A Framework for Creating Domain-specific Process Modeling Languages

Henning Heitkötter
Department of Information Systems, University of Münster, Münster, Germany

Keywords: Business Process Modeling, Domain-specific Languages, Model Transformation, BPMN 2, EMF.

Abstract: Using domain-specific modeling languages to capture business processes can greatly enhance quality and efficiency of process modeling, because language and models are more expressive, concise and easy to understand. The development of domain-specific languages (DSLs) with accompanying tools and transformations is, however, a complex, time-consuming, and costly task. An efficient and simple approach to creating process modeling languages (PMLs) for specific business domains by reusing common parts is needed, where each resulting language is still optimally adjusted to its domain. For each of these languages, the abstract and concrete syntax have to be defined as well as transformations to more general languages.
This paper presents DSLs4BPM, a generic framework for PMLs, which employs DSL modularization concepts to allow the derivation of domain-specific PMLs. The framework provides elements common to process modeling and a basic transformation to the generic Business Process Model and Notation 2.0. DSLs are created by adding own types to the framework language and own rules to the transformation at predefined extension points. The approach has been implemented based on the Eclipse Modeling Framework.

1 INTRODUCTION

Business process modeling (BPM) represents enterprise processes as models, often using general-purpose process modeling languages (PMLs). PMLs like Business Process Model and Notation (BPMN, (Object Management Group, 2011)) qualify as general-purpose in the sense of not being constrained to processes of a particular business domain. They feature only domain-neutral concepts like the generic term "Activity" and are particularly suited for analysis purposes and technical tasks. The latter includes, for example, workflow management and service orchestration. Since there are several tools and execution engines supporting popular languages like BPMN, process modelers often use these general-purpose languages directly to describe their business processes. However, because of their widespread area of application, these languages tend to be rather complex (Recker et al., 2009). They offer a low level of abstraction in order to be applicable to all general scenarios and have a highly technical appearance. Consequently, models in these languages tend to be verbose and difficult to understand. The modeling process itself may be time-consuming and error-prone.

Domain-specific languages (DSLs) try to address these issues by focusing on a particular (business) domain. They trade generality for an optimal representation of concepts from their domain (van Deursen et al., 2000) and try to reach an adequate level of abstraction (Mernik et al., 2005). Instead of complex combinations of generic elements, domain concepts can be expressed through exact, concise and semantically rich elements. For example, in the domain of banking, a single element could represent a domain-typical, complex series of process steps, like formal assessment of a case. Business process models using a DSL are more expressive and easier to create and understand, also and especially for domain experts.

Model-to-model transformations can still generate models in lower-level general-purpose languages by explicitly expressing the semantics of the DSL using low-level technical artifacts. Thus, existing execution engines or analysis tools designed for a technical language like BPMN can be used, even though they cannot handle the DSL itself. A transformation from the aforementioned banking DSL to BPMN might, for example, replace each instance of the formal assessment element by a complex network of BPMN activities expressing the same semantics. In this sense, domain-specific models are an important component of model-driven development (MDD). The integration of DSL and transformation combines some advantages of general-purpose and domain-specific lan-

Heitkötter H..
A Framework for Creating Domain-specific Process Modeling Languages.
DOI: 10.5220/0004024201270136
In *Proceedings of the 7th International Conference on Software Paradigm Trends* (ICSOFT-2012), pages 127-136
ISBN: 978-989-8565-19-8

guages.

Developing useful DSLs requires considerable effort and cost, not least because they consist of several parts: They have an abstract syntax describing the structure of their domain in terms of concepts and their relationships. One or more concrete syntaxes define the textual or—more common in process modeling—graphical notation of the DSL. A mature tooling infrastructure is needed to make modeling with a DSL efficient. As motivated above, DSLs should provide transformations to lower-level modeling or programming languages, taking into account the semantics of both, DSL and target language. Developing these model transformations from scratch is not a trivial task, either.

An efficient and simple approach to creating domain-specific PMLs with all their components would decrease the initial investment needed to set up the DSL. This paper presents our *DSL Framework for Business Process Modeling* (*DSLs4BPM*), from which domain-specific PMLs can be derived in a straightforward way. The framework consists of a basic language and a transformation to BPMN 2. The framework language provides elements common to process modeling languages and means for quickly adapting the framework language to a business domain. New, domain-specific concepts are added to the framework when designing a domain-specific language. The transformation to BPMN 2 that complements the language maps the language elements of the framework to corresponding concepts in BPMN 2. The transformation provided by the framework is adapted concurrently with the language. Thus, a DSL and its mapping to a general-purpose language are optimally adjusted to the domain. At the same time, common parts are reused.

