WSCDL to WSBPEL: A Case Study of ATL-based Transformation

Ravi Khadka[1], Brahmananda Sapkota[2], Luís Ferreira Pires[2], Marten van Sinderen[2], and Slinger Jansen[1]

[1] Utrecht University, PO Box 80.089, 3508TB Utrecht, The Netherlands, {ravi, s.jansen}@cs.uu.nl
[2] University of Twente, PO Box 217, 7500AE Enschede, The Netherlands, {b.sapkota, l.ferreirapires, m.j.vansinderen}@ewi.utwente.nl

Abstract. The ATLAS Transformation Language (ATL) is a hybrid transformation language that combines declarative and imperative programming elements and provides means to define model transformations. Most transformations using ATL reported in the literature show a simplified use of ATL, and often involve a single transformation. However, in more realistic situations, multiple transformations may be necessary, especially in case the original input/output models are not represented in the metametamodeling representation expected by the transformation engine. In this paper, we discuss a model transformation from service choreography (WSCDL) to service orchestration (WSBPEL), which cannot be performed in a single ATL transformation due to the mismatch between the concrete XML syntax of these languages and the metametamodeling representation expected by the ATL transformation engine. This requires auxiliary transformations in which this mismatch is resolved. In principle, the required auxiliary transformations can be implemented using XSLT or a general-purpose programming language like Java. However, in our case study, we evaluate the use of ATL to perform these transformations. We exploit ATL by leveraging the ATL's XML injection and the XML extraction mechanisms to perform the overall transformation in terms of a transformation chain.

Keywords: MDE, ATL, Model Transformation, XML Injector, XML Extractor, Model-Driven Service Composition, AM3, WSCDL, WSBPEL

1 Introduction

Model-Driven Engineering (MDE) aims at facilitating the development of software applications by creating abstractions that shield the technical details of the underlying computing environment (i.e., hardware and software environments) [23]. In MDE, models are the primary development artifacts, and model transformations are one of the most important operations applied to models [11]. In a model transformation, a target model conforming to a target metamodel is generated from a source model conforming to a source metamodel. The transformation from a source to a target model is driven by transformation definitions,

usually expressed in transformation languages. A number of transformation languages are available to specify model transformations. A typical example of a transformation language is ATL [8, 10], which is a hybrid language that allows both declarative and imperative constructs to be used to define transformations. ATL has been developed to support transformations in the scope of Model-Driven Architecture (MDA).

In this paper, we present a model transformation case study that uses ATL as transformation language. The case study consists of the transformation from service choreography to service orchestration(s) and aims at automating the service composition processes. Most model transformations using ATL reported in the literature show a simplified use of ATL, and often involve a single transformation. This is understandable, since these works mainly aimed at promoting the model-driven transformation approach and the related languages and tools, and addressed the coding of transformation rules with the ATL syntax elements, which can be better understood with relatively simple transformation problems. Typical examples are the transformations from class to relational models [10], from eXtensible Stylesheet Language Transformation (XSLT) to XQuery [5] and from Roman numbers to Arabic numbers [10]. However, in more realistic transformation cases, additional *auxiliary transformations* may be necessary due to the mismatch in technological spaces [16]. For instance, the ATL transformation engine expects the input/output models and metamodels to be serialized in XML Metadata Interchange (XMI) format [9], while the original input/output models in realistic transformations may not necessarily be represented in XMI. To resolve these mismatches, auxiliary transformations are needed to convert the original input format to XMI, or to convert from XMI to the final output format, or both.

This paper discusses a transformation from Web Service Choreography Description Language (WSCDL) [12] to Web Service Business Process Execution Language (WSBPEL) [2], which cannot be performed in a single ATL transformation. WSCDL and WSBPEL are specified in concrete XML syntax so that the situation sketched above applies to this transformation. In principle, the auxiliary transformations necessary in this case can be implemented using either eXtensible Stylesheet Language Transformation (XSLT) or a general-purpose programming language like Java [24]. However, in our case study, we have used ATL to perform these auxiliary transformations, by leveraging the ATL's XML injection and the XML extraction mechanisms, while the core transformation has been performed in ATL. This paper evaluates the resulting transformation chain and the use of ATL to perform these transformations.

This paper is further structured as follows: Section 2 motivates service composition and the transformation from service choreography to service orchestration(s). In Section 3, we describe the approach adopted in the transformation and justify the need of the auxiliary transformations. In Section 4, we discuss the implementation of the transformations. Section 5 discusses the lessons learned with this case study. In Section 6, we present some of the related work, and finally Section 7 gives our conclusions.

