ISSN 1884-0760

GRACE TECHNICAL REPORTS

On the use of Bidirectional Transformations for Translational Semantics

Florent Latombe and Soichiro Hidaka

GRACE-TR 2014–01 Apr 2014

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

On the use of Bidirectional Transformations for Translational Semantics *

Florent Latombe
IRIT, Université de Toulouse,
Toulouse, France

Soichiro Hidaka
National Institute of Informatics
Tokyo, Japan

April 10, 2014

Abstract

In this work, we aim at defining the semantics of Executable Domain-Specific Modeling Languages (xDSMLs) in a translational way by using Bidirectional Transformations written with the GRoundTram framework[1], where the backward transformation of GRoundTram enables us, between each step of execution of the target model conforming to the target xDSMLs (where the state of the model changes), to propagate the changes back to the state of the model conforming to the source xDSML, in coherence with the forward transformation that provides the translational semantics. It is possible to define most elements composing an xDSML using Model-Driven tools like the Eclipse Modeling Framework (EMF), however this creates a technical gap with GRoundTram which is relevant to graphs. We propose a workflow and the first steps of its implementations which allows us to transform graphs into models and vice-versa for the implementation of translational semantics of an xDSML.

1 Introduction

This section introduces the context of this work in subsection 1.1, the goal of this work in subsection 1.2 and the running example used for experimentation in subsection 1.3.

*The work presented in this technical report was done between the 7th of January 2014 and the 26th of March 2014 during an international internship at the National Institute of Informatics (Tokyo, Japan) under the supervision of Soichiro Hidaka.

[1] `http://www.biglab.org/`

1.1 Context

1.1.1 Bidirectional Transformation

The work presented in this document relies on the GRoundTram Framework[5, 7] developed by the BiG Project[2] of the NII. For more information about GRoundTram, see the project website at `http://www.biglab.org/` or contact Soichiro Hidaka. For bidirectional transformations in general, please refer to [2, 8].

1.1.2 Definition of Executable Domain-Specific Modeling Languages

The context of this work is the definition of **Executable Domain-Specific Modeling Languages**, hereafter refered to as xDSMLs. xDSMLs are, in particular, **Domain-Specific Languages** or DSLs. There exists a lot of litterature concerning DSLs and it is not the aim of this document to go into details about the practice of DSL design. What is essential to understand, is that DSLs are usually very small languages, typically not Turing-complete and designed for a very small audience (usually experts in a given field). Readers who want to know more about DSLs can have a great overview in Voelter et al.'s book *DSL Engineering*. xDSMLs are also, in particular, **Domain-Specific Modeling Languages** or DSMLs. DSMLs are a flavour of DSLs where the goal of the language is to model systems and their behaviours. Empirical studies have proven that the use of DSMLs by the industry is growing and requiring new techniques and tools [9]. Lastly, xDSMLs are also *Executable*, usually through interpretation or generation (akin to compilation) and thus need clear semantics.

Defining a DSL [4] is usually done by defining an Abstract Syntax (AS) which are the concepts and relations manipulated by the domain at hand, one or several Concrete Syntaxes (CS) in order to create instances (programs), and a mapping towards a Semantic Domain (SD). The AS can easily be defined in a Model-Driven Engineering (MDE) way by defining a MetaModel (MM) using, in our case, the Ecore editor of the Eclipse Modeling Project[3]. Concrete Syntaxes are usually textual (using xText[4] for instance) or graphical (using Sirius[5] for instance). The mapping towards the SD can be realized in different ways which will not be detailed here but can usually be categorized among one of these three families: Operational Semantics, Denotational Semantics, Axiomatic Semantics.

1.1.3 GEMOC: On the Globalization of Modeling Languages

GEMOC[6] is an open initiative exploring the necessary breakthrough in software languages to support a global software engineering. GEMOC

[2] `http://www.biglab.org/`
[3] `http://eclipse.org/modeling/`
[4] `https://www.eclipse.org/Xtext/`
[5] `http://www.eclipse.org/sirius/`
[6] `http://www.gemoc.org`

investigates tools and methods in software language engineering (SLE) for the design and implementation of collaborative, interoperable and composable modeling languages. In particular, GEMOC aims at providing technical and methodological means to reify the concurrency of languages at the language level [1]. Details about GEMOC will not be provided in this document as they are not directly relevant to the issue of translational semantics using bidirectional transformations, but any question can be redirected to Florent Latombe.

