repetition of executions as defined in the benchmark configuration is done by the benchmark. This means, for 5 runs, the specified command-line for a particular view will be called 5 times. These runs should all have the same prerequisites. In particular, solutions must not save intermediate data between different runs. Meanwhile, all iterations of the *Update* phase are executed in the same process and solutions are allowed (and encouraged) to save any intermediate computation results they like, as long as the results are correct after each change sequence.

The root path of the input models and changes, the run index, the number of iterations and the name of change sequences are passed using environment variables `ChangePath`, `RunIndex`, `Sequences` and `ChangeSet`. To demonstrate the usage of these environment variables, the benchmark framework also contains a demo solution which does nothing but print out a time csv entry using the provided environment variables.

4.3 Running the benchmark

The benchmark framework only requires Python 2.7 or above and R to be installed. R is required to create diagrams for the benchmark results. Furthermore, the solutions may imply additional frameworks. We would ask solution authors to explicitly note dependencies to additional frameworks necessary to run their solutions.

If all prerequisits are fulfilled, the benchmark can be run using Python with the command `python scripts/run.py`. Additional command-line options can be queried using the option `-help`.

```
{
  "Views": [
    "OutageDetection",
    "OutageAvoidance"
  ],
  "Tools": ["ModelJoin"],
  "ChangeSets": [
    "changeSequence1"
  ],
   "Sequences": 100
   "SequenceLength": 10
  "Runs": 5
}
```

Listing 4: The default benchmark configuration

The benchmark framework can be configured using JSON configuration files. The default configuration is depicted in Listing 4. In this configuration, both views are computed, using only the reference solution in ModelJoin, running the change sequence `changeSequence1` contained in the `changes` directory 5 times each. The exact commands created by the benchmark framework are determined using the solution configuration files described below.

To execute the chosen configuration, the benchmark can be run using the command line depicted in Listing 5.

```
&> python scripts/run.py
```

Listing 5: Running the benchmark

Additional commandline parameters are available to only update the measurements, create visualizations or generate new change sequences.

5 Evaluation

Solutions of the proposed benchmark should be evaluated by their completeness, correctness, conciseness, understandability, batch performance and incremental performance.

For each evaluation, a solution can earn 5 points for Task 1 and 5 points for Task 2. In the latter, we explain how the points are awarded for Task 1. The points for Task 2 are awarded equivalently.

5.1 Completeness & Correctness

Assessing the completeness and correctness of model transformations is a difficult task. In the scope of this benchmark, besides manual assessment by opponents, solutions are checked for the correct number of model elements in the result after each change.

Points are awarded according to the following rules:

- **0 points** The task is not solved.

- **1-4 points** The task is solved, but the number of elements in the result is either too high or too low.
- **5 points** The task is completely and correctly solved.

5.2 Conciseness

Detecting and especially forecasting an outage in a smart grid heavily relies on heuristics. Therefore, it is important to specify views in a concise manner.

- **0 points** The task is not solved.
- **1 point** The solution is the least concise.
- **5 points** The solution is the most concise.
- **1-5 points** All solutions in between are classified relative to the most and least concise solution.

To evaluate the conciseness, we ask every solution to note on the lines of code of their solution. This shall include the model views and glue code to actually run the benchmark. Code to convert the change sequence can be excluded. For any graphical part of the specification, we ask to count the lines of code in a HUTN[2] notation of the underlying model.

5.3 Understandability

Similarly to conciseness, it is important for maintainance tasks that the solution is understandable. However, as there is no appropriate metric for understandability available, the assessment of the understandability is done manually. For solutions participating in the contest, this score is collected using questionnaires at the workshop.

5.4 Performance

For the performance, we consider two scenarios: batch performance and incremental performance. For the batch performance, we measure the time the solution requires to create the view for existing models. For the incremental solution, we measure the time for the solution to propagate a given set of changes. Points are awarded according to the following rules:

- **0 points** The task is not solved.
- **1 point** The solution is the slowest.
- **5 points** The solution is the fastest.
- **1-5 points** All solutions in between are classified according to their speed.

The measurements for batch performance and for incremental performance are done separately. This means, solutions are allowed to run in a different configuration when competing for the batch performance than they use in the incremental setting. This is because in an application, usually only one of these aspects is particularly important.

5.5 Overall Evaluation

Due to their importance, the points awarded in completeness & correctness and understandability are doubled in the overall evaluation. Furthermore, due to the importance of incremental updates, we give the incremental performance a double weight.

Thus, each solution may earn up to 80 points in total (40 for each task).

Acknowledgements

We would like to thank Victoria Mittelbach for the permission to use her master thesis results for this case.

[2] `http://www.omg.org/spec/HUTN/`

References

[1] V. Mittelbach, "Model-driven Consistency Preservation in Cyber-Physical Systems," Master's thesis, Karlsruhe Institute of Technology (KIT), Germany.

[2] E. Burger, V. Mittelbach, and A. Koziolek, "Model-driven consistency preservation in cyber-physical systems," in *Proceedings of the 11th Workshop on Models@run.time co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016)*, (Saint Malo, France), CEUR Workshop Proceedings, 2016.

[3] "Iec 61970 energy management system application program interface (ems-api) - part 301 common information model (cim) base," 2011.

[4] "Iec 61850 communication networks and systems for power utility automation," 2015.

[5] D. U. Association, "Excerpt from companion specification for energy metering cosem interface classes and obis identification system," 2014.

[6] E. Burger, J. Henß, M. Küster, S. Kruse, and L. Happe, "View-Based Model-Driven Software Development with ModelJoin," *Software & Systems Modeling*, vol. 15, no. 2, pp. 472–496, 2014.

[7] L. Qianqian, X. Zeng, M. Xue, and L. Xiang, "A new smart distribution grid fault self-healing system based on traveling-wave," in *Industry Applications Society Annual Meeting, 2013 IEEE*, 2013, pp. 1–6.

[8] G. Szárnyas, O. Semeráth, I. Ráth, and D. Varró, "The TTC 2015 train benchmark case for incremental model validation," in *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 24, 2015.*, 2015, pp. 129–141.

[9] G. Hinkel, "NMF: A Modeling Framework for the .NET Platform," Karlsruhe Institute of Technology, Tech. Rep., 2016.

An NMF solution to the Smart Grid Case at the TTC 2017

Georg Hinkel
FZI Research Center of Information Technologies
Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany
hinkel@fzi.de

Abstract

This paper presents a solution to the Smart Grid case at the Transformation Tool Contest (TTC) 2017 using the .NET Modeling Framework (NMF). The goal of this case was to create incremental views of multiple models relevant in the area of smart grids. Our solution uses the incremental model transformation language NMF Synchronizations and the underlying incrementalization system NMF Expressions.

1 Introduction

Models should represent a system in a very abstract form. However, very often, the model is still too complex for humans to understand it or make use of it. Furthermore, necessary information is split among multiple models. Therefore, it is often beneficial for practical applications to reduce the complexity for human modelers through the use of views that combine the information from multiple models and reduce it to those parts of a model that are relevant for a particular task.

The Smart Grid Case of the Transformation Tool Contest (TTC) 2017 [Hin17] proposes a benchmark for such a scenario. Here, the modeled system is a smart grid where the necessary information to detect or predict is split among multiple models according to existing standards. The views originate from a model-based outage management system [Mit, BMK16] implemented using existing model view technology [BHK+14].

If the source model changes, the view has to be adapted to the changed source. For large models, it becomes very slow to recompute the entire model from scratch, in particular, since changes usually only affect small parts of the model. Rather, it is much more efficient to only propagate the changes to the view in an incremental manner. However, implementing such a change propagation manually can be a very laborious task that further conceals the code intention, i.e. the view that is actually being computed.

This paper presents a solution to the proposed benchmark using the incremental model transformation language NMF Synchronizations [HB17], integrated into the .NET Modeling Framework (NMF, [Hin16]). The solution is publicly available on Github[1]. We first give a very brief introduction into synchronization blocks, the formalism underneath NMF Synchronizations in Section 2 before Section 3 presents the solution. Section 4 evaluates the solution against the reference solution in ModelJoin and finally Section 5 concludes the paper.

2 Synchronization Blocks

Synchronization blocks are a formal tool to run model transformations in an incremental (and bidirectional) way [HB17]. They combine a slightly modified notion of lenses [FGM+07] with incrementalization systems. Model properties and methods are considered morphisms between objects of a category that are set-theoretic products of a type (a set of instances) and a global state space Ω.

In: A. Garcia-Dominguez, F. Krikava and G. Hinkel (eds.): Proceedings of the 10th Transformation Tool Contest, Marburg, Germany, 21-07-2017, published at http://ceur-ws.org

[1] `https://github.com/georghinkel/ttc2017smartGrids`

A (well-behaved) in-model lens $l : A \hookrightarrow B$ between types A and B consists of a side-effect free GET morphism $l \nearrow \in Mor(A, B)$ (that does not change the global state) and a morphism $l \searrow \in Mor(A \times B, A)$ called the PUT function that satisfy the following conditions for all $a \in A, b \in B$ and $\omega \in \Omega$:

$$l \searrow (a, l \nearrow (a)) = (a, \omega)$$
$$l \nearrow (l \searrow (a, b, \omega)) = (b, \tilde{\omega}) \quad \text{for some } \tilde{\omega} \in \Omega.$$

The first condition is a direct translation of the original PUTGET law. Meanwhile, the second line is a bit weaker than the original GETPUT because the global state may have changed. In particular, we allow the PUT function to change the global state.

An unidirectional (single-valued) synchronization block $\boldsymbol{S}$ is an octuple $(A, B, C, D, \Phi_{A-C}, \Phi_{B-D}, f, g)$ that declares a synchronization action given a pair $(a, c) \in \Phi_{A-C} : A \cong C$ of corresponding elements in a base isomorphism Φ_{A-C}. For each such tuple in states (ω_L, ω_R), the synchronization block specifies that the elements $(f(a, \omega_L), g \nearrow (b, \omega_R)) \in B \times D$ gained by the function f and the lens g are in the dependent isomorphism Φ_{B-D}.

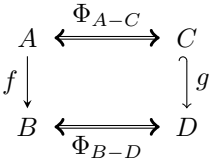


Figure 1: Schematic overview of unidirectional synchronization blocks

A schematic overview of a synchronization block is depicted in Figure 1. The usage of lenses allows these declarations to be enforced automatically[2]. The engine simply computes the value that the right selector should have and enforces it using the PUT operation.

A multi-valued synchronization block is a synchronization block where the lenses f and g are typed with collections of B and D, for example $f : A \hookrightarrow B*$ and $g : C \hookrightarrow D*$ where stars denote Kleene closures.

Synchronization Blocks have been implemented in NMF Synchronizations, an internal language integrated into C# [HB17].

3 Solution

We discuss the solutions to the outage detection and the outage prevention tasks separately in Sections 3.1 and 3.2.

3.1 Outage Detection

In NMF Synchronizations, the support for multiple input pattern elements is rather limited. As a reason, we experienced with NTL [Hin13] that multiple input elements is a rare case, but required a tremendous amount of code to support it. At the same time, the advantages of a true support for multiple input elements over transformation of tuples is limited.

Therefore, the easiest way to support multiple input pattern elements in NMF Synchronizations is to simply use tuples as inputs. Then, the model matching has to be adapted to match tuples instead of elements. Therefore, the main rule synchronizes a tuple of the CIM model and the COSEM model with the resulting view model.

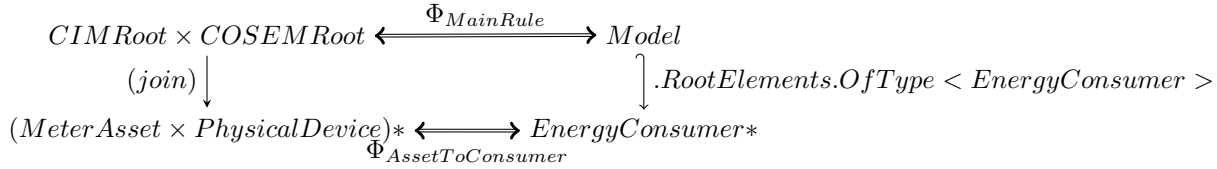


Figure 2: The join in the outage detection task formulated in a synchronization block

In a synchronization block, the main join of meter assets with physical devices is depicted in Figure 2, where we abbreviated the join expression. The implementation of this matching is depicted in Listing 1.

[2]If f was also a lens, then the synchronization block can be enforced in both directions.

```
public class MainRule : SynchronizationRule<Tuple<CIMRoot, COSEMRoot>, Model> {
  public override void DeclareSynchronization() {
    SynchronizeManyLeftToRightOnly(SyncRule<AssetToConsumer>(),
      sg => from pd in sg.Item2.PhysicalDevice
            join ma in sg.Item1.IDobject.OfType<IMeterAsset>()
            on pd.ID equals ma.MRID
            select new Tuple<IMeterAsset, IPhysicalDevice>(ma, pd),
      target => target.RootElements.OfType<IModelElement, OutageDetectionJointarget.IEnergyConsumer>());
  }
}
```

Listing 1: The implementation of the main rule for outage the outage detection task

In particular, the definition of the synchronization block in Listing 1 is implemented in a call to the `SynchronizeManyLeftToRightOnly`. The types and the base isomorphism `MainRule` used in the synchronization block can be inferred from the context and the explicitly specified dependent synchronization rule `AssetToConsumer`.

Because .NET has a hard implementation of generics[3], a type filter can be easily specified by passing generic type arguments. NMF also contains an overload of the `OfType` type filter that accepts two type arguments and keeps the collection interface.

In particular, the incrementalization system NMF Expressions that is used in NMF Synchronizations does support joins available through the query syntax of C#. A second synchronization rule then implements the kept attributes for every such a tuple, as depicted in Listing 2.

```
public class AssetToConsumer : SynchronizationRule<Tuple<IMeterAsset, IPhysicalDevice>, IEnergyConsumer> {
  public override void DeclareSynchronization() {
    SynchronizeLeftToRightOnly(
      asset => Convert.ToInt32(asset.Item2.AutoConnect.Connection), e => e.Reachability);
    SynchronizeLeftToRightOnly(asset => asset.Item2.ElectricityValues.ApparentPowermL1, e => e.PowerA);
    SynchronizeLeftToRightOnly(asset => asset.Item1.ServiceDeliveryPoint.EnergyConsumer.MRID, e => e.ID);
    SynchronizeLeftToRightOnly(
        asset => asset.Item1.ServiceDeliveryPoint.EnergyConsumer is ConformLoad ?
        ((ConformLoad)asset.Item1.ServiceDeliveryPoint.EnergyConsumer)
              .LoadGroup.SubLoadArea.LoadArea.ControlArea.MRID :
        ((NonConformLoad)asset.Item1.ServiceDeliveryPoint.EnergyConsumer)
              .LoadGroup.SubLoadArea.LoadArea.ControlArea.MRID,
      e => e.ControlAreaID);
    SynchronizeLeftToRightOnly(SyncRule<LocationToLocation>(),
      asset => asset.Item1.Location, e => e.Location);
  }
}
```

Listing 2: Implementation of kept attributes and references in the outage detection task

Listing 2 essentially consists of five synchronization block where each synchronization block is responsible for the synchronization of an attribute or reference of the target model. In case no synchronization rule is provided (like for the first four synchronization blocks), implicitly the identity of the inferred type is used. Two further synchronization rules synchronize location and position point.

3.2 Outage Prevention

In the implementation of the outage prevention task, the principle approach to use tuples to synchronize multiple inputs is the very same approach as in the outage detection task.

```
public class MainRule :
    SynchronizationRule<Tuple<CIMRoot, COSEMRoot, Substandard>, Model> {
  public override void DeclareSynchronization() {
    SynchronizeManyLeftToRightOnly(SyncRule<MMXUAssetToVoltageMeter>(),
      dr => dr.Item1.IDobject.OfType<IMeterAsset>()
              .Join(dr.Item3.LN.OfType<IMMXU>(),
                    asset => asset.MRID,
                    mmxu => mmxu.NamePlt.IdNs,
                    (asset, mmxu) => new Tuple<IMeterAsset, IMMXU>(asset, mmxu)),
      model => model.RootElements.OfType<IModelElement, IPMUVoltageMeter>());

    SynchronizeManyLeftToRightOnly(SyncRule<DeviceAssetToPrivateMeterVoltage>(),
      dr => dr.Item1.IDobject.OfType<IEndDeviceAsset>()
              .Join(dr.Item2.PhysicalDevice,
                    asset => asset.MRID,
                    pd => pd.ID,
```

[3]This means that the generic type arguments are still available at runtime.

```
                    (asset, pd) => new Tuple<IEndDeviceAsset, IPhysicalDevice>(asset, pd)),
      model => model.RootElements.OfType<IModelElement, IPrivateMeterVoltage>());
  }
}
```

Listing 3: The implementation of the main rule in the outage prevention task

The implementation of the main rule is depicted in Listing 3. In this listing, we used the alternative method chaining syntax for the join. Both syntaxes are equivalent, as the compiler converts the query syntax into the method chaining syntax.

To handle the different transformation of the various subtypes of a power system resource, we utilize the rule instantiation feature of NMF Synchronizations. With a rule instantiation, the isomorphism represented by a synchronization rule can be refined for a subset of model elements.

```
public class PowerSystemResource2PowerSystemResource
     : SynchronizationRule<IPowerSystemResource, IPowerSystemResource> {
  public override void DeclareSynchronization() {}
}
public class ConductingEquipment2ConductingEquipment
     : SynchronizationRule<IConductingEquipment, IConductingEquipment> {
  public override void DeclareSynchronization() {
    SynchronizeManyLeftToRightOnly(SyncRule<Terminal2Terminal>(),
        conductingEquipment => conductingEquipment.Terminals, equipment => equipment.Terminals);
    MarkInstantiatingFor(SyncRule<PowerSystemResource2PowerSystemResource>());
  }
}
```

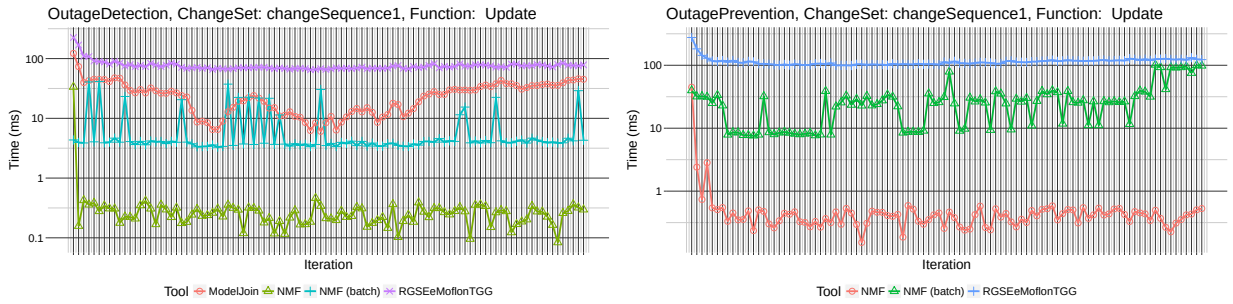
Listing 4: Transforming power system resources

An example of synchronization rule instantiation for conducting equipment is depicted in Listing 4. This means that whenever a power system resource is a conducting equipment, also its terminals are synchronized.

4 Evaluation

Our solution is quite concise as it only consists of 58 lines of code for the outage detection scenario and 195 lines of code for the outage prevention scenario. Both numbers include empty lines as well as lines that only contain braces. Another 140 lines of code actually run the benchmark.

The performance results recorded on an Intel i7-4710MQ clocked at 2.50Ghz in a system with 16GB RAM are depicted in Figure 3 that list the time to update the view model after every iteration, each applying 10 changes. The results are available for the NMF solution both in incremental and in batch mode, the reference solution in ModelJoin and the solution by Peldszus et al. using eMoflon.



(a) Outage Detection Task

(b) Outage Prevention Task

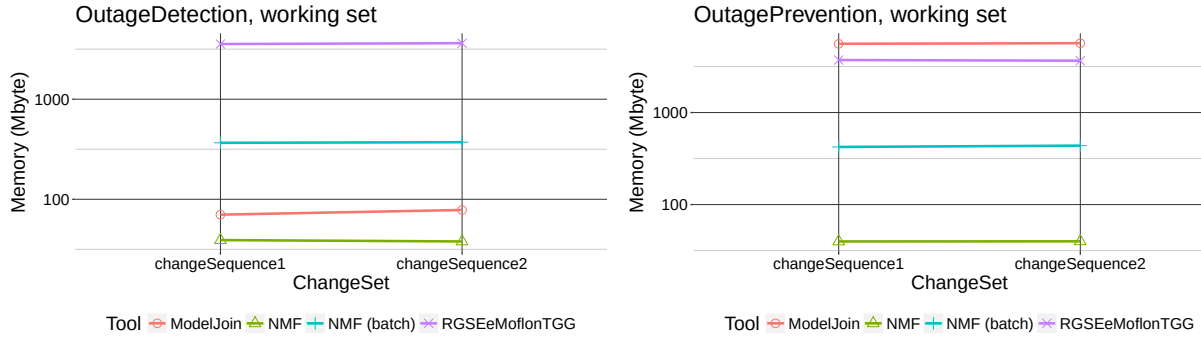
Figure 3: Update times for change sequence 1

For the *Outage Prevention* task, unfortunately the reference solution in ModelJoin ran out of memory and hence, it is not shown on the plot in Figure 3b.

The results indicate that the NMF solution in batch mode is the fastest among the batch implementations. Furthermore, if one switches the execution mode to incremental, then this yields another speedup of roughly more than a magnitude.

The performance curve for the incremental change propagation is more rough than the performance for the batch execution. This is because propagating the change depends much more on the actual changes than

rerunning the view computation on the entire (changed) model. However, interestingly, we see some spikes in the otherwise smooth curve for the batch execution. We think that this is due to garbage collection.



(a) Outage Detection Task

(b) Outage Prevention Task

Figure 4: Memory consumption (Work space size)

Interestingly, we noted that the incremental execution mode of NMF has the least memory consumption compared to the other solutions. Incremental tools usually have a memory overhead but the advantage of the incremental execution here is that only the changes of the models have to be loaded and not all of the entire models in each iteration.

5 Conclusion

In this paper, we presented the NMF solution to the Smart Grid case at the TTC 2017. The solution shows how synchronization blocks, in particular their implementation in NMF Synchronizations can be used to perform incremental view computations. The resulting solution is faster than the reference implementation by multiple orders of magnitude. In particular, the ability of NMF to run the solution incrementally yields a very good performance.

References

[BHK+14] Erik Burger, Jörg Henß, Martin Küster, Steffen Kruse, and Lucia Happe. View-Based Model-Driven Software Development with ModelJoin. *Software & Systems Modeling*, 15(2):472–496, 2014.

[BMK16] Erik Burger, Victoria Mittelbach, and Anne Koziolek. Model-driven consistency preservation in cyber-physical systems. In *Proceedings of the 11th Workshop on Models@run.time*. CEUR Workshop Proceedings, October 2016.

[FGM+07] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3), May 2007.

[HB17] Georg Hinkel and Erik Burger. Change Propagation and Bidirectionality in Internal Transformation DSLs. *Software & Systems Modeling*, 2017.

[Hin13] Georg Hinkel. An approach to maintainable model transformations using an internal DSL. Master's thesis, Karlsruhe Institute of Technology, October 2013.

[Hin16] Georg Hinkel. NMF: A Modeling Framework for the .NET Platform. Technical report, Karlsruhe Institute of Technology, Karlsruhe, 2016.

[Hin17] Georg Hinkel. The TTC 2017 Outage System Case for Incremental Model Views. In Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, editors, *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences*, CEUR Workshop Proceedings. CEUR-WS.org, July 2017.

[Mit] Victoria Mittelbach. Model-driven Consistency Preservation in Cyber-Physical Systems. Master's thesis, Karlsruhe Institute of Technology (KIT), Germany.

Detecting and Preventing Power Outages in a Smart Grid using eMoflon

Sven Peldszus, Jens Bürger, Daniel Strüber
{speldszus,buerger,strueber}@uni-koblenz.de

University of Koblenz and Landau

Abstract

We present a solution to the Outage System Case of the Transformation Tool Contest 2017, based on the bidirectional model transformation language *eMoflon*. The case comprises two tasks, in which the goal is to produce custom model views on a set of input models from a smart-grid system, eventually allowing power outages to be detected and prevented. To facilitate understandability, our solution uses eMoflon's declarative transformation rules. Moreover, to address performance, we use a general-purpose-language preprocessing step contributing to the identification of elements participating in the considered views.

1 Introduction

Model-driven engineering emphasizes the use of models to facilitate the development of software systems. As the involved systems grow in size and complexity, maintaining a single model to represent the overall system becomes infeasible. Complex scenarios usually involve a *multitude* of models to represent distinct concerns of the system and its subsystems, working together in some well-defined way.

When working with such a multitude of models, two issues may arise: First, the information relevant for specific tasks may be scattered over multiple models. To represent this relevant information in an easily digestible way, custom *model views* on a set of models are required. Second, in the case of frequently changing models, it is necessary to update these views after each modification. However, computing all views from scratch each time an underlying model changes can be prohibitively expensive. A solution to this issue are *incremental views* that reuse results from earlier view computations. Since the creation of incremental views is essentially a model transformation problem, it is promising to address it with the available transformation tools.

In this context, the *Outage System Case* [Hin17] of the 2017 Transformation Tool Contest, based on a smart-grid setting, provides an interesting benchmark scenario. To capture information about a power supply system, the benchmark features three kinds of models: The *Common Information Model* (CIM) describes physical components, measurement data, control and protection elements. The *Companion Specification for Energy Metering* (COSEM) model supports data exchange for meter reading, tariff and load control in power metering. The *Substation Configuration Description Language and Logical Node* (SCL/LN) model fosters interoperability of intelligent electronic devices in substation automation systems.

The Outage System Case comprises two tasks that each involve an incremental view on this set of models: (1) an *outage detection* task to identify power outages based on a lack of responsiveness by a smart meter, and (2) an *outage prevention* task based on disturbances that can be observed by comparing the current phasor data

In: Antonio Garcia-Dominguez, Georg Hinkel, Filip Křikava (eds.): Proceedings of the 10th Transformation Tool Contest, Marburg, Germany, 21-07-2017, published at http://ceur-ws.org

to historical ones. Technically, each of the views required for these tasks can be expressed as a model, based on a view-specific meta-model (see Fig. 1a and Fig. 3 in [Hin17]).

In this paper, we present a solution to the Outage System Case based on *eMoflon* [ALPS11], a bidirectional model transformation language and tool based on the paradigm of triple graph grammars (TGGs, [Sch95]). The main use-case of TGGs is the synchronization of two models, called left-hand side and right-hand side model (LHS, RHS). The user specifies a set of *rules*, which map LHS meta-model classes to RHS meta-model classes by using a dedicated *correspondence model*. From these correspondence rules, one can automatically generate transformation code for realizing model synchronization in two directions: the forward direction for propagating changes of the LHS to the RHS model, and the backward direction for propagating changes of the RHS to the LHS model.

