Generation of Workflow Code from DSMs

Stephan Roser, Florian Lautenbacher, and Bernhard Bauer

Programming Distributed Systems Lab, University of Augsburg, Germany
{roser|lautenbacher|bauer@ds-lab.org}

Abstract. To use process models not only for documentation purposes but also for execution with workflow engines, models need to be seamlessly transformed into executable workflow code. In practice however, existing model and code generations show a number of limitations: different process engines require different workflow code and domain-specific models need (often complex) graph transformation algorithms to come from a graph-based to a block-based structure. In this paper we describe these issues in detail and develop a model and code generation framework that fosters workflow code generation from domain-specific models.

1 Introduction

Model-driven software development (MDSD) embodies software engineering approaches focusing on creating models rather than program code. MDSD uses models to raise the level of abstraction at which developers create and evolve software [9] and reduces complexity of the software artifacts by separating concerns and aspects of a system under development [10]. The OMG™'s Model Driven Architecture® (MDA®) [18], a specific MDSD approach, suggests to apply models at three levels of abstraction. Multiple model transformations have to be developed transforming and integrating knowledge captured in the various models. However, this is an overhead that prevents many projects from increasing their productivity. The software factory initative from Microsoft® [9] focuses on domain specific languages (DSLs) and customized development processes. Software factory schemas are used to describe the assets for software development like DSLs, patterns, frameworks, and tools. The Generic Modeling Environment (GME) [1] is a configurable toolkit for creating domain-specific modelling and program synthesis environments. MetaEdit+ from MetaCase [2] allows the defintion of DSLs and customized diagram types. Code generators can be written that improve productivity by generating code directly from higher-level models.

In the AgilPro project[3] [4] - *Agil*e Business *Pro*cesses - we have developed a modelling language, a modelling tool, and a model and code generation framework that allow the definition of DSLs for process-oriented application domains and the generation of executable workflow code. In this paper we present the techniques we use in the generation framework. The main achievements of the generation framework are to provide component-based design, which fosters the reuse and composition of parts of generation solutions, and to decouple domain aspects from computational aspects in the code generation. The generation framework aims to enable people with no or little experience in code generation and workflow technology to generate workflow code from higher-level models in reasonable time.

2 Context

During the last decade in the ERP domain agile processes got more and more important. In order to improve existing products or customize them to the needs of the end-user most changes need

[1] http://www.isis.vanderbilt.edu/projects/gme/
[2] http://www.metacase.com/
[3] http://www.agilpro.eu/

to be done on the flow of services but not on the offered services. This results from changes in jurisdiction, new products, standards, or requirements of the customers. In most organisations there is a need for loosely-coupled components. Service-oriented Architecture (SOA) and Web service technology as an implementation of SOA are one way for achieving this. AgilPro is a tool-suite and process integration framework based on SOA. It allows the user to model their business processes, preview and execute them on a process engine.

The core of the AgilPro solution is a domain-specific model (DSM) that conforms to a DSL for process modelling and contains predefined model elements (e.g. applications, services, etc.) for a particular domain. The AgilPro modelling tool further provides the possibility to define multiple concrete syntaxes, for example one that especially suits the ERP domain. The AgilPro modelling tool offers (at least) two views on the DSM, a business view and a technical view. The business view abstracts from technical details such as which web services are invoked, how the data mapping between different applications works, etc.. This is part of the technical view where an IT expert can specify the relevant data for an execution of the process - if not already predefined in the DSM. The model and code generation framework is used to generate executable workflow code.

The AgilPro metamodel is graph-based as most business process languages and rests upon the UML 2 metamodel for activity diagrams. It extends it with information about responsibilities or functions like in ARIS [23] or data and events similar to BPMN [19]. Henceforth, it tries to combine the best practices of the currently existing process modelling languages. To enable domain-specific modelling for e.g. ERP, CRM, or financial service applications, the metamodel and the AgilPro modelling tool allows to define model templates. These model templates contain predefined elements of and information about the application domain, but also syntactical information. This are for example data types or applications with specific (execution) information or icons from the ERP domain. These predefined elements become automatically part of AgilPro modeller's modelling palette and complete together with the AgilPro metamodel the specific DSL. Since such DSLs cannot be executed on current process engines directly, one needs to transform them to an executable language. We use WS-BPEL [5], which is the quasi standard for orchestrating web services and supported by several process engines, for illustration purposes in the rest of this paper.