1.2 Main Goal

The methodology and tools developed in GEMOC for defining xDSMLs focus on defining the semantics of xDSMLs through an Operational Semantics manner. However this technical report presents an approach towards defining the semantics of an xDSML through a Translational Semantics manner. In other words, our goal is to provide the semantics of an xDSML (refered to as *source* xDSML) using the already well-defined semantics of another xDSML (refered to as *target* xDSML). Thus the need for a transformation from source xDSML to target xDSML, also refered to as the *Forward Transformation* in the Bidirectional Transformations context. However, simply being able to execute a language using another language's semantics is not enough. We want to be able to display (textually or through a graphical animation) the execution while it is happening. We also want to be able to take reactive systems into account, where the user or an environment (in the case of GEMOC, usually other xDSMLs) will have the opportunity to make choices on the flow of the execution. In order to be able to do that, the AS of the source xDSML must be able to encode the state of a model conforming to the xDSML. We also need the same capacity in the target xDSML's AS. On top of that, between each step of execution (where the state of the executed model changes), we need to propagate the changes in the state of the model conforming to the target xDSML up to the state of the model conform to the source xDSML, in coherence with the Forward Transformation mentioned before. This second transformation is also refered to as the *Backward Transformation* in the Bidirectional Transformations context.

Why translational semantics ? Implementing the semantics of an xDSML in a translational way can sometimes feel more expensive and time-consuming than in an operational way. This may be due to the fact that one needs to understand two domains and their respective semantics instead of one in order to be able to write the transformation between source and target xDSML. One also needs to be familiar with the Transformation Language used. However, translational semantics provide huge steps forward in reusability and modularity. In particular, GEMOC proposes to write the semantics of an xDSML in an operational way while reifying the concurrency at the language level. This architecture provides a lot of advantages in terms of reusability and analysis of the concurrency models used by xDSMLs, but at a cost: mapping the model of concurrency (as defined in GEMOC) with the operations triggering changes in the model comes with its own methodology and tools. Therefore, being able

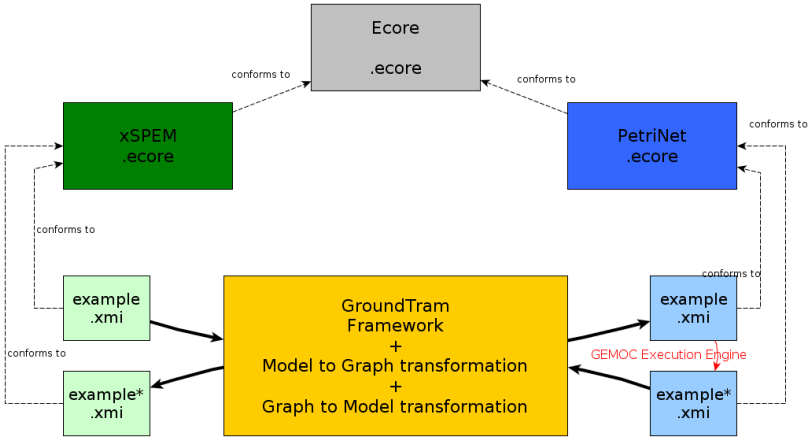

Figure 1: Expected Workflow for the example detailed in subsection 1.3

to reuse existing xDSMLs written using the GEMOC methodology and tools can provide many advantages: one still has the advantages of having a reified model of concurrency, while not having to deal with the most complex parts of designing an xDSML in the GEMOC methodology.

1.3 Example

The running example used for experimentation is the translational semantics of a simplified version of SPEM[7] into Petri Nets.

Source xDSML SPEM is a modeling language used to model processes. The MetaModel used is visible on figure 2. A `Process` owns several `ProcessElements`, which are either `WorkDefinitions` or `WorkSequences`. A `WorkSequence` has a *kind* which is either `startToStart`, `startToFinish`, `finishToStart`, `finishToFinish` which represents the nature of the relation between its two referenced `WorkDefinitions`, *predecessor* and *successor*. A `WorkDefinition` knows its outgoing and incoming `WorkSequences` and has an *ExecutionState* which represents the current state of the `WorkDefinition` during execution, and is either `Ready`, `Running`, or `Finished`.