In our application of eMoflon to the Outage System Case, we interpret the set of input models as the LHS, and the output view as the RHS. Since the views in our scenario are not required to be editable, we only need the forward direction in order to produce and update the required views. We provide the code for our solution at `https://github.com/SvenPeldszus/rgse.ttc17.emoflon.tgg`.

The rest of this paper is structured as follows: In Sec. 2, we provide an overview of our solution. In Sec. 3, we show details, including example rules. We evaluate our solution in Sec. 4, and give a conclusion in Sec. 5.

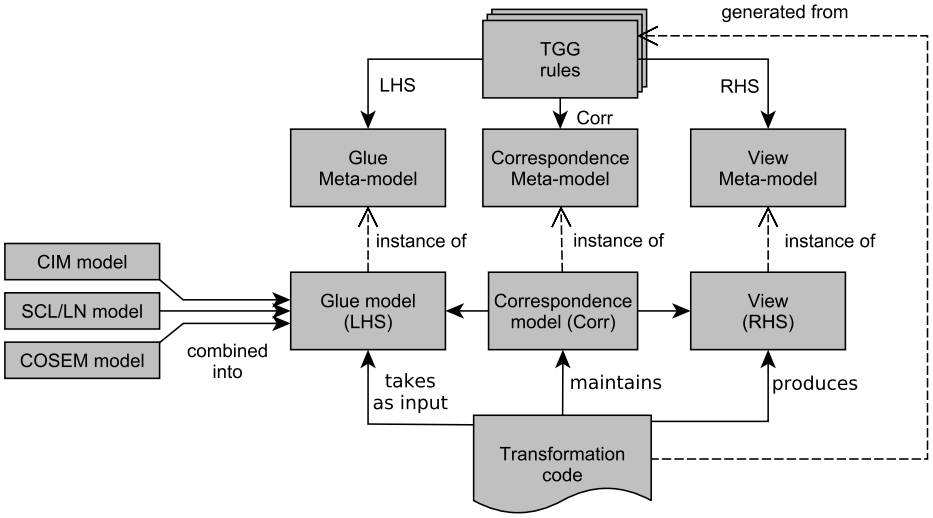


Figure 1: Overview of our view computation.

2 Overview

Fig. 1 shows an overview of the view computation. In eMoflon, the LHS has to be an instance of exactly one meta-model. To accomplish this, we introduce a *glue meta-model* which integrates the three given meta-models and adds a root class with containment references to the root nodes of these meta-models. Based on this meta-model, the three input models can be combined into one model that we call *glue model*, instantiating the glue meta-model.

A set of rules describes the triple graph grammar (TGG) used to construct the model view. These rules are composed of graph patterns, where the contained nodes and edges refer to elements of the involved meta-models, specifically, the glue, correspondence and view meta-models. Consequently, each TGG rule includes a pattern of glue-model elements which is matched against the input glue model. Whenever the glue-model pattern of a rule can be fully matched, the respective model elements of the view and the correspondence model are created. The transformation is realized by running transformation code generated by eMoflon from the TGG rules.

Note that our solution does not support the incremental execution of the transformation; we recompute the views from scratch after each change. In fact, eMoflon supports an incremental execution mode out of the box; however, since our use of the glue meta-model makes some additional preprocessing necessary, the preprocessing needs to be adapted to the incremental mode as well. Doing so is left to future work.

3 Details

In this section, we present the details of our solution. For each task, we first explain the preprocessing step, followed by an explanation of the transformation.

3.1 Task 1: Outage Detection

Preprocessing. The preprocessing step has two goals: First, it unifies the input models into the glue model used as input for the transformation. Second, it enriches this model with auxiliary references and model elements to support a more compact specification and a more efficient matching.

To unify the input models, we introduce a dedicated container element called `Root`. A `Root` holds a CIM model, a SCL/LN model and a COSEM model. In addition, it has some references to enable a more convenient retrieval of involved elements: in particular, all `MeterAsset`s of the CIM model are stored in a reference called `assets`. Finally, it holds a set of `MeterAssetsAndPhysicalDevicesPair`s to identify pairs of `PhysicalDevice`s and `MeterAsset`s with the same ID.

In a purely declarative specification, finding these pairs can be highly inefficient, since the underlying implementation may consider *all* pairs of `PhysicalDevice`s and `MeterAsset`s, filtering them to keep only those with matching IDs. To overcome this, we compute these pairs upfront using a Java method of 15 lines of code, where we consider each `PhysicalDevice` and `MeterAsset` element only once, while maintaining a a hash map. This enables us to identify the required element pairs in linear time.

Transformation. The view produced to solve Task 1 includes three distinct classes: `EnergyConsumer`, `Location`, and `PositionPoint`. Our transformation includes a TGG rule for producing each of them, amounting to three basis TGG rules in total. These basis rules are further extended by *rule refinements* [ASLS14] for dealing with special cases, such as the *if-then-else* in the ModelJoin specification.

To understand eMoflon rules, it is important to consider that they declaratively specify *how the considered models should be related* in the end. They do *not* specify the process of how these relationships are established—instead, this process is realized in the generated forward and backward transformation code.

In Fig. 2 we present an example of a TGG rule and a related refinement. The upper half of Fig. 2 illustrates the `EnergyConsumer` rule. The base rule represents lines 2 to 4 of the ModelJoin source of Task 1. The rule includes green nodes and edges for specifying newly created elements, and black nodes and edges for specifying existing ones. LHS model elements are shown with a yellow background, and RHS elements are shown with a red background. Elements of the correspondence model are indicated as diamonds. Beneath the graph, the rule includes two attribute conditions, specifying equality between pairs of values. Lax equality (`laxEq`) is used to compare a boolean to an integer, based on an underlying Java implementation. Note that the shown graphical representation of rules is a read-only representation generated by eMoflon; the actual rule creation and development was performed using eMoflon's textual syntax.

In a nutshell, this rule specifies the correspondence between a `MeterAsset`–`PhysicalDevice` pair and an `EnergyConsumer`. The forward transformation generated from the rule establishes that for each such pair in the input model, a corresponding `EnergyConsumer` will be generated in the output model. The attribute conditions ensure that the attributes of the newly created `EnergyConsumer` obtain the values according to the specification. Note that `MeterAssetPhysicalDevicePair` was not contained in the original model; as argued above, we added this during the preprocessing to enable a more compact specification and improved performance during matching.

The refinement `EnergyConsumerWithID` shown in the lower half of Fig. 2 deals with the following fact: For the `ServiceDeliveryPoint` of the considered `MeterAsset`, an existing `EnergyConsumer` may or may not exist. If it exists, it needs to be treated according to the view specification. We implement this distinction using rule refinement. The underlying semantics is to *glue together* the refinement with all of its super-rules, based on name equalities of the included elements. In the example, this allows us to apply the additional attribute condition to the target model, enabling us to propagate the LHS `EnergyConsumer`'s `mRID` to the RHS `EnergyConsumer`'s `ID` field, in case the former exists.

3.2 Task 2: Outage Prevention

Preprocessing. The preprocessing for task 2 follows the same basic ideas as for Task 1. We unify the input models, populate certain convenience references and add auxiliary model elements to facilitate a compact specification and optimized matching. The custom parts of interest in the glue meta-model is a reference `mmxus`

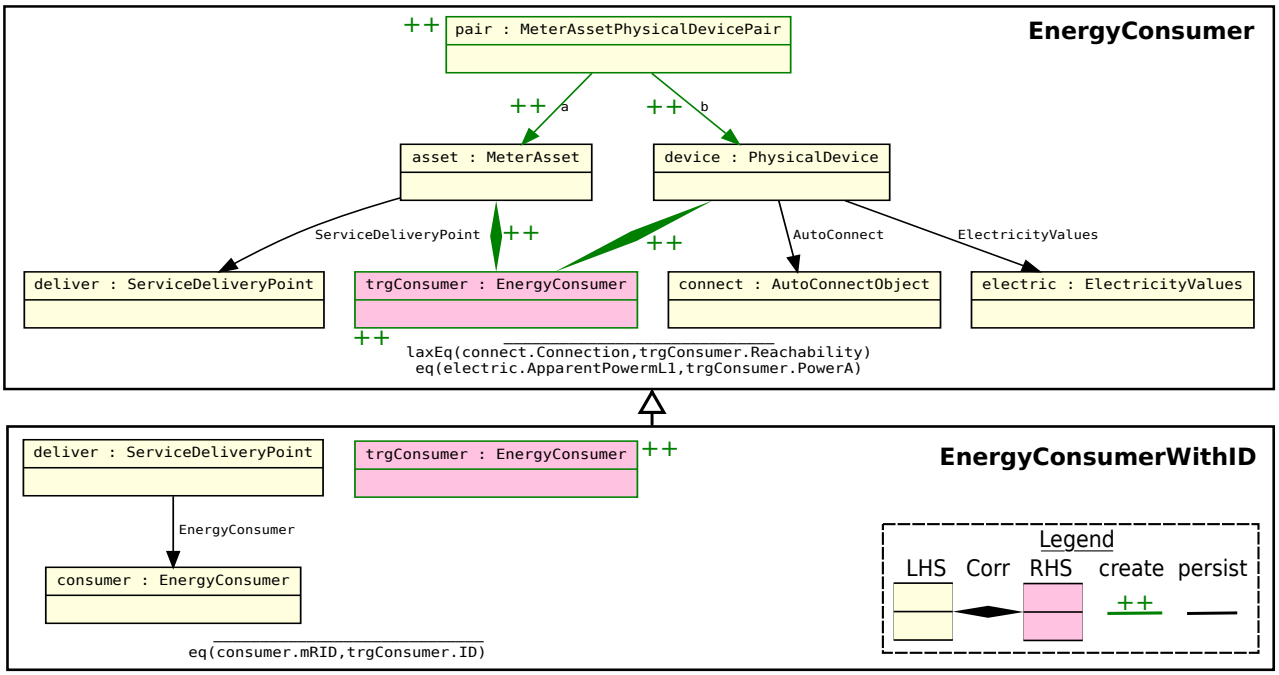


Figure 2: Example of a TGG rule and its refinement to compute the view in Task 1.

keeping track of the `MMXU` instances in the `SCL/LN` model, and a `MeterAssetMMXUPair` class maintaining pairs of `MeterAsset`s and `MMXU`s with the same ID. The purpose of these parts is the same as described for Task 1.

Transformation. Similar to Task 1, we organize the overall TGG rule specification into rules for specific classes from the view meta-model and rule refinements for dealing with special cases thereof. This time, more different elements types are involved and there are fewer special cases per element type: We distinguish 19 rules in total, only three of them needing refinements.

Fig. 3 illustrates two of these rules. The `Location` rule specifies the correspondence of `Location` elements in the LHS and RHS models, which are related to an existing `MeterAsset` and `PMUVoltageMeter`, respectively. To comprehend why we need to specify the related `MeterAssetMMXUPair` as well, we first give some details on how the matching process works.

In essence, the LHS is matched as a first step and after that, the RHS is matched. In TGGs, it is assumed that for each LHS match, we can identify a corresponding RHS match. However, in our particular scenario, the issue is that we do not transform all of the `MeterAssets`, but only some of them, strictly speaking those with a corresponding `MMXU`. Therefore, specifying the `MeterAssetMMXUPair` (which, again, is produced during preprocessing) saves us from matching `MeterAsset`s with no RHS counterpart, which would lead to an error.

Based on the correspondence manifested in the `Location` rule, the `Position` rule specifies the correspondence of `PositionPoint` classes in the LHS and RHS. Again, we use attribute conditions to specify the equality of involved attributes, which will be used during the forward transformation to propagate the values from the LHS element to the RHS counterpart. In the overall forward transformation process, all `Location`s will be transformed first, which enables all `PositionPoint`s to be transformed subsequently.

4 Evaluation

In this section, we evaluate two aspects of our solution: conciseness and performance.

According to the specification, conciseness is to be evaluated in terms of the number of Lines of Code (LoC). The measurement shall include "the model views and glue code to actually run the benchmark", while the code for converting the change sequences can be ignored. We present the results in Table 1a. Since our solution comprises declarative rule specification as well as some imperative orchestration code, we list both separately, amounting to 1312 LoC for rules and 364 for code. The glue code for running the benchmark comprises 119 LoC.

For the performance evaluation, we show the execution times for both tasks when applied to the input change sequences. All experiments have been performed on an Ubuntu 16.04 LTS PC with an Intel i5-6200U and 8 GB of RAM of which our implementation was allowed to allocate up to 6 GB for the experiments.

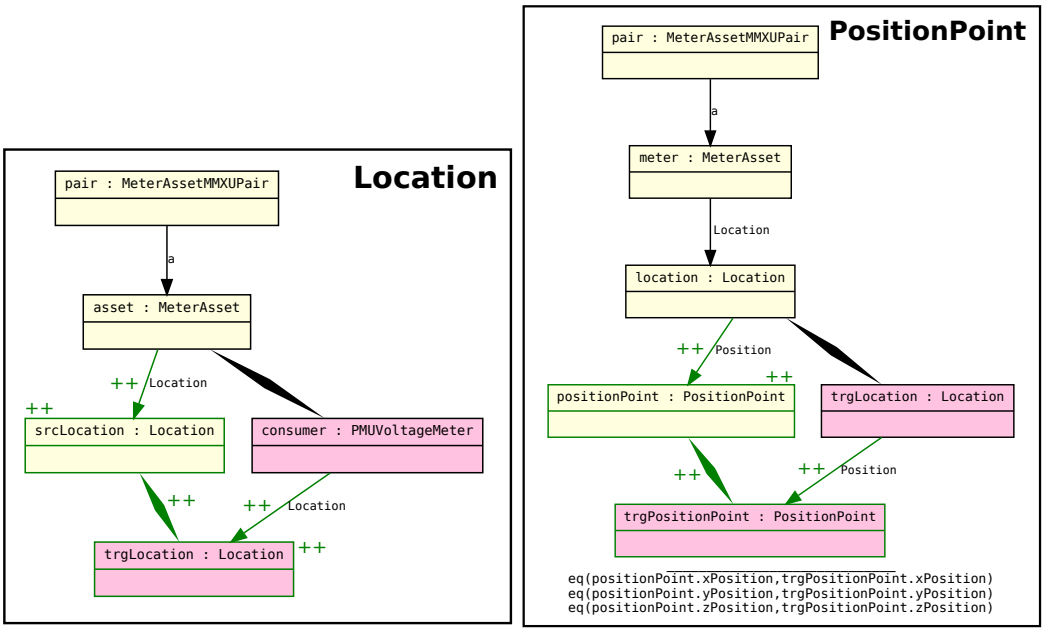


Figure 3: Two related TGG rules for producing the view in Task 2 (Legend: see Fig. 2).

Part	**LoC**
Rules	1312
Orchestration code	364
Benchmark code	119
Total	1795

(a) Conciseness.

Input	**Mode**	**Time[ms]** det/prev	**Input**	**Mode**	**Time[ms]** det/prev
seq1.out000	batch	847.9/1139.2	seq2.out000	batch	825.9/975.0
seq1.delta001	batch	379.7/497.8	seq2.delta001	batch	350.2/504.9
seq1.delta002	batch	387.9/598.2	seq2.delta002	batch	483.3/477.1
seq1.delta003	batch	321.5/308.2	seq2.delta003	batch	281.4/312.2
seq1.delta004	batch	246.1/477.5	seq2.delta004	batch	256.7/329.1
seq1.delta005	batch	210.8/350.7	seq2.delta005	batch	217.1/235.5

(b) Performance (task 1) (c) Performance (task 2)

Table 1: Results.

5 Conclusion

We presented a solution to the TTC 2017 Outage System Case based on eMoflon. Following a simple preprocessing step in Java, we facilitated a compact declarative specification as well as an efficient execution of our solution. We believe that the overall solution combines *the best of both worlds* by providing a mostly declarative specification, while using imperative code for performance-critical parts of the rule application process.

References

[ALPS11] Anthony Anjorin, Marius Lauder, Sven Patzina, and Andy Schürr. eMoflon: Leveraging EMF and Professional CASE Tools. *Informatik*, 192:281, 2011.

[ASLS14] Anthony Anjorin, Karsten Saller, Malte Lochau, and Andy Schürr. Modularizing triple graph grammars using rule refinement. In *International Conference on Fundamental Approaches to Software Engineering*, pages 340–354. Springer, 2014.

[Hin17] Georg Hinkel. The Outage System Case for Incremental Model Views. *10th Transformation Tool Contest (TTC 2017)*, 2017.

[Sch95] Andy Schürr. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1995.

Part II.

Families to Persons Case

The Families to Persons Case

Anthony Anjorin
Paderborn University, Germany
anthony.anjorin@upb.de

Thomas Buchmann
Univ. of Bayreuth, Germany
thomas.buchmann
@uni-bayreuth.de

Bernhard Westfechtel
Univ. of Bayreuth, Germany
bernhard.westfechtel
@uni-bayreuth.de

Abstract

The Families to Persons case is a well-known example problem for bidirectional transformations. This paper proposes an implementation of this case within the recently developed Benchmarx framework [2], based on previous conceptual work [1].

1 Introduction

Bidirectional transformations (bx) are transformations which may be executed from source to target and vice versa. They are required in a variety of scenarios, including bidirectional data converters, round-trip engineering, or view updates [5].

A wide variety of bx languages and tools have been developed which differ with respect to underlying data models, supported scenarios, incremental behaviour, etc. To compare these approaches, the need for benchmarks has been recognized for long [5]. In response to this need, a conceptual framework for bx benchmarks was proposed in [1].

Only recently, however, the conceptual proposal was materialized into an implemented infrastructure called Benchmarx [2]. The tool transformation case described below exploits this framework to provide for an implementation of a well-known bx case dealing with the transformation between heterogeneous databases. One of these maintains a collection of families and their members; its opposite contains a flat collection of persons.

Section 2 describes the Families to Persons case. Section 3 provides a survey of the tool architectures supported by the Benchmarx framework. Section 4 explains the test cases against which the solutions are executed. Section 5 sketches the steps to be performed for implementing the benchmark in a specific tool. Section 6 is devoted to the evaluation of the proposed solutions. Section 7 concludes the paper.

2 Case Description

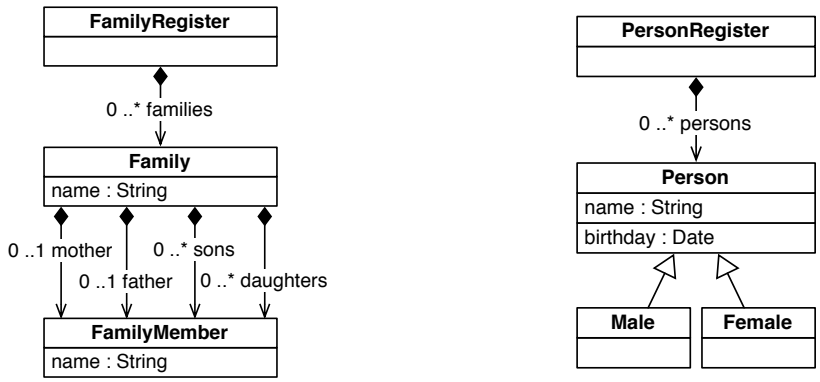
The Families to Persons case is part of the ATL[1] transformation zoo[2] and was created as part of the "Usine Logicielle" project. The original authors are Freddy Allilaire, Frédéric Joault, and Jean Bézivin, all members of the ATLAS research team at the time it was created (December 2006) and published (2007). The variant proposed for the TTC is described below.

We selected this case for several reasons: (1) The underlying data structures are rather simple. Thus, the case can be implemented e.g. in tools operating on trees rather than on graphs. (2) The case is rather small and simple to understand. (3) Furthermore, it may be implemented with acceptable effort. (4) Finally, it demonstrates a number of problems typically occurring in the transformation between heterogeneous data structures (information loss, flattening of hierarchies, different representation of the same information, etc.).

In: A. Garcia-Dominguez, F. Krikava and G. Hinkel (eds.): Proceedings of the 10th Transformation Tool Contest, Marburg, Germany, 21-07-2017, published at http://ceur-ws.org

[1] The Atlas Transformation Language: `https://www.eclipse.org/atl/`

[2] Available from `https://www.eclipse.org/atl/atlTransformations/#Families2Persons`



(a) Families metamodel

(b) Persons metamodel

Figure 1: Source and target data structures

Although solutions to the case need not be implemented in model transformation tools and do not have to live in a specific technological space such as EMF, we assume EMF models in the following explanation. Thus, the data structures to be manipulated are defined by Ecore models in Figure 1.

We assume a unique root in each model. A family register stores a collection of families. Each family has members who are distinguished by their roles. The metamodel allows for at most one father and at most one mother as well as an arbitrary number of daughters and sons. A person register maintains a flat collections of persons who are either male or female. Note that key properties may be assumed in neither model: There may be multiple families with the same name, name clashes are even allowed within a single family, and there may be multiple persons with the same name and birthday. Furthermore, all collections are assumed to be unordered.

A families model is consistent with a persons model if a bijective mapping between family members and persons can be established such that (i) mothers and daughters (fathers and sons) are paired with females (males), and (ii) the name of every person p is "$f.name$, $m.name$", where m is the member (in family f) paired with p.

After running a transformation in any direction, it is required that the participating models are consistent according the definition given above. However, this requirement does not suffice to define the functionality of transformations in a unique way. Below, we first consider batch transformations, where the target model is created from scratch.

The functionality of a forward transformation is straightforward: Map each family member to a person of the same gender and compose the person's name from the surname and the first name; the birthday remains unset. The backward transformation is more involved: A person may be mapped either to a parent or a child, and persons may be grouped into families in different ways.

To reduce non-determinism, we introduce two boolean parameters controlling the backward transformation, resulting in a configurable backward transformation: PREFER_CREATING_PARENT_TO_CHILD controls whether a person is mapped to a parent or a child. PREFER_EXISTING_FAMILY_TO_NEW controls whether a person is added to an existing family (if available), or a new family is created along with a single family member. If both parameters are set to true, the second parameter takes precedence: If a family is available with a matching surname, but there is no matching family with an unoccupied parent role, the member is inserted into an existing family as a child. Please note that the configuration parameters do not completely resolve non-determinism in the presence of multiple matching candidate families if PREFER_EXISTING_FAMILY_TO_NEW is set to true.

In this scenario, information loss occurs in both directions: In forward direction, the distinction between parents and children is lost as well as the aggregation into families. In backward direction, birthdays are lost. Altogether, this case constitutes an example of a round-trip engineering scenario, where both models may be edited and changes have to be propagated back and forth.

For incremental transformations, updates such as insertions, deletions, changes of attribute values, and move operations have to be considered. In forward direction, insertion of a family has no effect on the target model. Insertion of a member results in insertion of a corresponding person; likewise for deletions. If a family is deleted, all persons corresponding to its members are deleted. If a member is renamed, the corresponding person is renamed accordingly. If a family is renamed, all persons corresponding to family members are renamed. If a member is moved, different cases have to be distinguished. If the gender is retained, the corresponding person object is preserved; otherwise, it is deleted, and a new person object with a different gender is created whose

attributes are copied from the old person object. A local move within a family does not affect the corresponding person's name; a move to another family results in a potential update of the person's name.

In backward direction, the effect of an update depends on the values of the configuration parameters, which may be changed dynamically. Please note that the parameter settings must not affect already established correspondences; rather, they apply only to future updates. Deletion of a person propagates to the corresponding family member. If a person is inserted, it depends on the configuration parameters how insertion propagates to the families model (see above). Persons cannot be moved because the persons model consists of a single, flat, and unordered collection. Changes of birthdays do not propagate to the opposite model. If the first name of a person is changed, the first name of the corresponding family member is updated accordingly. Finally, if the surname of a person is changed, this change does not affect the current family and its members: The family preserves its name even if it does not contain other members; thus, the update has no side effects on the existing family. Rather, the corresponding family member is moved to another family, which may have to be created before the move; the precise update behaviour depends on the parameter settings.

3 Tool Architectures

The Benchmarx infrastructure has been designed such that it supports a wide range of tool architectures. To this end, tools are considered as black boxes with respect to the data structures maintained by the tools. As mentioned earlier, the data structures to be synchronized may be represented in a tool-specific way; EMF models merely serve as an example in this paper. Furthermore, the tools may be based on different architectures which are sketched below (see [2] for a more comprehensive description).

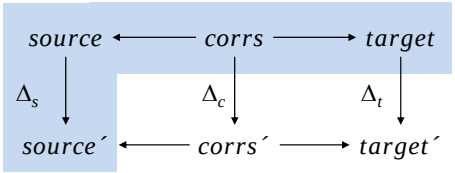


Figure 2: Input data for update propagation

The shaded region in Figure 2 comprises the input data that may be used for update propagation. Tools may rely on correspondences between source and target elements (horizontal arrows). Furthermore, they may assume deltas between old and new states (vertical arrows). A delta may be operational (o-delta, a sequence of operations from the old to the new state) or structural (s-delta, defined in terms of differing elements). The output produced by a tool comprises at least a new target state, as well as optional correspondences and deltas.

In [2], we identified seven tool architectures which differ with respect to their input and output data. A batch tool takes the new source and produces a new target from scratch. All other types of tools operate incrementally inasmuch as they update an already existing target. A state-based tool takes the new source and the old target and produces an updated target. A corr-based tool relies on stored correspondences to improve update behaviour; in addition to an updated target, it produces updated correspondences. An s-delta-based tool takes the old source, the old target, and an s-delta between the old and the new source state, and calculates an s-delta between the old and the new target state (and thus implicitly an updated target). An s-delta/corr-based tool additionally relies on stored correspondences, which are updated along with the calculation of the s-delta on the target. Similarly, a distinction is made between o-delta-based and o-delta/corr-based tools.

Selection of a tool architecture constitutes a classical engineering trade-off. On one hand, the preciseness of update propagation grows with the amount of available input data. For example, stored correspondences may be used to improve update propagation when correspondences cannot be inferred from source and target alone (e.g., when there are two family members with the same full name). On the other hand, a tool relying on input data such as deltas and correspondences may be used only in contexts where these data are available. For example, a tool relying on stored correspondences may not be applicable if source and target have been created independently. In this sense, there is no optimal tool architecture.

Figure 3 depicts a feature model for bx tool architectures used to classify the different bx approaches. The nodes of the tree are the "features" that a given bx tool architecture can possess. Features can be optional or mandatory, children features can only be chosen together with their parent feature, and children of the same parent can be either exclusive or non-exclusive alternatives. Features are abstract (grey fill) if they can be

implied from the set of concrete features (white fill) in a given product. The feature model depicted in Fig. 3 yields exactly the bx tool architectures described above in terms of involved computation steps.