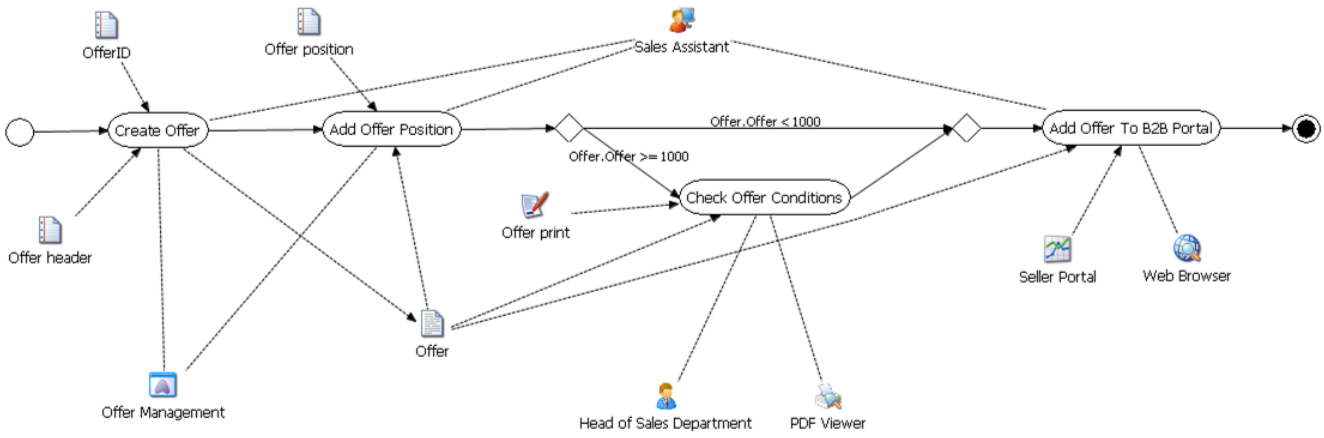

Fig. 1. Create Offer Process Modelled with AgilPro Light Modeller

Figure 1 shows a *Create Offer* process that is modelled with the AgilPro Light Modeler (LiMo). A complete process model comprises processing steps, input and output data, applications that

are used to execute the processing steps, and roles that perform the processing steps. In the *Create Offer* process the processing step *Create Offer* has an *OfferID* and *Offer header* as input data and produces an *Offer* as output. *Create Offer* is performed by the *Sales Assistant* and makes use of the *Offer Management* system for execution. If the offer has a value that is greater than or equal to 1,000, the *head of the sales departmen*t has to check the offer. Finally, the offer is added to a B2B portal by the *sales assistant*. What is not shown in the process diagram is the additional information of the predefined elements that complete the DSL. In the *Create Offer* process, these are for example the attributes of the data types like *'dioParameter'* for *OfferID* or information about the AgilPro integration framework Java adapters, which are called for the applications like *'eu.emundo.agilpro.fw.fe.intf.GenericUi'* for *Offer Management*. This additional information is used by the code generation to directly generate executable BPEL code.

3 Challenges of the Code Generation

To develop a code generation that generates executable workflow code directly from higher-level process descriptions like in AgilPro, one has to deal with a variety of challenges.

3.1 Process (Graph) Transformation

An important challenge is the translation of the higher-level processes into constructs that are provided through the target process execution language. BPEL for example is a so-called block-structured language. There, language elements that represent the control flow are composed in a cycle-free tree-structure without goto-statements. [16, 22] describe approaches of how to translate process graphs into block-structured BPEL code. The process graph has to be analyzed and even restrutured in order to map the graph to the BPEL elements. These approaches are based on the identification of *single-entry-single-exit* (SESE) components in the control flow. To identify SESEs, algorithms like [12] from compiler theory or the token flow algorithm [8, 19] can be used.