2 Background

In our case study, we perform a model transformation from service choreography to service orchestration(s) aiming at automating service composition process. The service composition process not only realizes required, value-added composite functionality, but also accelerates application development through reuse at service level [14]. Service compositions can be considered at different abstraction levels, notably at choreography and orchestration levels [20, 4]. A choreography is a decentralized perspective, which describes the public message exchanges, and thus defines how participating services should interact with each other. At a lower level, it is necessary to define how to realize the responsibilities specified at the choreography level in terms of the concrete processes. An orchestration is a centralized coordination of participating services, which defines the message exchanges along with the necessary internal actions, like data transformations and internal function invocations [20]. In this paper, we report on our approach to (semi-)automate the transformation from choreography to orchestration(s) using ATL. We refer to this process as *model-driven service composition* [15]. We defined the metamodels for choreography and orchestration, and defined transformation specifications based on mappings between elements of these metamodels.

In our approach, we specify choreographies and orchestrations using WSCDL (CDL in short) and WSBPEL (BPEL in short), respectively, because of their wide industry acceptance. We defined the CDL and BPEL metamodels based on the concrete XML syntax of these languages. In the core transformation, the CDL metamodel is the source metamodel and BPEL metamodel is the target metamodel. We defined mappings between elements of the CDL metamodel and the BPEL metamodel that reflect our design decisions. These mappings were implemented in ATL transformation specifications that can be executed by the ATL engine. Figure 1 gives an overview of our transformation approach, indicating the relations between models, metamodels and transformations.

3 CDL to BPEL Transformation Chain

We defined the CDL and BPEL metamodels from their language specifications, and represented these metamodels with the Eclipse Modeling Framework (EMF). Initially, we generated the metamodels of CDL and BPEL from their respective schemas as explained in [7]. Many unnecessary model elements were generated in this way, which complicated the transformation process unnecessarily. Hence, we re-developed these metamodels manually from the language specification of CDL [12] and BPEL [2]. The CDL and BPEL metamodels are depicted in Figure 6 and Figure 7, respectively.

ATL transformations are unidirectional so that they operate on read-only source models and generate write-only target models. During a transformation execution, the source model may be navigated but cannot be modified, while target models cannot be navigated. In the CDL to BPEL transformation, the

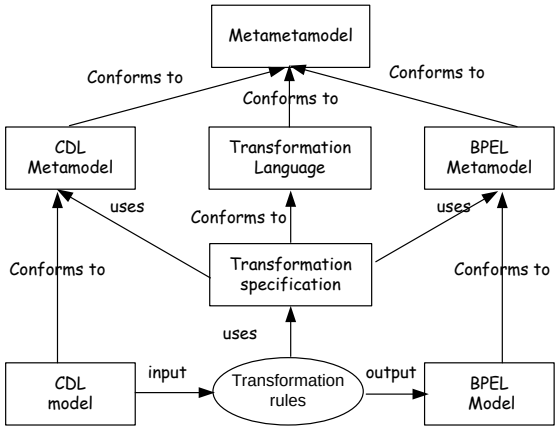

Fig. 1. Overview of transformation approach

original input model (CDL) and final output model (BPEL) are specified in XML format. However, the ATL engine expects all input/output models and input/output metamodels to be serialized in XMI format [9]. Hence, there is mismatch between original input/output model (CDL/BPEL in XML format) and input/output model (XMI format) of the ATL engine. To resolve these mismatches we need a transformation chain that consist of following transformations:

- *Auxiliary Transformations*, consisting of a CDL XML model to CDL XMI model transformation (T1) and a BPEL XMI model to BPEL XML model transformation(T3).
- *Core Transformation*, consisting of a CDL XMI model to BPEL XMI model transformation (T2).

Figure 2 shows the resulting transformation chain.

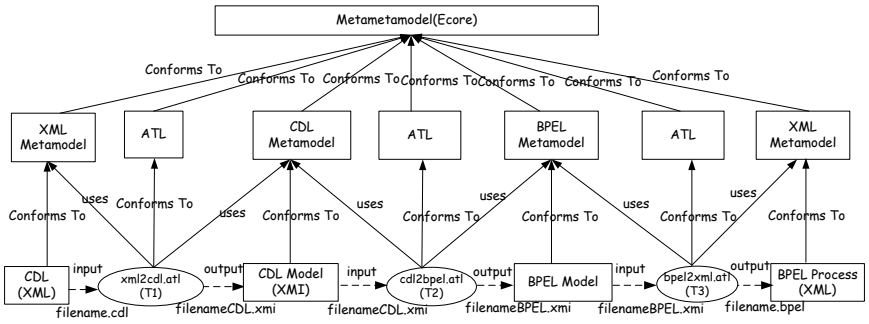

Fig. 2. Transformation chain

The core transformation (T2) of our approach is performed by executing the ATL transformation specifications that are derived from the transformation mapping between CDL and BPEL metamodel elements. The ATL engine reads a CDL XMI model as input, executes the transformation rules, and generates a BPEL XMI model as defined in the ATL transformation specification. The details of the transformation mappings between CDL and BPEL metamodel elements have been reported in [15, 13].