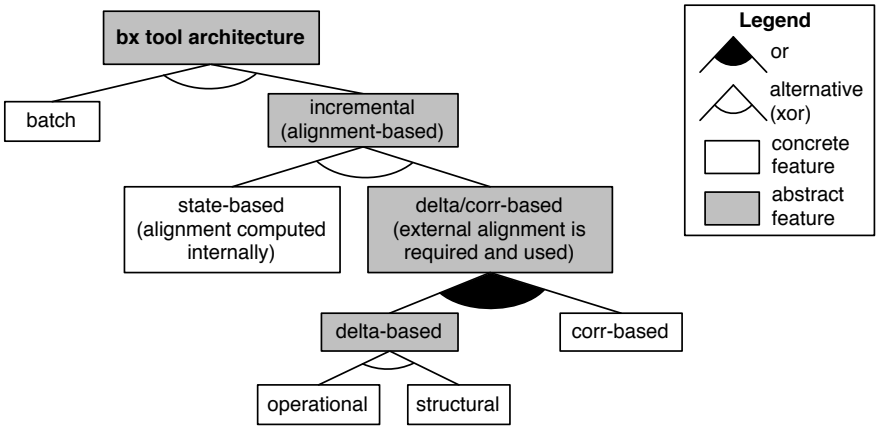


Figure 3: Bx tool architecture variability as a feature model

4 Test Cases

As an extension of the feature model for bx tool architectures, Fig. 4 depicts a feature model for benchmarx test cases. Every benchmarx test case must state the required bx tool architecture (cf. Fig. 3), its direction to be fwd (forward), bwd (backward), or both (which means round-trip), the combination of different change types applied in the test, and the required update policy to successfully pass the test. The set of possible change types, currently del (deletion), add (addition), move, and attribute (attribute manipulation) can be extended in the future to accommodate more expressive frameworks. Note that a test case can require an update policy that is a mixture of fixed, i.e., design time preferences and conventions, and runtime configuration (in the Families to Persons example, only the backward direction needs configuration parameters).

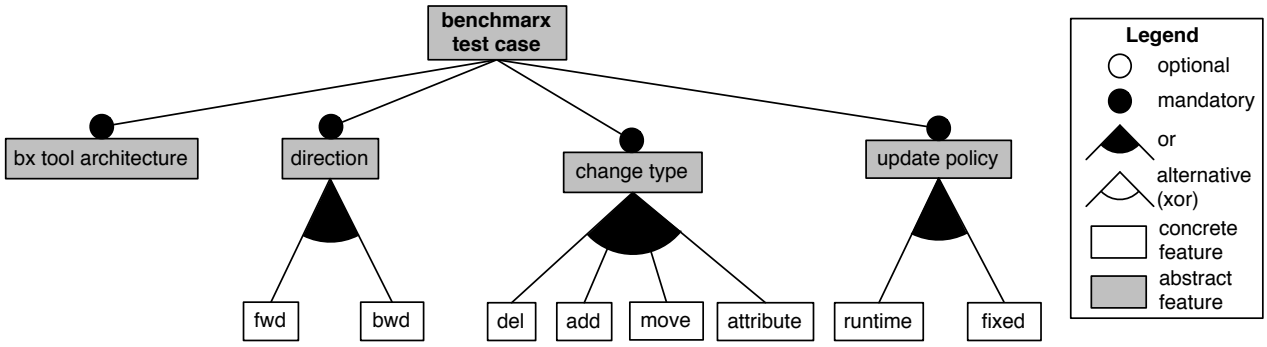


Figure 4: Test case variability as a feature model

The test cases provided for the Families to Persons case are separated into two different categories: (1) batch test cases and (2) incremental test cases. Each category comprises test cases for each transformation direction. Please note that in the following the term *forward* is used for transformations from the Families model to the Persons model. Accordingly, a *backward* transformation describes a transformation from the Persons model to the Families model.

While the batch test cases are used to check if a given source model is transformed into a target model correctly, incremental ones also take updates of the corresponding source models into account. This comprises in particular, renaming, deleting and moving family members or persons respectively. Please note that we tried to keep the number of test cases manageable. To this end, only the batch category contains separate test cases for each combination of configuration parameters in backward transformations. In the incremental category, the parameters are changed dynamically within a backward test case.

While the behaviour is clear for almost all test cases, testIncrementalRenamingDynamic needs some clarification: In the precondition for this test case, several persons are created, who are then transformed into corresponding families and members in the family model. In an edit delta, a person is renamed (full name is changed), such that the surname matches another existing family. The subsequent propagation to the Families model with parameter set to PREFER_EXISTING_FAMILY_TO_NEW should then move the existing family member to the corresponding family and change the first name accordingly. Please note that afterwards an empty family remains in the family register in case the old family only had one member (c.f., test case testIncrementalRenamingDynamic in class IncrementalBackward).

5 Implementation

5.1 Test Case design

Since we strive to provide a generic framework for benchmarking BX tools, and not all of the tools provide access to internal data structures like correspondence models or output deltas of a synchronization run, we decided to design each test case as a *synchronisation dialogue*, always starting from the same agreed upon consistent state, from which a sequence of deltas are propagated. Only the resulting output models are to be directly asserted by comparing them with expected versions.

A benchmarx test case is depicted schematically to the left of Fig. 5, with a concrete test case for our example to the right, following the proposed structure and representing an instantiation with JavaDoc, Java, and JUnit.

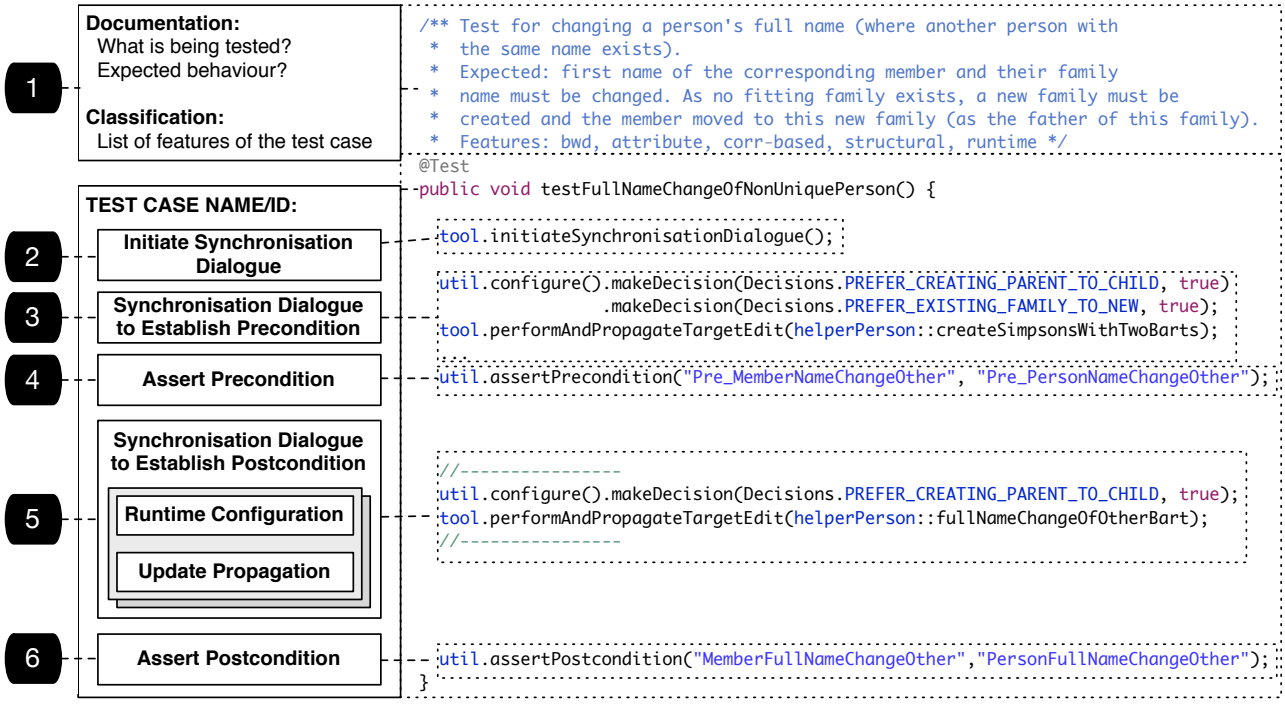


Figure 5: A benchmarx test case as a synchronisation dialogue

Every test case should be documented (cf. Label 1 in Figure 5) stating (i) what is being tested, (ii) expected behaviour, and (iii) a list of the concrete features of the test case taken from Fig. 4 to clarify at a glance if a given bx tool can pass the test or not.

Every test starts with an initialisation command invoked on the bx tool under test (Label 2), giving it the chance to establish the agreed upon starting point (e.g., for the Families to Persons benchmarx this comprises a single empty family register and corresponding single empty person register), and create all necessary internal auxiliary data structures.

The next part of a test case (Label 3) is a series of propagation steps, used to establish the precondition of the test. Although this creates a dependency to other tests (asserting exactly this precondition), this simplifies handling the required correspondence model, as the bx tool can build up the necessary internal state that must go along with the precondition. This means that the old consistent corr is “passed” implicitly to the bx tool via a

series of preparatory propagation steps. The precondition is finally asserted (Label 4), representing a well-defined starting point. If the test requires a runtime update policy, this is configured just before propagating the actual input delta (Label 5). The last part of a test case (Label 6) is an assertion of the postcondition, checking if the final source and target models are as expected.

In the concrete exemplary test case, a number of persons are created in the person register and then backward propagated to establish a consistent family register that is asserted as a precondition. As part of the actual test, a person named `Simpson, Bart` is now renamed in the person register; this change is backward propagated with the update policy to prefer creating parents (if possible) to creating children. The interested reader is referred to the actual test cases[3] for all further details.

5.2 Supplied Test Cases

In its current state, the Benchmarx Framework provides in total 34 pre-defined test cases (see documentation on Github) for the Families to Persons example. They cover both batch and incremental scenarios for each transformation direction. As stated above, incremental backward tests change the parameter configuration at runtime during test case execution to avoid specifying a fixed test scenario for each possible parameter combination. In the batch cases, empty and non-empty families, families with duplicate names and families with duplicate member names are transformed in forward direction. The backward direction for the batch mode checks for each parameter combination the transformation of male or female persons and how duplicate names are handled. The incremental test cases take deletions, insertions, attribute changes (e.g., renaming), moving and a combination of deleting and inserting into account. However, if some cases are missing they may be supplied following the schema of the sample test case as discussed in the previous section.

5.3 Implementing the Test Runner

In order to integrate a specific BX tool into our Benchmarx Framework, the interface BXTool needs to be implemented. Please note, although the Benchmarx Framework itself is written in Java, it is possible to also use it with non-JVM-based BX tools (c.f., our reference implementation for the tool BiGUL [6]). Fig. 6 gives an overview and explanation of the methods to be implemented.

Reference implementations for *eMoflon* [7], *BiGUL* [6], *medini QVT*[4] and *BXtend* [4] may be found in the Github repository. Please note that for EMF-based tools, an abstract class BXToolForEMF already exists where tool integrators may subclass from. This class already contains implementations for both assert methods. As stated above, the method initiateSynchronisationDialogue is invoked in the beginning of every test and is used to establish a common starting point, including a single empty family register and its corresponding single and empty persons register as well as all necessary and tool-specific internal data structures. The methods performAndPropagateTargetEdit and performAndPropagateSourceEdit are called from the test cases when corresponding edit deltas should be performed and propagated on the corresponding models. Contrastingly, the methods performIdleTargetEdit and performIdleSourceEdit are used to modify source and target models respectively, without propagating the change. This method should be used whenever a change in the respective model does not affect the opposite model. In the test cases these methods are used to create empty families or edit the birthdates of persons in the person register.

Please note, that we use parameterized test cases, where the BXTools are the parameters. In order to run the test cases for a single tool, the respective test runner has to be instantiated in the corresponding collection in class FamiliesToPersonsTestCase.

6 Evaluation

The goal of our benchmark is a qualitative evaluation of bx tools and the design of the test cases aims at revealing weaknesses or limitations of the chosen approaches. Due to the heterogeneity of bx tools, it is important to be able to easily and quickly distinguish between:

- A test that fails because it requires features that the tool does not support, is referred to as a *limitation*.
- A test that fails even though the tool should pass it (based on its classification), is referred to as a *failure*.

[3] Available from `https://github.com/eMoflon/benchmarx`
[4] http://projects.ikv.de/qvt

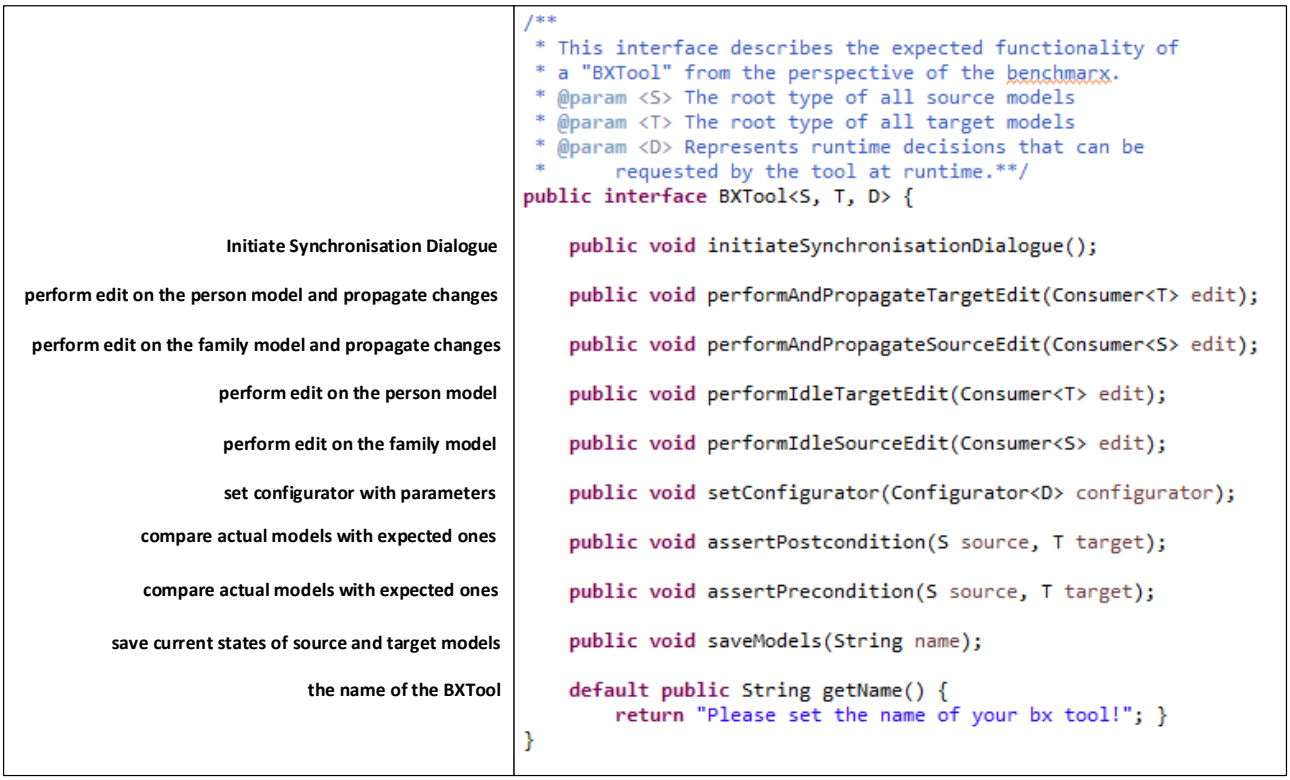


Figure 6: The BXTool interface

Limitations confirm the tool's set of (un)supported features, while failures indicate potential bugs in (the implementation of the benchmarx with) the bx tool. Similarly, one should clearly distinguish between:

- A test that the tool should (based on its classification) pass and passes, is referred to as an *expected pass*.
- A test that the tool passes even though this is unexpected in general, is referred to as an *unexpected pass*.

Expected passes confirm that the tool is correctly classified and behaves as expected, while unexpected passes indicate that the test case can either be improved, as it is unable to reveal the missing features of the tool, or that the bx tool has not been classified correctly.

Scalability of the transformation approaches is evaluated in simple transformations carried out on models while the number of model elements is being raised constantly. Our benchmark measures the time needed for both batch and incremental transformations in forward and backward direction.

Furthermore, we are interested in the metrics `[Lines of Code, Number of Words, Number of Characters]` of the transformation specifications. Those numbers should also be supplied in the spreadsheet[5].

Submissions to the Families to Persons case for the TTC should be made by cloning the provided SHARE VM [3] and contain the following:

- An implementation of the transformation including the test runner (c.f., Section 5.3).
- A spreadsheet containing the numbers retrieved from running the supplied test cases.
- The metrics of the transformation specification (see above).
- A plot similar to the ones shown in [2] based on the numbers retrieved from the performance test.

7 Conclusion

In this paper, we propose a well-known example for bidirectional transformations as a case for the transformation tool contest. We propose to use the benchmarx [2] framework as a test environment to evaluate the solutions. The need for a benchmark for bidirectional transformations has been recognised for long in the bx community. The

[5]not possible for graphical approaches

benchmarx presented in this paper is targeted at classifying and evaluating heterogeneous bx tools and approaches using a common transformation scenario. The propsed transformation tool case helps us to disseminate, evaluate, and improve the benchmarx infrastructure and provides a rich overview about existing bx tools and their benefits and drawbacks on the other hand. Several reference implementations for the transformation case created with *eMoflon*, *BiGUL*, *medini QVT* and *BXtend* are also available in the repository[6].

References

[1] Anthony Anjorin, Alcino Cunha, Holger Giese, Frank Hermann, Arend Rensink, and Andy Schürr. Benchmarx. In K. Selccuk Candan, Sihem Amer-Yahia, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014)*, CEUR Workshop Proceedings, pages 82–86. CEUR-WS.org, 2014.

[2] Anthony Anjorin, Zinovy Diskin, Frédéric Jouault, Hsiang-Shang Ko, Erhan Leblebici, and Bernhard Westfechtel. Benchmarx reloaded: A practical framework for bidirectional transformations. In Romina Eramo and Michael Johnson, editors, *Sixth International Workshop on Bidirectional Transformations (BX 2017)*, CEUR Workshop Proceedings, April 2017.

[3] anthony.anjorin@upb.de. Online demo: Benchmarx. `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu12LTS_BenchmarX.vdi`, 2017.

[4] Thomas Buchmann and Sandra Greiner. Handcrafting a triple graph transformation system to realize round-trip engineering between UML class models and java source code. In Leszek A. Maciaszek, Jorge S. Cardoso, André Ludwig, Marten van Sinderen, and Enrique Cabello, editors, *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 2: ICSOFT-PT, Lisbon, Portugal, July 24 - 26, 2016.*, pages 27–38. SciTePress, 2016.

[5] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In Richard F. Paige, editor, *Proceedings of the Second International Conference on Theory and Practice of Model Transformations (ICMT 2009)*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283, Zurich, Switzerland, June 2009. Springer-Verlag.

[6] Hsiang-shang Ko, Tao Zan, and Zhenjiang Hu. BiGUL: a formally verified core language for putback-based bidirectional programming. *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation - PEPM 2016*, pages 61–72, 2016.

[7] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. Developing emoflon with emoflon. In Davide Di Ruscio and Dániel Varró, editors, *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings*, volume 8568 of *Lecture Notes in Computer Science*, pages 138–145. Springer, 2014.

[6] `https://github.com/eMoflon/benchmarx`

An NMF solution to the Families to Persons case at the TTC 2017

Georg Hinkel
FZI Research Center of Information Technologies
Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany
hinkel@fzi.de

Abstract

This paper presents a solution to the Families to Persons case at the Transformation Tool Contest (TTC) 2017 using the .NET Modeling Framework (NMF). The goal of this case was to bidirectionally synchronize a simple model of family relationships with a simple person register. We propose a solution based on the bidirectional and incremental model transformation language NMF Synchronizations.

1 Introduction

According to the general model theory of Stachowiak [Sta73], models always have a purpose. In many practical applications, one tries to reuse existing metamodels, usually because valuable instrastructure is built on top of it. However, because the different purposes require different abstractions, the information concerning an entity is often split among multiple of these models, which makes a pure transformation approach infeasible as neither source nor target model can be fully reconstructed from the respective other model. Instead, the models must be synchronized to make sure that they are consistent with regard to some correspondence rules.

The Families to Persons case at the Transformation Tool Contest (TTC) 2017[ABW17] demonstrates this problem in the scenario of a well known example model transformation, the Families to Persons transformation, and provides a benchmark framework to compare solutions in terms of abilities and performance. Here, a structured model of family relations should correspond to a flat model of persons where the information of a persons family is encoded only through that person's full name. Further, the information of the gender of a person is encoded differently: While the Families model encodes this information through the position of a model element, the same information is represented through subtypes in the Persons model. Finally, the Persons model contains the birthday of a person, an information that is missing in the Families model. Therefore, the information contained in one model cannot be fully reconstructed using the other model. Nevertheless, there is a clear correspondence as there should be a family member for each person and vice versa.

In this paper, we present a solution to this model synchronization challenge using the incremental and bidirectional model transformation language NMF Synchronizations [Hin15, HB17], part of the .NET Modeling Framework (NMF, [Hin16]). There, incremental model transformations run as a live transformation [HLR06] that continuously monitors both source and target models for incremental changes and propagates them to the other model. Because the benchmark framework runs in Java and NMF Synchronizations only runs on the .NET platform, our solution runs as a separate process that communicates with the benchmark framework using standard input and standard output. However, this adds a considerable serialization and deserialization overhead on the solution, which is why the benchmark times cannot be compared with other solutions. Instead, we included a custom time management and discuss how large the serialization overhead actually is.

In: A. Garcia-Dominguez, F. Krikava and G. Hinkel (eds.): Proceedings of the 10th Transformation Tool Contest, Marburg, Germany, 21-07-2017, published at http://ceur-ws.org

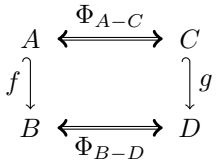


Figure 1: Schematic overview of synchronization blocks

Our solution is publicly available on GitHub[1].

The remainder of this paper is structured as follows: Section 2 gives a very brief introduction to synchronization blocks, the formalism used in NMF Synchronizations. Section 3 presents our solution. Section 4 evaluates our approach before Section 5 concludes the paper.

2 Synchronization Blocks

Synchronization blocks are a formal tool to run model transformations in a bidirectional way [HB17]. They combine a slightly modified notion of lenses [FGM+07] with incrementalization systems. Model properties and methods are considered morphisms between objects of a category that are set-theoretic products of a type (a set of instances) and a global state space Ω.

A (well-behaved) in-model lens $l : A \hookrightarrow B$ between types A and B consists of a side-effect free GET morphism $l \nearrow \in Mor(A, B)$ (that does not change the global state) and a morphism $l \searrow \in Mor(A \times B, A)$ called the PUT function that satisfy the following conditions for all $a \in A, b \in B$ and $\omega \in \Omega$:

$$l \searrow (a, l \nearrow (a)) = (a, \omega)$$
$$l \nearrow (l \searrow (a, b, \omega)) = (b, \tilde{\omega}) \quad \text{for some } \tilde{\omega} \in \Omega.$$

The first condition is a direct translation of the original PUTGET law [FGM+07]. Meanwhile, the second line is a bit weaker than the original GETPUT because the global state may have changed. In particular, we allow the PUT function to change the global state.

A (single-valued) synchronization block $\boldsymbol{S}$ is an octuple $(A, B, C, D, \Phi_{A-C}, \Phi_{B-D}, f, g)$ that declares a synchronization action given a pair $(a, c) \in \Phi_{A-C} : A \cong C$ of corresponding elements in a base isomorphism Φ_{A-C}. For each such a tuple in states (ω_L, ω_R), the synchronization block specifies that the elements $(f \nearrow (a, \omega_L), g \nearrow (b, \omega_R)) \in B \times D$ gained by the lenses are in the dependent isomorphism Φ_{B-D}.

A schematic overview of a synchronization block is depicted in Figure 1. The usage of lenses allows these declarations to be enforced automatically in both directions. The engine simply computes the value that for example the right selector should have and enforces it using the PUT operation. An unidirectional specification is also possible where f is no longer required to be a lens. More details can be found in previous publications [HB17].

A multi-valued synchronization block is a synchronization block where the lenses f and g are typed with collections of B and D, for example $f : A \hookrightarrow B*$ and $g : C \hookrightarrow D*$ where stars denote Kleene closures.

Synchronization Blocks have been implemented in NMF Synchronizations, an internal DSL hosted by C# [HB17, Hin15].

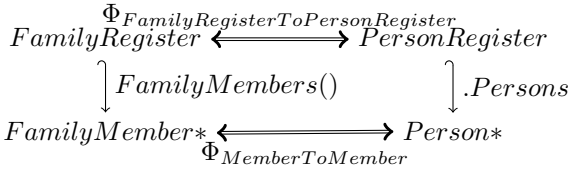
3 Solution

To solve the FamiliesToPersons case, we see two correspondencies that need to be synchronized:

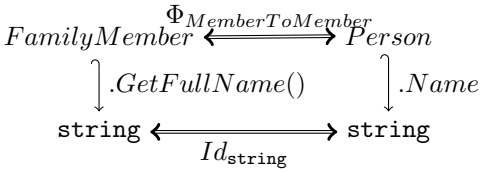
1. All family members contained in a family need to be synchronized with the people in the Persons model and
2. The full name of family members that consists of the name of the family and the name of the family member needs to be synchronized with the full name of the corresponding person.

Using synchronization blocks, these correspondences can be formulated in the diagrams of Figure 2. In the internal DSL of NMF Synchronizations, the implementation is depicted in Listing 1.

[1] `http://github.com/georghinkel/benchmarx`



(a) Synchronization block to synchronize family members with person elements



(b) Synchronization block to synchronize names

Figure 2: Synchronization Blocks in the solution

```
1  public class FamilyRegisterToPersonRegister : SynchronizationRule<FamilyRegister, PersonRegister> {
2    public override void DeclareSynchronization() {
3      SynchronizeMany(SyncRule<MemberToMember>(),
4                      fam => new FamilyMemberCollection(fam),
5                      persons => persons.Persons);
6    }
7  }
8  public class MemberToMember : SynchronizationRule<IFamilyMember, IPerson> {
9    public override void DeclareSynchronization() {
10     Synchronize(m => m.GetFullName(), p => p.Name);
11   }
12 }
```

Listing 1: Implementation of main synchronization blocks

In particular, the definition of the synchronization blocks in Listing 1 are implemented in a call to the methods `Synchronize` and `SynchronizeMany`. The types and the base isomorphisms `FamilyRegisterToPersonRegister` and `MemberToMember` used in the synchronization block can be inferred from the context and the explicitly specified dependent synchronization rule `MemberToMember`. In the second synchronization block, the identity on strings is used as isomorphism.

