A Pattern-Based Approach to Business Process Modeling and Implementation in Web Services

Steen Brahe[1] and Behzad Bordbar[2]

[1] Danske Bank and IT University of Copenhagen, Denmark
stbr@danskebank.dk
[2] School of Computer Science, University of Birmingham, UK
b.bordbar@cs.bham.ac.uk

Abstract. There are often three groups of experts involved in the design and implementation of business processes in a service oriented enterprise; *business analysts*, *solution architects* and *system developers*. They collaborate with each other to transform a high-level design created by a business analyst to a final executable workflow, based on a service composition language such as the Business Process Execution Language (BPEL). In this paper, we present a new approach to support and semi-automate this transformation process, thus producing applications of higher quality in shorter time. The idea is to capture existing knowledge in the enterprise, which is required for transforming models from one abstraction level to another, as reusable, parameterized patterns. These patterns are used for tool based model transformations of the business processes. To support our approach, we shall make use of Domain Specific Modeling Languages (DSMLs) designed for each enterprise to capture models of a business process at different levels of abstraction, each suitable for the use of one of the groups of experts. The presented approach bridges the gap between business and IT by providing customizable language-, tool- and transformation support for the different groups of experts within the enterprise and is illustrated by an example.

1 Introduction

Information technology is undergoing a rapid change of role from being a mere provider of support for businesses, to an active role in driving the revenue and profit [1]. There is an ever-increasing pressure on modern enterprises to adapt to the changes in their environment by evolving to respond to any opportunity or threat [2]. To address such challenges, Service Oriented Architecture (SOA) has received considerable attention as it provides the foundation for implementing business processes via composition of (existing) services.

Using SOA and service composition requires a collaborative effort of different groups of experts; *business analysts* model the process at a high conceptual level, *solution architects* map such conceptual designs to architectural models, and *system developers* implement architectural models in a service composition language such as Business Process Execution Language (BPEL) [3]. However, there is a gap between business and IT, due to different terminology, levels of granularity, varied models, approaches, tools and method that each employ [2].

D. Georgakopoulos et al. (Eds.): ICSOC 2006 Ws, LNCS 4652, pp. 166–177, 2007.
© Springer-Verlag Berlin Heidelberg 2007

In this paper we present a new approach to close the gaps between different model representations of a business process by using tool-based transformations from one model to another. The main idea of the approach is to capture knowledge required for the transformations as reusable, parameterized patterns, which can be used to conduct the transformations via software tools. To achieve this, we combine Model Driven Development (MDD) techniques [4] and Domain Specific Modeling Languages (DSMLs) [5,6] fitted specifically for the enterprise. DSMLs are used to capture models of the business process at different abstraction levels for the three groups of experts. This enables creation of precise, machine-readable models, which are also easier to communicate. MDD techniques are used for automatic transformations of models captured in domain specific languages. Hence, the presented approach aims to assist the experts belonging to each of the three groups to create precise models of the business process at their abstraction level and to support automatic propagation of changes in the model created by the analyst to the model created by the architect and further to the model created by the developer

The paper is organized as follows. Section 2 provides a brief introduction on DSML, MDD and service composition. Section 3 presents the outline of our approach. Section 4 illustrates the approach with the help of an example of a mortgage approval process in an imaginary bank. Section 5 evaluates the approach. Section 6 introduces a prototype implementation and section 7 contains the conclusion.

2 Preliminaries

This section describes concepts and notions used in the rest of the paper. It introduces the use of Domain Specific Modeling Languages, Model Driven Development, and service composition as an implementation to support business processes.

