A methodology for domain-specific business process modelling and implementation

Steen Brahe*

Danske Bank,
Holmens Kanal 2-12,
1092 Copenhagen K, Denmark
E-mail: stbr@danskebank.dk
*Corresponding author

Behzad Bordbar

School of Computer Science,
University of Birmingham,
Edgbaston, Birmingham,
B15 2TT, UK
E-mail: B.Bordbar@cs.bham.ac.uk

Abstract: Design and implementation of a business process in an enterprise often requires three groups of experts: business analysts, solution architects and developers. They collaborate to transform a high-level business process to a final executable system based on e.g. BPEL. Since enterprises often utilise their own domain concepts and use technologies in their own specific ways, standard modelling notations and transformations are insufficient. In this paper, we present a methodology to support and semi-automate the transformation of models into an implementation. It advocates the use of a set of domain-specific modelling languages explicitly designed for an enterprise where each language is tailored for the use of one of the three groups of experts. This enables creation of precise and machine-readable models by using domain concepts familiar to the experts. Further, the domain knowledge required for transforming models from one language to another is captured as reusable transformation patterns.

Keywords: model driven development; MDD; domain-specific modelling languages; DSML; transformations; patterns; business process management; service composition; SOA; BPEL.

Reference to this paper should be made as follows: Brahe, S. and Bordbar, B. (2009) 'A methodology for domain-specific business process modelling and implementation', *Int. J. Business Process Integration and Management*, Vol. 4, No. 1, pp.5–17.

Biographical notes: Steen Brahe is an Industrial PhD candidate at Danske Bank and IT University of Copenhagen. He has worked within the software industry for several years, particularly with SOA, MDD and BPM. His primary research topic is model driven development of business processes.

Behzad Bordbar is a Lecturer at the School of Computer Science, Birmingham University, UK. His current research is focused on model driven development techniques and their applications to large distributed systems.

1 Introduction

Information technology is undergoing a rapid change of role from being a mere provider of support for the business, to an active role in driving the revenue and profit (Wagner et al., 2006). Enterprises are beginning to adopt web services to become better equipped to evolve and adapt to changes in their environments (Alonso et al., 2004). In particular, the speed and precision in adapting an IT infrastructure to support a new business idea are crucial factors. To transfer a business idea into an IT system often requires a collaborative effort of a team of experts, many of whom with no or little IT expertise.

A common scenario is the implementation of a business process in web services, which mostly involves three groups of people: business analysts, solution architects and system developers. Firstly, business analysts must describe the business process; a set of logically related tasks performed to achieve a defined business outcome. A task is an atomic activity handled by either an application or a human. Such descriptions which are mostly captured in informal representations (e.g., in English and graphical depictions) must be refined by a solution architect to incorporate information about the IT infrastructure and systems supporting the company. Finally, the development team further refines the design to a format suitable for the creation of executable code such as BPEL (BPEL, 2003).

This paper presents new methodology and framework for business process modelling and implementation that aims at automating the repetitive part of the above process, resulting in shorter implementation time and better quality of the implemented process. As a proof of concept, a prototype tool has been developed to implement the methodology and is described in the paper.

The presented framework relies on domain-specific modelling languages (DSML) for capturing business processes at different abstraction levels. In this context, an enterprise defines and utilises its own modelling languages for modelling its business processes. This builds on Brahe and Østerbye (2006), who describe a method of using DSMLs for specifying business processes using the UML profiling mechanism and activity diagrams (UML2.0, 2004). The argument is that models based on enterprise specific languages are more precise compared to models based on a general process modelling language like BPMN (White, 2006).

A major challenge for the automation is that, since, a model created by a business analyst is at a higher level of abstraction, it lacks information regarding the architecture of the system. Such extra information must be incorporated during the transformation from the language of the business analyst to the language of the solution architect. A similar situation arises when transforming from the language of the architects to the language of the developers. To overcome the above obstacle, the proposed methodology and framework rely on design patterns (Gamma et al., 1994; van der Aalst et al., 2003) to capture knowledge required for transforming a model from one abstraction level to another. In this context, patterns are parameterised. Hence, to conduct our transformations, the value of parameters must be specified. This poses a new challenge, as the conventional model transformation methods in model driven development (MDD) must be extended to handle the parameters. We shall describe a prototype software tool that implements our approach by incorporating the value of parameters and conduct the transformation automatically.

We illustrate our methodology and framework by using an example of mortgage approval process in an imaginary enterprise called 'Estate Bank'. We define three modelling languages each tailored for the use of a group of experts within Estate Bank and illustrate that the process of 'implementing a new business idea', which starts from a design created by the business analysts and terminates in the final code, can be seen as a series of model transformations. Firstly, of models created in the language designed for the business analysts to the language for the solution architects, and subsequently, from the language for the solution architects to the language for the developers.

The paper is structured as follows. Preliminary information is given in Section 2. Section 3 describes the challenges that we address in the paper and introduces our methodology. It is followed by a detailed description of the methodology in Section 4. Section 5 applies the methodology to the mortgage process in Estate Bank. Section 6 describes a prototype implementation, which is developed on the basis of the presented framework. Section 7 contains related work and Section 8 includes concluding remarks and future work.

2 Preliminaries

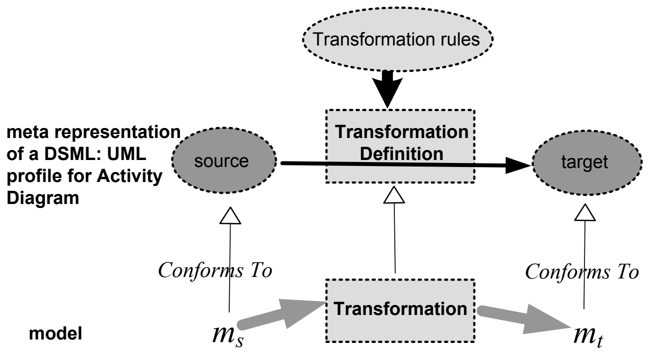
MDD (Stahl et al., 2006; Kleppe et al., 2003) provides an approach to application design and implementation, where systems and applications are specified through models based on meta-model representations. A meta-model for a specific domain is also called a DSML as it can be used as a language for creating models.