While NMF is able to convert the simple member accesses for the persons into lenses, this does not hold for the helpers `FamilyMemberCollection` and `GetFullName` that we used in this implementation. Therefore, we have to explicitly provide an implementation of Put for these two items.

For the `GetFullName`-method, the Put operation needs to be specified through an annotation. In addition, because NMF does not parse the contents of a method (only of lambda expressions), we need to specify an explicitly incrementalized version of the given helper method. To do this, we can reuse the implicit incrementalized lambda expression and also use that for the batch implementation to avoid code duplication. A sketched implementation is depicted in Listing 2.

```
1  private static ObservingFunc<IFamilyMember, string> fullName =
2    new ObservingFunc<IFamilyMember, string>(m =>
3        m.Name == null ? null : ((IFamily)m.Parent).Name + ",␣" + m.Name);
4
5  [LensPut(typeof(Helpers), "SetFullName")]
6  [ObservableProxy(typeof(Helpers), "GetFullNameInc")]
7  public static string GetFullName(this IFamilyMember member) {
8    return fullName.Evaluate(member);
9  }
10 public static INotifyValue<string> GetFullNameInc(this IFamilyMember member) {
11   return fullName.Observe(member);
12 }
13 public static void SetFullName(this IFamilyMember member, string newName) {
14   ...
15 }
```

Listing 2: Implementation of the `GetFullName` lens

In the case of `FamilyMemberCollection` which as the name implies is a collection, we only have to provide the formula how the results of this collection are obtained and implement the methods `Add`, `Remove` and `Clear`. A schematic implementation is depicted in Listing 3.

```
1  private class FamilyMemberCollection : CustomCollection<IFamilyMember> {
2    public FamilyRegister Register { get; private set; }
3    public FamilyMemberCollection(FamilyRegister register)
4      : base(register.Families.SelectMany(fam => fam.Children.OfType<IFamilyMember>()))
5    { Register = register; }
6
```

```
7     public override void Add(IFamilyMember item) { ... }
8     public override bool Remove(IFamilyMember item) { ... }
9     public override void Clear() { ... }
10  }
```

Listing 3: Implementation of the `FamilyMemberCollection`

However, to add a family member to a family, the `Add` method has to know the family name of a person as well as its gender – information that is encoded using the containment hierarchy in the Families model and therefore unavailable before the element is added to a family. Therefore, we carry this information over from the corresponding element of the Person metamodel using a temporary stereotype: In NMF, all model elements are allowed to carry extensions. We use this to add an extension that specifies the last name and whether the given element is male. The stereotype is deleted as soon as a family member is added to a family.

Furthermore, the fact that different genders are modeled through different classes in the Persons model, the synchronization rule `MemberToMember` needs to be refined to allow NMF Synchronizations to decide whether to create a `Male` or `Female` output element. This can be done in NMF Synchronizations through an instantiating rule.

The implementation of both of these concepts is depicted in Listing 4.

```
1   public class MemberToMale : SynchronizationRule<IFamilyMember, IMale> {
2     public override void DeclareSynchronization() {
3       MarkInstantiatingFor(SyncRule<MemberToMember>(),
4           leftPredicate: m => m.FatherInverse != null || m.SonsInverse != null);
5     }
6     protected override IFamilyMember CreateLeftOutput(IMale input, ...) {
7       var member = base.CreateLeftOutput(input, candidates, context, out existing);
8       member.Extensions.Add(new TemporaryStereotype(member) {
9         IsMale = true,
10        LastName = input.Name.Substring(0, input.Name.IndexOf(','))
11      });
12      return member;
13    }
14  }
```

Listing 4: The `MemberToMale`-rule

4 Evaluation

The actual transformation consists of 196 lines of which 73 are either empty or consist only of a single brace and whitespaces. We therefore think that our solution is very concise.

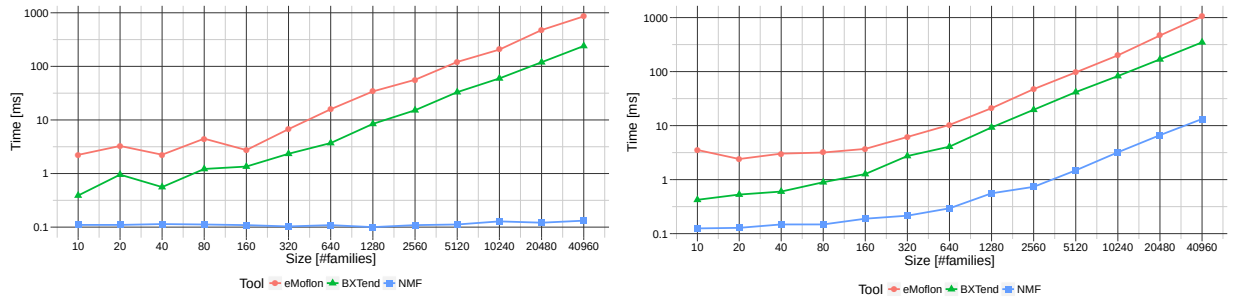
The solution passes all batch test cases defined in the test suite. For the incremental test cases, there are still some failures, but these are not due to the inabilities of the transformation but rather due to inaccurate change models: A move for example is transcribed as a deletion and an addition. However, for a new family member, NMF Synchronization will not reuse the `Person` element for the deleted member and thus, the information on the birthday is lost. Unfortunately, detecting a move and distinguishing it from deleting an element and inserting a new one is more complex and so far, we did not implement such a case distinction for the TTC.[2]

Due to the fact that the NMF solutions does not use EMF and does not use Java, there is a large serialization overhead attached to it. Depending on the test case, this overhead can outweigh the change propagation by multiple orders of magnitude. Thus, we adjusted the time measurement for our solution to eliminate this overhead.

The results for the incremental scalability test cases for NMF and the fastest reference solutions eMoflon and BXtend recorded on an Intel i7-4710MQ clocked at 2.50Ghz in a system with 16GB RAM is depicted in Figure 3. The plot shows the time to add a family member or a person in a model of the given number of families (each consisting of five members). Both axes are logarithmic. We only depicted the times for the fastest reference solutions as other provided reference solutions were much slower.

In the results for the *Incremental Forward* scenario, we can see that the resulting curve for the NMF solution is flat meanwhile the curve for eMoflon and BXtend is not. This indicates that unlike the other two, the NMF solution is the only fully incremental solution as the time to add a new family member does not depend on the size of the model. Furthermore, even for the smallest model with only 10 families, the incremental update

[2]Even without this case distinction, the class to convert EMF notifications to model changes in the NMF format is already more complex than the entire model synchronization.



(a) Incremental Forward

(b) Incremental Backward

Figure 3: Performance results for the incremental scalability tests

is faster than recomputing the *Persons* model from scratch. For the largest size, this leads to a speedup of more than three orders of magnitude.

The results for the *Incremental Backward* benchmark in the opposite direction are depicted in Figure 3b. Unlike the incremental forward where the scalability tests inserts a new family member and the propagation simply adds the corresponding person which can be done in amortized $O(1)$, the incremental backward test has to look for a suitable family, which implies a $\Omega(n)$ complexity. Therefore, the speedups in this case do not grow equally with the size. Instead, a saturation happens at a speedup of slightly more than one order of magnitude.

5 Conclusion

In this paper, we applied the bidirectional and incremental model transformation language NMF Synchronizations to the Families to Persons synchronization example. The solution demonstrates the easy adaptation of NMF Synchronizations by providing custom lens implementations.

Our solution is a live transformation that runs on the .NET platform, not on a JVM, and therefore requires a serialization and deserialization overhead to communicate with the benchmark driver. After eliminating this overhead, the results show that our solution is fully incremental and is faster than BXtend and eMoflon in the incremental scalability tests by multiple orders of magnitude.

References

[ABW17] Anthony Anjorin, Thomas Buchmann, and Bernhard Westfechtel. The Families to Persons Case. In Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, editors, *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences*, CEUR Workshop Proceedings. CEUR-WS.org, July 2017.

[FGM+07] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3), May 2007.

[HB17] Georg Hinkel and Erik Burger. Change Propagation and Bidirectionality in Internal Transformation DSLs. *Software & Systems Modeling*, 2017.

[Hin15] Georg Hinkel. Change Propagation in an Internal Model Transformation Language. In Dimitris Kolovos and Manuel Wimmer, editors, *Theory and Practice of Model Transformations: 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings*, pages 3–17, Cham, 2015. Springer International Publishing.

[Hin16] Georg Hinkel. NMF: A Modeling Framework for the .NET Platform. Technical report, Karlsruhe Institute of Technology, Karlsruhe, 2016.

[HLR06] David Hearnden, Michael Lawley, and Kerry Raymond. Incremental model transformation for the evolution of model-driven systems. In *International Conference on Model Driven Engineering Languages and Systems*, pages 321–335. Springer, 2006.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer Verlag, Wien, 1973.

The SDMLib Solution to the TTC 2017 Families 2 Persons Case

Albert Zündorf
Kassel University
zuendorf@uni-kassel.de

Alexander Weidt
Kassel University
alexander.weidt@uni-kassel.de

1 Introduction

The TTC 2017 Family to Persons Case asks for bidirectional transformations between a Family model and a Persons model. Each model provides informations that is not contained in the other model. Thus, the case asks to keep some kind of correspondences between the elements of the two models. In addition, the case asks for incremental handling of model changes.

Figure 1 shows the SDMLib class model for this case. SDMLib comes with its own code generation, i.e. we do not use the EMF class model nor the EMF code generation. SDMLib class models do not provide aggregation as this is an unusual concept within a graph like object model. Thus, the four associations between `Family` and `FamilyMember` are modeled as bidirectional associations with explicit roles at the `Family` side. To deal with these four associations easily, `FamilyMember` provides the additional method `getFamily()` that looks up all four associations in order to find the corresponding family.

To maintain the correspondences between the two model parts, we have added a famReg–persReg association between `FamilyRegister` and `PersonRegister`. In addition we use a cp–cfm association connecting `FamilyMember` objects with corresponding `Person` objects.

Adressing the incremental requirement, we use unidirectional `c`hanged links between the register objects and their `FamilyMember` and **Person** objects, respectively. The set methods mark changed objects with c links, as a side effect. Furthermore, the cp–cfm association has a life-dependency semantics: if one object is explicitly destroyed, the corresponding partner is deleted as well. Again, this is implemented via side effects in the corresponding `removeYou()` methods.

2 The rules

Figure 2 shows our forward transformation rule in graphical notation. Pattern matching starts with the pattern object `fr` that matches the `FamilyRegister` object passed as parameter. Pattern object `pr` matches the corresponding `PersonRegister`. Next, pattern object `fm` looks for `FamilyMember` objects marked as changed via a `c` link. This `c` link is shown in red color as it is deleted on rule execution (in order to unmark handled objects).

Now there are two cases: there might already exist a corresponding `Person` object or not. The first case is handled by the optional subpattern 1 shown in a dashed box in the bottom right corner of Figure 2. If pattern object `oldP` finds a match, the pseudo condition `ensureNameAndGender(oldP)` is checked. This methods, is shown below:

```
1    public boolean ensureNameAndGender(Person p) {
2       FamilyMember fm = p.getCfm();
3       Family f = fm.getFamily();
4       Class gender = Male.class;
5       if (fm.getMotherOf() != null || fm.getDaughterOf() != null) {
6          gender = Female.class;
```


In: A. Garcia-Dominguez, F. Krikava and G. Hinkel (eds.): Proceedings of the 10th Transformation Tool Contest, Marburg, Germany, 21-07-2017, published at http://ceur-ws.org

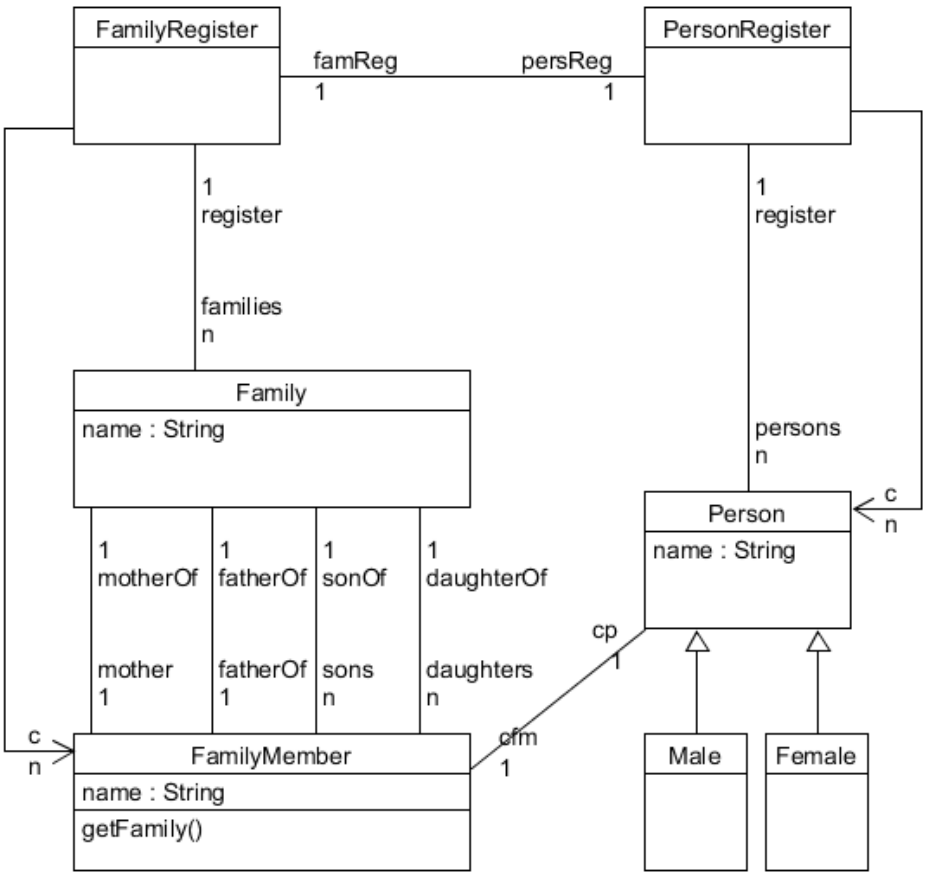


Figure 1: Adapted Class Model

```
7         }
8         if (p.getClass() != gender) {
9            try {
10               Person newP = ((Person) gender.newInstance())
11                     .withRegister(p.getRegister()).withBirthday(p.getBirthday())
12                     .withCfm(p.getCfm()).withName(p.getName());
13               p.removeYou();
14               p = newP;
15           } catch (InstantiationException | IllegalAccessException e) {
16               e.printStackTrace();
17           }
18        }
19        String fullName = String.format("s, s", f.getName(), fm.getName());
20        p.withName(fullName);
21        return true;
22     }
```

Listing 1: Example Graph Query in Java

The Families to Persons case requires that the mother and the daughters of a `Family` shall map to an object of type `Female` within the `PersonRegister`. The father and the sons shall map to an object of type `Male`. As a `FamilyMember` might have changed e.g. from the `sons` role of one family to the `mother` role of another family, the corresponding `Person` object needs to change from type `Male` to type `Female`. As Java objects are not able

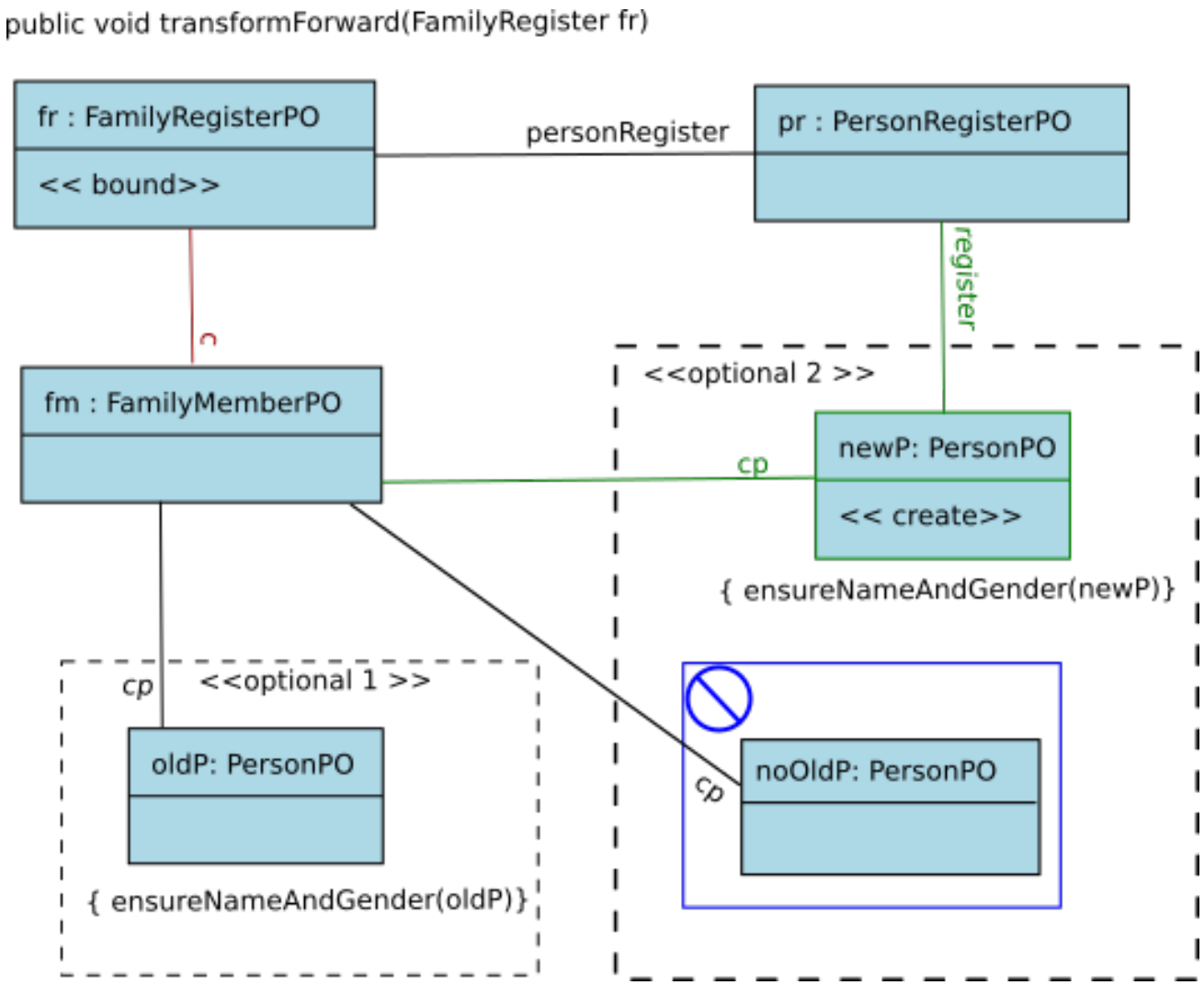


Figure 2: Forward Transformation Rule

to change their type, lines 10 to 12 of Listing 1 simply create a new Object of the appropriate gender and copy all attributes and references from the old object. Finally, method `ensureNameAndGender()` assigns the current full name to the `Person` object.

The second case is handled by the second optional subpattern of Figure 2. The second subpattern contains a negative application condition shown as a solid blue box with a forbidden sign in its upper left corner. This negative application condition again looks for an old corresponding `Person` if there is no match the negative application is not violated and the surrounding pattern continues its matching. In our case, the second optional subpattern just creates a new object of type `Person`. As discussed, we actually need an object either of type `Female` or of type `Male`. This is again handled by calling method `ensureNameAndGender()` within the pseudo condition shown below pattern object `newP`.

Figure 3 shows our backward transformation rule. Pattern matchin starts with the `PersonRegisterPO` pattern object `pr` shown in the up right corner. The match for `pr` is provided as parameter. The rule first looks for a `Person p` that is marked as changed via a `c` link. This `c` link is destroyed (as indicated by the red color). The backward transformation rule now handles two cases: there is an old corresponding `FamilyMember` or not. The first case is handled by the optional subpattern shown at the bottom of Figure 3. The pattern object `oldFM` matches a `FamilyMember` that has a `cfm` (corresponding family member) link to our `Person p`. Next pattern object `oldF` looks up the corresponding `Family`. This uses method `getFamily()` which tries `motherOf`, `fatherOf`, `daughterOf`, and `sonOf` links to navigate from a `FamilyMember` to its `Family`. Now there are two

conditions. Condition 1 is a pseudo condition that actually assigns the given name of the current person to the current `FamilyMember`. After that, the second condition checks, whether the current `Family` has the right name. The second condition holds, if the current Family's does not match current person's family name. In that case, we need to look for another Family with a matching name. This is done by the second subpattern of our rule. Thus, in case of a conflict with the family name, the first subpattern matches (all conditions) and it is executed, i.e. the `oldFM` object is destroyed. (Causing the second subpattern to create a new `FamilyMember` object, see below). If the family name matches, the second condition fails and the pattern is not executed but the `oldFM` object survives. In that case, the first pseudo condition has already adjusted the given name and the case of an old existing `FamilyMember` is handled, completely.

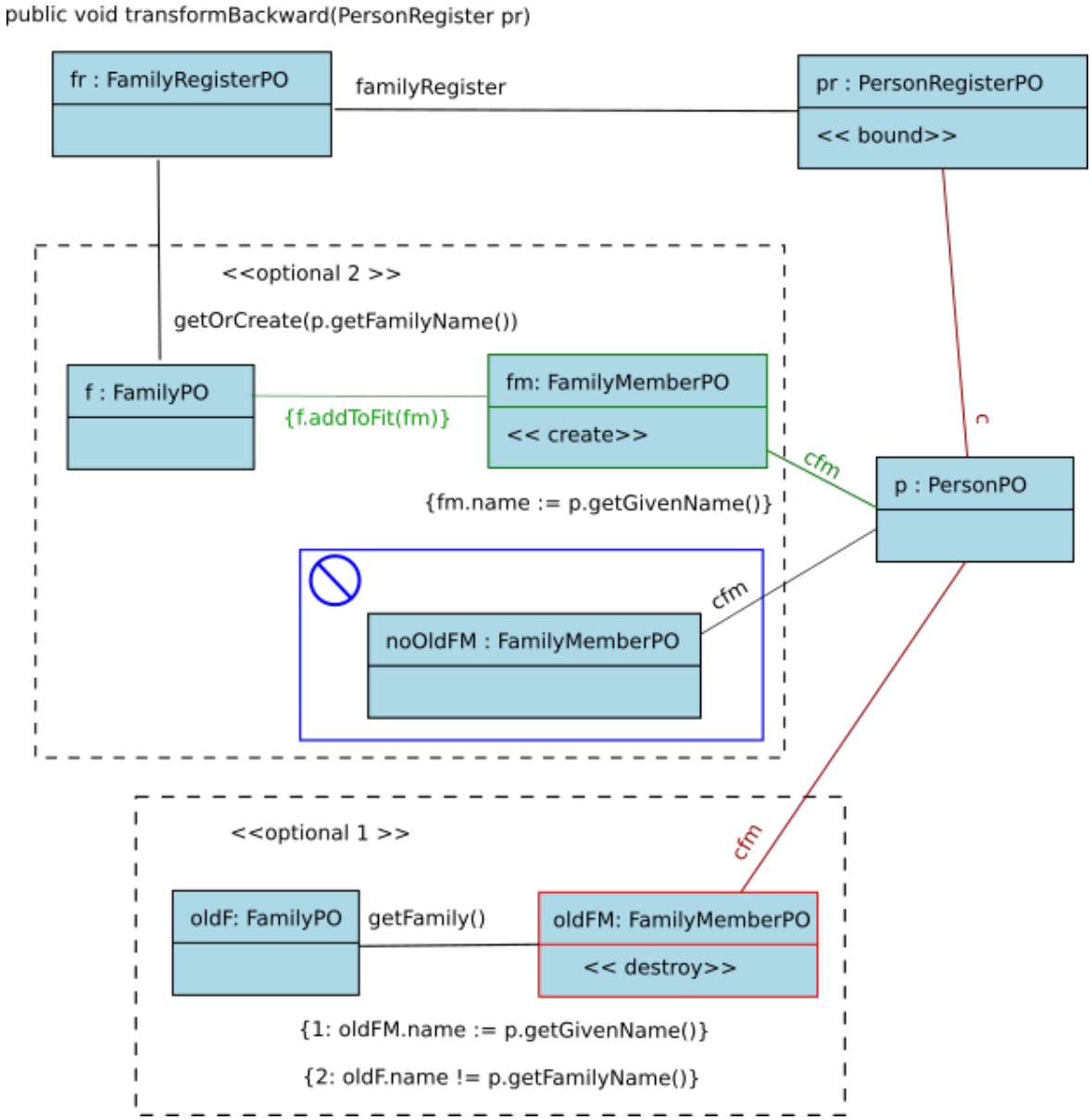


Figure 3: Backward Transformation Rule

The second optional subpattern of our backward transformation rule creates a new `FamilyMember` object that corresponds to our `Person p`. This is done by the pattern object `fm`. The pseudo condition below `fm` assigns the given name. We also need a `Family f` for the new `FamilyMember`. The `Family` is derived from the `FamilyRegister fr` via the path expression `getOrCreate(p.getFamilyName())`. This operation searches our `FamilyRegister` for a matching `Family`. If there is no `Family` with the right name, a new `Family` object is

created. Actually, the `getOrCreate()` methods also respects the `Decisions` parameter required by the case description and creates always a new `Family`, if required. Now, we have to add our new `FamilyMember` to its `Family`. This is done using the pseudo condition `f.addToFit(fm)`. Method addToFit() just looks for the gender of the current person and whether the corresponding parent role is still vacant and whether it should be used or whether the new `FamilyMember` should become a daughter or a son, respectively. Note, method `addToFit()` might easily have been modeled as a model transformation rule, but the cases are pretty straight forward and thus we just programmed it in plain Java.