2.1 Domain Specific Modeling Language

A general purpose process modeling language such as the Business Process Modeling Notation (BPMN) [7] or UML activity diagrams [8] are not designed to support enterprises in creating models using their own vocabulary and terminology. In contrast, a DSML created specific for an enterprise allows the experts to create models using locally known domain concepts and to provide domain specific information to model precisely. In this paper we shall make use of domain specific modeling languages, which are based on UML activity diagrams and extended for a particular domain by a UML profile [8]. A profile is constructed by using the extensibility elements: stereotypes, tagged values, and constraints [8], which are machine readable modeling construct used by UML tools. For example, in an activity diagram we may wish to specify, if a task is carried out by a software system or a human agent. To do so, a profile containing the stereotypes <<*Automatic*>> and <<*HumanActivity*>> can be applied to the activity diagram. Such stereotypes clarify if a task is carried out by software or by a human being. A stereotype is applied to a task to indicate the task type. Using these stereotypes or specialized task types extends activity diagrams into a new (here, rather simplistic) language.

Through out the paper we use the term task for the single actions or activities that make up a business process. We use the term task type to classify various tasks. For example, *HumanActivity* is a task type, which embodies tasks such as posting a letter or assessing a risk related to a mortgage by a human actor. A domain specific process modeling language consists of a number of task types that can be used for modeling.

2.2 Model Driven Development

In the Model Driven Development (MDD) paradigm, models are treated as primary software artefacts, from which the implementation is created with the help of software tools [4]. Adopting MDD in a software development process is expected to speed up development time and improves the quality of the delivered system.

The Model Driven Architecture initiative (MDA) [9] implements the MDD approach around a set of technologies and standards like MOF, UML and XMI. Central to the MDA is the idea of model transformations. Defining a transformation from one kind of model, the source model, to another kind of model, the target model, one is able to reuse that transformation for all source models of the same type. MDA provides mechanisms to define DSMLs and a conceptual framework for defining transformations between different DSMLs. Models are created by using constructs from meta-models. Meta-models are models, which formally defines the syntax of which models can be created. A meta-model defined for a specific domain can be seen as a Domain Specific Modeling Language. Using MDA technologies, a meta-model is defined either by using MOF, a meta modeling language, also called a meta-meta-model [9] or by using the UML profiling mechanism [8]. A transformation is a set of rules that specify mapping between the source and the target language. Several methods exist for defining model transformations ranging from complex frameworks utilizing languages as ATL and QVT to simple Java based frameworks as SiTra. For simplicity, we describe transformation rules in English.

2.3 Service Composition

Enterprises that adopt a Service Oriented Architecture often require combining services to support their business processes. As a result, service composition languages, such BPEL, are designed to allow combining and coordinate service invocations. BPEL is an XML based-language for describing business processes and business interaction protocols.

Research into the application of MDD techniques to the web service domain has recently received considerable attention. A popular area of research is model transformations from platform independent languages to Web service languages, among others, Class Diagrams to WSDL [10] and Activity diagrams to BPEL [11].

3 A Pattern Based Approach to Model Transformations

This section illustrates the outline of our method for bridging the gaps between Business and IT using DSMLs and MDD techniques as depicted in Fig. 1.

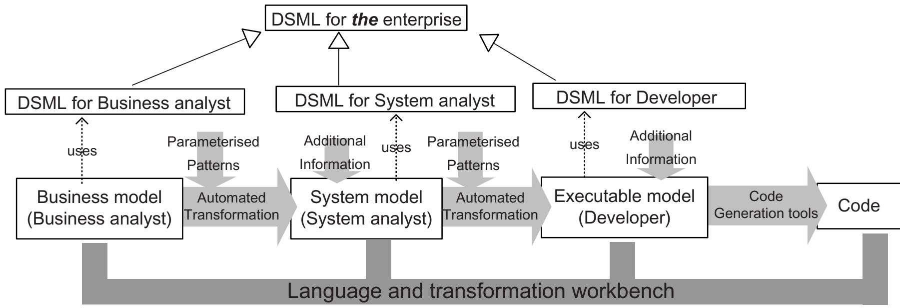

Fig. 1. A pattern based approach for modeling collaboration

The analyst creates a business model of the process. The architect transforms this model to an *architectural model* by applying a predefined and automatic transformation to the business model. The transformation uses *parameterized patterns* to create the architectural model. These patterns represent knowledge previously kept by the architect of how to map business models to architectural models in the enterprise. The patterns are parameterized, hence, the architect is asked to include values of *Additional Parameters* required by the transformation. Additional Parameters are information that are required in the architectural model, and which is not represented in the business model. Following the creation of the architectural model by the architect, a developer transforms it to an *executable model* in a similar fashion. The architect and the developer do not change generated models; instead the information they must provide to the final implementation is given as values of additional parameters during transformation. The transformation workbench incorporates this information into the generated models automatically. We shall now describe the approach and the use of parameterized patterns in further details.