Target xDSML Petri Nets are a mathematical modeling language used for the description of distributed systems. In our flavour of Petri Net, whose MetaModel is visible on figure 6, a `PetriNet` is composed of several `PetriNetElements`, which are either `Nodes` or `Arcs`. `Arcs` are of kind either "normal" or "read" (when read arcs are traversed, marking of the source `Place` is not decremented) and link two `Nodes` (references *source* and *target*). But these two `Nodes` must be of different natures: both *source*

[7] `http://www.omg.org/spec/SPEM/2.0/`

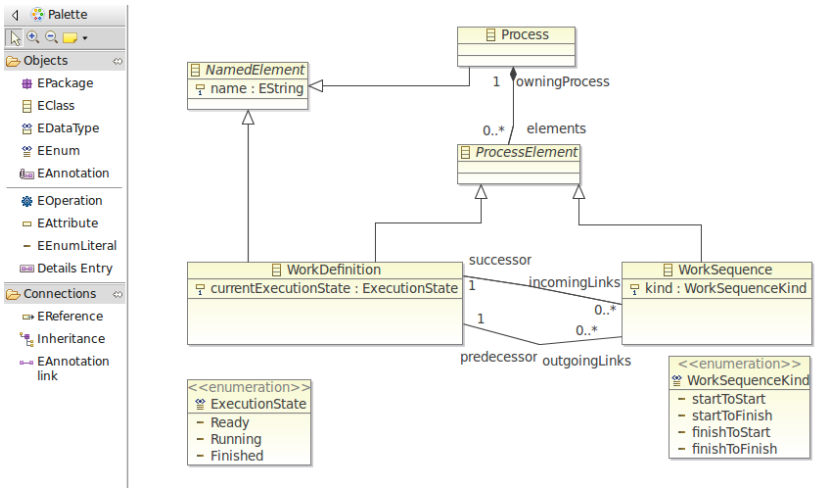

Figure 2: MetaModel of the xSPEM xDSML

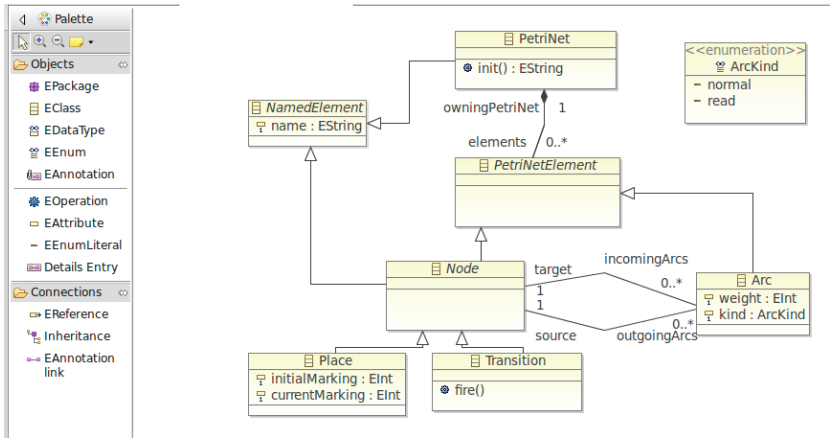

Figure 3: MetaModel of the PetriNet xDSML

and *target* cannot be a `Place` at the same time, or be a `Transition` at the same time. `Place` and `Transition` are the two concrete types of `Node`s. `Place`s have an *initialMarking* and a *currentMarking* attributes encoded as integers. `Node`s know about their outgoing and incoming `Arc`s.

Previous work Faiez Zalila has created and used a bidirectional transformation between SPEM and PetriNet in a previous internship at NII for model-checking purposes [13]. However, the main differences with the work presented here lie in the fact that the MetaModels used are different, as we are using bidirectional EReferences (marked as EOpposite) in our MetaModels whereas Faiez Zalila was not ; and the source and target

languages SPEM and PetriNet are only examples in our case and could have been different, whereas Faiez Zalila was using Petri Nets to do model checking and therefore relied heavily on properties inherent to Petri Nets.

1.4 Rest of the document

Besides the current section (1), which introduces the work and its context, section 2 presents the current state of the implementation, while section 3 presents the upcoming work left to do to complete the propose workflow. Section 4 will conclude this tehnical report. Lastly, the Appendix section A is about the dead-end met when trying to use Bidirectional Transformations for a slightly different topic: reversable pattern-matching of events.

2 Implementation

In this section are shown the current implemented elements. The overall goal can be seen on figure 4. This figure is not to be taken literally ; it is only there to illustrate the kind of structure we are supposed to have in the end. Notably, the approach using generic higher-order transformations [10] and their outputs (transformations from model to graph and transformation from graph to model, with fixed MetaModels of respectively input and output) is not implemented. Instead the implementation uses a transformation which takes a MetaModel as input (interpretative approach instead of the generative approach presented in the figure).