The auxiliary transformations T1 and T2 are carried out using AtlanMod Megamodel Management (AM3)[1] framework. We used the ATL's XML injection mechanism to perform transformation T1. An XML injection uses an XML injector, which is a tool that implements the injection process that transforms the CDL XML model to a CDL XMI model (instance of the CDL metamodel), and uses ATL rules to perform transformation. To perform transformation T3, we used the ATL's XML extraction mechanism, in which the XML extractor transforms the BPEL XMI model (instance of the BPEL metamodel) to a BPEL XML model.

4 Implementation

The implementation steps of our transformation chain are depicted in Figure 3. We used the Pi4soa[2] CDL editor to model the choreography specification. Each transformation is discussed in the sequel.

4.1 Core Transformation

The core transformation T2 has been implemented in ATL. The transformation rules are defined based on the transformation mapping reported in [15]. Listing 1 presents the code snippet of transformation T2 in which the *roleType* construct of CDL is transformed to *process* of BPEL. A *roleType* construct is used to specify the observable behavior of a participant in the collaboration. In BPEL, each process represents a role in collaboration so we generate a *process* for each *roleType* of CDL. Since, in our research we restrict ourselves to the centralized orchestrator, we generate the process for centralized orchestrator such that the name of the *process* is derived from the *name* of the *package*, and *targetNamespace* of *process* is derived from *targetNamespace* of *package* of CDL. The *variable* associated with the *process* is derived from the *variable* of CDL, *partnerlinks* of the *process* from other *roleType* of CDL, *activity* of the *process* from *activity* of CDL. The transformation is diagrammatically represented in Figure 4.

The code snippet (see Listing 1) contains mandatory header section (*module* and *create*), attribute and operation helpers (*orchestrator* and *isOrchestrator()*), and transformation rules (*roleType2BPELProcess* and *varTovar lazy rule*). The *create* statement specifies the target model (*BPEL*) and the source

[1] http://wiki.eclipse.org/AM3
[2] http://sourceforge.net/apps/trac/pi4soa/wiki

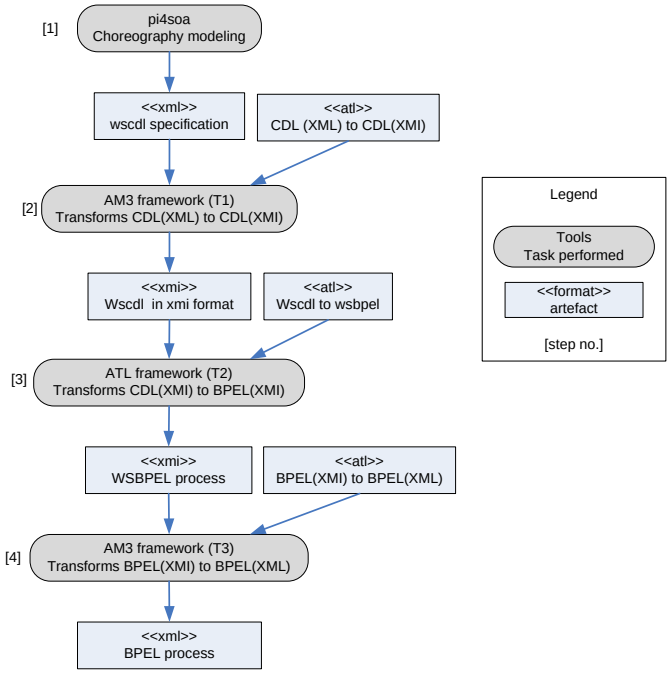

Fig. 3. Implementation steps

model (*CDL*). The transformation rules are the basic constructs of ATL used to express the transformation logic. The *roleType2BPELProcess* is the declarative rule (*matched rule*) that triggers *varTovar* lazy rule. A lazy rule in ATL is triggered by other rules and may be applied multiple times on a single match, each time producing a new set of target elements [10]. The *roleType2BPELProcess* rule is used to transform the *roleType* construct of CDL that is defined as *Orchestrator* to a BPEL process with same name as of *roleType*.