3.1 Parametrized Patterns

Derived from Alexander's work on architectural patterns, and now commonplace in software engineering [12], patterns have been embraced by the workflow and business process community [13,14]. A pattern describes a recurring problem that occurs in a given context, and based on a set of guiding principles, suggests a solution. In our approach, a *pattern* is a common architectural, or implementation solution to reoccurring tasks of same type. For instance, a business analyst may for simplicity model a single task in a process, but describe that it should be executed a number of times. The architectural pattern for the task is an iteration over a service invocation. Each time such a task is modeled by the analyst, the architect creates the same kind of solution. We use patterns to capture and describe such common solutions to tasks of the same type.

The patterns described in this paper are domain, or enterprise, specific, i.e. they are specific to each individual enterprise. They make use of attributes and parameters related to the models. Hence, we shall use the phrase *parameterized patterns* [15] to distinguish such patterns from high level patterns described in [12]. In our approach,

a parameterized pattern includes three pieces of information; a *pattern template*, *additional parameters* and *transformation rules*. Pattern templates capture the overall structure of a task type in the source language represented at a lower level of abstraction and is defined in the target language. Additional parameters specify information required for fitting and customizing the pattern template for a specific task. Transformation rules use values of the additional parameters and attribute values of the task to change and fit the pattern template into the target model.

3.2 Automated Transformation with the Help of Parameterized Patterns

Fig. 2 depicts an outline of our approach for conducting model transformation between different DSMLs using the information captured as design patterns. This results in refinement of a model to a lower level of abstraction as depicted in Fig. 1.

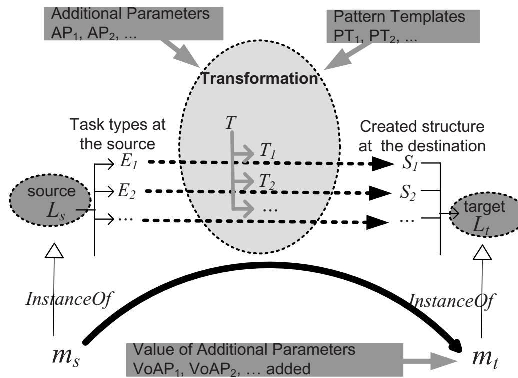

Fig. 2. Model transformation between DSMLs with the help of patterns

Let us consider a source DSML L_s and a target DSML language L_t. For example, in transformation from the Business model to the System model, see Fig. 1, L_s and L_t are DSMLs for business analysts and system analysts, respectively. Suppose that L_s consists of a number of domain specific task types $E_1, E_2, \ldots$ The aim is to transform a source model m_s defined in the language L_s to a target model m_t defined in the language L_t. To achieve this, a transformation T, which contains transformation rules for mapping tasks from L_s to tasks of L_t, is used. The transformation T consists of a number of sub transformations T_j, responsible for the transformation of one task type E_j in the source model to a structure S_j in the target language L_t. The global transformation T orchestrates and coordinates which sub transformations should be executed at the different tasks contained in the source model m_s, collects all generated structures by the sub transformations and connects the generated structures together to the target model m_t.

A sub transformation T_j captures and represents a parameterized pattern, and hence it represents domain specific knowledge of how to represent a task type at a lower level of abstraction in the target language L_t. This makes the sub transformations the

most essential part of the transformation. The sub transformation T_j is defined by the following elements:

1. **Pattern template** PT_j**.** A model template defined in the target language L_t. The model template represents the structure of the source task E_j transformed to L_t.
2. **Additional Parameters** AP_j**.** When transforming a source task E_j to a lower abstraction level (L_t), additional information may be required to enrich and customize the pattern template so the structure S_j defined in the L_t can be generated.
3. **Transformation rules.** Rules that specify how the pattern template PT_j is customized into the structure S_j. The rules make use of Values of Additional Parameter ($VoAP_j$) and values of attributes at the source task E_j.