The model driven architecture (MDA) initiative (MDA, 2007) is an implementation of the general MDD approach for developing software around a set of standards like Meta-Object Facility (MOF), UML, CWM, etc. UML is set of visual languages for specifying, constructing and documenting software systems (UML2.0, 2004). One of these, the activity diagram, has modelling of organisational processes as one of its purposes. UML is defined by the MOF. MOF is a meta-meta model because it is used for defining other meta-models like UML. When using MDA standards, there are two possible approaches for creating a DSML. The first approach is the definition of a new language based directly on MOF. Such a language becomes an alternative to UML. The second approach is based on specialisation of the existing UML entities using UML profiles. The intention of profiles is to give a straightforward mechanism for adapting UML with constructs that are specific to a particular domain, platform or method. Business process models are sufficiently similar to the fundamental abstractions of activity diagrams. Therefore, we will use UML activity diagrams and profiles to describe DSMLs defined throughout the paper. However, the presented method is independent of the use of UML and profiles.

Central to MDD is the automated model transformation. In this paper, we shall deal with transformations between models compliant to a UML profile of activity diagrams. In such transformations, a source model, m_s must be transformed to a target model, m_t, as depicted in Figure 1. *Source* and *target* represent meta-representation of the models m_s and m_t, respectively. A *transformation definition* includes a set of *transformation rules* that specify mapping between elements in the source and target language. Several tools and technologies exist for defining meta-models and transformation rules. The Eclipse Modelling Framework (EMF), an implementation of a subset of the MOF specification, is widely used for creating tool based meta-models in Eclipse. The Eclipse UML model, which is used by the prototype in this paper is, e.g., implemented using EMF. Transformation rules can be specified in specialised transformation languages as ATL (Jouault and

Kurtev, 2005) and QVT (QVT, 2007) or in, e.g., plain Java. The transformation rules are executed by a transformation engine, which generates the target model from the source model.

Figure 1 MDA model transformation (see online version for colours)


Transformation rules
meta representation of a DSML: UML profile for Activity Diagram
source
Transformation Definition
target
Conforms To
Conforms To
model
m_s
Transformation
m_t

3 Domain-specific customised tools

In this section, we first describe challenges of using standard modelling notations, tools and transformations for an enterprise that requires its own modelling notations and uses implementation technology in specific ways. Then, we introduce our methodology which addresses these challenges by combining DSML and MDD techniques.

3.1 Challenges

Various software development methodologies identify the role of the three groups of experts introduced in Section 1 and provide support for them. For example, using Rational Unified Process (Kroll and Kruchten, 2003), the business analyst models the business process in the inception phase of the project, while, the architect and the developer create their models at the elaboration and construction phases, respectively.

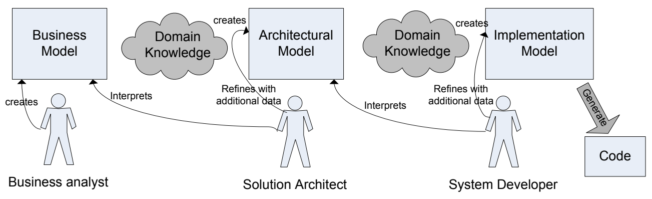
An enterprise that has defined it own development process and domain concepts often has a development process as illustrated in Figure 2. Transformation of models and generation of code is handled manually by humans. Three challenges exist that have to be addressed with the current approach to achieve an efficient development process:

1 *How to define precise models*. Domain-specific concepts cannot be modelled precisely in a general modelling language as, e.g., BPMN (White, 2006). Such languages only contain general modelling concepts. Furthermore, information related to a domain-specific concept in form of attribute values may be required for precise models. For instance, a business analyst may often have to model a risk task and provide attribute values such as the kind of risk calculation to be executed and a responsible department.

2 *How to achieve automated model transformations*. The transformation of a business model or an architectural model requires domain-specific knowledge from an architect or a developer. Such domain knowledge about how to transform a model from one abstraction level to the next must therefore be captured by tools to allow automatic model transformations.

3 *How to refine models without losing the additional information at next transformation*. Additional information has to be provided by the architect and the developer when they create their models. This information must be persisted to be used by future model transformations.

Figure 2 An enterprise-specific development process without customised tool support (see online version for colours)



Now, let us look into how our methodology may address these challenges.

3.2 Combining DSML and model transformations

Most recently the use of DSML has received considerable attention. The underlying idea is to capture knowledge and expertise within an enterprise in precise languages for the different groups of experts. The use of a DSML enables

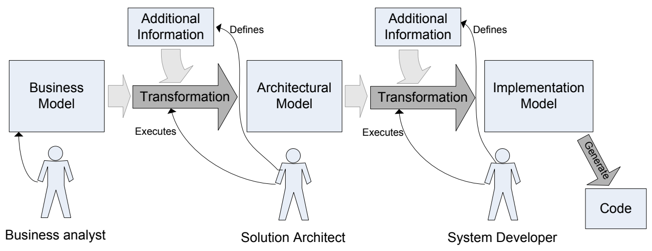
- creation of precise, machine-readable models applicable for automated model transformations
- customised tool support to enhance usability and performance by, e.g., allowing specific wizards for data collection and validation rules
- easier creation of models as the experts can use domain concepts and graphical representations familiar to them. Such models will also be easier to understand.

Hence, the use of DSMLs addresses the first challenge described above.

Our research builds on the idea of combining DSML and MDD transformation techniques and provides a methodology for developing a chain of languages and tools to support the collaborative effort of the three groups of experts with the goal to generate an implementation of the business process. In this context, MDD is now widely accepted; methodologies such as business-driven development (Mitra, 2005) advocate the use of model transformations to develop IT solutions that directly satisfy modelled business requirements.