Our main contribution consists of this framework, which allows the easy and efficient creation of process modeling DSLs for different domains in combination with adapted transformations to BPMN 2. Besides, our approach uses extensibility as proposed in the research literature about reusing and modularizing DSLs (Spinellis, 2001; Krahn et al., 2008; Voelter, 2010). Hence, our paper also demonstrates on a more general level how a language can be designed with reuse and adaptability in mind by offering extension points for extending language concepts and transformation rules in parallel. It also elaborates on the support for these means in the well-known and mature Eclipse Modeling Framework (EMF, (Steinberg et al., 2009)), proving that modular DSLs can be built with the pragmatic mechanisms of EMF.

The remainder of this paper is structured as follows. Section 2 on related work discusses approaches for creating domain-specific PMLs as well as general literature on reusable DSL design. It is followed by an introduction into process modeling, both from a general (BPMN 2) and from a domain-specific viewpoint (PICTURE). Both languages are prerequisites for the following sections. Section 4 describes the general design of DSLs4BPM. The language part of the framework is presented in section 5, section 6 outlines the transformation. Section 7 gives some details regarding the implementation and describes how to create derived DSLs by adapting the framework. We discuss and evaluate our framework in Section 8. The paper concludes with a summary and an outlook.

2 RELATED WORK

Research and praxis have proposed and evaluated several general-purpose PMLs. Besides BPMN, covered in the next section, the most prominent of these are Activity Diagrams from the Unified Modeling Language (UML, (Object Management Group, 2010)), studied in (Dumas and ter Hofstede, 2001), and the more technical Business Process Execution Language (BPEL, (OASIS Standard, 2007)). These languages do not consider any particular business domain. PICTURE, also described in section 3, is an example of a domain-specific PML. The focus of our work, however, does not lie on a single modeling language, but on creating DSLs for process modeling in an efficient manner and on combining these with corresponding transformations.

Creating domain-specific PMLs efficiently is also the focus of jABC (Steffen et al., 2007) and its predecessors (Steffen and Margaria, 1999; Margaria and Steffen, 2004). jABC employs a building block-based approach to the model-driven development of services and applications, similar to DSLs4BPM. Executable process models are composed from so called Service Independent Building Blocks (SIB). A SIB corresponds to a Java class with execution code, so that models are directly executable or can be transformed to application code. As our approach instead targets BPMN 2, it provides a higher abstraction level and supports a stepwise model-driven development, while retaining executability through the respective features of BPMN. jABC is a self-contained framework, whereas the process modeling language framework presented in this paper is implemented on top of standard modeling technologies (EMF), enabling its integration into larger MDD projects.

Brahe and Østerbye use UML activity diagrams and the profiling mechanism of UML to facilitate the definition of domain-specific PMLs (Brahe and Øster-

bye, 2006). Their tool generates UML profiles for individual domains. Profiles allow the specialization of generic UML constructs and are an example of an adaptation mechanism in general-purpose modeling languages. BPMN 2 offers a similar extensibility concept. DSLs4BPM does not make use of this approach to adaptation in order to avoid the overhead and suboptimal domain representation of a general-purpose language. Instead, the framework language provides a minimal set of common concepts and has to be extended for specific domains, which ensures flexibility and expressiveness. In contrast to our framework, Brahe and Østerbye do not consider model transformations for their DSLs.

DSLs4BPM uses modularization concepts for DSLs proposed in the literature. Spinellis identifies recurring patterns in the development of textual DSLs, of which some describe approaches to reuse (Spinellis, 2001). Our framework can be seen as an application of the extension pattern, because new elements are added to an existing language. In our framework, however, extensions only take place at particular points (Section 4). These extension points provide some guidance to language developers, which is not considered by the pattern.

Völter identifies several modularization concepts available in the language workbench JetBrains MPS (Voelter, 2010), one of them being extension, i.e., inheritance between languages and language concepts, in combination with transformation overriding. Other concepts available in MPS like embedding of languages are not needed in our framework. Our framework does not use JetBrains MPS but EMF, which provides less modularization concepts, and offers the remaining concepts less prominently and less easily accessible. However, as shown by our implementation, EMF's capabilities regarding DSL modularization are sufficient for creating a powerful framework for the efficient creation of process modeling DSLs.

3 PROCESS MODELING

3.1 General-purpose: BPMN 2

The Object Management Group's (OMG) specification Business Process Model and Notation (Object Management Group, 2011) describes a PML that is widely used and supports analysis and execution of business processes. Version 2.0 introduced a complete metamodel not available in previous versions and enhanced the support for executable models.

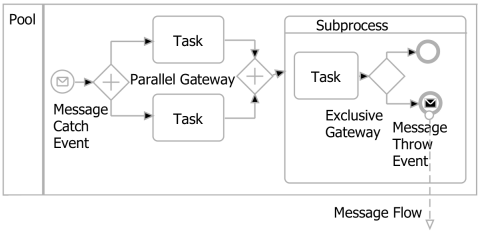

Figure 1: Example model of a BPMN process.