As a result, one has to implement quite complex graph transformation algorithms to realize (e.g. BPEL) workflow generation from higher-level process graphs. It is certainly possible to implemented these transformations (some good examples can be found in [22]) with nearly any model-to-model or model-to-code transformation approach. However, most model-to-model and model-to-code transformation approaches qualify themselves a lot better for describing relationsships between elements and implementing generation patterns than for implementing complex graph transformation algorithms. To realize preprocessing and graph transformation algorithms like [8, 12, 22] common programming languages like Java or C are better suited.

3.2 Usage of Process Execution Environments and Engines

Though process execution languages have a well-defined syntax and semantics, different process execution programs can be used to achieve the same external behaviour (effects) of a process (execution) engine. Hence, code generations templates have to be adjusted in the way process engines are used and implement the particular execution patterns. Moreover, code generation templates also encode domain knowledge, like the specific data types. The following examples illustrate two different ways of using BPEL to execute the invocation of a *CreateOffer* service. While the first example represents code generation from process models whose semantics is similar to BPEL code, the second example demonstrates code generation from higher-level process models.

The second example demonstrates quite well, that for a range of realistic application scenarios sophisticated execution and invocation patterns have to be encoded into code generations.

Example 1 The BPEL generation developed in the SPL4AOX project [20, p.78ff] was based on the action semantics of UML. Similar BPEL code generations have been described for example in [5]. Listing 1.1 depicts the BPEL code that is generated to invoke the *CreateOffer* service, that gets an *OfferID* as input and provides an *Offer* as output (see Figure 1 in Section 2).

Listing 1.1. BPEL code generated in SPL4AOX

```
<invoke name="CreateOffer"
 partnerLink="CreateOffer_Prov" portType="CreateOffer"
 operation="in" inputVariable="OfferID" outputVariable="Offer">
</invoke>
```

Example 2 For our usage of the JBoss workflow engine in the AgilPro project multiple BPEL instructions are necessary to obtain the same computational result as in example 1. Listing 1.2 depicts the sample code that is necessary to invoke the *CreateOffer* service depicted in Figure 1. One important issue to recognize is, that the process execution engine has an execution context in which information about the process execution can transiently be stored. *Variables* like *nextActionReq* are containers in this execution context which consist of attributes (*parts*) like *ticketnumber* or *NameIN*. We use the *ticketnumber* for correlation in the JBoss engine. *DataTypeIN*, *ValueIN*, and *NameIN* contain the information that is also used by the data object of the AgilPro integration framework. The code of Listing 1.2 is generated as follows:

1. Before the processing step *CreateOffer* is started, the input data *OfferID* is copied to the execution context. This is done by an assign and an invoke for each input data (line 1-25).
2. In the lines 26-39 the execution of the processing step *CreateOffer* is started.
3. The receive statement (line 40-45) waits for the completion of the processing step, which can also be human interaction input. Line 46-58 stops the task execution in the process engine.
4. Finally, the result data *Offer* (line 59-83) is fetched from the process execution context.