An outline of our methodology is depicted in Figure 3. A business analyst produces a model of the business process captured in business analyst language. An automatic transformation that incorporates domain knowledge previously held by the solution architect is able to transform the model into an architectural model defined in the solution architect language. The transformation also uses another model which contains additional information required for completing the architectural model. This information was previously entered directly into the architectural model when the solution architect refined it. By merging these two models into the architectural model, the solution architect can be sure that the refinement information is not lost when executing the transformation next time. Similarly, a transformation is defined to generate the implementation model based on the architectural model and additional information required for the implementation. Code can be generated directly from the implementation model.

Figure 3 Using several DSMLs in a MDD process (see online version for colours)



While business process models capture the behaviour of the business, information models capture static information related to it. The business analyst needs, e.g., to know about the organisation structure, the architect needs information about process metrics and the developer needs details about transaction scopes. The additional information models in Figure 3 refer to models of such information. In this paper, we restrict the use of information models to capture domain knowledge required for implementing a business process.

To underpin the methodology Brahe and Østerbye (2006) presents an approach and introduces two Eclipse based tools, ADModeler and ADSpecializer, for creating such languages and additional editors. Hence, for the method depicted in Figure 3 we are able to create DSMLs and supporting tools for each of the three groups of experts. The primary focus of this paper is to deal with the last two challenges presented above by defining a transformation framework that allows definition of customised model transformations and required additional information.

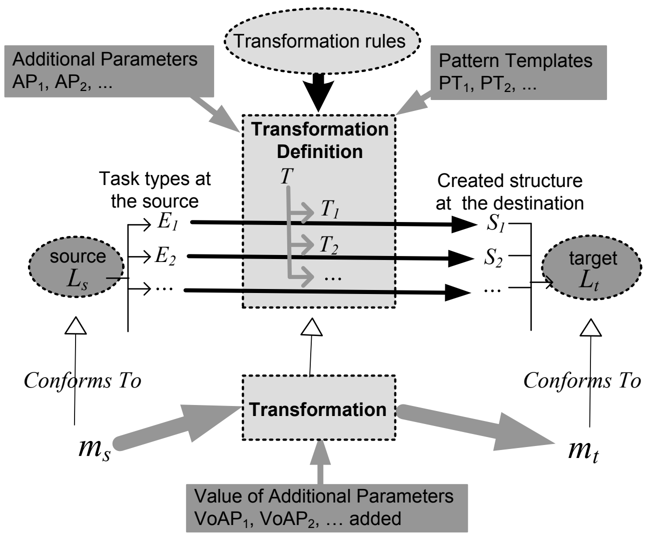
4 A methodology for customising languages and transformation

In this section, we present a framework for addressing the last two challenges previously described. The framework enables capturing knowledge required for the transformation of a model from a high-level to a lower-level of abstraction by introducing *reusable patterns* into the transformation mechanism. Derived from the work by Alexander (1964) on architectural patterns and now commonplace in software engineering (Gamma et al., 1994), they have been embraced by the workflow and business process community (Eriksson and Penker, 2000; van der Aalst et al., 2003). A *pattern* describes a recurring problem that occurs in a given context and based on a set of guiding principles, suggests a solution. However, using conventional patterns would not be sufficient to address the third challenge described above; we shall introduce the notion of *parameters* into the patterns and their use in the model transformation. Hence, we shall use the phrase *parameterised patterns* (MacDonald et al., 2002) to distinguish such patterns from high level patterns described by Gamma et al. (1994). We use ‘patterns’ to describe recurring solutions within one enterprise. This is in contrast to the well-known and general patterns described by Gamma et al. (1994) and van der Aalst et al. (2003). We shall include three pieces of information in each parameterised pattern: a *pattern template*, some *additional parameters* and *transformation rules*. A pattern template 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. In the context of this paper, a *structure* is defined as a number of tasks connected within a control flow. Additional parameters specify information required for fitting and customising 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.

The use of parameterised patterns will capture the knowledge of how to transform a domain-specific task type from one abstraction level to another and hence, addresses the two challenges. Having defined a parameterised pattern for a specific task type, tools can now collect the required information. This information has to be provided by the developer or architect, but they are not required to remember or know details about the patterns and which additional parameters are required, as the tool can prompt the user to include such information. A model transformation framework can be used to execute the transformation of a model to the lower abstraction level. Hence, the repetitive manual part of the development process is eliminated, resulting in faster development cycle and better quality models and implementation. Consequently, the challenge of modelling and implementing business processes then becomes one of identifying and defining domain-specific task types, DSMLs and transformations between different DSMLs.

Now, we shall introduce a transformation framework based on two DSMLs and knowledge captured by patterns, rules and additional transformation data. The framework is applicable for control flow based DSMLs like, e.g., UML activity diagrams extended by UML profiles. Figure 4 depicts an outline of our approach for conducting model transformation between different DSMLs, which results in refinement of a model to a lower level of abstraction.

Figure 4 Methodology with a pattern based model transformation between two DSMLs (see online version for colours)



Let us consider a source DSML L_s and a target DSML language L_t. Suppose that L_s consists of a number of domain-specific task types E_1, E_2, ….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 general transformation definition T, is used. It recursively traverses the source model to produce the target model. It does not contain any transformation rules. The transformation definition T uses a number of subtransformations T_j that contain the transformation rules. A subtransformation is 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. When transforming a source model m_s the global transformation T orchestrates and coordinates which subtransformations should be executed at the different tasks contained in m_s, collects all generated structures by the subtransformations and connects the generated structures together to the target model m_t. During the transformation, values of additional parameters, $VoAP_j$, must be provided. These values are instances of the metadata descriptions of required additional parameters AP_J that are required for the different subtransformations.

A subtransformation T_j captures and represents a parameterised 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 Tj 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 customise the pattern template so, the structure S_j defined in L_t can be generated.

3 *Transformation rules*. Rules that specify how the pattern template PT_j is customised into the structure S_j. The rules make use of values of additional parameters $VoAP_j$ and values of attributes at the source task E_j.