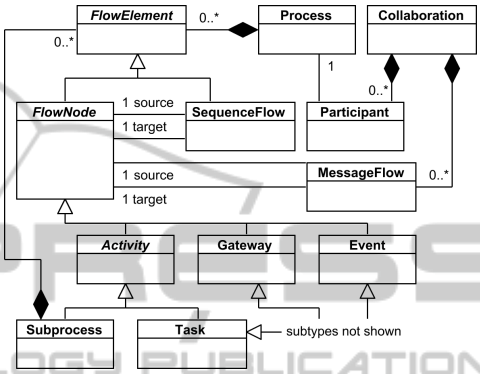

Figure 2: BPMN metamodel (simplified excerpt).

Tools like Activiti[1] can directly execute processes modeled in BPMN 2 by managing the workflow of processes and by orchestrating services. Due to its popularity and good tool support, BPMN is the target of our framework's transformation. To introduce the concepts that are important for the transformation, this section gives a short overview of BPMN 2.

The example of figure 1 highlights important elements of BPMN. It depicts a *Process*, which forms a central model element of BPMN. It is a graph of so called *Flow Nodes* connected by directed *Sequence Flow* edges. BPMN provides several types of flow nodes, including *Task*, *Gateway* and *Event*. Hierarchical nesting is supported by the concept of *Subprocesses*. Tasks represent units of work and are provided in different specializations to accommodate different kinds of activities, for example *User Tasks* or *Service Tasks*. Gateways are used to split the control flow of a process into parallel or exclusive branches as well as to merge such branches. Events symbolize that a process either triggers some incident (*Throw Event*) or waits for its occurrence (*Catch Event*). Additionally, events are differentiated with regard to the type of incident, for example message or timer. In BPMN 2's metamodel (figure 2), flow nodes and sequence flow are subsumed under the term *Flow Element*. A process is a collection of flow elements and is performed within an organization.

[1] http://www.activiti.org

A business process is made up of several such BPMN processes, each representing the work carried out within one organization, called *Participant* in this context. A *Pool* is the graphical representation of a participant in a BPMN diagram and contains the visual depiction of the participant's process. A set of interacting processes forms a *Collaboration*. *Message Flow* symbolizes the exchange of information between two different processes.

As BPMN is domain-independent and has a rather technical appeal, it is not an appropriate source model for MDD. Recker et al. analyze the modeling process with BPMN (Recker et al., 2010; Recker, 2010). They find BPMN to be complex and in some part confusing for modelers. Deficiencies of the language include ambiguous and redundant elements, as well as lack of support for business rules and process decomposition. However, these mainly impact the modeling process. As described before, BPMN is useful as the technical specification of a process. Our approach renders direct modeling in BPMN, which was found to be a non-trivial task, unnecessary, but provides BPMN models via a transformation.

3.2 Domain-specific: PICTURE

While there are several general-purpose PMLs, individual DSLs have not yet surfaced as prominently in process modeling as they have in other areas. This might be due to the effort necessary for developing such a DSL. However, as the discussion above and similar approaches from related work highlight, the usage of DSLs could enhance quality of process models. PICTURE, a DSL for business processes in public administration (Becker et al., 2007), demonstrates the general viability of DSLs for process modeling. It was developed to efficiently model the process landscape of public administrations and was subsequently transferred to the domain of banking (Becker et al., 2009b). It restricts the modeler's freedom in order to reduce complexity. Models in PICTURE follow a clear hierarchical structure and use straightforward, comprehensible means of expressing control flow. Typical activities in public administration or banking, respectively, are available as building blocks to describe the individual steps of a process in a condensed manner. As the DSL furthermore does not feature redundant or ambiguous elements, a modeler with experience in public administration or banking can efficiently model processes with PICTURE.

In summary, PICTURE allows for creating concise process models that are nevertheless expressive and include all information relevant for their respective purpose. PICTURE has been used and evaluated successfully in several projects (Karow et al., 2008; Matzner et al., 2009) and through experiments (Becker et al., 2009a). Therefore, it offers a good basis for the language of DSLs4BPM. Our framework language has the same structural approach to process modeling as PICTURE, but encompasses only those core elements of PICTURE that are domain-independent. As PICTURE has been successfully transferred to other domains, it is a viable basis for the framework language. PICTURE models for public administrations have been successfully transformed to BPMN (Heitkoetter, 2011). This suggests that it will also be possible to transform models based on a framework language that has been derived and generalized from PICTURE's structural parts.

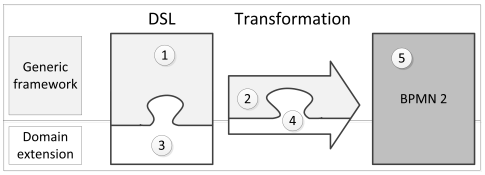

Figure 3: Overview of framework (①+②) and an extension for a particular domain (③+④).