Listing 1. Code snippet of T2 transformation

```
-- @path CDL=/test2Bpel/CDL.ecore
-- @path BPEL=/test2Bpel/BPEL.ecore
module cdlToBpel;
create OUT : BPEL from IN : CDL;

helper def : orchestrator : String = 'Manufacturer';
helper def : isOrchestrator() : CDL!RoleType =
    CDL!RoleType.allInstances() ->
        select(r | not r.isConnectedToChannelType()
    and r.name = thisModule.orchestrator)->first();

rule roleType2BPELprocess{
 from
   s: CDL!RoleType (not s.isConnectedToChannelType()
```

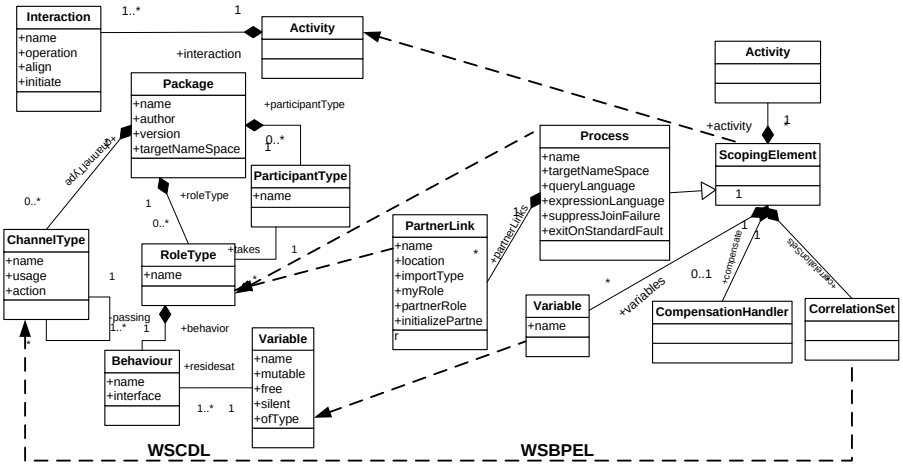

Fig. 4. RoleType to Process transformation

```
    and s.name = thisModule.orchestrator)
to
  t: BPEL!Process(
  name <- s.name+'Process',
  targetNameSpace <- s.getTargetNamespace(),
  variables <- s.getVariables()
  ->collect(v | thisModule.varToVar(v)),
  partnerLinks <- s.getRoleTypes(),
  scopeElementActivity <- CDL!Activity.allInstances(),
  correlationset <- CDL!ChannelType.allInstances()
  )    }

lazy rule varToVar {
  from v : CDL!Variable
  to bv : BPEL!Variable (
    name <- v.name)    }
```

4.2 Auxiliary Transformations

The auxiliary transformations T1 and T3 have been implemented by leveraging the ATLs XML injector and extractor for injecting and extracting XML models into and from the XMI metamodel syntax, respectively. We used the AM3 framework to implement XML injector (T1) and XML extractor (T3). In order to perform these transformation, we have used the XML metamodel shown in Figure 5).

Listing 2 shows the code snippet of transformation T1 in ATL between the XML source model and the CDL XMI target model. However, running this code directly in ATL engine does not result in the desired CDL XMI format, so that

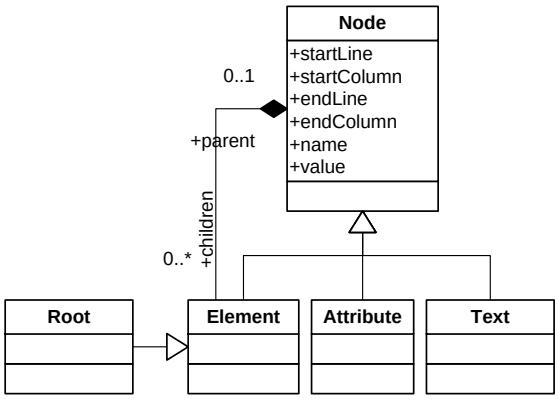

Fig. 5. XML metamodel

we had to use the XML injection mechanism of the AM3 framework. Similarly, in the transformation T3 we used the XML extraction mechanism of the AM3 framework to extract BPEL XML code from the BPEL XMI format.