Next, we shall describe the methodology with the help of an example of a mortgage approval process in the imaginary 'Estate Bank'.

5 Example of a mortgage process

Figure 5 depicts various tasks in a mortgage process as it is handled in Estate Bank via a UML activity diagram. The first task is to collect the applicant's personal details and information (*GetCustomerInfo*). Then, a number of parallel tasks occur:

- the applicant's details and her/his financial situation are verified with the help of a credit reference agency (*CreditReferenceAgency*)
- details of the property, including the property ownership status, are verified (*CheckHouse*)
- if the applicant is a current customer of the bank an internal credit check is made (*CheckCredit*).

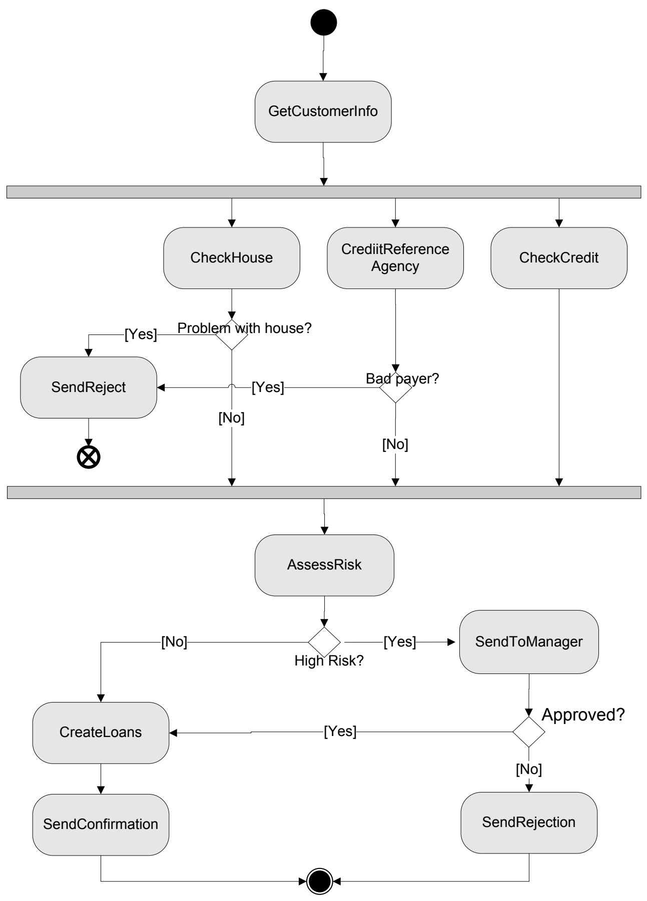
In case of a major problem with the house or the applicant's financial/credit status, the application is rejected and a rejection letter is sent to the customer (*SendReject*). A risk analysis is conducted (*AccessRisk*) after the three tasks have been completed. If the risk is high, the decision about approval or rejection is handed to a manager (*SendToManager*). If the risk is low the requested loans are created (*CreateLoans*) and a confirmation letter is sent to the customer (*SendConfirmation*).

To illustrate the different DSMLs and how to map from one model to the next, we shall apply our methodology to the *CreateLoans* task from Figure 5:

CreateLoans task

When a customer applies for a mortgage, it is possible for him/her to apply for more than one loan for the same property. For example, the mortgage can be divided into several smaller loans, each with a specific interest rate and financial terms. As a result, when the business analyst defines a CreateLoan task, the description may be along the following lines: '…several loans should be created depending on the information provided on the mortgage application by the customer…'

Figure 5 Activity diagram of the mortgage approval process (see online version for colours)



The solution architect uses this description to infer that the services/applications responsible for the creating loan should be executed iteratively, until all loans requested on the application form are created. In addition, the solution architect must obtain the name and version of the service responsible for creating a loan and add this information to the architectural model.

The system developer interprets the architectural model for creating an implementation model based on BPEL. The developer knows that an iteration of multiple tasks at the architectural level is implemented as an *assign* node followed by a *loop* node. The *service* task described in the architectural model is transformed to a suitable assign node followed by an *invocation* node. The developer has to find or create a WSDL file based on the service name and version, which are specified by the architect and annotate the invocation node suitably.

We start by defining subsets of three languages, one for the business analyst, one for the architect and one for the developer. The languages are meant to be specific to estate bank, i.e., they are not designed for the use in other enterprises. Then, we define required domain-specific patterns for Estate Bank at the architectural and the development level, to allow transformation of the mortgage process first to the architectural level and then to the implementation.

Transformations rules can be specified using an exiting formalism like QVT, ATL or Java (QVT, 2007; Jouault and Kurtev, 2005; Akehurst et al., 2006). However, producing such specifications is not a focus of this paper. Hence, we describe the transformations in plain text.

We describe in detail how one task, *CreateLoans*, from the mortgage process can be generalised as a domain-specific task type, which we refer to as a Bundle at the business level. Using subtransformations, we demonstrate transformation of the *CreateLoans* task, first to the architectural level and then, to the development level with only limited interference from the architect and the developer. Using the *bundle* task type to model the *CreateLoans* task in the business model helps the business analyst to create a precise model and the architect and developer does not have to manually transform the task to their abstraction level.

5.1 A DSML for business analysts

In this section, we shall present a DSML for the business analysts affiliated to Estate Bank. Table 1 depicts a set of five task types and their corresponding tasks from the mortgage example at Figure 5. To define the task types E_j, we have used illustrative names as follows.