Note: The repository containing the sources will be made available online[8] *once a more elaborated version of the work presented in this report has been published.*

[8] `http://www.prg.nii.ac.jp/projects/tsbx/`

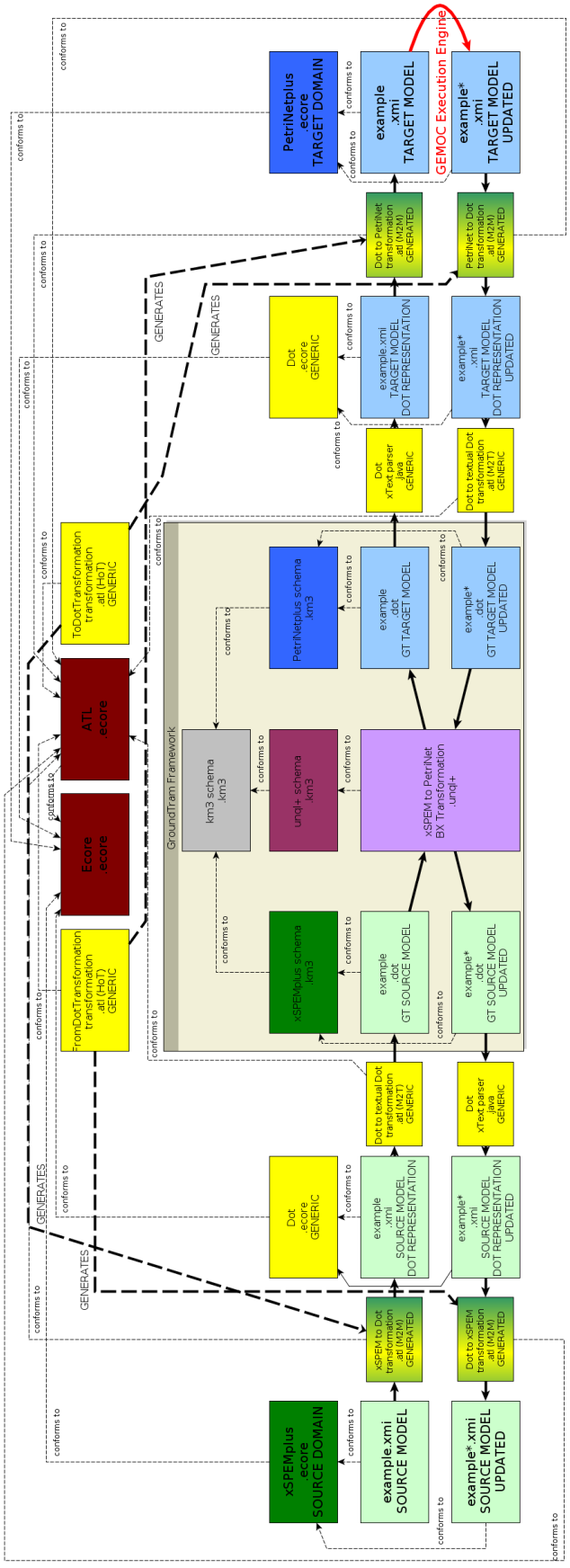

Figure 4: Zoomed-in view of the workflow for our example

2.1 Starting Point

The starting point of the current work is at the *GT SOURCE MODEL* element of figure 4, that is the graph representation of the source model in a format the GRoundTram framework accepts as input: thus the Dot [3] graph with labels exclusively on edges. Let us use an example, refered to as "sample 1" in the repository[9]. It consists in a `Process` named "TalkDrinkWorkExample" composed of 3 `WorkDefinition`s and 4 `WorkSequence`s. The `WorkDefinition`s are named "Talk", "DrinkCoffee" and "Work". Their *ExecutionState* is NULL as they have not been initialized yet. In our modeled process, we may only start drinking coffee if we have started talking (`WorkSequence` startToStart from "Talk" to "DrinkCoffee") and we may finish talking only if we have finished drinking coffee (`WorkSequence` finishToFinish from "DrinkCoffee" to "Talk"). We may also only start working if we have finished both talking and drinking coffee (`WorkSequence`s finishToFinish from "Talk" to "Work" and from "DrinkCoffee" to "Work").

2.2 Bidirectional Transformation

The bidirectional transformation written using GRoundTram is specific to both xSPEM and PetriNet. It takes a graph corresponding to an xSPEM model as input (the starting point mentioned earlier in subsection 2.1) and produces the equivalent PetriNet model as output. Both graphs can only have labels on edges as this is the input and output format used by the GRoundTram framework.

More precisely, a `Process` is transformed into a `PetriNet`. A `WorkDefinition` named "x" is transformed into 4 `Place`s: "x_ready", "x_running", "x_started", "x_finished" with initialMarking of 1 for "x_ready" and 0 for the other `Place`s and currentMarking of NULL for all `Place`s ; 2 `Transition`s: "x_start" and "x_finish" ; and 5 arcs: from "x_ready" to "x_start", from "x_start" to "x_running", from "x_start" to "x_started", from "x_running" to "x_finish", from "x_finish" to "x_finished". The left half of Figure 6 depicts these components.