Listing 1.2. BPEL code generated in AgilPro

```
<assign name="set_DTO_OfferID">
 <copy>
  <from part="Ticketnumber" variable="nextActionReq"/>
  <to part="Ticketnumber" variable="setValueToObjectReq"/>
 </copy>
 <copy>
  <from expression="string('dioParameter')"/>
  <to part="DataTypeIN" variable="setValueToObjectReq"/>
 </copy>
 <copy>
  <from expression="string('ID')"/>
  <to part="ValueIN" variable="setValueToObjectReq"/>
 </copy>
 <copy>
  <from expression="string('OfferID')"/>
  <to part="NameIN" variable="setValueToObjectReq"/>
 </copy>
 </assign>
<invoke name="setValueToObject_OfferID"
  portType="agi:AgilproIssuer" partnerLink="agilpro"
  operation="setValueToObject" inputVariable="setValueToObjectReq">
 <correlations>
  <correlation pattern="out" set="atmInteraction"/>
 </correlations>
</invoke>
<assign name="startAction_CreateOffer">
```

```
30  ...
   </assign>
32 <invoke name="startAction_CreateOffer"
     portType="agi:AgilproIssuer" partnerLink="agilpro"
34   operation="startAction" inputVariable="startActionReq">
    <correlations>
36   <correlation pattern="out" set="atmInteraction"/>
    </correlations>
38 </invoke>
   <receive
40   portType="atm:FrontEnd" partnerLink="atm"
     operation="nextAction" variable="nextActionReq">
42  <correlations>
     <correlation set="atmInteraction"/>
44  </correlations>
   </receive>
46 <assign name="endAction_CreateOffer">

50  ...
   </assign>
52 <invoke name="endAction_CreateOffer"
     portType="agi:AgilproIssuer" partnerLink="agilpro"
54   operation="endAction" inputVariable="endActionReq">
    <correlations>
56   <correlation pattern="out" set="atmInteraction"/>
    </correlations>
58 </invoke>
   <assign name="get_DTO_Offer">

75  ...
76 </assign>
   <invoke name="getValueFromObject_Offer" partnerLink="agilpro"
78   portType="agi:AgilproIssuer" operation="getValueFromObject"
     inputVariable="getValueFromObjectReq" outputVariable="getValueFromObjectRes">
80  <correlations>
     <correlation pattern="out" set="atmInteraction"/>
82  </correlations>
   </invoke>
```

3.3 Challenges Summary

Most people and organisations aiming to develop model or code transformations only want to be concerned with those parts of their solution that are really specific to their usage scenario. They want to be able to reuse parts of existing solutions and compose them with minimal effort. As described in the previous sections, there arise a variety of challenges when people want to derive executable process descriptions (code or models) from higher-level process models. In Section 5 we present a model and code generation framework that allows us as far as possible to address the various challenges separately. The following list summarizes the requirements for the framework.

- Since there exists a hugh diversity of modelling languages and projects providing means to model processes (like UML activities, BPDM, PIM4SOA [20, p.51ff], AgilPro, etc.), the framework shall allow to decouple code generation from the format of the input models.
- It is necessary to apply more or less complex graph transformation algorithms to translate higher-level process models to workflow executable code like BPEL or XPDL [26].
- Depending on the process execution environment the code generation has to encode complex invocation patterns that include knowledge about the respective workflow execution engine. The framework has to provide means to easily describe, maintain and reuse these generation patterns independent of other information that is necessary for the code generation.

4 Technological Background

According to Czarnecki and Helsen [6] the majority of currently available MDSD tools support template-based model-to-text generation (e.g., openArchitectureWare [17], JET [11], or AndroMDA [3]). A template usually consists of the target text containing slices of metacode to access information from the source and to perform code selection and iterative expansion. Template approaches usually offer user-defined scheduling in the internal form of calling a template from within another template. In the case of our application scenario templates are a good choice to implement the (complex) invocation patterns for the respective workflow language and platform. Templates are close to the structure of the code to be generated and are perfectly suitable to iterative development as they can be easily derived from examples.

However, template-based approaches are not the best choice when the generated output of the code generation depends on some structure or additional information of the model. In our application scenario this is the control flow of the process. In workflow code generation the generated code often has a fix sequence of processing steps, like for example in BPEL, that depends on the control flow of the described process. Such problems can be addressed with visitor-based code generation approaches, for examples see Jamda[4] or MetaEdit+. These approaches provide a visitor mechanism to traverse the internal representation of a model that triggers the code generation.

Finally, our solution also needs to solve the problem of (often complex) graph analysis and transformation. This can be done in a preprocessing step before the real code generation. Graph transformation algorithms are implemented in programming languages like Java and work on an internal model representation. Hence, the graph analysis and transformation can be seen as a model-to-model transformation that realizes a direct manipulation approach [6].

5 Model & Code Generation Framework

To deal with the challenges described in Section 3.3 we applied the separation of concerns paradigm to the model and code generation framework. Solutions of the described challenges can be integrated in the framework from separate components, which overlap as little as possible. This allows flexible reuse and combination of components in the model and code generation framework.