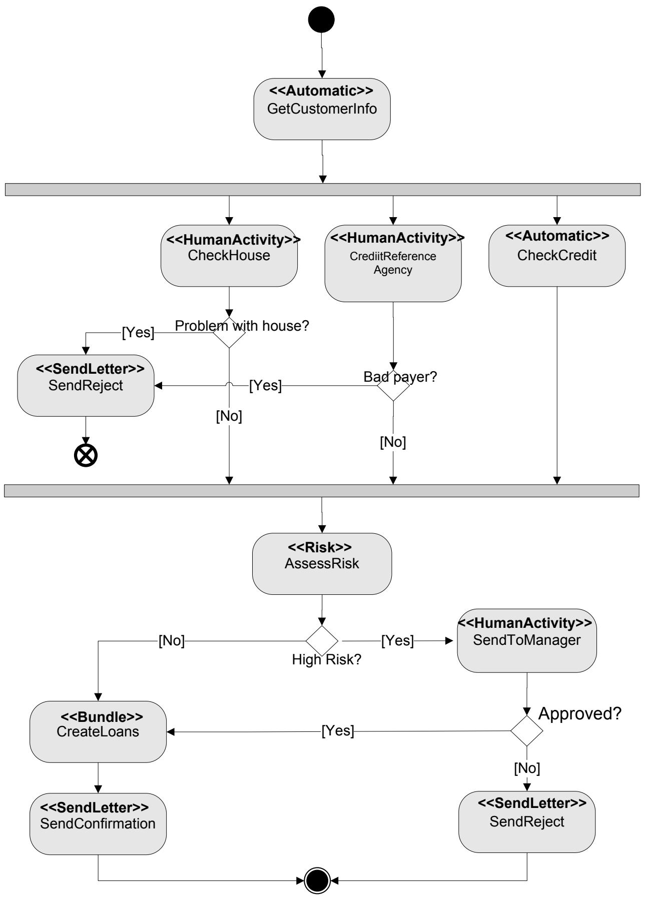
Table 1 Business analyst task types and application at the mortgage process tasks

Task type E_j	*Task in mortgage process*
Automatic$_B$	CheckCredit, GetCustomerInfo
HumanActivity$_B$	SendToManager, CheckHouse, CreditRefAgency
Risk$_B$	AssessRisk
SendLetter$_B$	SendReject, SendConfirmation
Bundle$_B$	CreateLoans

An *Automatic$_B$* task type is any task which can be executed by Estate Bank's IT system, whereas, a *HumanActivity$_B$* task type is any manual activity handled by an employee. A task of type *Risk$_B$* estimates the risk involved in giving a loan of a certain amount, on a property at a specific location to a specific customer. The *Bundle$_B$* task type is used for executing an activity several times. In the mortgage process, this type can be used to model the *CreateLoans* task to create several loans. We call the business analyst language L_B, where index B stands for business. Using this language and specific task types, the business analyst can model the mortgage process as illustrated in Figure 6. Here, stereotypes are used visually to indicate different task types.

The task types are exclusive to Estate Bank. They can be used for modelling many different business process scenarios and provide business analysts in the Bank with a common domain-specific vocabulary to be used in modelling business processes. The use of domain-specific types helps the analyst to reuse common types and ensure that required information for that type is defined. This means precise machine-readable models, which are easier to understand and which are applicable for semi-automated model transformations.

Figure 6 The mortgage process modelled in the business analyst language (see online version for colours)



5.2 A DSML for solution architects

The solution architect refines a model created by a business analyst. As a result, the DSML used by the solution architect requires more details than the DSML used by the business analyst. In this section, we shall explain only three of the task types used by the solution architects; *Loop*$_A$, *Service*$_A$ and *Receive*$_A$, which are used in refining the task type *Bundle*$_B$, explained in previous section. We call the architect language L_A, where A stands for architect.

The *Loop*$_A$ task type indicates that an iteration should be executed over a sequence of tasks. The architect may, for example, use a loop task to indicate that a certain service must be called a few times. The *Service*$_A$ task type indicates calling a specific service available for the use of Estate Bank. 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. For instance, the service could be responsible for calculating a risk profile for a customer or creating a specific loan. The *Receive*$_A$ type indicates that the process is waiting for external events to occur. For example, the *Receive*$_A$ type can be used in a process to indicate waiting for a customer to accept conditions send by e-mail. When the customer accepts the condition, for example, by logging into a website and confirming, the process can continue.

5.3 DSML for developers

The developer uses a language similar to BPEL. Hence, the DSML for the developer requires more details 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 four exemplary task types: *Assign*$_D$, *Invoke*$_D$, *Loop*$_D$ and *Receive*$_D$. We call the developer language L_D, where index D stands for developer.

An *Assign*$_D$ task type maps data between variables and is used to initialise input data to service invocations. An *Invoke*$_D$ task type is similar to BPEL's invoke and is described by a WSDL document. A *Loop*$_D$ task type iterates over a sequence and can be compared with a 'for' or a 'while' loop in traditional programming languages. A *Receive*$_D$ task type waits to be called from outside the process and is defined as a web service. Using a *Receive*$_D$ task type makes it possible for others services and processes to call the process from the outside. Models created in this DSML 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 Task types and their attributes

DSML	*Task types*	*Attributes*	*Description*
Business L_B	Bundle$_B$	Description	A description of what is bundled
		Iteration	The number of iterations, if it is known
Architect L_A	Loop$_A$	Iterations	The number of iterations
		KnownAtBuildTime	Number of iterations is known in build time?
	Service$_A$	Name	The name of the service to invoke
		Version	The version of the service to invoke
Developer L_D	Assign$_D$	Data mappings	Mapping of data between variables
	Invoke$_D$	wsdl	Document describing the service to call

5.4 Attributes at task types

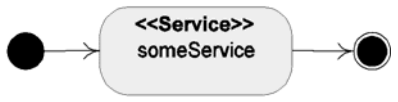
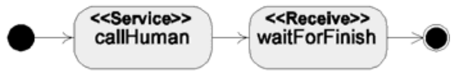
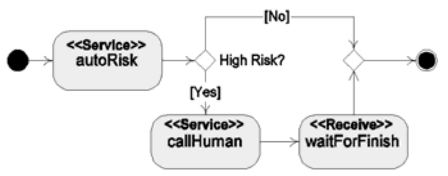
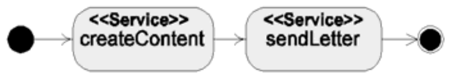
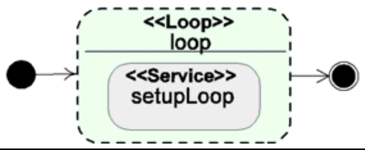
Now, we have defined fractions of three languages at three different abstraction levels where each language consists of a number of task types. Each task type is identified by its name and contains a number of attributes. When a modeller is creating a task of a certain type, he/she must specify values of the required attributes defined for the task type. Attributes for the defined task types in our languages can be found in Table 2. Only task types relevant to the CreateLoans task example have been illustrated.