3 Results

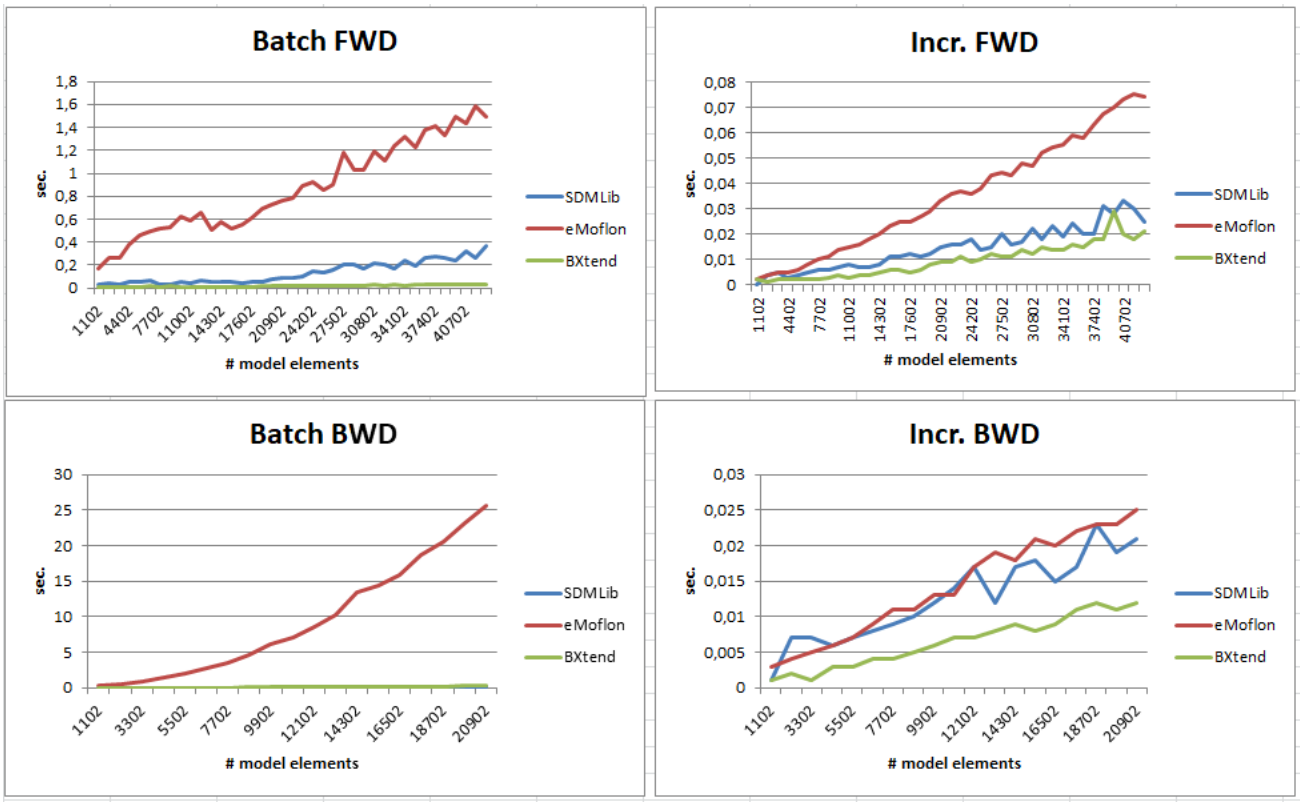


Figure 4: Performance

Figure 4 shows the performance measured by the predefined scalability test of the Families to Persons case. We have measured this performance on a laptop with 2 GHz Intel I7 CPU giving the Java process 6 GB heap space. We compared SDMLib only to eMoflon and BXtend as the Medini and the FunnyQT approach used a magnitude more time. Final performance results will be delivered by the case organizers that are able to compare all solutions on one computer. Thus, Figure 4 gives some relative results, only. Still, SDMLib performance almost similar to BXtend while eMoflon is 2 or 3 times slower. All tools deal well with the incremental transformations.

References

[SDMLib] SDMLib - Story Driven Modeling Library *www.sdmlib.org* May, 2017.

[F2PCase] Anthony Anjorin, Thomas Buchmann and Bernhard Westfechtel The Family to Persons Case *www.transformation-tool-contest.eu/TTC_2017_paper_2.pdf* May, 2017.

Solving the TTC Families to Persons Case with FunnyQT

Tassilo Horn
tsdh@gnu.org

The GNU Project

Abstract

This paper describes the FunnyQT solution to the bidirectional Families to Persons transformation case of TTC 2017. The solution is simple and concise and passes all batch transformations and some incremental tests.

1 Introduction

This paper describes the FunnyQT[1] [Hor16, Hor15] solution of the Families to Persons case [ABW17] of TTC 2017. With only 52 lines of declarative code, the solution is able to pass all batch transformation tests and some incremental tests. The solution project is available on Github[2] and it is integrated into the *benchmarx*[3] framework.

FunnyQT is a model querying and transformation library for the functional Lisp dialect Clojure[4]. Queries and transformations are Clojure functions using the features provided by the FunnyQT API.

Clojure provides strong metaprogramming capabilities that are used by FunnyQT in order to define several *embedded domain-specific languages* (embedded DSL [Fow10]) for different querying and transformation tasks.

FunnyQT is designed to be extensible. By default, it supports EMF [SBPM08] and JGraLab[5] TGraph models. Support for other modeling frameworks can be added without having to touch FunnyQT's internals.

The FunnyQT API is structured into several namespaces, each namespace providing constructs supporting concrete querying and transformation use-cases, e.g., model management, functional querying, polymorphic functions, relational querying, pattern matching, in-place transformations, out-place transformations, bidirectional transformations, and some more. For solving the Families to Persons case, its bidirectional transformation and relational model querying DSLs have been used.

2 Solution Description

This section explains the FunnyQT solution. First, Section 2.1 explains the actual transformation solving the case. Then, Section 2.2 explains how the solution is integrated into the *benchmarx* framework.

2.1 The Families to Persons Transformation

As a first step, relational querying APIs for the two metamodels are generated.

```
1 (rel/generate-metamodel-relations "metamodels/Families.ecore" f)
2 (rel/generate-metamodel-relations "metamodels/Persons.ecore" p)
```

This makes the relations for the families metamodel available with the namespace alias `f` and those of the persons metamodel with alias `p`. Relations are generated for every metamodel element. For example, there is a

In: A. Garcia-Dominguez, G. Hinkel and F. Krikava (eds.): Proceedings of the 10th Transformation Tool Contest, Marburg, Germany, 21-07-2017, published at http://ceur-ws.org

[1] `http://funnyqt.org` [2] `https://github.com/tsdh/ttc17-families2persons-bx` [3] `https://github.com/eMoflon/benchmarx/`
[4] `http://clojure.org` [5] `https://github.com/jgralab/jgralab`

relation (`f/Family f family`) which succeeds for every `family` in the given family model `f`. There is a relation (`f/name f el n`) which succeeds if `el` is a member or family of the family model `f` whose name is `n`. And there are reference relations like (`f/->sons f parent child`) which succeed if `child` is a son of `parent`.

Before the transformation itself, we define a helper relation which defines the possible kinds of relationships between a `family` and a family `member` depending on if we prefer to create parents over creating children (parameter `pref-parent`). This is a higher-order relation in that the two remaining parameters are a parent relation `prel` (either `f/->father` or `f/->mother`) and a child relation (either `f/->daughters` or `f/->sons`).

```
3  (defn relationshipo [pref-parent f family member prel crel]
4    (ccl/conda
5     [(bx/target-directiono :right)     ;; (1)
6      (ccl/conde
7       [(prel f family member)]
8       [(crel f family member)])]
9     [(bx/existing-elemento? member)]   ;; (2)
10    [(ccl/== pref-parent false)        ;; (3)
11     (crel f family member)]
12    [(bx/unseto? f family prel member) ;; (4)
13     (prel f family member)]
14    [(crel f family member)]))         ;; (5)
```

`ccl/conda` is like a short-cutting logical disjunction. The n-th clause is only tried if all preceeding clauses fail. The first clause succeeds when we are transforming into the direction of the right model, i.e., the person register. In this case, `member` may be in a parental role of family (`prel`), or it might be in a child role (`crel`). We don't really care but want to ensure that all members of the given `family` are reachable, thus we use a non-short-cutting `ccl/conde` which gives both clauses a chance to succeed.

All other clauses deal with transforming a person model to a family model, i.e., the backward transformation. The second clause deals with the case where `member` is an already existing element, i.e., not created by the current execution of the transformation. Here, we assume that this member is already properly assigned to a family, so we simply succeed without doing anything.

In the third clause, if we do not prefer assigning to parental roles, then the child relation `crel` must succeed between the `family` and the `member`. The child relations can always succeed because a family can have an arbitrary number of daughters and sons. Thus, in the remaining clauses we have to deal only with the case where assigning to parental roles is preferred.

In the fourth clause, if the `family`'s parental role is still unset or already assigned to `member`, then the parental relation must succeed between the `family` and the `member`.

Lastly, if no clause has succeeded until now, then the child relation has to succeed. As said, this goal[6] can always succeed.

In the following, the actual transformation definition is explained. It is defined using the `deftransformation` macro provided by FunnyQT.

```
15 (bx/deftransformation families2persons [f p prefer-parent prefer-ex-family]
16   :delete-unmatched-target-elements true
17   :id-init-fn bx/number-all-source-model-elements
```

The name of the transformation is `families2persons` and it declares four parameters. The parameter `f` is the family model (the left model), `p` is the persons model (the right model), `prefer-parent` is a Boolean flag determining if we prefer creating parents to creating children, and `prefer-ex-family` is a Boolean flag, too, determining if we prefer re-using existing families over creating new families for new family members.

By default, bidirectional FunnyQT transformations will never delete elements from the current target model, i.e., the model being created or modified by the synchronization. The reason for that behavior is that it allows to run the transformation in one direction first and then in the other direction in order to perform a full synchronization where missing elements are created in each of the two models. Thus, after running a transformation, e.g., in the direction of the right model, it is only ensured that for each element (considered by the transformation's rules) in the left model, there is a corresponding counterpart in the right model. However, the right model might still contain elements which have no counterpart in the left model. With option `:delete-unmatched-target-elements` set to `true`, this behavior is changed. Elements in the current target model which are not required by the current source model and the transformation relations are deleted.

[6]The application of a relation is called a *goal*.

The next option, `:id-init-fn`, has the following purpose. In this transformation case, family members and persons do not have some kind of unique identity. For example, it is allowed to have two members named Jim with the same name in the very same family Smith. However, FunnyQT BX transformations have forall-there-exists semantics where it would suffice to create just one person in the right model with the name set to "Smith, Jim". But the case description mandates that we create one person for every member and vice versa, no matter if they can be distinguished based on their attribute values. For such scenarios, FunnyQT's bidirectional transformation DSL provides a concept of synthetic ID attributes. The value of `:id-init-fn` has to be a function which returns a map from elements to their synthetic IDs. The built-in function `bx/number-all-source-model-elements` returns a map where every element in the source model gets assigned a unique integer number. These synthetic IDs are then used in a transformation relation which is discussed further below.

The first transformation relation, `family-register2person-register`, transforms between family and person registers.

```
18  (^:top family-register2person-register
19   :left  [(f/FamilyRegister f ?family-register)]
20   :right [(p/PersonRegister p ?person-register)]
21   :where [(member2female :?family-register ?family-register :?person-register ?person-register)
22           (member2male :?family-register ?family-register :?person-register ?person-register)])
```

It is defined as a top-level rule meaning that it will be executed as the entry point of the transformation. Its `:left` and `:right` clauses describe that for every `?family-register` there has to be a `?person-register` and vice versa. We assume that there is always just one register in each model.

The `:where` clause defines that after this relation has been enforced (or checked in checkonly mode), then the two transformation relations `member2female` and `member2male` have to be enforced (or tested) between the current `?family-register` and `?person-register`[7].

The next transformation relation, `member2person`, describes how family members of a family contained in a family register in the left model correspond to persons contained in a person register in the right model. As it can be seen, there is no goal describing how the `?family` and the `?member` are connected in the `:left` clause and in the `:right` clause we are dealing with just a `?person` of type `Person` which is an abstract class. As such, this relation is not sufficient for the complete synchronization between members in the different roles of a family to females and males. Instead, it only captures the aspects that are common in the cases where mothers and daughters are synchronized with females and fathers and sons are synchronized with males. Therefore, this relation is declared abstract.

```
23  (^:abstract member2person
24   :left  [(f/->families f ?family-register ?family)
25           (f/Family f ?family)
26           (f/name f ?family ?last-name)
27           (f/FamilyMember f ?member)
28           (f/name f ?member ?first-name)
29           (id ?member ?id)
30           (ccl/conda
31            [(ccl/== prefer-ex-family true)]
32            [(bx/existing-elemento? ?member)
33             (id ?family ?last-name)]
34            [(id ?family ?id)])]
35   :right [(p/->persons p ?person-register ?person)
36           (p/Person p ?person)
37           (p/name p ?person ?full-name)
38           (id ?person ?id)]
39   :when  [(rel/stro ?last-name ", " ?first-name ?full-name)])
```

So what are these common aspects? Well, a `?member` of a `?family` (where we have not determined the role, yet) contained in the `?family-register` passed in as parameter from `family-register2person-register` corresponds to a `?person` (where we have not determined the gender, yet) contained in the `?person-register` passed in as the other parameter from `family-register2person-register`. The `:when` clause defines that the concatenation of the `?family`'s `?last-name`, the string `", "` and the `?member`'s `?first-name` gives the `?full-name` of the `?person`.

What has not been described so far are the `id` goals in lines 29, and 34 and the `ccl/conda` goal starting in line 30. The first two define that the `?member` and the corresponding `?person` must have the same synthetic ID. Remember the `:id-init-fn` in line 17 which assigned a unique number to every element in the respective source

[7] Transformation relations are called with keyword parameters. The two calls in the `:where` clause state that the current `?family-register` will be bound to the logic variable with the same name in the called relation and the same is true for the `?person-register`.

model of the transformation. With these synthetic IDs, the transformation is able to create one person for every member and vice versa even in the case where two elements are equal based on their attribute values.

Lastly, the `ccl/conda` goal starting in line 30 of the `:left` clause handles the preference of re-using existing families, i.e., assigning new members to existing families, over creating new families for new members. By default, FunnyQT would always try to re-use an existing family. Thus, if the `prefer-ex-family` parameter is `true`, nothing needs to be done. Likewise, if `?member` is an existing element for which we assume she is already assigned to some family, we can also just stick to the default behavior but define the ID of the `?family` to be its name (although it is probably not unique). If the first two `ccl/conda` clauses fail, i.e., `prefer-ex-family` is `false` and `?member` is a new member which is just going to be created by the enforcement of this relation, then we define that the ID of the `?family` must equal to the IDs of the `?member` and `?person`. Thus, in this case and only in this case, new members force the creation of a new family even when there is already a family with the right name.

The last two transformation relations extend the `member2person` relation for synchronizing between members in the role of a family mother or daughter and female persons and between family fathers or sons and male persons.

```
40  (member2female
41   :extends [(member2person)]
42   :left  [(relationshipo prefer-parent f ?family ?member f/->mother f/->daughters)]
43   :right [(p/Female p ?person)])
44  (member2male
45   :extends [(member2person)]
46   :left  [(relationshipo prefer-parent f ?family ?member f/->father f/->sons)]
47   :right [(p/Male p ?person)]))
```

In the `:left` clauses we use the `relationshipo` helper relation described in the beginning of this section which chooses the right female or male role based on the preference parameter `prefer-parent` and the current state of the family, i.e., by checking if the respective parental role is still unset. In the two `:right` clauses, we only need to specify that the `Person ?person` is actually a `Female` or `Male`.

These 33 lines of transformation specification plus the 12 lines for the `relationshipo` helper, and two lines for the generation of the metamodel-specific relational querying APIs form the complete functional parts of the solution. The only thing omitted from the paper are the namespace declarations[8] consisting of 5 lines of code.

2.2 Gluing the Solution with the Framework

Typically, open-source Clojure libraries and programs are distributed as JAR files that contain the source files rather than byte-compiled class files. This solution does almost the same except that the JAR contains the solution source code, FunnyQT itself (also as sources) and every dependency of FunnyQT (like Clojure) except for EMF which the *benchmarx* project already provides.

Calling Clojure functions from Java is really easy and FunnyQT transformations are no exception because they are plain Clojure functions, too. The `BXTool` implementation `FunnyQTFamiliesToPerson` extends the `BXToolForEMF` class. Essentially, it has just a static member `T` which is set to the transformation.

```
public class FunnyQTFamiliesToPerson extends BXToolForEMF<FamilyRegister, PersonRegister, Decisions> {
    private final static Keyword LEFT = (Keyword) Clojure.read(":left");
    private final static Keyword RIGHT = (Keyword) Clojure.read(":right");
    private final static IFn T; // <-- The FunnyQT Transformation

    static {
        final String transformationNamespace = "ttc17-families2persons-bx.core";
        // Clojure's require is similar to Java's import.  However, it also loads the required
        // namespace from a source code file and immediately compiles it.
        final IFn require = Clojure.var("clojure.core", "require");
        require.invoke(Clojure.read(transformationNamespace));
        T = Clojure.var(transformationNamespace, "families2persons");
    }
```

All Clojure functions implement the `IFn` interface and can be called using `invoke()`. And exactly this is done to call the transformation.

```
    private void transform(Keyword direction) {
        T.invoke(srcModel, trgModel, direction,
                 configurator.decide(Decisions.PREFER_CREATING_PARENT_TO_CHILD),
                 configurator.decide(Decisions.PREFER_EXISTING_FAMILY_TO_NEW));
    }
```

This corresponds to a direct Clojure call `(families2persons src trg dir prefer-parent prefer-ex-family)`.

[8] The Clojure equivalent of Java's package statement and imports.

3 Evaluation and Conclusion

In this section, the test results of the FunnyQT solution are presented and classified as requested by the case description [ABW17]. Since the bidirectional transformation DSL of FunnyQT is state-based and not incremental at all (and by design), many of the incremental tests are mostly out of scope. However, in its core use case, non-incremental bidirectional transformations, the solution passes all tests.

BatchForward.* All tests result in an *expected pass*.
BatchBwdEandP.* All tests result in an *expected pass*.
BatchBwdEnotP.* All tests result in an *expected pass*.
BatchBwdNotEandP.* All tests result in an *expected pass*.
BatchBwdNotEnotP.* All tests result in an *expected pass*.
IncrementalForward.* The solution *expectedly passes* the tests `testStability`, `testHippocraticness`, and `testIncrementalInserts`. All remaining tests *fail expectedly* because those tests require incremental abilities. For example, in some tests the birthdates are set manually in the persons model. The re-execution of the transformation causes deletion and re-creation of those persons, however then they get assigned default birthdates and not the manually edited ones as the transformation does not consider this attribute at all.
IncrementalBackward.* The solution *expectedly passes* the tests `testStability`, `testIncrementalInsertsFixedConfig`, `testIncrementalOperational`, and `testHippocraticness`. The test `testIncrementalInsertsDynamicConfig` *fails unexpectedly*. When we neither prefer existing families nor assigning to parental roles, the transformation still adds the new Seymore to an existing family in a parental role. All other tests are *expected fails* due to the fact that they require incremental capabilities.

In summary, the correctness is satisfying. There is only one test which fails unexpectedly. All other fails are expected and can hardly be solved by a non-incremental approach.

A very weak point of the solution is its performance. It is at least an order of magnitude slower than the other solutions already integrated in the benchmarx project (BiGUL, eMoflon, BXtend, MediniQVT). Where their runtimes are in the tenth of seconds or below, the FunnyQT solution takes seconds. With models in the size of thousands of elements, you might have to wait a bit for the transformation to finish. One reason is that the FunnyQT BX implementation is state-based and thus needs to recompute the correspondences between all elements in the source and target models on every invocation instead of just propagating small deltas like incremental approaches do. Furthermore, FunnyQT's bidirectional transformation DSL is built upon the relational programming library *core.logic*[9] which is not tuned for performance but for simplicity and extensibility of its implementation and FunnyQT probably does not use it in the best possible way.

Other good points of the solution are its conciseness and simplicity. With only 52 lines of code, it is by far the most concise solution currently integrated in the benchmarx project. And it is quite simple to understand. The only complexities it has arose from the different alternatives depending on external parameters.

References

[ABW17] Anthony Anjorin, Thomas Buchmann, and Bernhard Westfechtel. The Families to Persons Case. In Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, editors, *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences*, CEUR Workshop Proceedings. CEUR-WS.org, July 2017.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[Hor15] Tassilo Horn. Graph Pattern Matching as an Embedded Clojure DSL. In *International Conference on Graph Transformation - 8th International Conference, ICGT 2015, L'Aquila, Italy, July 2015*, 2015.

[Hor16] Tassilo Horn. *A Functional, Comprehensive and Extensible Multi-Platform Querying and Transformation Approach*. PhD thesis, University Koblenz-Landau, Germany, 2016.

[SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2 edition, 2008.

[9] `https://github.com/clojure/core.logic`

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,

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.

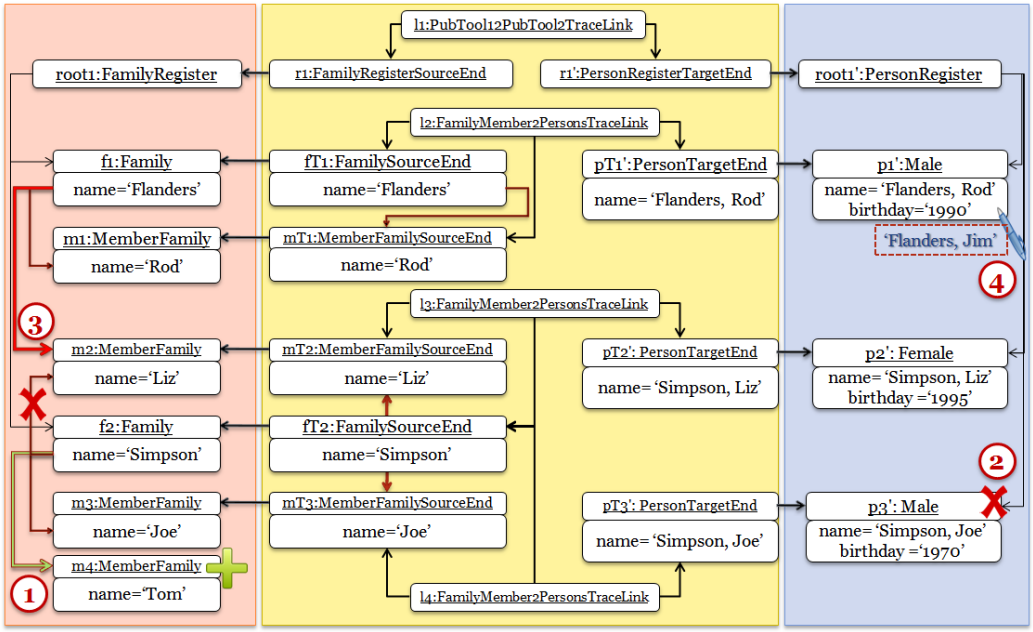


Figure 1: Example of consistent models

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).

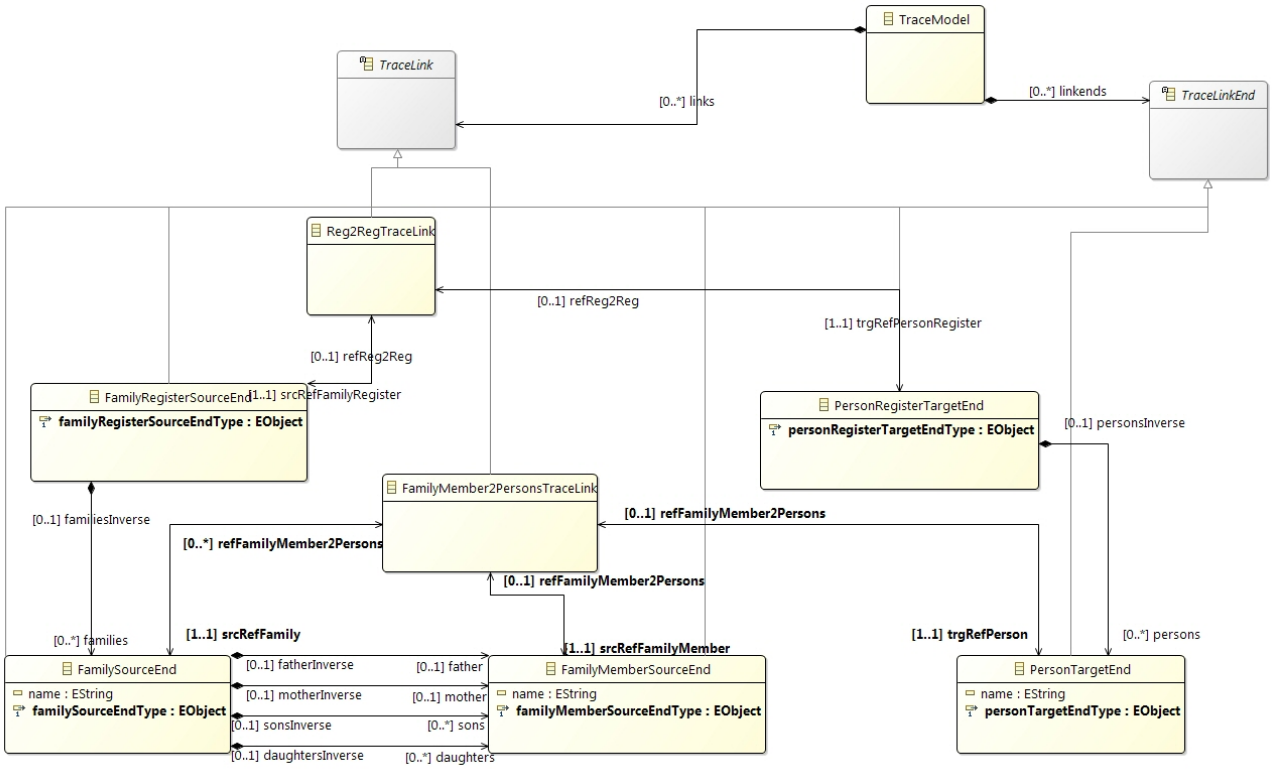


Figure 2: The case-specific trace metamodel for Families to Persons case study

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.

Part III.

State Elimination Case

State Elimination as Model Transformation Problem

Sinem Getir
Software Engineering, Humboldt University Berlin
getir@informatik.hu-berlin.de
Duc Anh Vu
Software Engineering, Humboldt University Berlin
vuducanh@informatik.hu-berlin.de
Francois Peverali
Model-driven Software Development, Humboldt University Berlin
peveralf@informatik.hu-berlin.de
Daniel Strüber
Institute for Computer Science, University of Koblenz and Landau
strueber@uni-koblenz.de
Timo Kehrer
Department of Computer Science, Humboldt University Berlin
timo.kehrer@informatik.hu-berlin.de

Abstract

State elimination has been proposed in the literature as a viable technique for transforming finite state automata (or finite state machines) into equivalent regular expressions. In this TTC case, we consider this well-known technique as a model transformation problem, aiming at evaluating the suitability, performance and scalability of dedicated model transformation techniques w.r.t. this problem.