4 Example: Process Modeling in Estate Bank

In this section we shall illustrate the above approach with the help of an example of an imaginary enterprise called Estate Bank. In contrast to a real business process, which can be quite complex, we use a simplified process as the purpose of the example is to illustrate our approach. Fig. 3 models a mortgage approval process inside Estate Bank. When a customer requests for a mortgage at the bank, a risk analysis (AssessRisk) task is executed. Based on the risk, either the loans for the mortgage is created (CreateLoans) or the request is rejected (Reject).

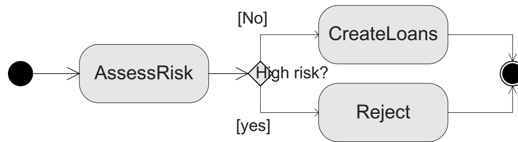

Fig. 3. A mortgage approval process in Estate Bank

A business analyst defines the above model of the mortgage approval process. The team of system architects and, subsequently, the team of developers must create an executable system from such a model. Due to space limitation we shall only define a subset of the modeling languages and transformations. Firstly, we describe subsets of the different languages used by the three groups of experts. Then, we shall define the essential sub-transformations for a selected number of task types from the different languages. Finally, we illustrate the transformation of the CreateLoans task in the mortgage process from the business level to the architect level and further to the development level by using the different sub transformations.

4.1 A DSML for Business Analysts

Consider a domain specific language L_B containing three task types named *HumanActivity* (E_1^B), *Automatic* (E_2^B) and *Bundle* (E_3^B). A task of type *HumanActivity*, as the name suggest, is a task which is handled by a human actor. For example, the AssessRisk

task used in Fig. 3 can be carried out by an employee at the bank, and hence the task is a *HumanActivity*. An *Automatic* task is a task, which is executed by a computer program. For example, the Reject task in the mortgage process is an *Automatic* task type as a computer program in Estate Bank automatically is able to send a rejection letter or an email. A *Bundle* task is one which is executed a number of times. For example, in the mortgage process, creating a number of different loans with different interest rate based on the customer request can be considered a bundle . These task types are high-level enough to be used by the business analyst for creating business process models. For a full-blown realistic example in a real enterprise, several additional types are required. However, the three task types are sufficient to explain our approach.

4.2 A DSML for Solution Architects

The solution architect refines models created by the business analyst. As a result, the DSML, called L_A, used by the solution architect requires more information than the DSML used by the business analyst. Here, we shall exemplify refinement of the task type Bundle from the previous sections. Two of the task types used by the solution architects are *Loop* (E_1^A) and *Service* (E_2^A), which are used in refining the task type *Bundle* from the analyst language. A task of type *Loop* indicates that an iteration should be executed over a sequence of other tasks. The architect may use a *Loop* to indicate that a certain service must be called a number of times, e.g. creation of several loans but with different interest rates. A task of type *Service* indicates calling a specific service available for the use of Estate Bank, for instance creation of a loan with a specific interest rate. Such services are identified by their name and version. The architect determines which service to be executed and specifies the name and version for the service task.

4.3 A DSML for Developers

The developer uses a language similar to BPEL and WSDL. Considering these languages express the system in lower level of abstraction, the DSML, called L_D, for the developer requires more information than the one for the solution architect. The language is not specific to Estate Bank as it is similar to the BPEL language. We present three exemplary task types: *Assign* (E_1^D), *Invoke* (E_2^D) and *Loop* (E_3^D). A task of type *Assign* maps data between variables and is used to initialize input data to service invocations. A task of type *Invoke*, similar to BPEL's *invoke*, is described by a WSDL document. A task of type *Loop* iterates over a sequence and can be compared with a "for" or "while" loop in traditional programming languages. Models created in the DSML for the developers can be compiled directly to BPEL code without any additional parameters required. The models must be defined completely, i.e. the models must be rich enough to be "executable".