4 DESIGN OF FRAMEWORK

Figure 3 gives an overview of our framework for the integrated creation of process modeling DSLs and transformations. DSLs4BPM consists of a generic PML (① in figure 3) and a transformation (②) mapping the generic concepts to BPMN 2 (⑤). Language and transformation provide the basic structure and are explicitly designed to be extended. Domain-specific languages can be derived from the framework by extending the generic language at predefined extension points with domain-specific constructs (③). At the same time, the transformation should be adapted to the new language elements and their semantics by overloading specified rules (④). The new rules should transform the domain-specific constructs into corresponding elements from BPMN. They can also adapt the behavior of the general transformation where necessary. These partial transformations are seamlessly integrated into the general transformation by the principle of Inversion of Control.

The idea behind this approach is to generalize similarities of PMLs and to provide a complete transformation framework for these common concepts. Business processes of a particular domain feature recurring patterns that are easily identifiable and enumerable. These typical activities, each consisting of several steps, have a meaning for domain experts and

are thus possible elements of an expressive DSL. The already mentioned formal assessment in banking is an example of such a typical activity. Besides these individual activities, differing from domain to domain, process models mainly feature similar elements across domains, for example various kinds of control flow and a hierarchical structure, that can be generalized and extracted into a generic language.

Hence, the framework is responsible for common concepts of structured processes, while the individual, derived DSLs handle domain-specific concepts, for example from banking, insurance or public administration. This allows for a less complex creation of business process DSLs that are nevertheless expressive and concise. Additionally, an adapted transformation to BPMN 2 is available with minor investment. Thereby, the process modeling framework aims to overcome the gap between expressive DSL models and widely usable general-purpose models, combining the advantages of DSLs and BPMN 2.

DSLs4BPM provides means for structuring a process as well as for describing the process flow in terms of sequences, variations and communication. Moreover, organizational aspects are considered. The main extension point offered by the general framework is the metamodel element *Process Building Block*. It is intended to be subclassed by derived DSLs. Each subtype should correspond to a typical, potentially complex, but clearly delimitable activity of that domain. The framework's core transformation contains a placeholder rule for building blocks. Derivations should provide a suitable mapping for their specific subtypes that replaces or augments the default behavior to ensure an adequate transformation. Hence, the additions to our framework made by derived DSLs and their transformations take place at predefined points by means of subclassing. This is a less general, but more accessible notion of extensibility than the analogous pattern for DSLs in (Spinellis, 2001), which covers arbitrary additions. The coupling of DSL extension with adaptation of a transformation is also special to our approach. Suchlike extension points allow a better semantic integration of new types via the transformation.

5 FRAMEWORK LANGUAGE

The framework language has been adapted from PICTURE by generalizing concepts and focusing on domain-neutral aspects. Since the general alignment and structure of PICTURE have been preserved, DSLs4BPM inherits the benefits of that well-tested DSL. Modeling of structured processes is easier and more efficient due to a lower complexity of the language. Models built with a DSL derived from DSLs4BPM are concise, expressive, and easy to understand. The easy and understandable syntax is achieved by focusing on processes that are at least moderately structured and for the most part linear. The number of options available to express a concept and by it the modeler's freedom is intentionally restricted in order to reduce variability. In the following, the domain-independent framework language is described along with its metamodel (figure 4) and extension points for derived DSLs. Figure 5 displays an example model in a derived DSL that resembles the original PICTURE for public administration.

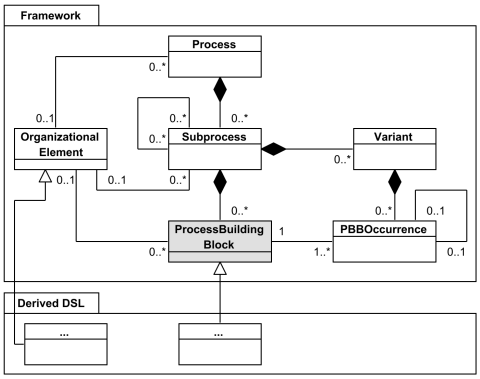

Figure 4: Metamodel of framework language (excerpt) with extension points.

A (business) *Process* is divided into several *Subprocesses* along organizational or functional criteria. Predecessor-successor relationships between subprocesses determine the order of execution. A subprocess can have one or more predecessors and successors. Its execution begins when all preceding subprocesses have been completed. Additional constraints enforce the absence of cycles, so that a process is a connected, directed acyclic graph of subprocesses. Roughly speaking, a subprocess is a linear sequence of *Process Building Blocks*. These blocks will generally be instances of domain-specific subtypes. The domain-specific building blocks themselves can encapsulate complex semantics, but on the subprocess level they are seen, in principle, as executed sequentially with their inner semantics hidden. The modeler chooses from the set of building block types available in the respective DSL and aligns instances in the order they appear in the process flow. Building blocks have attributes, e.g., name, which must or can be set on instances. Domain-specific building blocks will define their own attributes as needed. Through these attributes, instances can match their appearance in the process at hand and represent the respective step of the process more precisely. This simple way of ar-