A `WorkSequence` is transformed into an `Arc` of kind "read" from a `Place` to a `Transition`. Let us consider a `WorkSequence` of nature `startToStart` from the `WorkDefinition` named "x" to the `WorkDefinition` named "y". The element corresponding to this `WorkSequence` is an `Arc` between `Place` "x_started" and `Transition` "y_start" (depicted by a dotted line in Figure 6). Had the `WorkSequence` been of nature `startToFinish`, then it would have connected "x_start" with "y_finished". If it had been of nature `finishToStart` it would have connected "x_finished" with "y_start" and if it had been of nature `finishToFinish` it would have connected "x_finished" with "y_finish".

More technically, in our UnQL transformation, we first create a `PetriNet` with the name of the input `Process` using a *Select* query. Inside this query there is another *Select* query which is in charge of creating all the `PetriNetElement`s to place in the containment reference *elements* of `PetriNet`. First, the easier part is creating the 4 `Place`s,

[9] src/examples/test.transformation/src/sample1/inputs

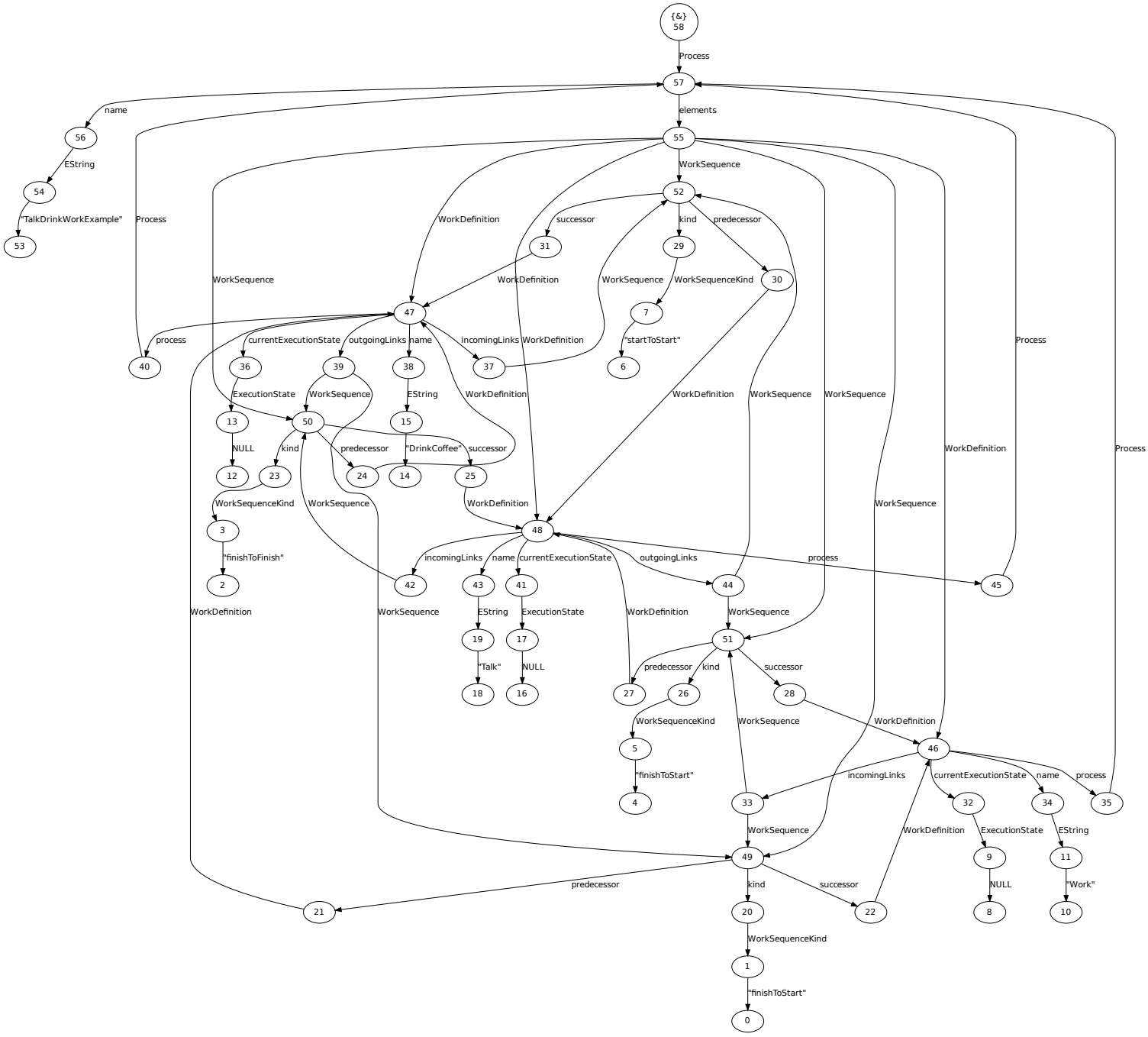

Figure 5: Graph representation of the input xSPEM model

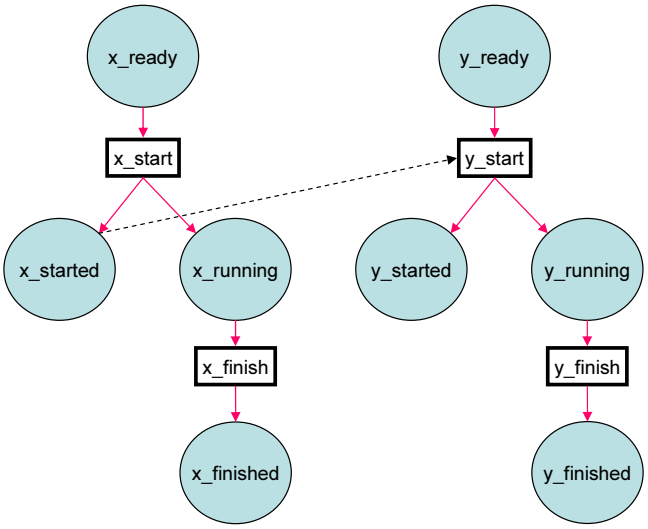

Figure 6: PetriNet components created from a `WorkSequence` of nature `startToStart` from `WorkDefinition` x to y

2 `Transitions` and 5 `Arcs` corresponding to each `WorkDefinition`. Afterwards is the harder part which consists in creating an `Arc` which, depending on its kind, will have a different *source* and a different *target*. For this, another *Select* query is created. This query looks up, in the elements created in the first part, the `Place` which has the following name: name of the *predecessor* `WorkDefinition` of the `WorkSequence` concatenated with either "_started" or "_finished" depending on the kind of the `WorkSequence`. The `Place` found will be the *source* of the `Arc` we are creating. Then, we have to look up for the `Transition` which will be the *target* of the `Arc` we are creating. This `Transition` must have the following name: name of the *successor* `WorkDefinition` of the `WorkSequence` concatenated with either "_start" or "_finish" depending on the kind of the `WorkSequence`.

