Transforming BPMN process models to BPEL process definitions with ATL

Guillaume Doux, Frédéric Jouault, Jean Bézivin

AtlanMod, INRIA Rennes Center - Bretagne Atlantique & Ecole des Mines de Nantes, France

{frederic.jouault, guillaume.doux, jean.bezivin}@inria.fr

Abstract. This paper presents a solution to the *Case Study: BPMN to BPEL Model Transformation*. This solution implements a bridge between two business process modeling languages, BPMN and BPEL. The proposed solution has been implemented using ATL.

Keywords: Model-Driven Engineering; Model Transformation; ATL; BPMN; BPEL.

1 Introduction

This paper presents our solution to the *Case Study: BPMN to BPEL Model Transformation* [1] for the GraBaTs 2009 Contest [7]. The presented solution answers to the first variation of the use case (transformation restricted to structured process models). This bridge is implemented using an ATL transformation chain.

This document is structured as follows. Section 2 introduces ATL. Then, section 3 presents the use case main task and its different variations. In Section 4, our solution will be developed. Section 5 contains a discussion about our solution and the evaluation criteria. Finally Section 6 concludes this study.

2 ATL

The AtlanMod Transformation Language [3][4] (ATL) is a general transformation language and tool [2] available from Eclipse.org [6]. ATL is a declarative language allowing the specification of transformation rules matched over the source model which create elements in the target model. It contains also an imperative part allowing handling cases that can be too difficult to manage declaratively. ATL is part of the AmmA [5] platform (AtlanMod Model Management Architecture) which provides support for various tasks such as model-based Domain-Specific Language implementation, and reverse engineering.

3 BPMN to BPEL Case Study

This use case is dedicated to the definition of a transformation between two languages: the Business Process Modeling Notation [8] (BPMN) and the Business Process Execution Language [9] (BPEL).

The main task is focused on the transformation between a subset of the BPMN metamodel and a subset of the BPEL metamodel. These metamodels are presented in [1]. Three variations of this task have been defined. They correspond to three levels of completeness of the transformation. The first one is restricted to structured process models. The second one adds the transformation of quasi structured process models. The last one covers the two preceding tasks and adds the transformation of synchronizing process models.

4 Proposed Solution

Our solution is implemented by using an ATL transformation chain as shown on Fig. 1. As it can be seen this bridge involves several metamodels and transformations. They are described in this section.

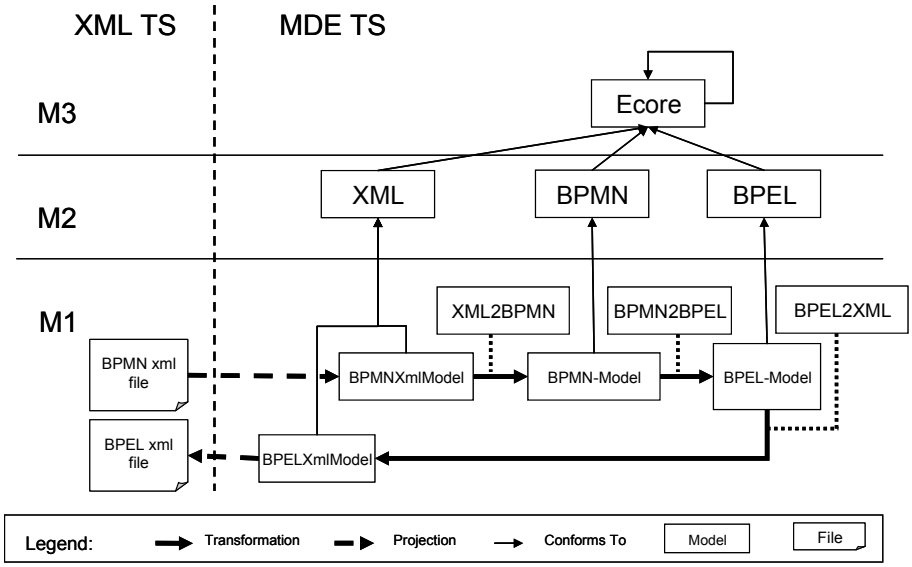

Fig. 1. - BPMN2BPEL Transformation Process Overview

4.1 The BPMN Metamodel

The BPMN metamodel used in the chain is created using EMF [14] from the *BabelBPMN.xsd* schema given with the sample Input/Output models of the use-case. The EMF toolchain natively supports this conversion. The reference mapping used for this translation can be found at [16].