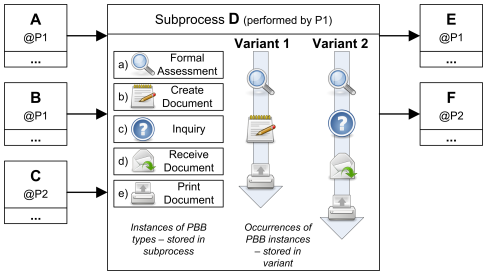

Figure 5: Example model of a process in a derived DSL (with six subprocesses A-F performed by two organizational elements P1, P2).

ranging the constitutive elements of a process linearly considerably simplifies modeling. A large part of process complexity is handled by the domain-specific building blocks, which nevertheless are intuitively accessible to the domain-experienced modeler, because they stand for known activities of the domain.

Some subprocesses require, however, the possibility to model variations in the sequence of building blocks, like skipped or additional blocks or a different order of execution depending on certain conditions. In the example of figure 5, the second variant of subprocess D replaces the *Create Document* instance of variant 1 with an *Inquiry* followed by a *Receive Document* building block. To accommodate these needs, the framework language uses the concept of subprocess *variants*. Actually, the way of modeling described above takes place entirely on the variant level. Each variant is a complete and separate linear sequence of building blocks. An instance of a building block including its attribute values can appear in one or more variants and is shared among the variants of a subprocess, enabling the reuse of parts of the subprocess. In figure 5, both variants contain the same *Formal Assessment* instance (*a*) and the same *Print Document* instance (*e*).

A typical workflow when modeling the course of a subprocess is to first create the main variant representing the normal case, which is then copied to a new variant and modified wherever the respective alternative differs from the original. The modeler is able to model every distinct case separately without influencing other cases. If the process flow has only a moderate amount of variation, this way of modeling proves to be very efficient (Becker et al., 2007). Additionally, the resulting models provide a high readability, as differences are clearly separated. The first and last building block must be identical across all variants of a subprocess. This constraint makes models easier to understand, guarantees strictly defined semantics and simplifies the transformation.

The underlying metamodel of the framework language (figure 4) incorporates the previous explanations with an *occurrence* concept. Variants as well as building block instances are stored at subprocess level, so that blocks can be shared among variants of their subprocess. Each variant contains a collection of building block occurrences referencing the building blocks that are present in the variant. A linear predecessor-successor-relationship defines the order of occurrences separately for each variant. The organizational view of the process is incorporated by a generic *organizational element*. This is another extension point for a derived DSL, as organizational structures will be domain-specific. Process, subprocess and building block specify the organization in charge of them by a reference to this organizational supertype. Resources and data can be attributed similarly to further describe building block instances.

6 TRANSFORMATION TO BPMN 2

The transformation to BPMN 2 aims to generate executable models. It thus enables or facilitates the implementation of supporting systems. The generated BPMN models explicitly represent both the complex activities embodied in domain-specific building blocks and the control flow implicitly existing in framework models. The transformation is an inherent part of the framework and a complete mapping of the framework language. Derived DSLs adapt the transformation by supplying custom transformation rules for their own elements.

The framework's generic transformation to BPMN 2 is based on a previous domain-specific transformation from PICTURE to BPMN 2 (Heitkoetter, 2011). That transformation was tied to the domain of public administration, while the focus here lies on the domain-neutral concepts and their generic mapping to BPMN 2. The rules from the original transformation that are concerned with domain-neutral structural concepts also apply to the framework transformation. The following gives a high-level overview of the transformation and otherwise focuses on the newly introduced extension points and their integration into the transformation. For more details and reasoning regarding the structural rules please refer to (Heitkoetter, 2011).

A process of the framework language is mapped to a BPMN collaboration with several participants. Each participant represents one particular organization of the input model. The corresponding pool depicts all subprocesses performed by that organization. Subprocesses are represented by the mapping of their building block instances. Every building block in-

stance is transformed only once, irrespective of the number of occurrences in variants, so that each occurrence maps to the same representation. Hence, this representation can have more than one predecessor and successor in BPMN, respectively. The transformation connects the representations of building blocks of a subprocess as determined by their occurrences in variants, using sequence flow and, if necessary, exclusive gateways to model the process flow. Through these transformation rules, a framework subprocess is represented in BPMN as a connected graph of representations for its building blocks. The transformation connects these individual graphs according to the connections between subprocesses, generating sequence or message flow edges, parallel gateways, and message events as necessary.