1 Introduction

Finite state automata (or finite state machines) are used in many applications such as program analysis, data validation, pattern matching, speech recognition as well as in the behavioral modeling of systems and software. Furthermore, quantitative extensions of automata are used to evaluate system behavior together with software profiles in software engineering. Such models are, like Markov chains, probabilistic automata, adding probabilistic distributions to state transitions [BKL08]. Even though the semantically equivalent counterparts of (probabilistic) finite state automata, namely (stochastic) regular expressions, are generally not considered as behavioral models, they may be used in the aforementioned applications interchangeably with finite state automata. In practice, regular expressions are often transformed into finite state automata, e.g. to obtain models from code, to extract behavioral models at runtime, etc. While this is not an expensive task, the transformation of a (probabilistic) finite state automaton into an equivalent (stochastic) regular expression is much more expensive and challenging.

In: Antonio Garcia-Dominguez, Georg Hinkel, Filip Křikava (eds.): Proceedings of the 10th Transformation Tool Contest, Marburg, Germany, 21-07-2017, published at http://ceur-ws.org

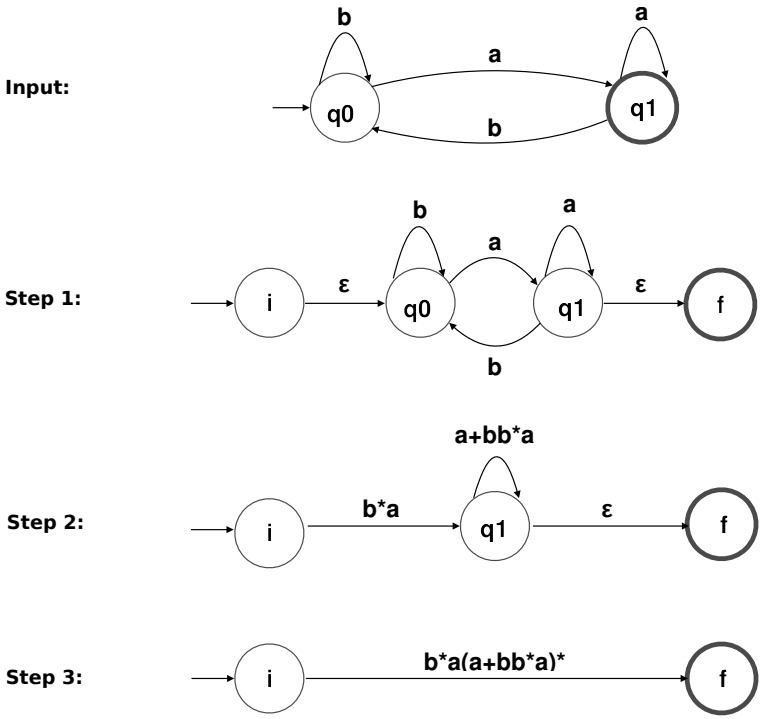


Figure 1: State Elimination example.

State elimination [DK01] has been proposed in the literature as a viable solution for doing this kind of transformation. In this TTC case, we consider this well-known technique as a model transformation problem, aiming at evaluating the suitability, performance and scalability of dedicated model transformation techniques w.r.t. this problem.

Section 2 briefly recalls basic principles of state elimination and presents a reference implementation based on a commonly used software package for experimenting with formal languages. Section 3 rephrases the involved problems as model transformation problems and provides the respective task descriptions, some of which are being considered as optional extensions to the main task of doing state elimination. Section 4 presents a set of quality characteristics to evaluate proposed solutions. All resources provided along with the case can be obtained from `https://github.com/sinemgetir/state-elimination-mt/`

2 Case Description

In this section, after recalling some basic definitions, we describe the state elimination algorithm, a reference implementation using JFLAP [PPR96], and our probabilistic implementation that is an extension to JFLAP in detail.

2.1 State Elimination

In the following, let Σ be a finite set of symbols, called alphabet, and Σ^* the words over Σ. A regular expression α over Σ represents a regular language $L(\alpha) \subset \Sigma^*$ for which, according to Kleene's theorem, there exists a finite state automaton accepting this language. An FSA is a tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is a set of final states. We say that an FSA is *uniform* if it has has a unique initial state with no incoming transitions and a unique final state with no outgoing transitions. Uniforming an FSA may be performed as follows:

- If $q_0 \in F$ or if there exists an $a \in \Sigma$ and $q \in Q$ so that $\delta(q, a) = q_0$, then add a new state i to Q with $\delta(i, \epsilon) = q_0$ and set i as the new initial state instead of q_0.
- If $|F| \geq 1$ or if there exists $p \in F$, $a \in \Sigma$ and $q \in Q$ so that $\delta(p, a) = q$, then add a new state f to Q, add the transition $\delta(p, \epsilon) = f$ for all $p \in F$ and set $F = \{f\}$.

The state elimination algorithm [DK01] takes a FSA as input and calculates a regular expression describing its accepted language as output. Given an FSA, state elimination works on its unified form. The idea is to iteratively eliminate states which are neither initial nor final. In each step, a new equivalent generalized transition graph (GTG) is created until only the initial and final states are left. A GTG generalizes an FSA in that transitions may be labelled by regular expressions instead of single symbols.

Let $\mathcal{R}(\Sigma)$ be the universe of regular expressions over a given alphabet Σ, then the transition function is generalized to $\delta : Q \times \mathcal{R}(\Sigma) \to Q$. Roughly speaking, regular expressions used as transition labels are utilized to keep track of eliminated states. The regular expression labelling the transition between the two states that remain at the end of the iteration represents the final output of the algorithm.

In Algorithm 1, let α_{xy} be the regular expression for the transition from state x to state y. Moreover, given an FSA $A = (Q, \Sigma, \delta, q_0, F)$, we refer to its unified form as $A' = (Q', \Sigma, \delta', i, \{f\})$ which we will interpret as GTG during state elimination.

Algorithm 1 State elimination

Input: A uniform FSA A'.
Output: A regular expression defining the regular language accepted by A'.

1: **repeat**
2: Randomly choose a state $k \in Q' \setminus \{i, f\}$ and remove it from A'.
3: Then for all pairs $p, q \in Q' \setminus \{k\}$ the new regular expression α'_{pq} for the transition from p to q is:

$$\alpha'_{pq} = \alpha_{pq} + \alpha_{pk}\alpha^*_{kk}\alpha_{kq}$$

4: **until** There are only the initial state i and the final state f left.
5: **return** α_{if}

Based on Algorithm 1, a small example that demonstrates how to convert an FSA into a regular expression is presented in Figure 1. Step 1 uniforms the given FSA, while steps 2 and 3 showcase the iterative elimination of states.

2.2 Reference Implementation

We use the common software package JFLAP [PPR96] to transform an FSA into an equivalent regular expression using the state elimination method. The difference between Algorithm 1 and the JFLAP implementation is that, for the latter one, the initial state can have incoming transitions and the final state can have outgoing transitions. Considering that, the transformation can be divided into three steps:

(1) Conversion of the given FSA into a simple FSA: A simple FSA is an FSA which is uniform and, in addition, for all pairs of states there is exactly one transition. The simple automaton is created in the following steps:

- If the automaton has more than one final state or the initial state is a final state, we create a new state s and for all final states f, add an $\epsilon - transition$ for $f \longrightarrow s$ and set s as the only final state. Note that in JFLAP $\epsilon - transitions$ are defined as $\lambda - transitions$ with the empty string as the label.
- If there are several transitions between two states, we combine them into a single transition; the new label is obtained as a disjunction of all transition's labels.
- If there are no transitions between two states, we add an empty transition between them. An empty transition is a transition with the label $\emptyset$.

(2) Conversion to a GTG. Converting a simple automaton obtained from step (2) into an equivalent GTG with only the initial state and the final state is described by Algorithm 2, which in turn uses the subroutine described in Algorithm 3 in order to obtain a regular expression for a state to be eliminated.

(3) Deriving the regular expression. We now have an equivalent GTG with exactly two states, the initial and the (different) final state. Algorithm 4 describes how we get the final regular expression.

Algorithm 2 convertToGTG

Input: A simple automaton A.
Output: An equivalent GTG A' with only the initial state and the final state.

```
for k : A.getStates() do
    newTransitions ← []
    if k is not initial or final then
        for p, q : A.getStates() do
            if p ≠ k and q ≠ k then
                pq ← getExpressionForElimination(k, p, q, A)
                newTransitions.add(pq)
        remove k and all its incoming and outgoing transition from A
        A.add(newTransitions)
```

Algorithm 3 getExpressionForElimination

Input: The state k to be eliminated, the state p that reaches k, the state q that is reached from p via k, the automaton A.
Output: The regular expression for the transition from p to q without k.

```
pq ← the regular expression for the transition from p to q
pk ← the regular expression for the transition from p to k
kk ← the regular expression for the transition from k to k
kq ← the regular expression for the transition from k to q
return (pq) + (pk)(kk)*(kq))
```

Algorithm 4 deriveFinalExpression

Input: The GTG A' with only the initial state and the final state.
Output: An equivalent regular expression for A'.

```
i ← A'.getInitialState()
f ← A'.getFinalState()
ii ← the regular expression for the transition from i to i
if ← the regular expression for the transition from i to f
ff ← the regular expression for the transition from f to f
fi ← the regular expression for the transition from f to i
return ((ii)*(if)(ff)*(fi))*(ii)*(if)(ff)*
```

2.3 Extension to Probabilistic Automata

The case of transforming an FSA into an equivalent regular expression may be extended to converting probabilistic (finite state) automata (PA) into stochastic regular expressions.

A PA is a tuple (Q, Σ, P, q_0, F) where Q, Σ, q_0 and F are defined analogously to FSAs, and $P : Q \times Q \times \Sigma \to [0, 1]$ is a transition probability function assigning a probability to each transition such that

- $\forall q \in Q \setminus F : \sum_{q' \in Q} \sum_{a \in \Sigma} P(q, q', a) = 1$, and
- $\forall q \in F : \sum_{q' \in Q} \sum_{a \in \Sigma} P(q, q', a) = 0$

We extend the state elimination algorithm to PAs and provide a reference implementation using JFLAP. When a PA is converted to a regular expression, this regex is also enriched with probabilities and rates, generally referred to as stochastic regular expression (SRE). We present the syntax briefly as follows:

$$E := \alpha \;\Big|\; \sum_i E_i[n_i] \;\Big|\; E_1 : E_2 \;\Big|\; E^{*f}$$

where $\alpha \in \Sigma \cup \{\epsilon\}$, $n_i \leq 1000 \in \mathbb{N}_0$, $f \in [0, 1] \subset \mathbb{Q}$ and E_i is a SRE:

- *Atomic Action* α: The action α is an atomic action.

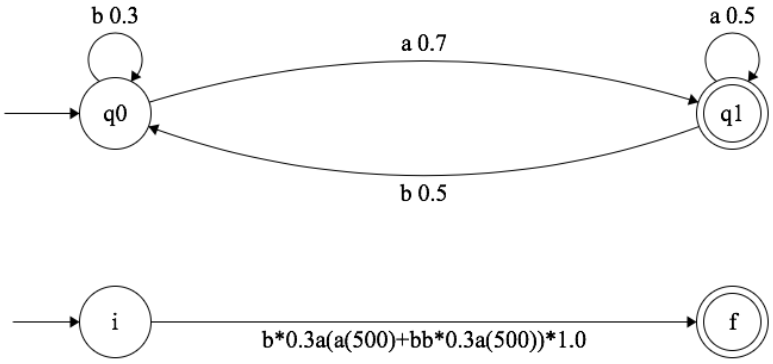


Figure 2: State Elimination example with probabilities.

- *Choice/Sum* $\sum_i E_i[n_i]$: One of the provided terms is randomly chosen. The n_i denotes the choice number for each term. The ith term is chosen with the probability $\frac{n_i}{\sum_j n_j}$.
- *Kleene Closure* E^{*f}: The term E is repeated random number of times. Each iteration occurs with a probability of f.The termination probability is $1 - f$.
- *Concatenation* $E_1 : E_2$: The terms E_1 and E_2 are successively interpreted. As the concatenation is associative, parenthesis may be omitted.

State elimination for PAs follows the same basic principles as for FSAs (see Section 2.2). The extended steps for the probabilistic version of the state elimination appears with the probability calculation. In the beginning, the probabilities of all outgoing transitions are recalculated where the probability of the loops remain unchanged. During the "label join", the probabilities are multiplied, remain unchanged and are uniformed to rates (e.g. Constant=1000) when expressions are concatenated, starred and choiced, respectively. During the recalculation of probabilities, we ignore loop transitions from a state to itself, epsilon and empty transitions. We recalculate the probabilities of the remaining outgoing transitions of each state using the follow steps: First, for all transitions, we add their probabilities to get the new "100%". Second, to get the new probability of each transition, we divide its old probability with the sum computed in the first step.

The reference implementation can be found in the resources provided along with this TTC case. Figure 2 showcases the probabilistic extension of state elimination by means of an example. Note that the probability of the last remaining transition is 1.0, so we omitted it in the figure.

3 Task Description

The basic idea of this TTC case is to rephrase the well-known problems sketched in Section 2 into dedicated model transformation challenges. A generic meta-model for transition graphs which may be interpreted as FSAs, CTGs and PAs is presented in Figure 3. The nodes of a transition graph represent the states while its edges represent the transitions of an FSA, CTG or PA. States are attached unique identifiers. Transitions may be labelled (carrying either symbols or regular expressions) and, in case of PAs, equipped with probabilities. Since virtually all model transformation tools available in the model transformation research community are based on the Eclipse Modeling technology stack, we provide an implementation of the meta-model presented in Figure 3 in EMF Ecore. Of course, other meta-modeling technologies are allowed in solutions as well. This requires a conversion of the input models serving as experimental data in terms of the evaluation (see Section 4), which we provide in EMF format.

3.1 Main Task

The main task is to implement a state elimination for FSAs as a model transformation. In this basic setting, we assume that the FSA given as input is uniform. Thus, the task boils down to iterative state removal until there

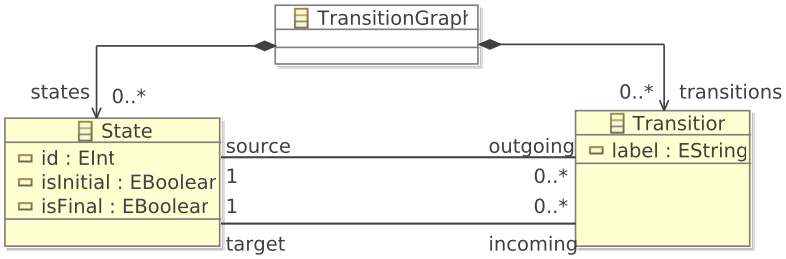


Figure 3: Generic transition graph meta-model for representing FSAs, CTGs and PAs.

is only the initial state and final state left, thereby updating regular expressions attached to transitions.

3.2 Extension 1

As a first additional yet optional task, we ask to uniform the FSM before applying state elimination. The task is to obtain a uniform FSA containing a unique initial state and a unique final state from a non-uniform FSA containing multiple initial and final states.

3.3 Extension 2

A second optional extension is to provide a model transformation taking a probabilistic finite state automation as input and producing a stochastic regular expression as output. Extension 2 can be considered independently from Extension 1.

4 Evaluation Criteria

To evaluate the quality of the proposed solutions, we give a set of quality characteristics which we consider to be relevant for the presented transformation case. The characteristics are inspired by the ISO/IEC 9126-1 standard, which defines quality characteristics in the broader scope of software engineering [BBC+04, JKC04]. We also draw inspirations from more recent work dedicated to the quality of model transformations [SG12, GSS14, LMT14]. We refine each quality characteristic into measurable attributes for each of the tasks presented in Section 3. To obtain concrete measures for their solutions, participants are kindly invited to use the evaluation framework provided with the case resources. This way, most of the measures may be obtained in an automated manner.

4.1 Correctness

A first important and rather obvious quality characteristic is the correctness of the provided solutions.

A transformation solving the main task (see Section 3.1) is correct if the regular expression obtained as a final result of the state elimination is correct. This means that it is semantically equivalent to the regular expression obtained from the reference solution using JFLAP presented in Section 2.2. Although semantically equivalence of regular expressions is decidable in general, we abstain from providing such an automated equivalence proof in our evaluation framework. We choose a testing approach instead, providing sets of positive and negative test cases, i.e., sets of strings to which the produced regular expression should match and should not match. This way, a result is considered to be correct if it passes all tests.

A solution to the first extension task (see Section 3.2) is correct if the produced model represents a uniform FSA which is equivalent to the one provided as input. This can be confirmed by checking equality of the resulting model to the one produced by a reference implementation in our evaluation framework. This correctness test may be automated using the model comparison tool EMF Compare [BP08], which should report no changes between a produced output model and the corresponding reference output model.

Correctness of solutions to the second extension task (see Section 3.3) can be checked in a similar way as for the main task. We use the same test cases as for the main task, ignoring probabilities of the stochastic regular expressions produced as output of the second extension task.

All tasks are scored by means of the provided test cases. A point is given for each test case which passes successfully, points are summarized over all test cases. This means that a correct solution for the main task may

get a maximum number of 11 points, while correct solutions to the first and second extension task may get a maximum of 3 and 9 points, respectively.

4.2 Suitability

Since this TTC case basically rephrases a set of well-known problems into a dedicated model transformation challenge, we are particularly interested in the suitability of the transformation specifications solving these problems. Since general quality metrics from traditional programming languages, such as complexity or modularity, are hardly generalizable for a set of different model transformation languages, we choose a rather simple qualitative criterion which can be referred to as the *level of abstraction*. We follow the approach presented in [LMT14]: The level of abstraction is classified as *High* for primarily declarative solutions, *Medium* for declarative-imperative solutions, and *Low* for primarily imperative solutions. All tasks presented in Section 3 are evaluated in the same way.

Since we acknowledge that classifying the abstraction level of a solution according to its declarativeness may be biased by subjective preferences, the scores that may be obtained for this quality characteristic are relatively low to not too much influence the overall results. All tasks and extension tasks are scored evenly with zero to three points. Zero means the task has not been tackled at all. If it has been solved, solutions may be scored with one, two, or three points, depending on the abstraction level ranging from low, medium, to high.

4.3 Performance

Performance is measured in terms of the *execution time* of a transformation solving the task. This includes the loading of input models and printing or serializing the produced output. The measuring of execution times can be automated by using the evaluation framework provided along with the case description. For the main task and each of its extensions, execution times should be measured for all input models already introduced in Section 4.1.

Performance results of a solution are scored by comparing the execution times with the other solutions. We consider the average execution time for the first ten input models (see *leader3_2* to *leader4_4* in Table 1). The fastest solution gets 14 points (maximum number of points for correctness and suitability of a solution to the main task). The remaining solutions get proportional scores w.r.t. their average execution times.

4.4 Scalability

Since the reference solution using JFLAP suffers from scalability problems when dealing with large input models (cf. Section 4.5), we are particularly interested in whether modern model transformation technologies can improve the scalability of the state elimination. In addition to the FSA models provided for evaluating correctness and scalability, we provide additional larger models comprising up to around one million states. Solutions should figure out which is the *largest model* they are still able to handle. Scalability is only considered for the main task presented in Section 3.1. As we can see in Table 1, the reference solution based on JFLAP scales up to model *leader4 4* containing 812 states.

Scalability is scored by comparing the solutions with each other. The solution which scales up to the largest input model gets a score of six points (analogously to performance). The rest of the solutions get scores which are proportional w.r.t. to the number of participating solutions.

4.5 Results for the Reference Solution

In this section, we present the evaluation results obtained for the reference solution using JFLAP. Please note that we give the results to better illustrate our evaluation criteria, while it is not intended to compare solutions using dedicated model transformation technologies with the reference solution, but mutually among themselves. We use models produced by the Synchronous Leader Election Protocol from the Prism Benchmark Suite [KNP12]. These models are Discrete Time Markov Chains (DTMC), but we interpret them as FSAs an PAs by using the target state names of transitions as labels. In case of FSAs, we ignore probabilities. For example, the DTMC triple (0,1,0.87) is interpreted as a transition from state *s0* to state *s1* assigned with the label "s1".

Table 1 summarizes our experimental results obtained for the reference solution using JFLAP. Participants in this case should use the same input models, all of them are available as EMF models conforming to the generic meta-model introduced in Section 3. In Table 1, sizes of input models are indicated by the number of states, execution times are reported in seconds. The used timeout duration was one hour per model. We ran all

Table 1: Evaluation results obtained for the reference solution using JFLAP. Sizes of input models are indicated by the number of states, execution times are reported in seconds.

	Correct	**Abstraction**	**Exec. Time (s)**	**Scalability**
model name (FSA uniform)				
leader3_2 (26)	yes	low	0,09	
leader4_2 (61)	yes	low	0,14	
leader3_3 (69)	yes	low	0,49	
leader5_2 (141)	yes	low	3,46	
leader3_4 (147)	yes	low	4,37	
leader3_5 (273)	yes	low	58,60	
leader4_3 (274)	yes	low	57,78	
leader6_2 (335)	yes	low	143,12	
leader3_6 (459)	yes	low	461,64	
leader4_4 (812)	yes	low	4786,58	leader4_4
leader5_3 (1050)	-	-	timeout	
model name (FSA non-uniform)				
zeroconf multiple initial states (1296)	yes	low		
zeroconf multiple final states (1296)	yes	low		
zeroconf multiple inital & final states (1296)	yes	low		
model name (Probabilistic Automata)				
leader3_2 (26)	yes	low		
leader4_2 (61)	yes	low		
leader3_3 (69)	yes	low		
leader5_2 (141)	yes	low		
leader3_4 (147)	yes	low		
leader3_5 (273)	yes	low		
leader4_3 (274)	yes	low		
leader6_2 (335)	yes	low		
leader3_6 (459)	yes	low		

experiments on a macOS with 1,6 GHz Intel Core i5 processor and 8 GB of memory (Java 1.8 with the maximum heap size of 2 GB).

Acknowledgments

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution.

References

[BBC+04] P Botella, X Burgués, JP Carvallo, X Franch, G Grau, J Marco, and C Quer. Iso/iec 9126 in practice: what do we need to know. In *Software Measurement European Forum*, volume 2004, 2004.

[BKL08] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of model checking*. MIT press, 2008.

[BP08] Cédric Brun and Alfonso Pierantonio. Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, 2008.

[DK01] Ding-Shu Du and Ker-I Ko. *Problem Solving in Automata, Languages, and Complexity*. John Wiley and Sons, 2001.

[GSS14] Christine M Gerpheide, Ramon RH Schiffelers, and Alexander Serebrenik. A bottom-up quality model for QVTO. In *Int. Conference on the Quality of Information and Communications Technology*, pages 85–94. IEEE, 2014.

[JKC04] Ho-Won Jung, Seung-Gweon Kim, and Chang-Shin Chung. Measuring software product quality: A survey of iso/iec 9126. *IEEE software*, 21(5):88–92, 2004.

[KNP12] M. Kwiatkowska, G. Norman, and D. Parker. The PRISM benchmark suite. In *Proc. 9th International Conference on Quantitative Evaluation of SysTems (QEST'12)*, pages 203–204. IEEE CS Press, 2012.

[LMT14] Kevin Lano, Krikor Maroukian, and Sobhan Yassipour Tehrani. Case study: Fixml to java, c# and c++. In *TTC@STAF*, pages 2–6, 2014.

[PPR96] M. Procopiuc, O. Procopiuc, and S. Rodger. Visualization and interaction in the computer science formal languages course with jflap. pages 121–125, 1996.

[SG12] Eugene Syriani and Jeff Gray. Challenges for addressing quality factors in model transformation. In *Int. Conference on Software Testing, Verification and Validation*, pages 929–937. IEEE, 2012.

An NMF solution to the State Elimination case at the TTC 2017

Georg Hinkel
FZI Research Center of Information Technologies
Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany
hinkel@fzi.de

Abstract

This paper presents a solution to the State Elimination case at the Transformation Tool Contest (TTC) 2017 using the .NET Modeling Framework (NMF). The goal of this case was to investigate, to which degree model transformation technology may help to make the specification of state elimination more declarative and faster than the reference implementation in JFLAP for smaller models.

1 Introduction

State elimination is a well known technique in theoretical computer science to extract the regular expression represented by a given state machine: The regular expression represented by a state machine is extracted by eliminating states and simultaneously constructing regular expressions for the remaining states. This state elimination is repeated until only one initial and an end state is left and the regular expression can be obtained from the edge between these states.

In the scope of the Transformation Tool Contest 2017, this problem has been reified as a model transformation problem[GVPK17] in order to reason on the understandability, but also on the scalability of modern model transformation systems.

In this paper, we present a solution for the state elimination case using the .NET Modeling Framework (NMF, [Hin16]). However, the state elimination case does not fit either of the model transformation languages NTL [Hin13] or NMF Synchronizations [Hin15, HB17]. Both of these languages share the common assumption that it is a characterizing element of a model transformation that there is correspondence between elements of the source model and elements of some target model. Based on this simple assumption, NTL offers unidirectional, imperative model transformations meanwhile NMF Synchronizations is more declarative and supports also bidirectional and incremental transformations on top of NTL.

As both of these languages do not fit, we still have NMF Expressions[1], the incrementalization system used in NMF. This incrementalization system supports the incremental execution of *arbitrary* analyses, but rests on the assumption that the analysis is side-effect free.

Therefore, we present a solution using standard C# based on the model API generated for the given Ecore model using NMF. However, we tried to demonstrate the usage of declarative C#-parts that also reach a good degree of declarativeness.

Our solution is publicly available online[2] but not on SHARE because the used version of NMF requires at least Windows Vista which is not available in SHARE.

In: A. Garcia-Dominguez, F. Krikava and G. Hinkel (eds.): Proceedings of the 10th Transformation Tool Contest, Marburg, Germany, 21-07-2017, published at http://ceur-ws.org

[1] There is no dedicated publication yet, but usage examples can be found in [HH15].

[2] `https://github.com/georghinkel/state-elimination-mt`

The remainder of this paper is structured as follows: Section 2 presents our solution. Section 3 evaluates our solution with regard to performance and conciseness. Section 4 concludes the paper.

2 Solution

Our proposed solution is completely standard general-purpose code. However, the language features of C# make this code already very concise such that we believe there is no further language necessary to further reduce the size of the code.

As the very first step, the solution has to load the input model. For this, we need to transform the Ecore metamodel to the NMeta format used within NMF and generate code for it. Both steps can be done together using the tool `Ecore2Code` that ships with the NMF Nuget Package `NMF-Basics`.

The code generator is aware of multiplicities, containments and opposite references and generates the code appropriately. However, we manually refined the generated NMeta metamodel to set lower bounds of the involved attributes to 1. As a reason, NMF generates nullable types for attributes with lower bound 0 and they are much more cumbersome to use.