Table 2 depicts the task type *Bundle*, of the DSML for the business analyst and its refinement by the architects and developers. Whenever a business analyst models a task as a *Bundle* type (E_3^B), for example the task CreateLoans in the mortgage process Fig. 3, she/he must specify values of the required attributes of the task as listed in Table 1. Firstly, the description attribute clarifies the purpose of the *Bundle*. Secondly, the iterations attribute, if the number is known at modeling time, specifies the number of times the *Bundle* should execute.

Table 1. Task types and their attributes

DSML	Task type	Attributes	Description
Business L_B	Bundle E_5^B	description	A description of what is bundled
		iterations	The number of iterations if it is known
Architect L_A	Loop E_1^A	iterations	The number of iterations
		knownAtBuildTime	Number of iterations is known at build time?
	Service E_2^A	name	The name of the service to invoke
		version	The version of the service to invoke
Developer L_D	Assign E_1^D	data mappings	Mapping of data between variables
	Invoke E_1^D	wsdl	Document describing the service to call

As illustrated in Table 2, the architectural pattern PT_3^{BA} for modeling the equivalent to a Bundle at the architectural level is a loop task type, and inside the loop, a service task type is present. The pattern expresses the common solution to reoccurring model elements of type *Bundle*. The loop task type requires values for two attributes :

-knownAtBuildTime: Boolean. True, if the iteration numbers is known at build time

-number:= the number of times the iteration should run.

Both these attributes can be extracted from the attributes of the *Bundle* task, so no additional information is required here. The *service* task type also requires data for two attributes:

Service name:= The name of the service which the bundle invokes multiple times.

Service version:= The version of the service to be invoked.

Table 2. Sub transformation for *Bundle* task type from business to architectural level

Pattern template PT_3^{BA}	Add. params AP_3^{BA}	Rules
<<Loop>> loop <<Service>> setupLoop	-Service name -Service version	Set name and version at $Service$ attributes

These attributes cannot be extracted from the *Bundle* task type at the business level, as they are information about the architecture of services in Estate Bank, so they must be provided as additional parameters AP_3^{BA} during the transformation. The business analyst has only provided a description of the purpose of the task of type *Bundle*. The architect uses his/her knowledge of Estate Banks services to describe which service and what version to call and specify the attribute values of the *Service* task. A sub transformation T_3^{BA} can be defined for transformation of the *Bundle* task type at the business level to the architectural level. Table 2 shows the pattern template, a textual description of the transformation rules and the required additional transformation parameters.

The *Bundle* sub transformation generates a model structure S_3^A defined in the architect language. This structure contains two tasks, one of type *Loop*, and one of type *Service*. The structure can be transformed to the development level by use of two different sub transformations, one sub transformation T_1^{AD} for the *Loop* task type and one (T_2^{AD}) for the *Service* task type.

Table 3 illustrates that a *Loop* task at the architectural level is transformed to an *Assign* task and a *Loop* task at the development level. The *Service* task at the architectural level is transformed to a sequence of an *Assign* task followed by an *Invoke* task at the development level. The two assign nodes at the development level both need additional parameters for determining how to map data for variables to the loop node and the invoke task respectively. This information can be provided at modeling time, however since the focus of the paper is on the control flow part of the models, we will not deal with this aspect here.

The *Loop* node requires a conditional statement (logic) to determine when is should terminate. This is similar to the conditional statements, for example in “if” and “while” clauses, in conventional programming languages. The *Invoke* node need to know the WSDL document defining the service to invoke. The logic and the document have to be provided for the transformations as values of additional parameters, $VoAP_j$.