It has then been modified to take in charge a new **NodeType** called **Task** which will be used as default type for nodes. This modification has been done in order to avoid an injection issue of EMF. This issue comes from the fact that EMF does not serialize attributes, which have as value the default value of an enumeration. In our case, it causes that all nodes that should be **Tasks** became **StatEvents**, because it was the default type. Our modification fixes this problem.

4.2 The BPEL Metamodel

The BPEL metamodel used here is based on the BPEL metamodel that can be found on the AtlanMod metamodel zoo [10]. This metamodel has been originally derived from the metamodel used in the Eclipse BPEL Project [11].

Two modifications have been done on this metamodel. The first one has been to change the cardinality of the **otherwise** reference in the **switch** class from 0-1 to 0-*. This modification has been done to be able to reproduce the pattern that appears in the *mediumStructured.bpel* sample. In this example, we can see a switch block containing two otherwise blocks. The second modification is to add a **value** attribute typed as string to the **condition** class to be able to store the expression corresponding to the condition.

4.3 The XML2BPMN Transformation

This transformation is used to inject the XML file representing a BPMN process into a BPMN model that conforms to the BPMN metamodel. As requirement, the XML file representing the BPMN process has to follow the schema presented in *BabelBPMN.xsd*. The mapping used in this transformation is quite simple because the metamodel used for BPMN is very close to the schema used in the XML input files. An excerpt of this mapping is presented in Table 1.

Source Model Element	**Target Model Element**
XML Element, with name = ‘process’	Process
XML Element, with name = ‘node’	Node
XML Element, with name = ‘arc’	Arc

Table 1 - Excerpt of the XML2BPMN Mapping

4.4 The BPMN2BPEL Transformation

This transformation is more complex than the one presented above. It implements the mapping proposed with the use case. Explanations on this mapping can be found in [12].

The main difficulties encountered in this transformation were in the identification of the different patterns appearing in the source model. As the input model contains only sequences of nodes and arcs and no additional structure, we need to write helpers that, from a given node, return if this node appears in a particular pattern. Once the inputs patterns have been clearly identified, the writing of the declarative rule producing the output element is less complicated. In the following paragraphs, a significant excerpt of ATL rule is shown, and then the five patterns involved in the transformation will be detailed.

The following ATL rule (Listing. 1) presents how an AND-Split node can be transformed into a flow. This sample is representative because as in the other rules, the main difficulty appears in the different used helpers. In the "from" part you can see the input element with a filter on its type attribute (here #ANDSplit), and in the "to" part the element that will be created during the matching. Here, a BPEL "flow" element will be created. Its name will be "flowComponent_" followed by an id. Its activities will be generated from the different nodes contained in the branch following it. The "getNodesInBranch" helper returns a sequence of sequence containing the nodes from the different branches. Each result sequence of this helper is then translated into a BPEL "sequence" containing the result of the contained nodes matching.

```
rule ANDSplit2Flows {
    from
        s : BPMN!Node  (s.type=#ANDSplit)
    to
        t : BPEL!Flow(
           name <- 'flowComponent_'+s.number.toString(),
           activities <- s.getNodesInBranch->iterate(c;
              acc : Sequence(OclAny)= Sequence{} |
                 acc->including(
                    if (c->asSet()->size()=1)
                       then c->first()
                       else thisModule.createSequence(c)
                    endif
                 )
        )
  }
```

Listing. 1. ATL rule sample.