As mentioned in Section 5.1, the CreateLoans task, which creates multiple loans, is of type *Bundle*$_B$. In the next section, the above task types are used to model pattern templates for, firstly transforming a *Bundle* task type to the architectural level and then, to transform the result to the development level.

5.5 Sample of patterns in Estate Bank

Each task type E_j^B in the business DSML has a pattern representation PT_j^{BA} in the architectural DSML. Equally has each task type E_j^A in architectural DSML a pattern representation PT_j^{AD} in the developer DSML. The super indices ^{BA} and ^{AD} indicates transformation from L_B to L_A and from L_A to L_D. The architectural and development pattern templates are illustrated in Table 3 and Table 4.

Table 3 Architectural pattern templates for business analyst task types (see online version for colours)

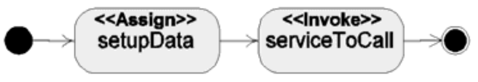
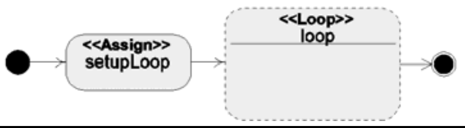
Analyst type	Architectural pattern template PT_j^{BA}
Automatic$_B$	
HumanActivity$_B$	
Risk$_B$	 <<Service>> autoRisk High Risk? [No] [Yes] <<Service>> callHuman <<Receive>> waitForFinish
SendLetter	
Bundle$_B$	

The architectural pattern for the *Automatic*$_B$ type is a *Service*$_A$ task which refers to service in Estate Banks IT systems. For the *HumanActivity*$_B$ type, the pattern consists of two tasks; a *Service*$_A$ task, responsible for calling a human activity system and a *Receive*$_A$ task waiting for the human activity system to signal back, that the activity has been completed. The pattern for a *Risk*$_B$ type contains several nodes; first, an automatic risk calculation is made. If the result from this task indicates a high risk, the human activity system is called to let a person manually evaluate the risk calculation. If the risk is low, no further action is required. For the *SendLetter*$_B$ type, the pattern consists of two *Service*$_A$ tasks; First, a content service is called to create the content and layout of the letter. Second, a send letter service is called to create the physical letter and send it by mail. The *Bundle*$_B$ type and the types and patterns found in Table 4 are described in details in the following paragraphs.

All patterns are modelled using UML activity diagrams and profiles.

In the next section we describe how the *Bundle*$_B$ task type first can be transformed to the architectural language and second to the development language by use of subtransformations containing pattern templates, transformation rules and additional transformation data.

Table 4 Developmental pattern templates of architectural task types (see online version for colours)

Architect type	*Pattern template* PT_j^{BA}
Service$_A$	
Loop$_A$	

5.6 Bundle task type and pattern

Whenever a business analyst models a task as a *Bundle*$_B$ type, for example CreateLoans, he must specify values of the required attributes of the task as listed in Table 2. Firstly, the *description* attribute clarifies the purpose of the Bundle. Secondly, the *iterations* attribute, if the number of iterations is known at modelling time, specifies the number of times the Bundle should execute. The architectural pattern PT_{Bundle}^{BA} for modelling the equivalent to a *Bundle*$_B$ is a *Loop*$_A$ task type, and inside the loop, a *Service*$_A$ task type is present. The *Loop*$_A$ task type requires values for two attributes to be completely defined:

1 *knownAtBuildTime*: Boolean if the iteration numbers is known at build time

2 *iterations*: the number of times the iteration should run.

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

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

2 *Service version*: The version of the service to be invoked.

Table 5 Subtransformation for *Bundle*$_B$ task type from business to architectural level (see online version for colours)

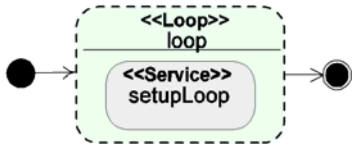
Pattern template PT^{BA}_{Bundle}	*Add. parameters* AP^{BA}_{Bundle}	*Rules*
	• Service name • Service version	Set name and version at << Service >> attribute

Table 6 Subtransformation of *Service*$_A$ and *Loop*$_A$ task type from architect to developer level