The actual deserialization of the input model is as straight forward as shown in Listing 1. We need to create a new repository and load the model into that repository.

```
1 var repository = new ModelRepository();
2 var transitionGraph = repository.Resolve(path).RootElements[0] as TransitionGraph;
```

Listing 1: Loading the input model

Our solution only solves the main task and extension task 1, due to time constraints. In the solution, we first select the initial state and create a new one, if the selected initial state has incoming edges. The implementation is depicted in Listing 2.

```
1  var initial = transitionGraph.States.FirstOrDefault(s => s.IsInitial);
2  if (initial.Incoming.Count > 0)
3  {
4    var newInitial = new State { IsInitial = true };
5    transitionGraph.Transitions.Add(new Transition
6    {
7      Source = newInitial,
8      Target = initial
9    });
10   initial = newInitial;
11 }
```

Listing 2: Creating a new initial state, if the initial state has an incoming state

The solution makes intensive use of the ability of C# to initialize objects inline. This syntax feature is also supported by NMF, at least for single-valued features. We also make use of the fact that NMF respects bidirectional references. Therefore, the assignments in Lines 7 and 8 of Listing 2 do not only set the `Source` and `Target` property of the newly created transition object, they also implicitly add the new transition to the `Incoming` and `Outgoing` collection for the respective states. For this to work, NMF generates special collection implementations for these two collections. This simplifies the manually written code.

Next, we select a final state. Because the logic for this is slightly more complex, we extracted it into a method whose implementation is depicted in Listing 3. At first, we obtain a list of all states that are currently set as final states. If this list only contains a single element, there is no need to change anything and we simply select the state as the new final state. If there are multiple states, we create a new final state and add transitions from the existing final states to that new state. Adding those transitions is done exactly in the same way as in Listing 2.

```
1  var finalStates = transitionGraph.States.Where(s => s.IsFinal).ToList();
2  if (finalStates.Count == 1 && finalStates[0].Outgoing.Count == 0)
3  {
4    return finalStates[0];
5  }
6  else
7  {
8    var newFinal = new State();
9    foreach (var s in finalStates)
10   {
11     transitionGraph.Transitions.Add(new Transition
12     {
13       Source = s,
```

```
      Target = newFinal
    });
  }
  transitionGraph.States.Add(newFinal);
  return newFinal;
}
```

Listing 3: Creating a new final state, if multiple final states exist

The next part of the solution is the removal of states. Before we describe the implementation of this step, we take a step back to review the algorithmics of the state elimination.

Roughly, removing a state implies to update $i \cdot o$ transitions where i is the number of incoming and o is the number of outgoing transitions.

If we converted the automaton to a simple automaton as suggested in the case description, this implies $i = n$ and $o = n$ where n is the number of states (which decreases as states are eliminated). This immediately yields an effort of $\Omega(n^3)$. For very large numbers of n, this is highly problematic. Therefore, we create transitions lazily to reduce the complexity. The example state machines are rather sparse, meaning that the average in and out degree of the states is low in comparison with the number of states.

Still, we have to create or update $i \cdot o$ transitions whenever we delete a state. If we create a transition, this raises the product $i \cdot o$ for later removed states. Therefore, it is reasonable to delete those states with a low product $i \cdot o$ first as we assume that they have the least effect on eliminating other states.

The implementation of this step is depicted in Listing 4. In the first line of this listing, we sort the states by the product $i \cdot o$ before starting to remove states. Sorting the collection can be done highly declarative in C# using the `OrderBy` query operator. Because .NET disallows modifications of a collection while it is iterated, we copy the ordered version into an array.

Eliminating a state, we first check whether there are any self-transitions for the state to be removed and join them. To make this joining more readable, we picked the C# query syntax to select the labels of all self-edges. If there is a self-edge with a non-empty label, we directly star it for the regular expression.

```
foreach (var s in transitionGraph.States.OrderBy(s => s.Incoming.Count * s.Outgoing.Count).ToArray())
{
  if (s == initial || s == final) continue;

  var selfEdge = string.Join("+", from edge in s.Outgoing
                                  where edge.Target == s
                                  select edge.Label);

  if (!string.IsNullOrEmpty(selfEdge)) selfEdge = string.Concat("(", selfEdge, ")*");

  foreach (var incoming in s.Incoming.Where(t => t.Source != s))
  {
    if (incoming.Source == null) continue;
    foreach (var outgoing in s.Outgoing.Where(t => t.Target != s))
    {
      if (outgoing.Target == null) continue;
      var transition = incoming.Source.Outgoing.FirstOrDefault(t => t.Target == outgoing.Target);
      if (transition == null)
      {
        transitionGraph.Transitions.Add(new Transition
        {
          Source = incoming.Source,
          Target = outgoing.Target,
          Label = incoming.Label + selfEdge + outgoing.Label
        });
      }
      else
      {
        transition.Label = string.Concat("(", transition.Label, "+", incoming.Label,
                                         selfEdge, outgoing.Label, ")");
      }
    }
  }

  s.Delete();
}
```

Listing 4: Removing states

Before we actually remove the state, we iterate through all of its incoming and outgoing transitions. For each pair of incoming and outgoing transition, we create a new transition that avoids the state to be deleted. The label of that new transition is the same as following the `incoming` edge to the state marked for deletion, then

making cycles in that state (only in case there actually are cycles) and then following the `outgoing` edge to the target state.

This iteration through every pairs of incoming and outgoing transitions can be regarded as imperative code, but still has many declarative elements. For example, we do not need to specify *how* the collections are iterated or filtered and neither do we specify how a matching shortcut transition is found. Though these elements only provide a very thin abstraction layer, we think that they make the code quite understandable.

Finally, we delete the state that we previously picked from the list of all states. Through the generated code of NMF, this operation boils down to a simple call to a `Delete` method. It will remove the state from the collection it is contained in, i.e. the collection of states in the transition graph. Further, it resets all references to that state[3]. Because transitions are independent of states in the metamodel with respect to the containment hierarchy, the incoming and outgoing transitions will still exist after the state has been deleted. However, after the deletion operation, they are no longer connected to the deleted state, but the `Source` and `Target` reference are simply empty, i.e. point to `null`.

We could go on and delete those transitions as they are no longer valid. However, in our experiments, we came to the conclusion that it is better to leave these transitions in there because the effort to delete them (removing an element from an ordered collection is an $O(n)$-operation!) is larger than the additional overhead they imply for the traversing incoming or outgoing transitions of other states.

Finally, we return the possible paths from the initial state to the target state. This can be done similarly as above. The implementation, this time as a one-liner, is depicted in Listing 5.

```
1 return string.Join("+", from edge in initial.Outgoing where edge.Target == final select edge.Label);
```

Listing 5: Calculating the overall regular expression

3 Evaluation

Our solution is very concise. Besides generated code for the metamodel and one line of metamodel registration, the entire solution consists of 102 lines, of which 31 lines are either blank or only contain braces.

Model	# Elements	Regex size	Time to transform (ms)	Correct	JFLAP (ms)
leader3_2	61	33	192	✓	90
leader3_3	166	95	193	✓	490
leader3_4	359	210	206	✓	4,370
leader3_5	672	398	207	✓	58,600
leader3_6	1,135	675	218	✓	461,640
leader3_8	2,631	1,571	298	✓	–
leader4_2	139	76	227	✓	140
leader4_3	630	354	219	✓	57,780
leader4_4	1,881	1,067	253	✓	4,786,580
leader4_5	4,492	2,558	395	✓	–
leader4_6	9,221	5,262	831	✓	–
leader5_2	315	172	197	✓	3,460
leader5_3	2,344	1292	286	✓	–
leader5_4	9,513	5,267	890	✓	–
leader5_5	28,544	15,833	5,277	✓	–
leader6_2	735	398	207	✓	143,120
leader6_3	8,248	4,487	739	✓	–
leader6_4	45,865	24,979	17,210	✓	–
leader6_5	173,194	94,408	395,616	✓	–
leader6_6	515,077	280,865	4,356,603	✓	–
leader6_8	2,886,813	–	–	–	–

Table 1: Execution times for the example input models

[3]NMF can also raise an event to notify clients that an element was removed from the containment hierarchy but this event is not raised in this case as no clients are subscribed to it.

The execution times for the test models are depicted in Table 1. The execution times have been recorded on an Intel i7-4710MQ clocked at 2.50Ghz on a system with 16GB RAM. For the largest model *leader6_8*, the solution ran out of memory. The recorded times include the initialization of NMF, initializing metamodels, loading model instances and computing the regular expression. It does not include serializing the generated regular expression to a file or testing the expression for the example inputs.

The times show a very good performance and scalability. For most of the models, the regular expression can be computed in less than a second. In particular, for the example model *leader4_4*, the largest that the reference solution in JFLAP is able to process, our solution is faster than JFLAP by more than three orders of magnitude. Only for the largest models, the transformation takes a lot of time.

4 Conclusion

In this paper, we discussed how the .NET Modeling Framework can be used to solve state elimination reified as model transformation task. Because there is no correspondence between source and target model required, the transformation languages in NMF are not well suited for this problem. Hence, our solution is essentially using general-purpose code, although the implementation of bidirectional references and automatically resetting references for deleted model elements makes the implementation more concise.

The performance results for the smaller models are very good. In particular, the solution is able to process all except for the very largest models.

References

[GVPK17] Sinem Getir, Duc Anh Vu, Francois Peverali, and Timo Kehrer. State Elimination as Model Transformation Problem. In Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, editors, *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences*, CEUR Workshop Proceedings. CEUR-WS.org, July 2017.

[HB17] Georg Hinkel and Erik Burger. Change Propagation and Bidirectionality in Internal Transformation DSLs. *Software & Systems Modeling*, 2017.

[HH15] Georg Hinkel and Lucia Happe. An NMF Solution to the TTC Train Benchmark Case. In Louis Rose, Tassilo Horn, and Filip Krikava, editors, *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences*, volume 1524 of *CEUR Workshop Proceedings*, pages 142–146. CEUR-WS.org, July 2015.

[Hin13] Georg Hinkel. An approach to maintainable model transformations using an internal DSL. Master's thesis, Karlsruhe Institute of Technology, October 2013.

[Hin15] Georg Hinkel. Change Propagation in an Internal Model Transformation Language. In Dimitris Kolovos and Manuel Wimmer, editors, *Theory and Practice of Model Transformations: 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings*, pages 3–17, Cham, 2015. Springer International Publishing.

[Hin16] Georg Hinkel. NMF: A Modeling Framework for the .NET Platform. Technical report, Karlsruhe Institute of Technology, Karlsruhe, 2016.

Transformation of Finite State Automata to Regular Expressions using Henshin

Daniel Strüber

University of Koblenz and Landau
strueber@uni-koblenz.de

Abstract

We present a solution to the State Elimination Case of the Transformation Tool Contest 2017, based on the Henshin model transformation language. The main task is to convert a finite state automaton (FSA) into a regular expression; two extensions include the simplification of FSAs as well as the conversion of probabilistic FSAs. The distinguishing feature of our solution is its largely declarative specification, based on Henshin's concepts of rules and composite units for specifying modifications and control flow. We present the results of the performance and scalability evaluation from the provided benchmark suite. Similar to the reference solution, our solution did not scale up to all test models in the benchmark suite; yet, compared to this solution, it achieved a speed-up by two orders of magnitude for some of the larger models.

1 Introduction.

Finite state automata (FSAs) are a modeling concept with many practical applications, including program analysis, pattern matching, speech recognition and behavioral software modeling. In its basic form, a FSA comprises a set of states and a set of labelled transitions between the states; some of the states are designated as initial and final states. An important feature of FSAs is their fundamental relationship to regular expressions. While converting a regular expression into a corresponding FSA is easy, the opposite task is more sophisticated; current solutions suffer from nonlinear algorithmic complexity [GVP+17].

The TTC 2017 *state elimination* case [GVP+17] aims to study how transformation tools may contribute to more efficient solutions. The case description specifies three tasks—a main task and two extensions—dealing with three kinds of FSAs: (i) *basic* FSAs with at least one initial and at least one final state, (ii) *simple* FSAs with exactly one initial and exactly one final state, and (iii) *probabilistic* FSAs, in which transitions are annotated with probabilities. Simple FSAs are a subset of basic ones, whereas probabilistic FSAs generalize basic ones. The main task is to transform a simple FSA to a regular expression. Extension 1 involves transforming a basic to a simple FSA, and extension 2 deals with transforming a probabilistic FSA to a stochastic regular expression.

In this paper, we present a complete solution based on the Henshin model transformation language [ABJ+10]. Henshin is a graph-based transformation language providing support for the declarative specification of in-place model transformations. The basic features of Henshin's tool set are a suite of editors and an interpreter kernel; more sophisticated features include code generation for parallel graph pattern matching and support for various transformation analyses.

In: Antonio Garcia-Dominguez, Georg Hinkel, Filip Křikava (eds.): Proceedings of the 10th Transformation Tool Contest, Marburg, Germany, 21-07-2017, published at http://ceur-ws.org

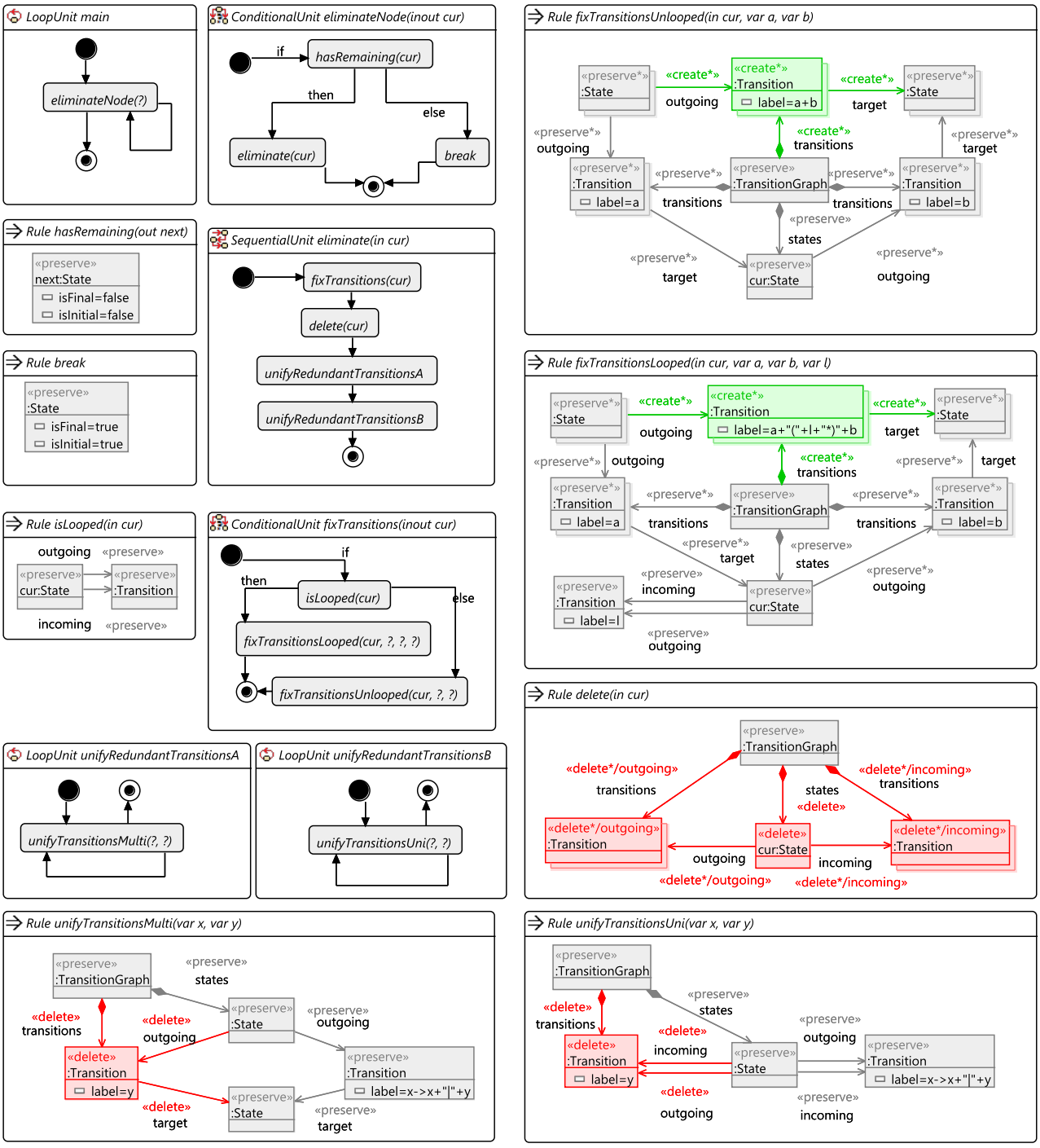


Figure 1: Solution for main task: state elimination in FSAs.

A distinguishing feature of Henshin is its expressive visual syntax which aims to facilitate usability during transformation development [SBG+17]. To provide a largely declarative solution, we use Henshin's concepts for the specification of control flow and model modifications. Control flow is specified using *composite units* that orchestrate the execution of a set of sub-units and rules, for example, by applying them in sequential order or in a counted loop. Model modifications are specified using *rules*, expressing basic match-and-change patterns.

2 Solution

In what follows, we present our solution in detail. We start with the main task and continue with the two extension tasks. The solution is available via GitHub at `https://github.com/dstrueber/stateelim-henshin`. The URL of the associated SHARE image is made available on the GitHub site.

2.1 Main Task: Converting FSAs to Regular Expressions

Our solution to the main task follows the state elimination algorithm presented in [GVP+17]. We implemented this algorithm using 8 rules, which are orchestrated in a control flow of 6 composite units, as shown in Fig. 1. As required by the specification, the produced result is a string representing the output regular expression. On the way to this result, we manipulate state labels using string operations, thus enabling a compact specification.

The overall goal is to remove all non-initial and non-final states from the FSA. To achieve this goal, our starting point is the loop unit *main* which executes its inner unit *eliminateNode* as often as possible. The aim of *eliminateNode* is twofold: first, it checks whether any state is to be removed and, if this is the case, identifies it. This is done using the rule *hasRemaining*, a simple rule for matching a non-initial and non-final state. Second, if such state is found, it is stored in the parameter *cur* (for *current*) and passed to the sequential unit *eliminate*. The *eliminate* unit proceeds in the following steps: first it "fixes" the incoming and outgoing transitions of state *cur* by replacing them, then it deletes *cur*, and finally, it unifies any redundant transitions arising in the process.

To fix the transitions, conditional unit *fixTransitions* first checks if *cur* is a looped state, that is, a state with a loop transition. Depending on the result, one of the rules *fixTransitionsUnlooped* and *fixTransitionsLooped* is triggered, which only differ in their treatment of the loop. Both rules make use of rule-nesting: they have a *kernel rule*—the "flat" states in the visual syntax—and a *multi-rule*—the "layered" states with asterisk signs. When applying either rule to the input model, the kernel rule is matched first, and then, based on the identified match, the multi-rule is applied with a *for-all* semantics, that is, as often as possible. Based on the provided state *cur*, all pairs of incoming and outgoing transitions are identified, and a new transition is created for each of these pairs. The label of the newly created transition is composed of the labels of the pair (plus in the loop case, the loop label), which are stored and propagated using the variables a, b (and l). Rule *delete* works in a similar manner by deleting state *cur* via the kernel rule, and its adjacent transitions via separate multi-rules.

As a result of the *fixTransitions* step, the model may temporarily contain pairs of states with multiple transitions. These redundant transitions are now unified, using separate loop units for cases where the transitions are non-loops and loops, respectively. Each of these units calls a rule which nondeterministically identifies redundant pair of transitions, removes one of the transitions, and joins its label with the label of the remaining transition.

The *main* unit terminates when there are no remaining nodes to be eliminated. To escape from the loop, a *break* rule is required, which specifies an impossible pattern (here, a node that is initial and final at the same time), so it always evaluates to *false*. After the whole process, there is only the initial and the final state left, possibly with various transitions between them. To obtain the final regular expression in a concise manner, a short Java routine puts together their labels according to Listing 4 in [GVP+17].

2.2 Extension 1: FSA simplification.

The conversion of a basic FSA into a simple one involves two steps: first, if there are multiple initial states, these states become non-initial states with an incoming ϵ-transition from a newly created unique

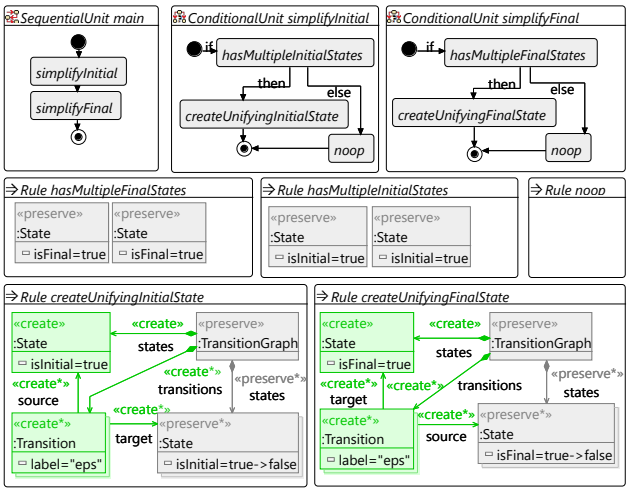


Figure 2: Solution for extension 1: Simplifying a FSA.

initial state. Second, if there are multiple final states, these states become non-final states with an outgoing ϵ-transition to a newly created final initial state. Our solution represents these steps via two sub-units *simplifyInitial* and *simplifyFinal*, which are applied in sequential order using a sequential *main* unit.

The *simplifyInitial* unit uses a rule *hasMultipleInitialStates* to check if the simplification treatment becomes necessary, and, if, this is the case, performs the simplification using the rule *createUnifyingInitialState*. Specifically, rule *hasMultipleInitialStates* checks whether two separate initial states exist in the input FSA. Rule *createUnifyingInitialState* again uses the concept of rule-nesting to achieve a for-all semantics. The kernel rule specifies the creation of an additional initial state. The multi-rule matches all earlier initialState, turns them into non-initial states and creates an incoming ϵ-transition for each of them.

The treatment of final states based on unit *simplifyFinal* is completely dual.

2.3 Extension 2: Converting probabilistic automata.

The solution for extension 2 extends the solution for the main task with small modifications concerning the treatment of labels and probabilities. Since the original case description did not specify how the labels and probabilities are to be computed (in fact, the probabilities in the computed regular expression are ignored in the correctness evaluation), we followed the specification of the reference implementation (`https://github.com/sinemgetir/state-elimination-mt/`). Details about the reference implementation's specification were obtained in our communication with the case authors.

According to this specification, the conversion process includes a preprocessing of the input automaton where the probabilities of the outgoing transitions for each state are recalculated, such that (i) loop and ϵ transitions are ignored, (ii) the probabilities of the other transitions are added up to derive "the new 100%", and (iii) each transition obtains its original probability divided by "the new 100%" as its new probability. Fig. 3 shows our transformation for implementing this specification: Unit *recalculateProbs* specifies that its two contained rules are applied in sequential order. Rule *recalculateProbsTemps* uses rule nesting to iterate over all outgoing non-loop/empty/ϵ transitions of all states, so that their probabilities are stored in variable a. The sum of all values of a is stored in a newly created `Trace` object, using Henshin's `Aggregations` helper class to compute the sum. Rule *recalculateProbsUpdate* performs the same iteration as before to update the probabilities of the involved transitions. Given the old probability b, the new value is b/a, where a is the aggregate percentage stored in the `Trace` object. In this process, the `Trace` objects become obsolete and are consequently deleted.

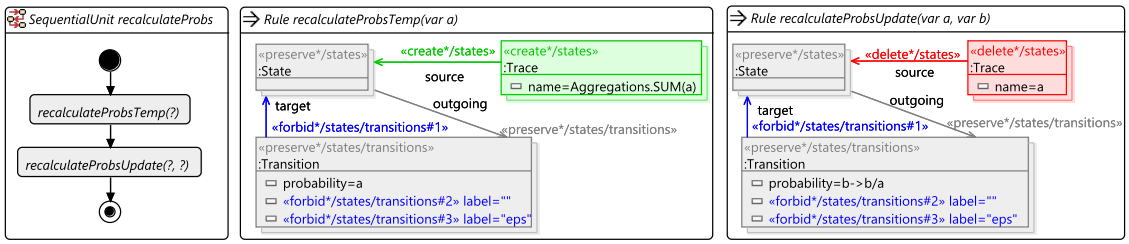


Figure 3: Solution for extension 2: pre-processing step.

The remaining conversion process is the same except for the label and probabilities handling, where we adhere to the specification: *"when we concatenate two labels we multiply their probabilities. When we 'or' two labels, we add their probabilities."*

For example, in the rule *fixTransitionsLooped*, for the transition beingly newly created, the probability needs to account for the probabilities of the original transitions, and the label needs to represent the probability of the loop transition. To this end, we modified this rule as shown in Fig. 3. During the matching process, we now store the labels as well as the probabilities of the matched transition in variables ($\{a, b, l\}$ and $\{pa, pb, pl\}$, respectively). In the multi-rule, we then use these variables in the label and probability of each newly created transition. The label includes the probability of the loop transition, the probabilities is made up from the probabilites of the input transitions.

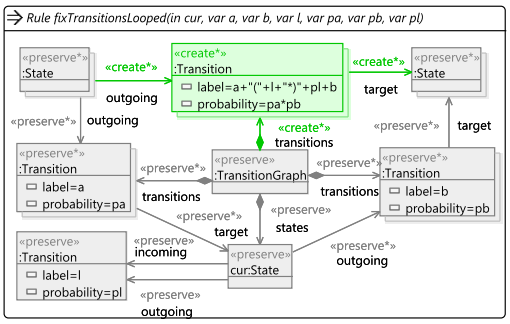


Figure 4: Solution for extension 2: excerpt.

3 Evaluation

In this section, we apply the evaluation criteria from the case description to our solution. The solution was executed on a Windows 10 system (Intel Core i7-5600U, 2.6 GHz; 8 GB of RAM, Java 8, 2 GB Xms). We ran the experiment with a timeout duration of max. 1 hour per model, following the reference solution.

Correctness. When applied to the provided test models, our solution produced correct results in all cases. For the main task and extension 2, we used the provided benchmark framework for verifying our solution. We slightly extended the framework so it supports the execution against user-specified timeout durations.

For extension 1, unfortunately, we could not follow the evaluation process described in the case description: The FSA data structure used by the reference implementation does not support input FSAs with more than one initial state, rendering it infeasible as a baseline for the correctnes check. Instead, we performed a light-weight validation by checking if the numbers of states, initial states, final states, and transitions changed as expected.