Table 3. Sub transformation of *Service* and *Loop* task type from architect to developer level

Task type	Pattern template PT_j^{AD}	Add. params AP_j^{AD}	Rules
Service	<<Assign>> setupData → <<Invoke>> serviceToCall	-WSDL file	Change the invoke node to use WSDL
Loop	<<Assign>> setupLoop → <<Loop>> loop	-logic	Set iteration number at loop

The described parameterized patterns allow the CreateLoans task, if modeled as a *Bundle* type, to be transformed into code with only limited work done by the architect and the developer. They only have to provide specific information during the transformations. The architect has to provide the service name and version of the service that in the IT systems fulfils the requirements specified by the business analyst. The developer has to provide a WSDL document based on the service name and version and logic for when the loop should terminate. Based on these additional transformation parameters, the described sub transformations in Table 2 and Table 3 handle the rest of the work of transforming the business model to an implementation. This is illustrated in Fig. 4.

Similarly, the other tasks, AssessRisk and Reject, of the mortgage process can be transformed by other subtransformations to an implementation. Fig. 5 illustrates the complete mortgage process transformed to the developers DSML where also the AssessRisk and the Reject task has been transformed. The different Assign tasks, map1,

map2, map3 and map4, are used for mapping data for service invocations; for the AssessRisk service which is handled by a human actor, for initializing the while loop for creating the different loans requested by the customer, for the CreateLoan service which create one loan and for the Reject service which sends a rejection to the customer.

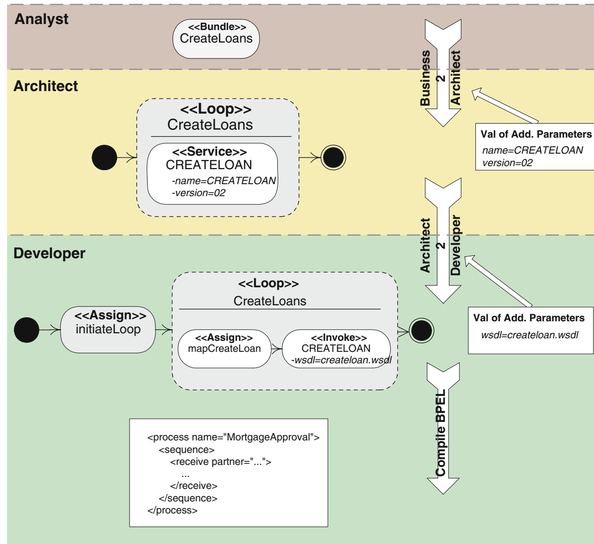

Fig. 4. Transformation of the CreateLoans task from analyst to architect to developer to code

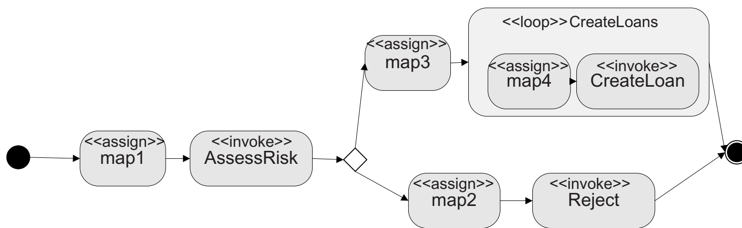

Fig. 5. Mortgage Approval process transformed to developer DSML

5 Discussion

As the example illustrates, the analyst and the architect are able to create precise, machine-readable models in well known domain specific concepts by using languages fitted specially for their needs. By using the suggested approach, i.e. having defined sub transformations for the specific domain concepts, tools can now collect the required information for the concrete tasks in a source models, automatically transform the source model to the target domain and finally generate the implementation code. The model

can be transformed to an implementation, where only required additional transformation parameters have to be provided by the architect and the developer. The developer and the architect are not required to remember or know all details about the patterns and which additional parameters are required. For example, the tool can provide assistance in form of wizards.

Following the gathering of information, the transformation of the task to the lower abstraction level is carried out automatically. As a consequence, the challenge of modeling and implementing business processes, then becomes one of identifying and defining domain specific concepts, DSMLs and transformations between different DSMLs. An outcome and a possible limitation of the approach is that it is not possible to introduce manual corrections to generated models. It is a subject to further research how manual changes applied to generated models can survive repeatable transformations. Due to space limitation, this paper only focuses on control flow part of the business process. Modeling the flow of messages is equally important. For example in the mortgage example it should be modeled which information that is provided to the process and what information the different tasks require. Our approach can be similarly used to model and transform the message flow of a business process.