We made the observation that solutions for some of the described challenges highly depend on the application domain, the modelling context, and the execution environment, while others are independent of the application domain. Hence, we introduced a common process modelling format and divided the framework into a domain specific and a domain independent part like in compiler theory [1], where an intermediate language is used to allow language independent code optimization. However, it is not totally correct to identify a front end and a back end like in compiler construction, since the front end (the *adapter for DSL model (I)*) and the code generator (the *generation templates (IV)*) both depend on the DSL specific process format and thus are not independent from each other. Figure 2 depicts a structural view on the generation framework.

– The *process transformer and optimizer* (II) and the *process visitor* (III) are domain independent. They address general graph transformation problems and graph traversing independent to any concrete process modelling languages. These components access process descriptions that are represented in a common process modelling format. The framework makes use of a

[4] http://sourceforge.net/projects/jamda

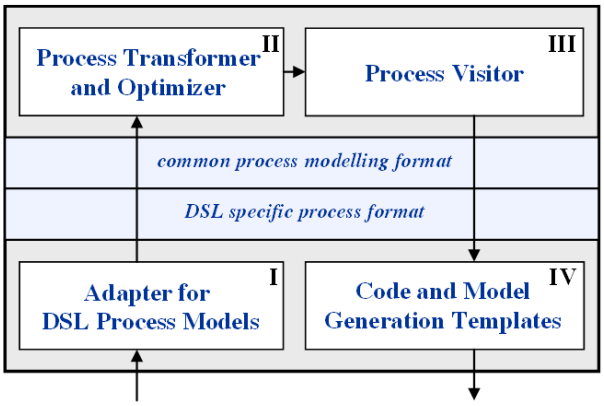

Fig. 2. Model & Code Generation Framework

process modelling format that was derived from the Standard Workflow Models [14]. For block-structured graphs it provides a common process modelling format that is based on BPEL.

– The other two components of the framework, the *adapter for DSL process models* (I) and the *code and model generation templates* (IV), directly access the process modelling format of the DSL that is used for modelling the input model. Hence, adapters and generation templates have always to be used in combination, i.e. they must use the same DSL specific process format.

The model and code generation framework not only allows to plug together components via common interfaces, but also provides a workflow that composes these components. Once the framework is configured, i.e. the components are registered and plugged in, the user only has to provide the input model and start the generation workflow of the framework. Figure 3 shows this generation workflow. It depicts the four states of the workflow execution (a)-(d) and the transitions.

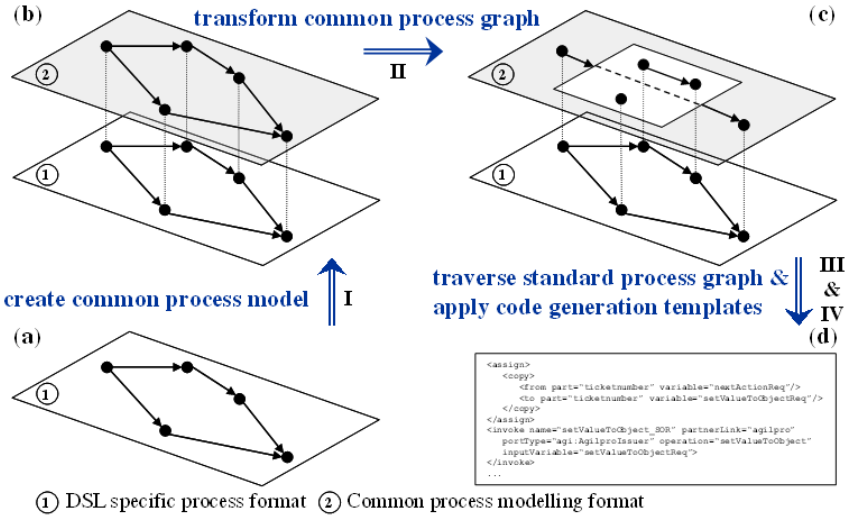

Fig. 3. Code Generation

(a) The first state of the workflow comprises the input process model that is represented in a format specific to the DSL used for modelling the input process.