Listing 2. Code snippet of Transformation T1

```
module cdl2xmi;
create OUT : CDL from IN : XML;

rule Root2Package{
 from
  s : XML!Root
 to
  t : CDL!Package(
   name<-'Package',
   children<-Sequence{name, author, tgnsp, version,
    s.informationType, s.relationType,
    s.participantType, s.roleType,
    s.tokenLocator, s.token,
    s.choregraphyPkg, s.channelType}),

  name:CDL!Attribute(
    name<-'name',
    value<-s.name),
  author:CDL!Attribute(
    name<-'author',
    value<-s.author),
   -- code continues--
   }
```

Listing 3 depicts the ATL Ant task that invokes the XML injection of transformation T1. Task *am3.loadModel* loads a model with injectors and task *am3.atl* executes the ATL transformation of Listing 2. Finally, task *am3.saveModel* is

used to save the model. This task also specifies the extractors that save the model in XML format in transformation T3. These ATL Ant tasks are documented in [3].

Listing 3. Illustrating XML injection of Transformation T1

```
<project name="CDL2BPEL" default="transformAll">
<!-- other tasks -->
<!--Inject source model -->
 <am3.loadModel modelHandler="EMF" name="xmlModel"
                metamodel="XML" path="/project/">
   <injector name="XML"/>
 </am3.loadModel>
<!-- Transform XML model into CDL model -->
 <am3.atl path="/project/xml2xmi.atl">
  <inModel name="IN" model="xmlModel"/>
  <inModel name="XML" model="XML"/>
  <inModel name="CDL" model="CDL"/>
  <outModel name="OUT" model="cdlModel"
            metamodel="CDL"/>
 </am3.atl>
<am3.saveModel model="cdlModel"
               path="/project/cdl.xmi"/>
</project>
```

The source code can be downloaded from here[3].

5 Discussion

In this case study, we used ATL to perform the auxiliary and core transformations that transform a given CDL specification to a BPEL process. Although there is support for the imperative definition of transformation rules, the presented solution uses a declarative approach, as recommended in [6]. The declarative transformation rules are called *matched rules* and the imperative transformation rules are known as *called rules*. Unlike the *called rules*, which are not triggered when a match is found but are called from another rule, the *matched rules* are triggered if a pattern matches successfully. Because of our declarative approach, the transformation is easier to read and understand, which is important in large transformation projects from an engineering and maintenance perspective. A significant part of the *matched rules* are written using *helpers*. In our case study, most of the navigation functions are implemented as helpers to keep the actual transformation rules free from complex navigation expressions. For instance, the *getTargetNamespace()*, *getVariables()*, *getRoleTypes()* of Listing 1 are helpers and quite often used throughout the transformation rules, which otherwise, if implemented in the actual transformation rules, create more complex navigation expressions. The separation of helpers and actual transformation

[3] `http://people.cs.uu.nl/ravi/source/source.zip`

rules fits the basic intention of these constructs: rules are used for creating target model elements and helpers are used for source model navigation [10].

ATL uses Object Constraint Language (OCL) [18] to represent data types and declarative expressions. The OCL expressions are common to all data types, which are helpful in the context of filtering. Moreover, facilities for performing calculations on several data types via OCL was proven to be powerful and expressive enough while specifying the transformation rules in our solution. Elements can simply be selected using the name of the reference. Hence, selection and traversing of instances of model elements are facilitated.

The pattern matching mechanism of ATL matches the specific elements from the source model and creates the right elements within the target model. In combination with OCL expressions, pattern matching saves a large number of lines of code compared with transformations written in general-purpose programming languages like Java or XSLT-based transformations [24]. Previously, transformation T1 of our approach was implemented using XSLT [15], which became lengthy and required considerable effort to be defined and maintained if compared with the latest ATL-based transformation.

The use of the AM3 framework for ATL's XML injection (T1) and XML extraction (T3) made our approach uniform, since it aligned these transformations with the ATL-based core transformation (T2). The XML injection and XML extraction facilitated the understandability of the transformation and resulted in a better maintainable transformation chain, if compared with our earlier solution (i.e., using XSLT).

Further, existing Integrated Development Environments (IDE) for the ATL and the AM3 framework are available as extensions to the Eclipse Modeling Project (M2M) and Generative Modeling Technologies (GMT) respectively, which provide support, such as syntax highlighting and debugging for developing transformations.