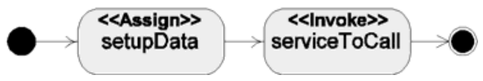
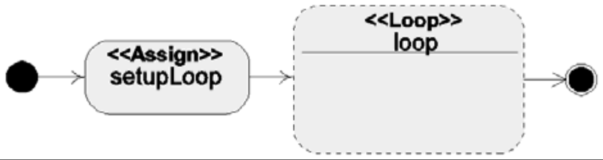
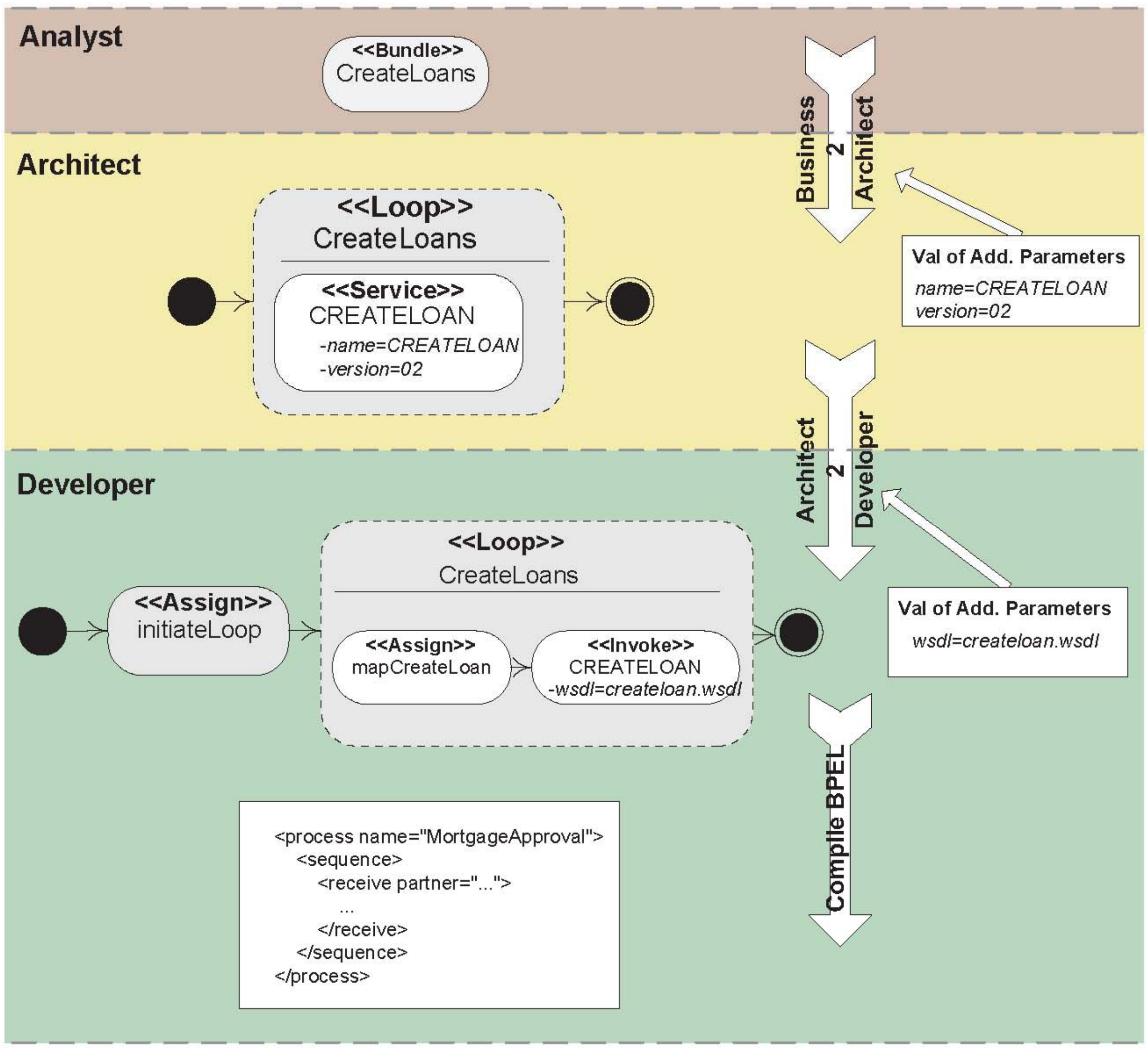
Task type	*Pattern template*	*Add. params.*	*Rules*
Service$_A$		WSDL file	Change the invoke node to use WSDL
Loop$_A$		logic	Set iteration number at loop

Figure 7 CreateLoan task transformed from business level, to architect level and to development level (see online version for colours)



```
<process name="MortgageApproval">
   <sequence>
      <receive partner="...">
         ...
      </receive>
   </sequence>
</process>
```

These attributes cannot be extracted from the *Bundle*$_B$ task type at the business level, so they must be provided as additional parameters AP^{BA}_{Bundle} during the transformation of a task of the *Bundle*$_B$ type. The business analyst has only provided a description of the purpose of the task of type *Bundle*$_B$. The architect must based on this description localise which service and what version to call and specify the attribute values of the service task. A subtransformation T^{BA}_{Bundle} can be defined for transformation of the *Bundle*$_B$ task type at the business level to the architectural level. Table 5 shows the pattern template, a textual description of the transformation rules and the required additional transformation parameters. The Bundle subtransformation generates a model structure S^{A}_{Bundle} defined in the architect language. This structure contains two tasks, one of type *Loop*$_A$ and one of type *Service*$_A$. The structure can be transformed to the development level by use of two different subtransformations, one sub transformation T^{AD}_{Loop} for the *Loop*$_A$ task type and one $T^{AD}_{Service}$ for the *Service*$_A$ task type.

As illustrated in Table 6, 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 modelling 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 needs logic to determine when is should terminate and 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*$_{Bundle}$.

Figure 7 illustrates how the CreateLoans task from the mortgage approval process, if modelled as a *BundleB* type, can be transformed into code with only limited work done by the architect and the developer. 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 data, the described subtransformations in Table 5 and Table 6 handle the rest of the work of transforming the business model to an implementation. Using the illustrated languages and subtransformations, the mortgage process can be transformed to an architectural and an executable model.

5.7 Discussion

By applying our proposed transformation framework at the CreateLoans task from the mortgage example, we have shown that knowledge of how to transform models between different abstraction levels can be formalised by definition of domain-specific task types, pattern templates, additional transformation parameters and transformation rules. By formalising this information, it becomes easier for the business analyst to create precise models, the architect and the developer does not have to remember dozens of different patterns and they do not need to remember what information to provide. Hence, the proposed methodology and framework has addressed the challenges previously described.

In the next chapter we briefly present a tool, that implements the presented transformation framework and that successfully has been applied at the mortgage approval example.

6 Prototype tool implementation

Brahe and Østerbye (2006) use UML activity diagrams as the semantic base for business process modelling and are using profiles to create DSMLs for different purposes inside an enterprise. They also present two Eclipse based tools, ADSpecializer and ADModeler, which are able to generate new DSMLs and customised tool support based on UML activity diagrams and the extension mechanism of Eclipse.

To implement the proposed methodology, we are developing a tool called ADTransformer. It is able to transform a model from a source language to a target language, that are both based on UML activity diagrams and profiles. ADTransformer therefore, suits well with ADModeler and ADSpecializer. The three tools together form a prototype of a complete language and transformation workbench, which supports creation of languages and modeling tools, creation of models using these languages and tools, and transformation of models between languages.