The compositionality of the UnQL language achieves the capturing of twig-like patterns to retrieve subgraphs encoding required model elements like `WorkDefinitions`, as well as traversals of intermediate results created during the first part of the above transformation.

There is a limitation in the semantics of GRoundTram which causes problems in creating loops in the output graph if these loops were not already present in the input graph. For instance, it is not possible to have, in the output graph, a nice representation of an EReference from an EObject to its container (in our case, from a `PetriNetElement` to its owning `PetriNet`). This problem also occurs for bidirectional EReferences like the EReferences *source*and *target* of type `Node` in the class `Arc`, whose opposites are *incomingArcs* and *outgoingArcs* in the class `Node`. This is due to how the "select", "replace", "extend" and "delete" queries of GRoundTram are implemented. Therefore, for now we put a placeholder

label on the edge which should lead to the EObject referenced. We choose to use the label "EOppositeReference". The value of this EReference will be computed later on in the transformation of subsection 2.4.

After the transformation, we have an UnCAL graph which we are able to transform into a Dot [3] graph. Unfortunately it is too big to be contained in this document, but can be found in the repository[10]

2.3 Transformation from graph to model

The Dot graph resulting from the output of the GRoundTram transformation is still a valid classical Dot graph. Thus, tools based on classical Dot format can be used. In our case, we will use the parser for Dot textual concrete syntax generated by Xtext so as to obtain an XMI model conform to the Dot MetaModel. The Dot MetaModel is independent from PetriNet or xSPEM so this parser will always be applicable to the output of a GRoundTram transformation. This parser can be found in the repository[11] too.