We validated our transformation with two example scenarios: a Purchase Order scenario [13] and a Build-To-Order scenario [15]. In both scenarios we modeled the choreography using Pi4soa and performed all the transformations (T1, T2, and T3). The T3 transformation successfully generated the BPEL skeleton from the given CDL specification. The choreography has higher abstraction level than the orchestration so choreography does not represent the internal details of the participating services in the collaboration. We, therefore, manually added some missing information (like branching conditions) to the generated BPEL process, which was validated against the executable schema of BPEL to ensure syntactical correctness. We later imported the BPEL process in the ActiveBPEL designer tool [1] to check the behavior of the orchestrator, and tested this behavior for correctness, with successful results.

6 Related Work

A number of transformation approaches from CDL to BPEL have been reported in the literature. Mendling et al. [17] proposed an XSLT-based transformation

approach to realize the transformation from CDL to BPEL. The transformation is bidirectional in nature, i.e., the XSLT script can generate a BPEL process from a CDL specification, and vice-versa. In [22], Rosenberg et al. developed CDL to BPEL transformation approach using Java. Similarly, Weber et al. [25] presented a CDL to BPEL transformation approach based on Java. In our previous work [15, 13], we used XSLT-based transformation to perform transformation T1 of the transformation chain (see Figure 3).

XSLT-based transformations require experience and considerable effort to define and maintain, whereas transformations written using general-programming languages like Java tend to be hard to write, comprehend, and maintain [24]. Metamodel-based transformations, like the ones we implemented in our transformation chain in this case study, are much more convenient for the sake of understandability and maintenance.

7 Conclusion

In this paper, we reported on our experience with a realistic model transformation using ATL. The transformation was developed to automate service composition design from choreography (CDL) level to orchestration (BPEL) level. In particular, we described the need of multiple transformations, subdivided in core and auxiliary transformations, and the use of ATL to perform all those transformations. Auxiliary transformation exploited the use of ATL's XML extraction and injection to provided uniformity and facilitated the maintainability of the resulting transformation chain. We have successfully validated the proposed transformation approach with two example scenarios. Our result shows a significant improvement over the manual transformation approaches that existed earlier. We also tested the behavior of the generated BPEL process for correctness, with successful results.

Overall, we conclude that the consistent use of ATL in the transformations of our case study has been beneficial from the perspective of maintainability and understandability. Possible future work can be to implement the CDL to BPEL transformation in other transformation languages like Query/View/Transformation (QVT) [21] and Yet Another Transformation Language (YATL) [19], and compare the resulting transformations, for example, by comparing their complexity and performance.

References

1. ActiveBPEL: ActiveBPEL Engine. Online (2011), Available at: `http://http://www.activebpel.org`
2. Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., et al.: Web Services Business Process Execution Language Version 2.0. OASIS, Available at: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (2007)
3. AM3/Ant: AM3/ANT Tasks. Online (2004), Available at: `http://wiki.eclipse.org/AM3_Ant_Tasks`

4. Barros, A., Dumas, M., Oaks, P.: Standards for web service choreography and orchestration: Status and perspectives. In: Business Process Management Workshops. pp. 61–74. Springer (2006)
5. BéZion, J., Dupé, G., Jouault, F., Pitette, G., Rougui, J.: First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In: 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture. p. 50 (2003)
6. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the Large and Modeling in the Small. Model Driven Architecture pp. 33–46 (2005)
7. EMF: Generating an EMF Model using XML Schema (XSD). Online (June 2004), Available at: `http://www.eclipse.org/modeling/emf/docs/2.x/tutorials/xlibmod/xlibmod_emf2.0.html`
8. Jouault, F., Allylamine, F., BéZion, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
9. Jouault, F., Allylamine, F., BéZion, J., Kurtev, I., Valourem, P.: ATL: a QT-like transformation language. In: Companion to the 21st ACP SKIPLANE symposium on Object-oriented programming systems, languages, and applications. pp. 719–720. ACP (2006)
10. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Satellite Events at the MODELS 2005 Conference. pp. 128–138. Springer (2006)
11. Jouault, F., Allylamine, F., BéZion, J., Kurtev, I., Valourem, P.: Atl: a qt-like transformation language. In: Companion to the 21st ACP SKIPLANE symposium on Object-oriented programming systems, languages, and applications. pp. 719–720. OOPSLA '06, ACP, New York, NY, USA (2006)
12. Kavantzas, N., Burdett, D., Ritzinger, G., Lafon, Y.: Web Services Choreography Description Language Version 1.0, W3C Candidate Recommendation, November 2005. World Wide Web Consortium (2005), Available at: `http://www.w3.org/TR/ws-cdl-10/`
13. Khadka, R.: Model-Driven Development of Service Compositions: Transformation from Service Choreography to Service Orchestrations. Master's thesis, University of Twente (August 2010), Available at: http://essay.utwente.nl/59677/
14. Khadka, R., Sapkota, B.: An Evaluation of Dynamic Web Service Composition Approaches. In: Proceeding of the 4th International Workshop on Architectures, Concepts and Technologies for Service Oriented Computing, ACT4SOC 2010. pp. 67–79. INSTICC Press, Athens, Greece (July 2010)
15. Khadka, R., Sapkota, B., Ferreira Pires, L., van Sinderen, M., Jansen, S.: Model-driven development of service compositions for enterprise interoperability. In: van Sinderen, M., Johnson, P. (eds.) Enterprise Interoperability, LNBIP, vol. 76, pp. 177–190. Springer (2011)
16. Kurtev, I., Bézivin, J., Aksit, M.: Technological spaces: An initial appraisal. In: Proceedings of Confederated International Conferences CoopIS, DOA, and ODBASE: Industrial Track. Springer, Irvine, CA, USA. (2002)
17. Mendling, J., Hafner, M.: From inter-organizational workflows to process execution: Generating BPEL from WS-CDL. In: On the Move to Meaningful Internet Systems 2005: OTM Workshops. pp. 506–515. Springer (2005)
18. OCL: Object Constraint Language OMG Specification Ver 2.0. Online (2002), Available at: `http://www.omg.org/spec/OCL/2.0/PDF`
19. Patrascoiu, O.: YATL: Yet another transformation language. In: Proceedings of the 1st European MDA Workshop, MDA-IA. pp. 83–90. Twente, The Netherlands (2004)