(b) The second state is reached by applying the adapter on the input model. The adapter creates a representation of the input process in the common process modelling format. To ensure traceability between the processing steps of the two process models, it further links the processing steps in the common representation format to the processing steps of the input model.
(c) The process transformer restructures and optimizes the process represented in the common process modelling format. For example it generates a block-structured graph. Though in state (c) the control flow of the two process representations differs, their processing steps are still linked to the respective processing steps of the other representation format.
(d) The last transition is used for model and code generation and ends in the final state (d) of the framework's workflow. Therefore the process visitor traverses the process of state (c) that is represented in the common process modelling format. The code or model generation templates are called via a notification mechanism provided by the framework.[5] Like we can see in Figure 3 the workflow terminates with state (d) when the code or a new model was generated.

The main power of the model and code generation framework lies in the *process transformer and optimizer (II)* and the combination of a visitor-based and a template-based code generation approach (*(III) and (IV)*). (II) identifies SESEs in the process descriptions [8, 12]. SESEs are the basis for graph transformations that allow to generated block-structured (BPEL) code [22]. SESEs are also used to test the soundness of the process's control flow in reasonable time [24]. (III) and (IV) combine the advantages of vistor-based and template-based code generation approaches. The process visitor traverses the process flow of the input model and calls templates for workflow code generation. The visitor allows to generate the workflow code in the sequence that is given be the process's control flow. This is especially important for e.g. *Sequences* in BPEL, where the process steps are performed according to the order they have in the BPEL text file. The template mechanism has the advantage that generation templates can easily be derived from examples [6].

The framework allows graph transformation and flexible higher-level process descriptions into constructs that are provided through the target process execution language and supports users to easily implement code generation for process execution via complex invocation patterns.

6 Case Study for Workflow Code Generation

This section presents a case study that illustrates the code generation for the *Create Offer* process introduced in Figure 1 of Section 2.

6.1 Configuration of the Generation Framework

To generate BPEL code the generation framework has to be configured first. Hence, the process transformer that transforms arbitrary processes into BPEL and the process traverser that can process common block-structured process models are registered at the framework. While the process traverser is already provided by the framework, we use the Token Analysis component[6] as a process transformer. The adapter for AgilPro LiMo models and the respective BPEL code generation templates for the AgilPro JBoss workflow engine (jBPM) are also registered at the framework. Now, the workflow of the generation framework can be executed as shown in Figure 3.

[5] The framework implements the notification mechanism with the publish-subscribe pattern [7].
[6] http://sourceforge.net/projects/tokenanalysis/

6.2 Creation of Common Process Model

In the first step of the framework's workflow the adapter for AgilPro LiMo process models generates a representation of the input process in the common process modelling format. The resulting process is depicted in Figure 4 in UML concrete syntax (the common process modelling format itself has no concrete syntax representation). Further the adapter connects the respective processing steps of the two process representations.

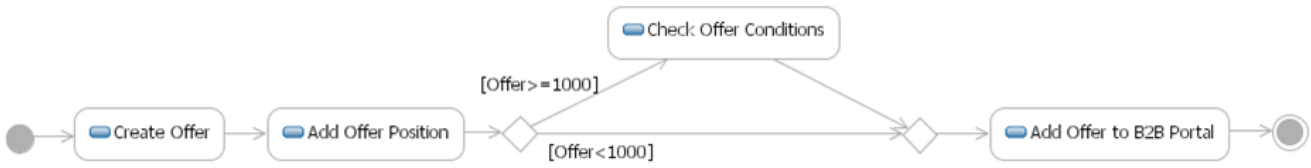

Fig. 4. Create Offer Process as Common Process Model

6.3 Transformation of Common Process Graph

The process transformer transforms the process generated by the adpater into a block-structured process. Figure 5 depicts this block-structured process. As we can see, an alternative block was generated from the decision and merge gateways.