2.4 Transforming from a Dot model into a PetriNet model

Now, we need to transform the XMI model conform to the Dot MetaModel into an XMI model conform to our PetriNet MetaModel. A transformation written in Java[12] takes as input the XMI model conform to the Dot MetaModel and creates the PetriNet XMI model equivalent. In particular, it decodes the bidirectional EReferences. EReferences which have an EOpposite but have no value attached to them in the graph are computed last, and by going through the whole graph looking for the opposite EReference we are able to re-construct the value of all the missing EReferences. For example, consider that in the input Dot model the value of the EReference *source* of a given `Arc` is present, then in order to add this `Arc` to the *outgoingArcs* EReference of the correct `Node`, we have to look through all the created `Node`s and compare them to the value of the *source* EReference of our `Arc`. When the match is found, then we can set the value of the EReference *outgoingArcs* (with respect to its multiplicity).

Afterwards, we have found that this approach was similar to the one Wider describes in subsections 4.1 and 4.2 in [12].

3 Upcoming work

3.1 Improvements on the existing transformation

The transformation mentioned in subsection 2.4, which we can abusively call a transformation from graph to model, is not entirely decorelated from the target xDSML used as example. Therefore, it could use some slight improvements to make it truly generic and only parameterized by

[10] src/examples/test.transformation/src/sample1/outputs
[11] src/core/org.gemoc.translational_semantics.dot.xtext/src/org/gemoc/translational_semantics/dot/xtext/Parser.java
[12] src/examples/test.javaparsing/src/test/javaparsing/DotXmiModelToPetriNetModel.java

the MetaModel of the target xDSML. Most of the algorithm though is still unreliant on the specific features of the MetaModel and thus these improvements should not be too big.

3.2 Transformation from model to graph

Our starting point was that of a xSPEM model encoded as a graph, but the true starting point should be of the model itself. Therefore we also need a transformation which, parameterized by the MetaModel of the source xDSML, will be able to encode models conform to the MetaModel of the source xDSML into graphs which can server as input for the bidirectional transformation.

3.3 Usage

Once the work mentioned in previous subsections 3.1 and 3.2 have been made, the generic workflow shown on figure 4 should be the following:

1. The input model conforming to the source xDSML MetaModel is transformed into a Dot model using the transformation from model to graph mentioned in 3.2
2. This Dot model is pretty-printed into the textual concrete syntax of Dot
3. The Forward Transformation of the GRoundTram Bidirectional Transformation is applied to the Dot graph obtained
4. The result of the transformation is parsed using the xText parser mentioned in 2.3
5. The output Dot model is then transformed into a model conform to the target xDSML MetaModel using the transformation mentioned in 2.4
6. A step of execution is made on this model (by the GEMOC Execution Engine in our case), which typically changes the value of some EAttributes and EReferences in the model.
7. The updated model is then transformed back into a Dot model, using the same transformation as in step 1 but parameterized this time by the target xDSML MetaModel
8. The Dot model is then pretty-printed into a Dot graph as in step 2
9. The Backward Transformation of the GRoundTram Bidirectional Transformation is applied to the Dot graph obtained
10. The output Dot graph is then parsed by the parser, as in step 4
11. The Dot model obtained is then transformed back into a model conform to the source xDSML MetaModel using the same transformation as in step 5 but parameterized this time by the source xDSML MetaModel

This workflow must then be repeated for every step of the execution resulting in a change in the model being executed.

4 Conclusion

The semantics of an Executable Domain-Specific Modeling Language designed in a Model-Driven way can be defined in a translational way using Bidirectional Transformations implemented in GRoundTram. The gap between the technical spaces of models and edge-labeled graphs can be closed by using a set of transformations which can ultimately be reduced down to only 2 transformations specific to either the source xDSML or the target xDSML used. These two transformations can be obtained either by generation using Higher-Order Transformations, or by interpretation by providing a MetaModel as argument.

The first of these transformations is a transformation which, given a MetaModel (of our source xDSML or of our target xDSML), can transform models conform to the given MetaModel into models conforming to the Dot MetaModel (with labels exclusively on edges so as to comply with GRoundTram formats). This transformation is then used to transform the original model we want to execute into a model corresponding to an edge-labelled graph so as to be pretty-printed and served as input to the GRoundTram transformation ; this same transformation is also used to transform the executed model conform to the target xDSML after it has been modified by a step of execution so as to be pretty-printed in order to be the input of the Backward Transformation.

The second transformation is a transformation which, given the MetaModel of the source xDSML or of the target xDSML, can transform Dot graphs (as before, with labels exclusively on edges) into a model conforming to the given MetaModel. This transformation is used first on the Dot edge-labeled graph obtained after the Forward Transformation so as to recreate a model conforming to the target xDSML from a graph ; it is then used as the very last step of the proposed workflow on the Dot graph resulting from the parsing of the output of the Backward Transformation so as to recreate, from the given graph, a model conforming to the source xDSML.