The generic transformation provided by the framework transforms a building block to a single task. Building block subtypes of derived DSLs can be mapped to arbitrary graphs of flow elements. The core transformation is prepared to handle these more complex representations. It only expects that the BPMN representation of a building block is a collection of flow elements (flow nodes and sequence flow) with one entry and one exit point. These restrictions enable the transformation to integrate any representation conforming to this basic skeleton into the BPMN process. Besides the default generic task, a building block can thus be mapped to a specific type of task (e.g., User Task or Service Task), to an event, to a subprocess which in turn contains a connected graph of flow nodes, or directly to such a flow graph. The restriction to flow graphs with unique source and sink is due to the linear nature of variants. The execution of a building block can nevertheless influence the process flow because it modifies the process state.

7 IMPLEMENTATION

The implementation of DSLs4BPM is based on the Eclipse Platform[2] and uses the Eclipse Modeling Framework (Steinberg et al., 2009), which provides means to define modeling languages and handles models in these languages. The metamodel of the framework language is described with EMF's Ecore language. An implementation of BPMN 2's metamodel with EMF is available through an Eclipse project[3]. The model-to-model transformation is written in the Operational Mapping language of OMG's Query/View/Transformation standard (Object Management Group, 2008), of which an EMF-based implementation is available in the Eclipse project QVT Operational[4] (QVTO).

All elements of the framework language from section 5 have been implemented in Ecore, including the generic type *Process Building Block*. EMF's *EClass* type, which is used to describe these metamodel elements, supports object-oriented inheritance. Therefore, implementing derived DSLs is straightforward. The DSL designer has to provide a new Ecore model with the domain-specific building block types as subclasses of the generic building block type from the framework metamodel. These subclasses can have their own attributes and references. Models built in the derived DSL resort to the structural elements from the framework language and to the individual building blocks from the DSL.

The core transformation included in DSLs4BPM takes a model written in the framework language as input and outputs a BPMN model that complies with BPMN 2's metamodel and semantics. It implements the transformation rules of section 6, which deal with elements from the framework language. Each core transformation rule creates all BPMN elements that make up the representation of the framework element handled by the rule, including helper elements and connections, and calls the rules for child elements. The additional rules for a derived DSL have to be defined in a separate transformation.

Listing 1: Excerpt from core transformation.

```
1  helper ProcessBuildingBlock::transformPBB()
2      : Collection(FlowElement) {
3    var result : List(FlowElement) := List{};
4    result += self.map toFlowGraph();
5    // create connections to successors
6    // create gateway, if necessary
7    return result;
8  }
9  mapping ProcessBuildingBlock::toFlowGraph()
10     : List(FlowElement) {
11   result->add(self.map toSingleTask());
12 }
13 query ProcessBuildingBlock::connectorIncoming()
14     : FlowNode {
15   return self.map toSingleTask()
16 }
```

The core transformation contains several operations related to building blocks (listing 1). `transformPBB` is responsible for transforming a building block and integrating it into the process (lines 1–8). The actual mapping of the building block to a flow graph as described in section 6 is expected to be performed by `toFlowGraph` (9–

[2]http://www.eclipse.org

[3]http://wiki.eclipse.org/MDT/BPMN2

[4]http://wiki.eclipse.org/M2M/QVTO

12). The core transformation resorts to the query `connectorIncoming` to determine the entry point of the representation when connecting it (13–16). An analog query exists for the exit point. In case of the generic building block, both of these methods return the single task created by `toFlowGraph`.

In order to represent each domain-specific activity accurately in BPMN, tailored mappings for domain-specific buiding block types have to be provided. The new transformation that transforms a derived DSL has to extend the framework transformation through QVTO's keyword `extends` (listing 2, lines 1–3). As part of the new transformation, the mapping `toFlowGraph` and both connector queries have to be overridden for each building block subtype that needs a specific mapping, in this example `SpecialPBB`. The mapping operation creates the flow nodes and sequence flow edges that make up the specific representation and returns them as a collection of flow elements (4–8). It can resort to the values of attributes defined for the subtype or to configuration parameters in order to further tailor the mapping, which thus does not have to be static. For example, it could use a project-specific parameter as the address of a Web service in a Service Task or modify the BPMN representation of a building block instance according to the instance's attribute values. The connector queries must return those flow nodes out of the flow graph that act as entry or exit point, respectively (9–11). When an instance of a subtype is encountered, the overridden methods are called instead of the default ones according to the Inversion of Control principle. This way the transformation can incorporate the elements of the adapted representation into the result model and create connections to the appropriate elements. In addition to this general mechanism, the framework offers shortcuts for certain kinds of extension, for example a mapping to a single Service Task.

Listing 2: Extended transformation (excerpt).

```
1 transformation dsl2bpmn2(in src : dsl,
2   out trgt : bpmn2)
3   extends transformation framework2bpmn2;
4 mapping SpecialPBB::toFlowGraph()
5     : List(FlowElement) {
6   result->add(self.map toTaskOne());
7   // create other elements of the flow graph
8 }
9 query SpecialPBB::connectorIncoming():FlowNode{
10   return self.map toTaskOne()
11 }
```

Besides the extended mappings, nothing has to be defined for an individual DSL, as the framework already defines most of the transformation. If necessary, a modification of core transformation rules is possible via QVTO. The extended transformation is defined during the development of the derived DSL. The modeler who eventually uses the DSL is not concerned with these tasks. QVTO's extension mechanisms in addition to the polymorphism in EMF thus enable an easy and straightforward way to adapt the framework's transformation.