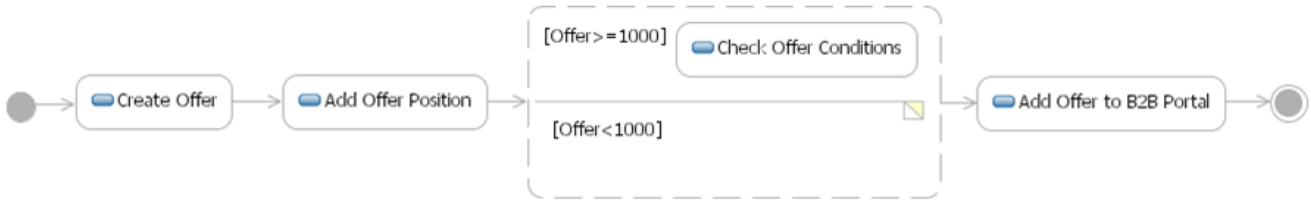

Fig. 5. Block-structured Create Offer Process as Standard Process Model in UML Syntax

6.4 Generating the BPEL Code

In a last step the block structured process is traversed and the BPEL code generation templates are applied. In the following we present some code excerpts generated by the code generation for the *Create Offer* process. Before the process visitor starts, statical information like *partnerLinks*, *variables*, and *correlationSets* are generated for the BPEL process. Further some initial invocations on the process engine are made. Then the visitor starts to traverse the block-structured process description. For each processing step a *scope* is generated that gets the name of the step and contains the further processing instructions that are necessary for this step. The first process step the visitor accesses is the *Create Offer* step. For this step the BPEL workflow code depicted in Listing 1.2 of Section 3.2 is generated within a *scope* element.

7 Conclusions

In the context of executable BPEL workflow code, there exist a variety of solutions that provide hardly more than another concrete syntax (graphical instead of textual) for BPEL (cp. UML profile for BPEL [2], Oracle BPEL Process Manager [21], or the ActiveBPEL designer [15]). These solutions do not narrow the gap between higher-level process descriptions and workflow execution. Tool chains that allow model-driven development and the generation of BPEL code like the IBM Tool Suite[7], still have restrictions that prevent all process models from being fully transformed [25, p.109]. These approaches further require manual model refinement at multiple abstraction levels. [13] describes a generic mapping approach of business process models to other process-oriented representations by the means of XPDL. To our experience a generation of XPDL from higher-level process models does not have to deal with the same challenges than a generation of BPEL, since the sequence of model elements in a XPDL model does not determine the process's control flow and XPDL does not require block-structured processes.

In this paper we developed a model and code generation framework that fosters the generation of executable workflow code. The generation framework separates tasks that occur during workflow code generation into separate, reusable components. Its main contribution is to decouple components that depend on the domain and the execution environment from components that deal with computational aspects like control flow analysis and transformation. This allows better reuse of the knowledge encoded into adapters, graph-transformation algorithms and code generation templates. If e.g. BPEL code in another version (2.0 instead of 1.1) shall be generated from a model, only the respective code generation templates have to be adjusted. The framework combines the advantages of visitor-based approaches, template-based approaches, and graph transformation techniques. The visitor allows to generate the workflow code in the sequence that is given be the process's control flow and the templates can be derived from examples. For the components that realize the process transformer and optimizer one is free in the choice of an implementation technology.

The generation framework can be applied to any DSL that is concerned with process description. For the DSMs only the respective adapter(s) have to be implemented. Adapters for DSMs can parse XMI files or access modelling APIs. The generation framework can even be applied, when the source model does not contain all necessary information to generate executable workflow code. In this case the generation templates are replaced with code that constructs a refined model. An example would be to generate a BPEL UML profile [2] model, which is manually refined.

We have implemented workflow code generations in various projects. In the SPL4AOX project (see Example 1 in Section 3.2) we had an effort of about 2.5 man month to specify and implement BPEL code generation with the oAW Xpand and the oAW Extend language. The input models for this transformation are constrained to block-structured process models. Our second implementation was done for a prototype of the AgilPro project (see Example 2 in Section 3.2). We used JET templates combined with Java code. The solution is able to deal with a limited set of cycles in the control flow. The effort was approximately 1 man month, where most of the time was spent on constraining and transforming the control flow of the input models. However, the parts of the this implementation did not lend themselves for reuse, since graph transformation and code generation was combined. Based on these experiences we developed the generation framework and a graph transformation component with an effort of 4 man month[8]. Finally, another person, which *had no experience* with code generation and workflows, implemented the code generation for AgilPro in

[7] http://www-306.ibm.com/software/websphere/

[8] The implementation is part of the Workflow Generation Framework project at http://sourceforge.net/projects/wf-codegen

3 days with our generation framework. This person simply had to copy the process graph of the input model in the AgilPro adapter and derive the generation templates from examples.