Suitability. With our solution, we aimed at providing a primarily declarative solution. We achieved this goal by specifying all parts of the state elimination algorithm (Listing 2 and 3 in [GVP+17]) using Henshin's declarative rule and control flow concepts. In addition, our solution includes two minor imperative parts, written in Java: A driver to trigger the execution of the Henshin interpreter with the specification, and a part to convert the output of state elimination to an expression (Listing 4 in [GVP+17]).

Performance. Table 1 shows the performance measurements in comparison to the available data for the reference solution. For the three smallest test models, we observed a slow-down. For all of the remaining models, we observed an increasingly growing speed-up, amounting to an order of two magnitudes in the case of the largest input model *4_4*. A complexity analysis to substantiate this observation is left to future work.

Model (#states)	Execution time (sec.) Reference	Henshin
3_2 (26)	0.09	**0.21**
4_2 (61)	0.14	**0.25**
3_3 (69)	0.49	**0.29**
5_2 (141)	3.46	**0.78**
3_4 (147)	4.37	**0.77**
3_5 (273)	58.60	**2.93**
4_3 (274)	57.78	**2.32**
6_2 (335)	143.12	**3.45**
3_6 (459)	461.64	**10.09**
4_4 (812)	4786.58	**48.15**

Table 1: Performance measurements.

Scalability. The largest case where our solution produced a result within one hour was *4_5*, taking 10:12 minutes for 1933 states. Given more time, we can transform larger models as well, e.g., *5_4* in 120:41 minutes for 4244 states.

4 Outlook

Conceptually, the state elimination case is a highly interesting scenario for our ongoing work on the variability of model transformations [SRA+16, SS16]. It features variability in two dimensions: variability in the language of the input automata (plain and probabilistic FSAs), and variability in the solution artifacts of the transformations (repeatedly, we handle a *looped* case differently from an *unlooped* case). We intend to use this scenario as a case study for extending our work towards language-level variability.

References

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: advanced concepts and tools for in-place EMF model transformations. *Model Driven Engineering Languages and Systems (MoDELS)*, pages 121–135, 2010.

[GVP+17] Sinem Getir, Duc Anh Vu, Francois Peverali, Daniel Strüber, and Timo Kehrer. State Elimination as Model Transformation Problem. *10th Transformation Tool Contest (TTC)*, 2017.

[SBG+17] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. Henshin: A Usability-Focused Framework for EMF Model Transformation Development. In *International Conference on Graph Transformation (ICGT)*, pages 196–208, 2017.

[SRA+16] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer, and Jennifer Plöger. RuleMerger: automatic construction of variability-based model transformation rules. In *Int. Conf. on Fundamental Approaches to Software Engineering (FASE)*, pages 122–140, 2016.

[SS16] Daniel Strüber and Stefan Schulz. A tool environment for managing families of model transformation rules. In *International Conference on Graph Transformation (ICGT)*, pages 89–101, 2016.

The SDMLib Solution to the TTC 2017 State Elimination Case

Alexander Weidt
Kassel University
alexander.weidt@uni-kassel.de

Albert Zündorf
Kassel University
zuendorf@uni-kassel.de

1 Introduction

The Transformation Tool Contest 2017 State Elimination Case [SECase] asks for the reduction of a Finite State Automaton into an equivalent regular expression. Basically, one replaces states with transitions that combine the regular expressions attached to incomming and outgoing transitions of the removed state. This paper shows the SDMLib [SDMLib] solution to this case.

2 The rules

At first, the example automaton has to be normalized: The automaton must have exactly one inital state and exactly one final state.

public void uniformInitial (TransitionGraph graph)

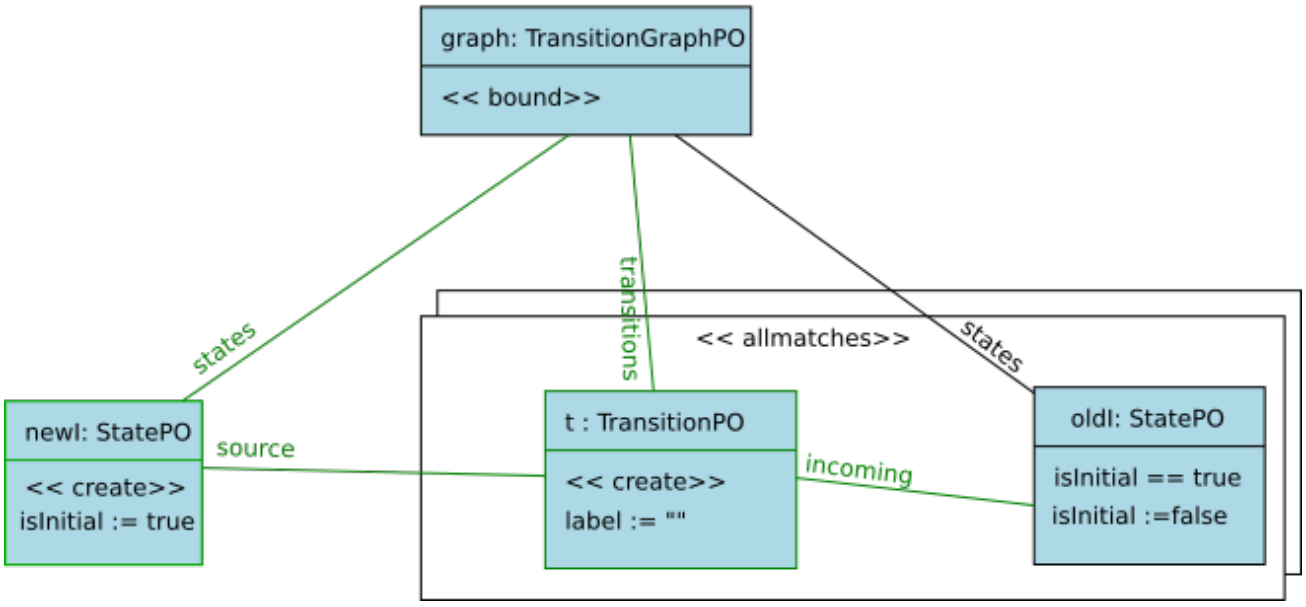


Figure 1: Adding single intial state

Figure 1 shows in graphical notation the SDMLib transformation rule that adds a new initial state to the current automaton and connects this new state to all previous initial states with empty transitions.

In: A. Garcia-Dominguez, F. Krikava and G. Hinkel (eds.): Proceedings of the 10th Transformation Tool Contest, Marburg, Germany, 21-07-2017, published at http://ceur-ws.org

Transformation `uniformInitial()` starts by matching the pattern object `graph` to the `TransitionGraph` object passed as parameter. Next, pattern object `newI` creates a new initial state. Then the sub-pattern rendered within two stacked boxes is executed. The pattern object `oldI` searches for old initial `State` objects attached to our `TransitionGraph` via a `states` link and with an `isInital` attribute equal to `true`. The `<<allmatches>>` sterotype attached to the sub-pattern shown in Figure 1 causes the sub-pattern to be executed for each match of `oldI`. Thus, for each old initial state a new `Transition t` with empty label is created. The new transition connects the new initial state `newI` and the current old initial state. In addition, the `isInitial` attribute of `oldI` is set to `false`.

We use a similar rule called `uniformFinal()` to introduce a single final state into the current automaton. The case description also proposes to add theta transitions between all states that are not yet connected via a transition. Then the reduction algorithm is save to assume that each pair of states is connected via a transition. Similarly, multiple transitions between two states are combined into a single transition. This reduces the number of case distinctions within the algorithm but increases the runtime. Thus, the SDMLib approach does not perform these normalization steps but deals with these cases in its transformation rules.

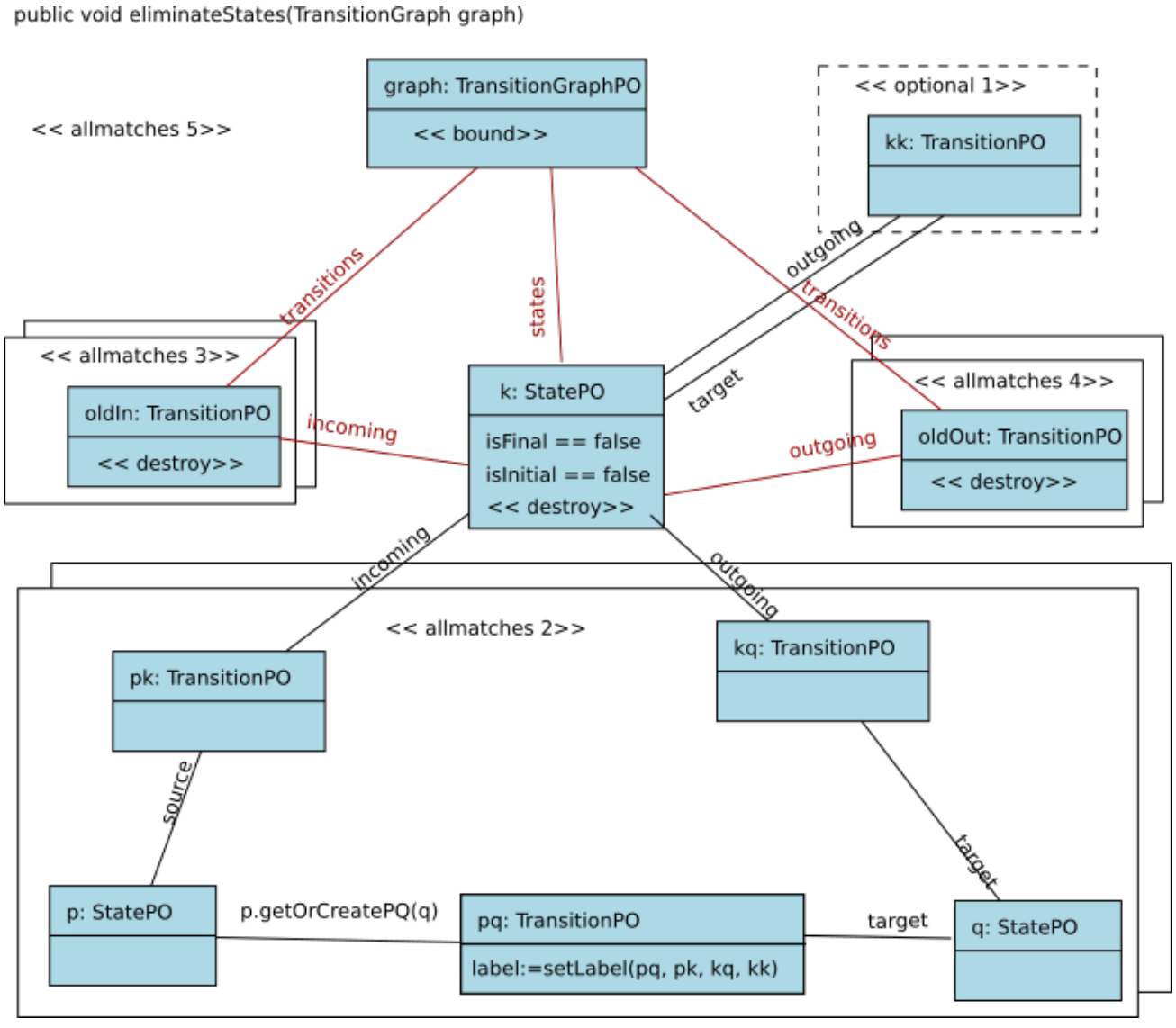


Figure 2: Eliminate States

Figure2 shows our `eliminateStates()` transformation. Again we start with a pattern object `graph` that matches the root of our finite automaton, passed as parameter. Next we look up `State k`, the state that shall be

replaced by transitions. `State k` shall neither be `isFinal` nor `isInitial`. Then, the sub-pattern `<<allmatches 2>>` looks up a `State p` that has a `Transition pk` going to `State k` and a `State q` that is reached from `State k` via `Transition kq`. The sub-pattern `<<allmatches 2>>` is iterated, that means it is executed for all possible matches. For each match, we call the pseudo path expression `p.getOrCreatePQ(q)`. This operation tries to find an existing `Transition pq` that connects the `States p` and `q`. The path expression creates such a `Transition`, otherwise. Operation `setLabel(pq, pk, kq, kk)` generates a label for transition `pq` computed as `pq.label + pk.label kk.label* kq.label`. The cases that `pq.label` is empty or that `kk` is null or that `kk.label` is empty are handled, specifically. Once all matches of sub-pattern `allmatches 2` are handled, sub-pattern `allmatches 3` and `allmatches 4` remove all old `Transitions` leading to or leaving `State k`, respectively. Finally, `State k` itself is destroyed. The whole top level pattern is iterated itself through the stereotype `<<allmatches 5>>`. Thus, transformation `eliminateStates` eliminates all states except the unique initial and final states introduced at the beginning. These initial and final states are then connected by exactly one `Transition` and this `Transition` has a regular expression as label that is equivalent to the original finite automaton.

3 Results

Figure 3 shows some measurements for our solution. We have executed the transformation on a laptop with about 1.6 Ghz. Intel I5 processor giving the Java process 6 GB heap space. We noticed that the evaluation of the regular expression uses a lot of time, thus we show our results with and without the time used for the evaluation of the resulting regular expression. We compare our results with the JFLAP algorithm times reported in the call for solution. We are waiting for the results of the other teams.

	SDMlib mit Evaluation	SDMlib ohne Evaluation	JFLAP
leader3_2	0.0385514360	0.052127528	0.09
leader4_2	0.0866275960	0.081792921	0.14
leader3_3	0.0993657960	0.063173148	0.49
leader5_2	1.1262002710	0.239402017	3.46
leader3_4	0.1605883720	0.098499445	4.37
leader3_5	0.5394560420	0.294551276	58.6
leader4_3	35.3694409170	7.921832035	57.78
leader6_2	24.6645100420	3.029549227	143.12
leader3_6	0.9771501780	0.771568491	461.64
leader4_4	8452.815857866	too big for memory	timeout

Figure 3: Execution Times (in seconds)

Overall, the solution was quite straight forward although our state elimination transformation needs quite a number of sub-pattern and we use some helper methods to handle different cases, uniformly.

References

[SDMLib] SDMLib - Story Driven Modeling Library *www.sdmlib.org* May, 2017.

[SECase] Sinem Getir, Duc Anh Vu, Francois Peverali, and Timo Kehrer State Elimination as Model Transformation Problem *www.transformation-tool-contest.eu/TTC_2017_paper_4.pdf* May, 2017.

Solving the State Elimination Case Study using Epsilon

Mohammadreza Sharbaf
m.sharbaf@eng.ui.ac.ir

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

Bahman Zamani
zamani@eng.ui.ac.ir

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

Abstract

The transformation of a finite state automaton into an equivalent regular expression is a challenging topic which is presented in TTC 2017. This paper presents a solution to State Elimination case using the Epsilon framework.

1 Introduction

The State Elimination case study includes both a model to model and a model to text transformation, which aims to transform Finite State Automata (FSA) or Finite State Machines (FSM) into equivalent Regular Expressions (RE). This is a challenging and expensive transformation. In this paper, we provide a solution to the transformation problem which eliminates states in an iterative manner. Our solution is based on random selection of a state and eliminating it from FSA. The solution is available as a Github repository[1].

Our solution is implemented using Epsilon[2] invoked from a Java application. Epsilon is an extensible set of languages and tools for model management which is built atop the Eclipse Modeling Framework (EMF) [1]. Epsilon can be used to perform all model management tasks, including in-place and out-place model transformations. Epsilon is an appropriate tool for solving the above mentioned case study that involves model modification, before generating equivalent regular expressions based on finite stated automata.

The remainder of this paper is structured as follows. Section 2 provides an introduction to the fundamental parts of Epsilon which are used for solving this problem. Section 3 provides our solution to the case study. Evaluation of the proposed solution is presented in section 4. Finally, section 5 summarizes our findings.

2 Epsilon Overview

Epsilon is a Java-based comprehensive framework which includes several languages for model management tasks such as model transformation, code generation, model refactoring and validation [2]. Following are the Epsilon languages which are used in our solution:

- **Epsilon Object Language (EOL):** EOL is an imperative language which is the core of Epsilon and can be used as a standalone generic language or used within other Epsilon model management languages. For tackling the state elimination problem, we have benefited from EOL in creation and deletion of states and transitions.

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] https://github.com/MSharbaf/TTC2017-StateElimination

[2] https://www.eclipse.org/epsilon

- **Epsilon Transformation Language (ETL):** ETL is a declarative model-to-model transformation language, which inherits the imperative feature of EOL to perform complex transformations. It accepts multiple input models and is able to generate multiple target models. However, for this solution, we only used a single source and target model. Each ETL program consists of several ETL modules. A module can contain any number of transformation rules, operations and optional *pre* and *post* blocks, which are executed before and after the transformation rules, respectively.
- **Epsilon Generation Language (EGL):** EGL is a template-based language for text generation, which facilitates the construction of model-to-text transformations. Each EGL template consists of static and dynamic parts that reuses the EOL mechanism for defining declarative operations.

The aforementioned languages are used to create scripts (Epsilon codes) that take one or more models and transform them into another model or text. The ETL and EGL scripts can be executed with launch configuration facilities or alternatively invoked directly from a Java program.

3 Solving the State Elimination Case Study using Epsilon

In TTC 2017 a state elimination case study [3] based on the state elimination algorithm for FSA has announced. This case study includes a main task and two extensions. The main task is to convert a uniform FSA to the equivalent RE. An FSA is uniformed if it has a unique initial state with no incoming transitions and a unique final state with no outgoing transitions. A regular expression is a mathematical expression which specifies the language generated or accepted by a uniform FSA. The first extension is to transform a non-uniform FSA into a uniform one. The second extension is to convert a probabilistic FSA to a stochastic RE.

The state elimination algorithm for the main task takes a uniform FSA and generates an equivalent regular expression. In this paper the Epsilon framework is used to solve the main task and the first extension. In the following sections the description of the solutions are provided in more detail.

3.1 Overview of the Main Task Solution

In order to solve the main task problem, the transformation specification takes the uniform FSA as input model, and generates the equivalent regular expression as output model. The transformation eliminates each state with corresponding transitions and adds a new equivalent transition for predecessor and successor states in each iteration, until there is only one initial and one final state. In this solution the transformation chain includes two main phases:

- **Phase 1:** Transformation of a uniform FSA to a generalized transition graph (GTG)
- **Phase 2:** Transformation of a GTG to a final regular expression

In the following section each phase is explained in more detail.

Phase 1: Transformation of a uniform FSA to a GTG

This phase is a model-to-model transformation, which is divided into three steps as follows:

- **Step 1:** Creating a GTG, with initial and final states, and an unlabeled transition between them
- **Step 2:** Selection and deletion of intermediate states (i.e., states other than initial and final) of FSA and their incoming and outgoing transitions
- **Step 3:** Labelled transition between initial and final states in GTG, equivalent to all transitions of the input FSA

These steps are implemented in an ETL module, which consists of ETL rules and sets of operations. In the following more details of this transformation is provided.

Step 1: Creation of GTG

In this step a GTG with a single initial and final state, and a single outgoing and incoming transitions is generated. Listing 1 illustrates the implementation of this rule in ETL. In this rule the initial and final states of input model is given to the transformation, and it then generates the model in the output with initial state, final state and a transition between them. Calculation of transition label is performed in the second and third steps.

```
rule StateAddition
  transform S1: input!State to S2: output!State{
  guard : S1.isInitial or S1.isFinal
  S2.id = S1.id ;
  S2.isInitial = S1.isInitial ;
  S2.isFinal = S1.isFinal ;
  if(S1.isInitial){
    trans.source = S2 ;
    S2.outgoing ::= trans ;
  }
  if(S1.isFinal){
    trans.target = S2 ;
    S2.incoming ::= trans ;
  }
}
```

Listing 1: ETL Rule demonstrating the *State Addition* operation

Step 2: Deletion of Intermediate states and transitions

In this step all the intermediate states and related transitions are eliminated and new labeled transitions corresponding to them are generated. The implementation of this step is provided in Listing 2. The EOL operations and expressions are used here for identification and deletion of the elements in the source model. In order to calculate the label of transition it is required to delete each intermediate state of FSA (for instance K), and transform its transitions into a new one with respect to its predecessor (P) and successor states (Q). These states are detected by examining the incoming and outgoing transitions for the intermediate state.

In addition it is required to check the existence of loop, i.e., a transition with an identical initial and final state, in the selected state (K) and to identify the direct transitions between predecessor (P) and successor (Q) states. Following that all the transitions between predecessor (P) and successor states are eliminated and new transitions are generated. Finally a transition is labeled according to the $(\alpha_{pk}\alpha^{*}{}_{kk}\alpha_{kq})$ formula [3].

```
var lbl_PKQ = "" ;
var flag_lbl_PKQ = false ;
for(tp in P.outgoing.select(tr|tr.target == K)){
  for(tq in Q.incoming.select(tr|tr.source == K)){
    if(flag_lbl_PKQ == true)
      lbl_PKQ += "+" ;
    lbl_PKQ += tp.label + lbl_K_Self_Loop + tq.label ;
    flag_lbl_PKQ = true ;
  }
  delete tp ;
}
```

Listing 2: EOL expression to calculate the label of new transitions corresponding to eliminated states

Step 3: Labelling the unlabeled transition in the GTG

In this step a single initial and final state with one or more transitions between them are remained in FSA. Additionally, it may be possible for each state to have loop. The transformation in this step generates a label equivalent to all the labels of transitions existed or generated in the previous step based on the state elimination rule. This label is then attached to the new transition between initial and final states in the GTG. The EOL statements are used for implementation of this part in the post condition of ETL module.

Phase 2: Transformation of a GTG to a final regular expression

In this phase the generated GTG is transformed to a textual file that only contains a final regular expression, which is implemented with an EGL script as indicated in listing 3. EGL has been used to transform model to textual artifact and automatically generate a textual file.

3.2 Overview of the First Extension Solution

For the main task it was assumed that the input FSA is uniformed. However, in the first extension it is possible to have more than one initial and final states. The extension is a model-to-model transformation. In our solution

```
[%
  var sb := new Native("java.lang.StringBuilder");
  sb.append(Transition.allInstances.selectOne(s|s.label.isDefined()).label);
%]
[%=sb.toString()%]
```

Listing 3: EGL template for transforming a GTG into an equivalent RE

Table 1: Abstraction level for our solution

Element		**Abstraction level**
Main Task		
Phase 1	Transformation of a Uniform FSA to a GTG	Medium
Phase 2	Transformation of a GTG to a final regular expression	High
Overall solution		Medium
Extension 1		
Overall solution		Medium

we have used ETL and EOL to solve this extension. In order to apply the state elimination algorithm to this cases, we should add a new initial state and change its *isInitial* property to true. Following that, a transition with a null label should be inserted from the new initial state to each original initial states and their *isInitial* property is changed to false. For the elimination of final states into a single state, we have followed a similar procedure.

In the proposed solution an operation is added to the ETL transformation for transforming non-uniform to uniform FSA, which generates a single initial and final state with its corresponding transitions. This operation is called in the *pre*-condition of the ETL module written for this extension.

3.3 Execution of the Solution

The proposed solution in this paper requires Epsilon Core for execution of EOL, ETL and EGL scripts. Therefore, we use the Eclipse Distribution which contains most of the required prerequisites of Epsilon. The complete solution uses the ANT Epsilon tasks to execute transformation chains in the specific workflow and enables the user to run it from the Eclipse toolbar.

4 Evaluation

In the case study description [3] a set of quality characteristics with its measurable attributes are defined for systematic evaluation of each solution. In the following sections the evaluation of our solution according to the correctness, suitability, performance and scalability are provided.

4.1 Correctness

According to the evaluation criteria, a solution for the main task is correct when the RE obtained as a final result of the State Elimination passes all sets of positive and negative test cases. Each test case is a set of strings to which the produced Regular Expression should match or should not match, which can be checked by the Evaluation Framework provided in the case description. Regarding the result of Evaluation Framework, our main task solution is correct and passed all the positive and negative test cases.

Additionally, the correctness of the solution for the first extension task generates a uniform FSA which is equivalent to the input FSA with some initial and final states. The Epsilon solution in this paper generates a correct output model for each input model in the extension part.

4.2 Suitability

The suitability in this case study is evaluated by the level of abstraction of transformation language, which is High for primarily declarative language and Low for primarily imperative language. Although our solution combines imperative EOL statements with ETL and uses EGL script as textual file generator which are primarily declarative, the overall level of abstraction in this solution is *Medium*. Table 1 shows the abstraction level for each phase of the main task, and an overall value for solutions of the main task and first extension.

4.3 Performance

The solution performance should be measured as the seconds spent for executing two transformation phases with the provided input models. This includes the loading of input FSA models and generating a text file as the equivalent regular expression output. The following table shows the execution time for our solution in seconds. We measured execution time automatically by Ant builder. This measures are the average execution time of ten consecutive executions of Ant builder with the specified input model. All tests were carried out on a standard Windows 7 PC using an Intel® Core™ i7 with 3.6 GHz processor and 8GB RAM.

Table 2: Evaluation results for our solution includes execution times and scalability level

Model name (FSA uniform)	Correct	Execution Time (s)	Scalability
leader3_2 (26)	yes	0.1476	
leader4_2 (61)	yes	0.1617	
leader3_3 (69)	yes	0.1769	
leader5_2 (141)	yes	0.2252	
leader3_4 (147)	yes	0.2398	
leader3_5 (273)	yes	0.3877	
leader4_3 (274)	yes	0.3639	
leader6_2 (335)	yes	0.4528	
leader3_6 (459)	yes	0.71	
leader4_4 (812)	yes	1	
leader5_3 (1050)	yes	2	
leader3_8 (1059)	yes	2	
leader4_5 (1933)	yes	7	
leader6_3 (3759)	yes	23	
leader4_6 (3962)	yes	29	
leader5_4 (4244)	yes	31	
leader5_5 (12709)	yes	341.6	
leader6_4 (20884)	yes	920.8	
leader6_5 (78784)	yes	9834.34	
leader6_6 (234210)	yes	102602	Leader6_6

4.4 Scalability

According to the evaluation criteria, scalability is a model with a maximum number of states which is correctly converted to an equivalent regular expression. According to the table 2 the transformation generates result for all the test cases specially for model leader6_6 with 234210 states.

5 Conclusion

In this paper we used Epsilon languages to transform a finite state automaton into an equivalent regular expression. The suitability of transformation languages in this solution is medium for implementation of the main task and first extension. Transformation generates correct results for all the test cases and the result of execution is extensively better than the results provided in the case description. Additionally, the scalability of transformation is high as it managed to execute the largest test case with 234210 states.

References

[1] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. 2008.

[2] D. Kolovos, L. Rose, R. Paige, and A. Garcıa-Domınguez, *The Epsilon Book*. 2016.

[3] S. Getir, D. A. Vu, F. Peverali, and T. Kehrer, "State Elimination as Model Transformation Problem," 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.