8 DISCUSSION

In the following, we evaluate DSLs4BPM with respect to the development effort needed when designing new DSLs and transformations. Afterwards, we study the conceptual advantages and contributions. We also highlight potential areas for improvement.

The savings that can be expected with our results can be approximated by looking at the source code of our framework and typical extensions compared to standalone implementation of DSLs. The framework language consists of 16 classes with just over 50 distinct features. The basic transformation to BPMN 2 included in the framework has been implemented in approximately 650 logical lines of code of QVTO. As this code transforms the basic process structure and control flow, nearly all of it would also be required in a standalone implementation that does not use the framework. The overhead of the framework transformation consists of statements due to provisions for adaptation. These lines, which would not be needed in a single, standalone transformation, amount to less than five percent of the transformation's source code. Regarding derived languages, only one additional class due to peculiarities of EMF is needed in a derived DSL besides the specific concepts introduced by that DSL, so that the overhead is minimal. With respect to an extended transformation, each new concept that has to be transformed to BPMN requires a corresponding adapted transformation rule. Each of these adaptations consists of as many statements as needed to generate the particular BPMN representation. These mostly perform simple, but laborious work to build up the corresponding flow graph and would be needed in either case, so that there are only 0-3 lines of overhead per concept, depending on the complexity.

In summary, DSLs4BPM introduces only a small amount of overhead, while relieving DSLs of the most complex parts of the transformation, namely the representation of structure and of control flow. This greatly reduces the development effort for implementing individual DSLs, which mainly does not extend beyond a straightforward implementation of concepts and mappings that have been identified during the de-

sign of the DSL. The expected savings have been confirmed in two case studies with DSLs for public administration and for banking, respectively.

Our framework suggests a streamlined adaptation process for deriving a DSL in combination with a transformation to BPMN 2. The design of the DSL requires the identification of domain concepts that should be added to the framework language, mainly as new building block types. For each of these new elements, relevant attributes have to be identified, as well as its mapping to BPMN. In case of building blocks, the latter should be described in the form of a BPMN subgraph that represents the typical activities behind this building block type. Currently, the implementation has to be done manually with help provided by the respective Eclipse tools. In principle, large parts of the adaptation process following the conceptual design of a DSL can be supported by a wizard and automated. The wizard would have to gather the aforementioned information from the language designer, i.e., the list of new concepts and their representation as a BPMN graph. For the latter, the wizard could resort to a BPMN diagram tool. Based on the gathered information, the wizard would create the derived DSL and the source code of an extension of the transformation in QVTO[5]. Only a relatively small amount of additional work would be needed to adapt the automatically generated transformation to special requirements, for example if attribute values influence the BPMN representation. Hence, our framework opens up even further improvements for creating DSLs, besides the already simplified process of DSL design and implementation. Future work will inspect this direction.

As to the trade-off between reuse potential and domain specificity, DSLs4BPM provides common structural elements and commissions derived DSLs to provide individual building block types representing typical domain-specific activities. This separation reduces the development effort while enabling expressive models. Transforming the concise and domain-specific process models to BPMN 2 yields advantages when analyzing or implementing processes. Due to the individual transformation rules that express domain-specific complex activities, the generated BPMN 2 models can exhibit a high level of detail suitable for service orchestration or workflow management. Therefore, using the framework allows for a valuable combination of expressive DSL models and executable BPMN models. The fact that adaptation takes place at predefined and documented extension points simplifies the derivation of DSLs from the framework.

The extension mechanisms provided by EMF and QVTO proved to be suited for creating modular and reusable modeling languages. Thus, using the mature EMF environment also enables more complex scenarios for DSL creation and reuse than simply defining each language from scratch. While specific scenarios might need the more advanced modularization techniques for DSL reuse described in the literature and available in other language development environments, the extensibility approach employed while designing the framework is sufficient on its own for reusing common parts. This is especially important and helpful because most environments for creating DSLs and model transformations offer some kind of extension mechanism.

In contrast to approaches that focus on DSL reuse, DSLs4BPM has an integrated, model-driven perspective and includes transformations to less abstract but widely-used languages as an inherent part. A DSL alone is often not as useful as a combined approach. As soon as a DSL shall not only to be used for documentation, but also for other purposes like analysis or implementation, customized tools are needed. Due to the specific nature of a DSL, these are most often not available and developing them would not be cost-efficient. Through the readily available transformation to BPMN, DSLs implemented within our framework can almost immediately profit from the execution support and other capabilities of BPMN.