Time saving using the generation framework in contrast to other approaches will always depend on the complexity of the input graphs (e.g. with or without cycles) and on the experience of the generation developer. However, our model and code generation framework makes it possible for people with no or little experience in code generation and graph transformation to produce workflow code at reasonable time.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.
2. J. Amsden, T. Gardner, C. Griffin, and S. Iyengar. Draft uml 1.4 profile for automated business processes with a mapping to bpel 1.0, version 1.1. *IBM developerworks*, 2003.
3. AndroMDA. Andromda. http://www.andromda.org/.
4. B. Bauer, G. Palfinger, F. Lautenbacher, and S. Roser. "agilpro": Modellierung, simulation und ausführung agiler prozesse. *Objekt Spektrum*, 2007.
5. BPEL4WS. Business process execution language for web services version 1.1, 2003.
6. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
7. E. Gamma, R. Helm, and R. E. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, 1995.
8. M. Götz, S. Roser, F. Lautenbacher, and B. Bauer. Using token analysis to transform graph-oriented process models to bpel. 2007.
9. J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley Publishing Inc., 2004.
10. B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006.
11. JET. Java emitter templates (jet). http://www.eclipse.org/modeling/m2t/.
12. R. Johnson, D. Pearson, and K. Pingali. The program structure tree: computing control regions in linear time. In *Conference on Programming language design and implementation*, pages 171–185. ACM Press, 1994.
13. J. Jung. Meta-modelling support for a general process modelling tool. In *5th OOPSLA Workshop on Domain-Specific Modeling*, 2005.
14. B. Kiepuszewski, A. H. ter Hofstede, and W. M. van der Aalst. Fundamentals of control flow in workflows. *Acta Informatica*, 39(3):143–209, 2003.
15. P. T. Maurer. Activebpel 3.0 from active endpoints, inc. *SOA World Magazine*, pages 22–23, 2006.
16. J. Mendling, K. Lassen, and U. Zdun. Transformation strategies between block-oriented and graph-oriented process modelling languages. In *Multikonferenz Wirtschaftsinformatik (MKWI)*, volume 2, pages 297–312. GITO-Verlag, 2006.
17. oAW. openarchitectureware (oaw). http://www.openarchitectureware.org/.
18. OMG. Mda guide version 1.0.1. omg/2003-06-01, 2003.
19. OMG. Business process modeling notation specification, final adopted specification. dtc/06-02-01, 2006.
20. OMG. Uml profile and metamodel for services - for heterogeneous architectures (upms-ha). ad/2007-06-02, 2007.
21. Oracle. *Oracle® BPEL Process Manager, Developer's Guide*, 2005.
22. C. Ouyang, M. Dumas, A. H. ter Hofstede, and W. M. van der Aalst. Pattern-based translation of bpmn process models to bpel web services. *International Journal of Web Services Research (JWSR)*, 2007.
23. A.-W. Scheer. Aris - vom geschäftsprozess zum anwendungssystem. Springer Verlag, 1998.
24. J. Vanhatalo, H. Völzer, and F. Leymann. Faster and more focused control-flow analysis for business process models through sese decomposition. In *5th ICSOC Conference*, LNCS, pages 43–55. Springer, 2007.
25. U. Wahli, L. Leybovich, E. Prevost, R. Scher, A. Venancio, S. Wiederkom, and N. MacKinnon. Business process management: Modeling through monitoring using websphere v6 products. IBM Redbook, 2006.
26. WfMC. Process definition interface – xml process definition language. WFMC-TC-1025, 2005.