The five patterns involved here are the followings:

1. The **sequence**, which can be identified as a path between nodes with one entry and one exit. This pattern is matched in ATL using Listing. 2:

```
rule Task2SequenceInvoke {
    from
        s : BPMN!Node
(if s.type.oclIsUndefined() then true else s.type=#Task endif and
s.isFirstFromSequence and not (s.isFirstInFlow or s.isFirstInSwitch))
```

Listing. 2. Task2SequenceInvoke Header.

This header matches all the tasks nodes which are the firsts of a sequence and which are not embedded in another structure.

2. The **flow**, which can be identified by an AND-Split node which is the beginning of different paths. These paths are then joined by an AND-Join node. This structure is matched using the header presented in Listing. 3.

```
rule ANDSplit2Flows {
    from
        s : BPMN!Node  (s.type=#ANDSplit)
```

Listing. 3. ANDSplit2Flows Header.

This header allows matching all the **ANDSplit** nodes.

3. The **switch**, which is characterized by a XOR-Split node followed by different path as alternatives. These alternatives are then joined by a XOR-Join node. The following header (Listing. 4) presents how the **switch** structure without otherwise is matched in ATL:

```
rule XORSplit2Switch {
    from
        s : BPMN!Node
((s.type=#XORSplit) and (not s.existArcFromXorJoin) and
(not s.existArc2XorJoin) and (s.getFollowingsArcs->select(c |
c.guard.oclIsUndefined())->isEmpty()))
```

Listing. 4. XORSplit2Switch Header.

Here, the **XORSplit** nodes having no arcs to or from a XOR Join and having no guard defined are matched.

4. The **pick**, it is a switch component where the conditions are based on events. Its pattern starts by an EBXOR-Split node followed on each alternative by an event detection node (for message or timer). In ATL it is directly matched using EBXOR-Split nodes has shown in Listing. 5:

```
rule EBXorSplit2Pick {
    from
        s : BPMN!Node (s.type=#EBXORSplit)
```

Listing. 5. EBXorSplit2Pick Header.

5. The **while** and **repeat**, they can be identified because they start with a XOR-Join node and finish by a XOR-Split. This XOR-Split node initializes two

paths. One to the XOR-Join node which marks the start of the loop. The second exits from the loop. The next rule header (Listing. 6) shows how the **while** pattern is matched in our transformation.

```
rule XORSplit2While {
    from
        s : BPMN!Node
  ((if (not s.type.oclIsUndefined()) then s.type=#XORSplit
       else false endif) and s.existArcFromXorJoin )
```

Listing. 6. XORSplit2While Header.

The rule having this header matches all the **XORSplit** nodes that have an arc coming from a **XORJoin** node.

4.5 The BPEL2XML Transformation

This transformation is used to transform the obtained BPEL model from BPMN2BPEL into an XML file following the schema defined in [9]. As in the XML2BPMN transformation, the mapping between the BPEL input model and XML output file is quite simple. An excerpt of this mapping can be viewed in Table 2.

Source Model Element	Target Model Element
Flow Element	XML Element with 'flow' as name and an XML attribute (having the name 'name' and the name of the Flow Element as value) and it's contained Activities as children.
Wait Element	XML Element with 'wait' as name and an XML attribute (having the name 'name' and the name of the Flow Element as value) as child
Empty Element	XML Element having 'empty' as name

Table 2 - Excerpt of the BPEL2XML Mapping

5 Discussion

In this section, we will develop how our solution, which allows the transformation of structured process models, answers to the different evaluation criteria.

The correctness and readability criteria are assumed to be filled by our solution. This is due to the fact that the obtained BPEL model are very close to the samples BPEL models given with the use case. The main difference is in the numbering of the created blocks which are used as identifiers in the models. As the transformation uses

essentially BPEL's block structured control-flow constructs such as 'sequence', 'while' or 'switch' for example, the result should be relatively readable.