We have provided the first steps of implementations for this second transformation and used it successfully on the example of defining the semantics of xSPEM using PetriNets. Upcoming work should build on that and the full workflow proposed should be completed, so as to provide a fully implemented solution to the gap existing between the models and graphs technical spaces. Once refined, this workflow could be implemented in the GEMOC tools so as to provide a way to define the semantics of new xDSMLs using previously well-defined xDSMLs.

Acknowledgments

Many thanks to Professor Zhenjiang Hu of the National Institute of Informatics. Also many thanks to Quang Minh Tran and the rest of the BX Team of NII for their inputs and discussions during this internship. Thanks to the NII International Internship Program for providing this collaboration opportunity.

References

[1] Benoit Combemale, Julien Deantoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry, and Robert France. Reifying Concurrency for Executable Metamodeling. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *SLE - 6th International Conference on Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 365–384, Indianapolis, IN, États-Unis, 2013. Springer. doi: 10.1007/978-3-319-02654-1_20. URL `http://hal.inria.fr/hal-00850770`. CNRS PICS Project MBSAR (http://gemoc.org/mbsar).

[2] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *International Conference on Model Transformation (ICMT 2009)*, pages 260–283. LNCS 5563, Springer, 2009.

[3] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz and Dynagraph - Static and Dynamic Graph Drawing Tools. In *GRAPH DRAWING SOFTWARE*, pages 127–148. Springer-Verlag, 2003.

[4] D. Harel and B. Rumpe. Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64–72, Oct 2004. ISSN 0018-9162. doi: 10.1109/MC.2004.172.

[5] S. Hidaka, Zhenjiang Hu, K. Inaba, H. Kato, and K. Nakano. Groundtram: An integrated framework for developing well-behaved bidirectional model transformations. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 480–483, Nov 2011. doi: 10.1109/ASE.2011.6100104.

[6] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing Graph Transformations. In *ACM SIGPLAN International Conference on Functional Programming*, pages 205–216. ACM, 2010.

[7] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations. *Progress in Informatics*, (10):131–148, March 2013. URL `http://dx.doi.org/10.2201/NiiPi.2013.10.7`.

[8] Zhenjiang Hu, Andy Schürr, Perdita Stevens, and James F. Terwilliger. Dagstuhl seminar on bidirectional transformations (bx). *SIGMOD Record*, 40(1):35–39, 2011.

[9] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, New York, NY, USA, 2011. ACM.

ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985858. URL http://doi.acm.org/10.1145/1985793.1985858.

[10] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the use of higher-order model transformations. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 18–33. Springer-Verlag, 2009.

[11] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. ISBN 978-1-4812-1858-0.

[12] Arif Wider. Implementing a Bidirectional Model Transformation Language as an Internal DSL in Scala. In K. Selçuk Candan, Sihem Amer-Yahia, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference*, number 1133 in CEUR Workshop Proceedings, pages 63–70, Aachen, 2014. URL http://ceur-ws.org/Vol-1133#paper-10.

[13] Faiez Zalila and Soichiro Hidaka. Facilitating verification results feedback on DSM verification context using bidirectional model transformation. unpublished manuscript, February 2013.

A Pattern matching of events using Bidirectional Transformations

Context We wanted to be able to create pattern matching of events from low-level events into higher-level events. We also needed to be able to reverse these patterns, so that according to another specification referencing higher-level events we could intervene on the lower-level events. For examples one such pattern matching could be

$$e_1 + e_2 \rightarrow F_1$$

meaning that the coincidence of an occurrence of event e_1 with an occurrence of event e_2 would trigger an occurrence of event F_1 ;

$$e_2|e_3 \rightarrow F_2$$

meaning that the union of an occurrence of event e_2 with an occurrence of event e_3 would trigger an occurrence of event F_2. Then, in another specification, we needed to be able to compute the reverse of F_1 or F_2.

Problem Although there are difficulties in how to define the reverse of $e_2|e_3$, this was not the main problem here. The main problem was that we needed to be able to compute the reverse of, say, F_2 **without having had an occurrence of** F_2 **actually happen beforehand**. This simple need (both specifications relative to the same higher-level events not being executed in sequence) is incoherent with the way the backward semantics of GRoundTram are done, relying on traces of the forward transformation [6].

Conclusion Bidirectional Transformations written in GRoundTram cannot be used to create bidirectional mappings called in an arbitrary order ; instead, it is to be expected that the forward and backward transformations will be called in sequence. They do not need to be temporarily close, but the input of the backward transformation should explicitly be the output of a forward transformation in the first place, and cannot just be used outside of this context.