6 Tool Implementation of the Approach

Our earlier paper [16] describes the tools ADModeler and ADSpecializer, which enable the creation and use of DSMLs based on UML activity diagrams and profiles. We are currently finalizing an extension of the above workbench by a new module called ADTransformer, a transformation engine feasible for transforming models based on different profiles for UML activity diagrams. ADTransformer implements the concepts of sub transformations, parameterized patters, patterns templates, transformation rules and additional parameters. Using the three tools together one is able to define and utilize DSMLs, and define and use transformations between different DSMLs.

7 Conclusion

This paper presents an approach for bridging the gap between business and IT by facilitating better interaction between experts involved in business process modeling and implementation. The main idea is to capture domain knowledge related to different groups of experts as domain specific modeling languages and reusable, parameterized transformation patterns. Using an example, the paper demonstrates that domain specific modeling combined with customizable model transformations can simplify the process of modeling and implementing business processes. Using our tool-based approach will result in shorter time to market from business process idea to implementation, higher quality of the resulting code based on automated transformations, an assurance for what is conceptually modeled is actually also implemented, and better interaction between different groups of experts.

References

1. Wagner, H.-T., Beimborn, D., Franke, J., Weitzel, T.: IT Business Alignment and IT Usage in Operational Processes: A Retail Banking Case. In: HICSS'06. Proceedings of the 39th Annual Hawaii International Conference on System Sciences, vol. 8, pp. 172–194 (2006)
2. Arsanjani, A.: Empowering the business analyst for on demand computing. IBM Systems Journal 44, 67–80 (2005)
3. BEA, IBM, Microsoft, SAP, A., Systems, S.: Business Process Execution Language for Web Services (BPEL4WS). Version 1.1 (2003), http://www-128.ibm.com/developerworks/library/specification/ws-bpel/
4. Stahl, T., Völter, M., Bettin, J., Haase, A., Helsen, S.: Model-Driven Software Development: Technology, Engineering, Management. Wiley, Chichester (2006)
5. Chen, K., Sztipanovits, J., Neema, S.: Toward a semantic anchoring infrastructure for domain-specific modeling languages. In: EMSOFT '05. Proceedings of the 5th ACM international conference on Embedded software, pp. 35–43. ACM Press, New York (2005)
6. van Deursen, A., Klint, P., Visser, J.: Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices 35, 26–36 (2000)
7. White, S.: Business Process Modeling Notation, Version 1.0, final adopted version (2006), Availablable at http://www.bpmn.org/Documents/OMG-02-01.pdf
8. UML2.0: UML 2.0 Superstructure Specification, Final Adopted Specification (2004), available at http://www.omg.org/docs/formal/05-07-04.pdf
9. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture–Practice and Promise. The Addison-Wesley Object Technology Series. Addison-Wesley, Reading (2003)
10. Bezivin, J., Hammoudi, S., Lopes, D., Jouault, F.: An Experiment in Mapping Web Services to Implementation Platforms. Technical report, LINA, University of Nantes (2004)
11. Bordbar, B., Staikopoulos, A.: On Behavioural Model Transformation in Web Services. In: Conceptual Modelling for Advanced Application Domain (eCOMO), Shanghai, China, pp. 667–678 (2004)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1994)
13. Eriksson, H., Penker, M.: Business Modeling with UML. Business Patterns at Work. John Wiley & Sons, Chichester (2000)
14. van der Aalst, W., Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow Patterns. Distributed and Parallel Databases 14, 5–51 (2003)
15. MacDonald, S., Szafron, D., Schaeffer, J., Anvik, J., Bromling, S., Tan, K.: Generative design patterns. In: IEEE International Conference on Automated Software Engineering, pp. 23–34. IEEE Computer Society Press, Los Alamitos (2002)
16. Brahe, S., Østerbye, K.: Business Process Modeling: Defining Domain Specific Modeling Languages by use of UML Profiles. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 241–255. Springer, Heidelberg (2006)