This solution is not reversible. ATL Transformations are unidirectional and the transformation from BPEL to BPMN has not been implemented. Nevertheless, as ATL transformations have been used to implement this bridge, the BPMN to BPEL transformation can be processed to automatically generate traceability information during the transformation execution [15]. With this information the user could then locate the elements of the source BPMN models that can be impacted by a localized modification in the target BPEL model.

Another advantage of our solution based on a transformation chain resides in its modularity. As the core transformation (from BPMN to BPEL) is separated from the management of the input and output format (in XML to BPMN and BPEL to XML), the core transformation can be easily reused with another kind of input or output. For example, to change the input format (e.g., textual, graphical), it needs only to write a transformation from the wanted input format to the actual BPMN metamodel, the last chain part staying unchanged. This kind of approach provides a powerful way to write reusable modules.

6 Conclusion

This work satisfies the first variation of the use case. This bridge allows the reproduction of the BPEL models from the BPMN models given as example for this variation. To take in charge the other variations, an idea to develop could be the use of intermediate representations and transformations between BPMN and BPEL. This method could ease the development of the transformations by allowing a more direct mapping between the representations and an easiest definition of the transformation rules. The current solution already shows that chaining transformations enables to solve different problems independently.

Acknowledgments

This work is being supported by the French IdM++ project.

References

[1] Marlon Dumas: Case Study : BPMN to BPEL Model Transformation. University of Tartu, Estonia. A GraBaTs 2009 Case Study. Available at: http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/cases/grabats2008translationcase.pdf

[2] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl: a model transformation tool. *Science of Computer Programming*, 72 (3, Special Issue on Second issue of experimental software and toolkits (EST)):31-39, 2008.

[3] F. Jouault and I. Kurtev. “transforming models with atl”. In *in Proc. Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, 2005.

[4] F. Jouault and I. Kurtev.On the interoperablity of model-to-model transformation languages. *Science of Computer Programming*, 68 (3, Special Issue on Model Transformation):114-137, 2007.

[5] Kurtev, J. Bézivin, F. Jouault, and P. Valduriez. Model-based dsl frameworks. In *OOPSLA ’06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and application*s, pages 602_616, New York, NY, USA, 2006. ACM.

[6] The Eclipse-M2M ATL (AtlanMod Transformation Language) project’s website: http://www.eclipse.org/m2m/atl/

[7] GraBaTs 2009 Workshop homepage: http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/

[8] OMG: Business Process Modeling Notation (BPMN) Version 1.0 OMG final Adopted Specification. OMG, February 2006. Available via http://www.bpmn.org/.

[9] D. Jordan and J. Evdemon.: Web Services Process Execution Language Version 2.0. Committee Specification. OASIS WS-BPEL TC, January 2007. Available via http://www.oasis-open.org/committees/download.php/22475/wsbpel-v2.0-CS01.pdf.

[10] The AtlanMod Metamodel Zoo: http://www.emn.fr/x-info/atlanmod/index.php/Zoos

[11] The Eclipse BPEL Project: http://www.eclipse.org/bpel/

[12] C. Ouyang, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede.: Translating BPMN to BPEL. Technical Report BPM-06-02, BPMcenter.org, January 2006. Available at: http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2006/BPM-06-02.pdf

[13] Jan Recker1 and Jan Mendling2.: On the Translation between BPMN and BPEL:Conceptual Mismatch between Process Modeling Languages. 1 Queensland University of Technology. j.recker@qut.edu.au . 2 Vienna University of Economics and Business Administration. jan.mendling@wu-wien.ac.at.http://wi.wu-wien.ac.at/home/mendling/publications/06-EMMSAD.pdf

[14] The Eclipse Modeling Framework Project: http://www.eclipse.org/modeling/emf/

[15] Frédéric Jouault.: Loosely Coupled Traceability for ATL. AtlanMod Team, INRIA-EMN. In: Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany. http://www.emn.fr/x-info/atlanmod/index.php/Publications_and_Presentations_2005

[16] EMF Documentation, XML Schema to Ecore Mapping: http://www.eclipse.org/modeling/emf/docs/overviews/XMLSchemaToEcoreMapping.pdf