9 CONCLUSIONS

The DSL Framework for BPM (DSLs4BPM) presented in this paper enables the efficient creation of domain-specific PMLs by reusing common, generic parts of a compact language. Thus, the framework makes it easier to realize the benefits associated with using an optimally adjusted DSL for process modeling. As a particular focus lies on model-driven development, the framework is complemented with an adaptable transformation to BPMN 2. By adaptation, the transformation can express the semantics of domain-specific elements. Implementing the framework based on EMF has demonstrated the viability of that environment for advanced and modular model-driven approaches.

Future work will deal with improving the tool support for developing derived DSLs, e.g., by providing a wizard, and for modeling with the framework. At the moment, tree-based editors generated by EMF are used during modeling. A more meaningful and extendable concrete syntax is needed. Our approach

[5]The generation of QVTO code would require a BPMN-to-QVTO code generator.

does not depend on any kind of concrete syntax, so that the introduction of a notation will be straightforward, even more so as the framework language lends itself to a graphical notation. The wizard mentioned above could also help in this regard by creating a graphical editor for a DSL based on a generic editor. Furthermore, it might be worthwhile to explore if the general approach taken in this paper—creating a framework from which to derive DSLs—can be transferred to other aspects of system modeling, for example requirements modeling, where DSLs play an important role as well.

REFERENCES

Becker, J., Algermissen, L., Pfeiffer, D., and Räckers, M. (2007). Local, participative process modelling - the PICTURE-approach. In *Proc. of the 1st International Workshop on Management of Business Processes in Government (BPMGOV)*.

Becker, J., Breuker, D., Pfeiffer, D., and Räckers, M. (2009a). Constructing comparable business process models with domain specific languages - an empirical evaluation. In *17th European Conference on Information Systems (ECIS)*, pages 1–13.

Becker, J., Weiss, B., and Winkelmann, A. (2009b). Developing a business process modeling language for the banking sector - a design science approach. In *Proc. of the 15th Americas Conference on Information Systems (AMCIS)*.

Brahe, S. and Østerbye, K. (2006). Business process modeling: Defining domain specific modeling languages by use of UML profiles. In *Model Driven Architecture – Foundations and Applications*, pages 241–255.

Dumas, M. and ter Hofstede, A. (2001). UML activity diagrams as a workflow specification language. In *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 76–90.

Heitkoetter, H. (2011). Transforming PICTURE to BPMN 2.0 as part of the model-driven development of electronic government systems. In *Proc. of the 44th Hawaii International Conference on System Sciences (HICSS 2011)*.

Karow, M., Pfeiffer, D., and Räckers, M. (2008). Empirical-based construction of reference models in public administrations. In *Multikonferenz Wirtschaftsinformatik 2008. Referenzmodellierung*, pages 1613–1624.

Krahn, H., Rumpe, B., and Völkel, S. (2008). MontiCore: Modular development of textual domain specific languages. In *Proc. of TOOLS EUROPE*.

Margaria, T. and Steffen, B. (2004). Lightweight coarse-grained coordination: a scalable system-level approach. *International Journal on Software Tools for Technology Transfer*, 5(2-3):107–123.

Matzner, M., Voigt, M., Alexandrini, F., Araujo, T. S., and Becker, J. (2009). Process modelling in brazilian public administrations: The domain-specific PICTURE approach. In *15th Americas Conference on Information Systems (AMCIS)*.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344.

OASIS Standard (2007). Web Services Business Process Execution Language Version 2.0.

Object Management Group (2008). Meta Object Facility (MOF) 2.0 Query/View/Transformation specification.

Object Management Group (2010). UML 2.3 superstructure specification.

Object Management Group (2011). Business Process Model and Notation 2.0 specification.

Recker, J. (2010). Opportunities and constraints: the current struggle with BPMN. *Business Process Management Journal*, 16(1):181–201.

Recker, J., Indulska, M., Rosemann, M., and Green, P. (2010). The ontological deficiencies of process modeling in practice. *European Journal of Information Systems*, 19(5):501–525.

Recker, J. C., Rosemann, M., Indulska, M., and Green, P. (2009). Business process modeling : a comparative analysis. *Journal of the Association for Information Systems*, 10(4):333–363.

Spinellis, D. (2001). Notable design patterns for domain-specific languages. *The Journal of Systems and Software*, 56(1):91–99.

Steffen, B. and Margaria, T. (1999). METAFrame in practice: Design of intelligent network services. In *Correct System Design, LNCS 1710*, pages 390–415.

Steffen, B., Margaria, T., Nagel, R., Jörges, S., and Kubczak, C. (2007). Model-driven development with the jABC. In *Proc. of the 2nd International Haifa Verification Conference (HVC)*.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Longman.

van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36.

Voelter, M. (2010). Implementing feature variability for models and code with projectional language workbenches. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, pages 41–48.