20. Peltz, C.: Web services orchestration and choreography. Computer 36(10), 46–52 (2003)
21. QVT: Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.1. Online (2011), Available at: http://www.omg.org/spec/QVT/1.1/PDF/
22. Rosenberg, F., Enzi, C., Michlmayr, A., Platzer, C., Dustdar, S.: Integrating quality of service aspects in top-down business process development using WS-CDL and WS-BPEL. In: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2007. p. 15. IEEE Computer Society (2007)
23. Schmidt, D.: Model-driven engineering. IEEE computer 39(2), 25–31 (2006)
24. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. Software, IEEE 20(5), 42–45 (2003)
25. Weber, I., Haller, J., Mulle, J.: Automated derivation of executable business processes from choreographies in virtual organisations. International Journal of Business Process Integration and Management 3(2), 85–95 (2008)

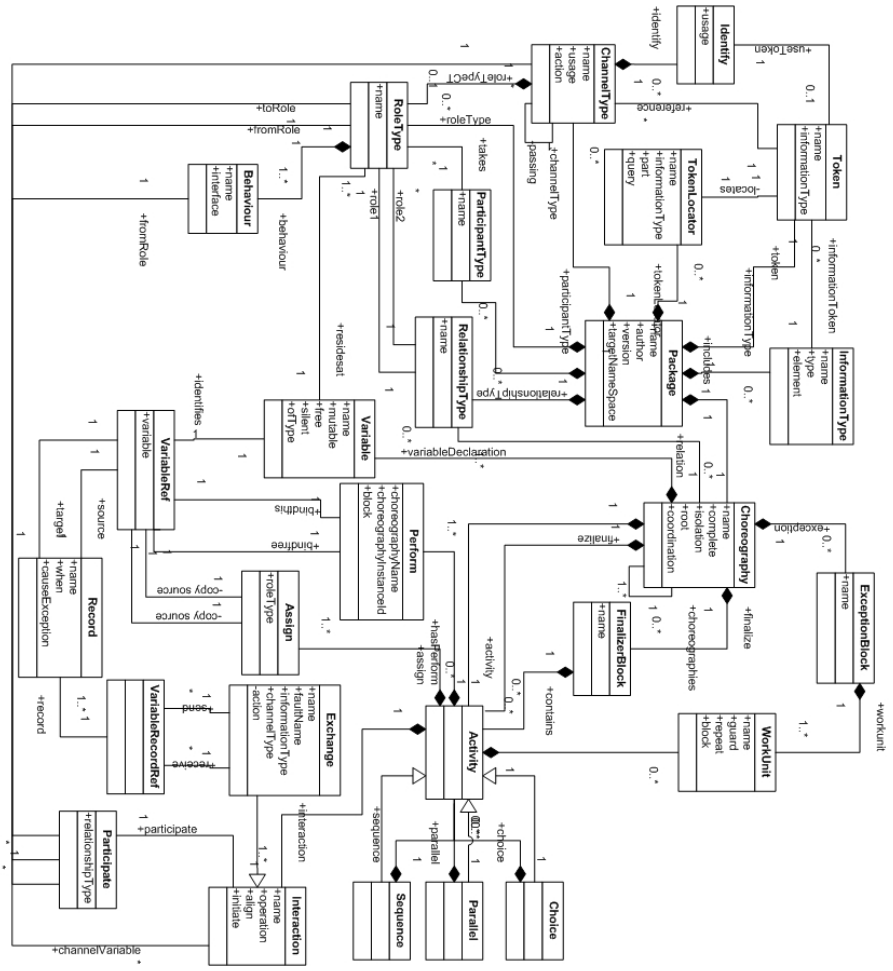

Fig. 6. CDL metamodel

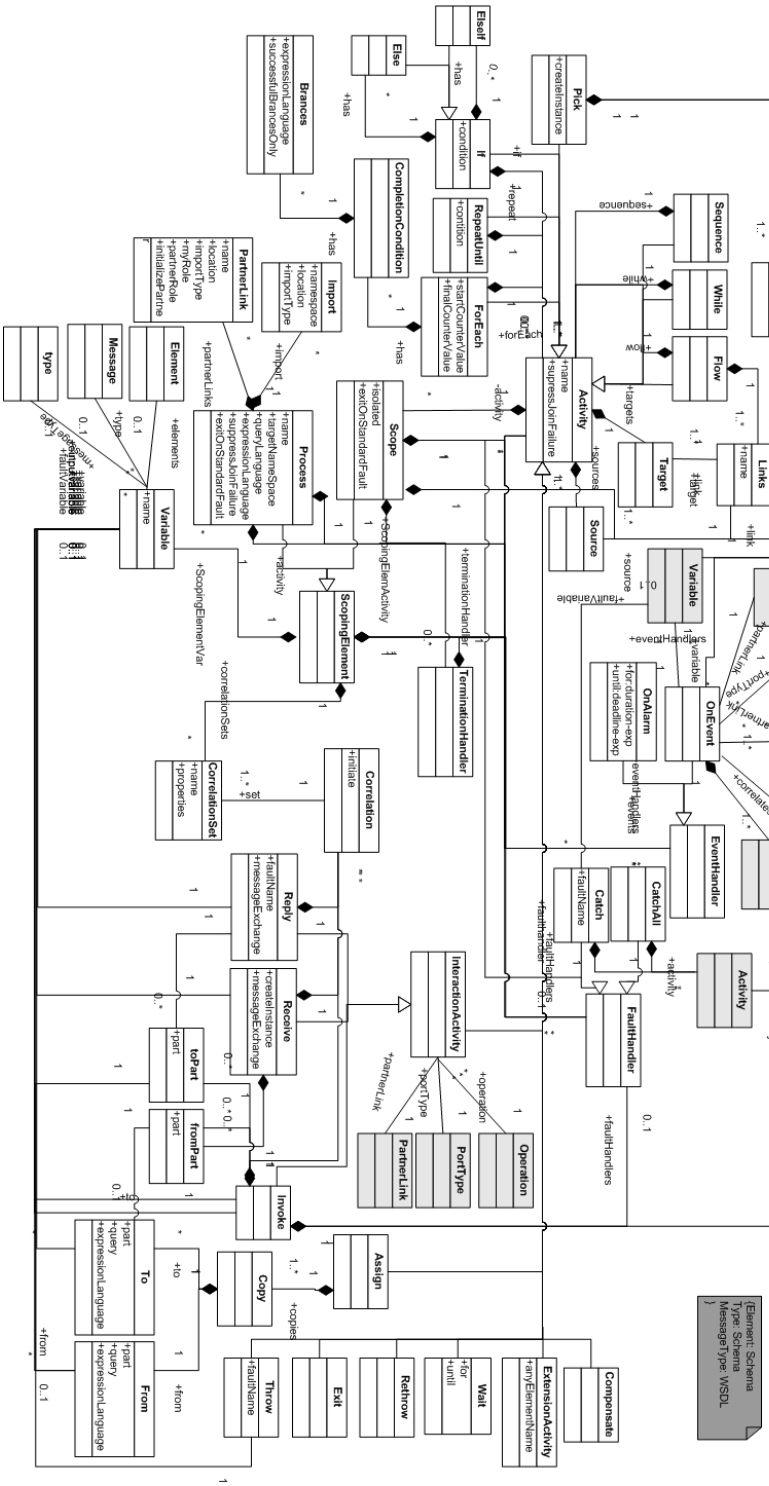

Fig. 7. BPEL metamodel