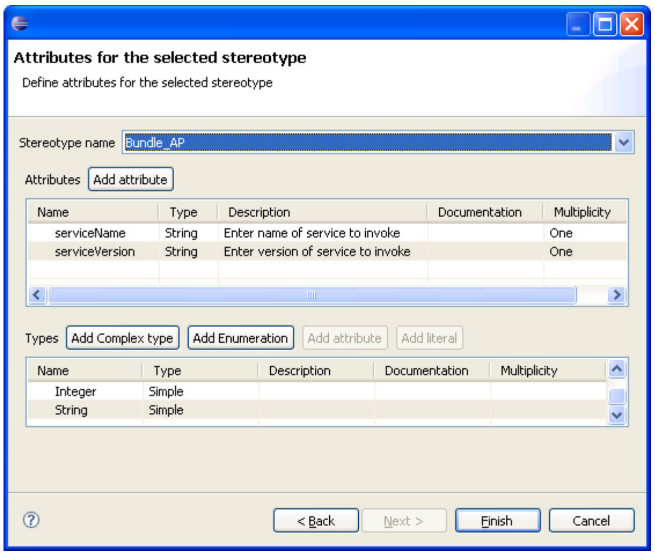
6.1 Implementation

ADTransformer has been implemented as an Eclipse based tool and consists of two parts. The first part is used by a tool developer to create a transformation definition. It uses the concepts of subtransformations and parameterised patterns and provides an extension point, which allows the definition of a transformation between a source and a target language. The transformation is specified by defining a subtransformation for each of the task types in the source language. The second part is used by a solution architect or a system developer to execute the transformation definition at a concrete model. It contains an execution engine that implements the global transformation *T* as a generic transformation. It recursively iterates through the control flow graph of the source model. For each task in the source model, the generic transformation executes the corresponding subtransformation that has been defined by a tool developer. The subtransformation generates a structure of a process model in the target language, which the generic transformation collects and puts into the target model.

6.2 *Defining a transformation*

ADTransformer has a wizard that allows a tool developer to generate an Eclipse plug-in containing empty definitions of all subtransformations between two languages. The wizard has three pages; The first page is used to select the source and target language. The second wizard page is used for information purposes only. It shows the names and locations of empty pattern templates and transformation rules that will be generated for each of the task types in the source language. The third wizard page is used to specify additional parameters needed by the transformation for transforming a task of a certain type from the source language to the target language (Figure 8). After finishing the wizard, the tool developer models the pattern templates in the target language (Figure 9) and defines transformation rules for the subtransformations in the generated Java classes. The generated Eclipse plug-in can now be distributed to architects and developers.

Figure 8 Wizard page for defining additional parameters required by the transformation (see online version for colours)



Attributes for the selected stereotype

Define attributes for the selected stereotype

Stereotype name: Bundle_AP

Attributes

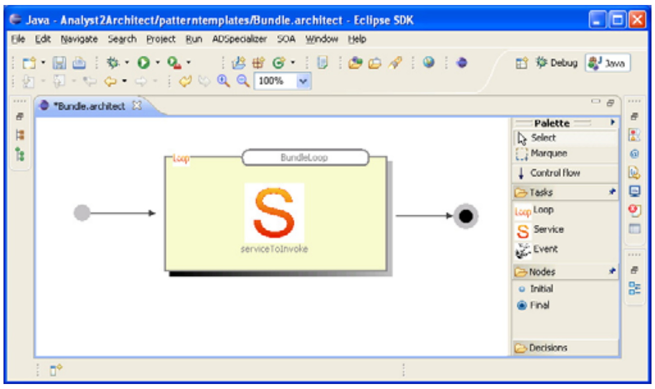
Name	Type	Description	Documentation	Multiplicity
serviceName	String	Enter name of service to invoke		One
serviceVersion	String	Enter version of service to invoke		One

Types

Name	Type	Description	Documentation	Multiplicity
Integer	Simple			
String	Simple			

Note: The transformation of a Bundle task type requires a service name and a service version

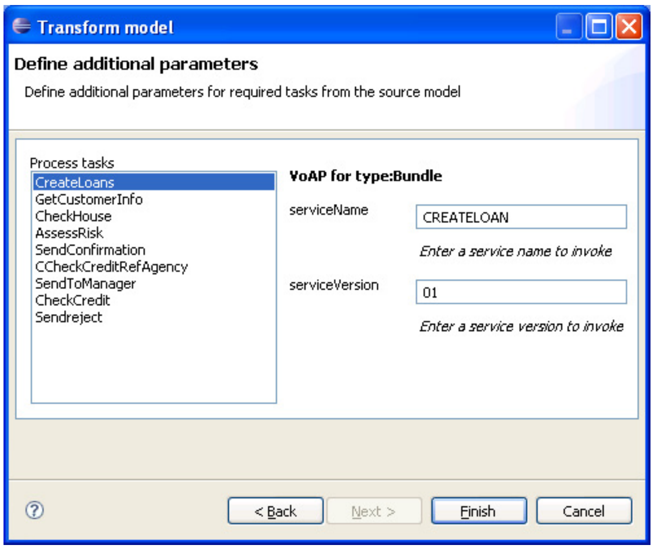
Figure 9 Architectural pattern template for the Bundle task type modelled in the architectural language in ADModeler (see online version for colours)


Java - Analyst2Architect/patterntemplates/Bundle.architect - Eclipse SDK
File Edit Navigate Search Project Run ADSpecializer SOA Window Help
Debug
Java
100%
*Bundle.architect
Loop
BundleLoop
serviceToInvoke
Palette
Select
Marquee
Control flow
Tasks
Loop
Service
Event
Nodes
Initial
Final
Decisions

6.3 *Executing a transformation*

Solution architects and system developers use ADTransformer to execute the transformation definitions. For instance, a solution architect right-clicks at the business model of the mortgage process and selects 'transform model'. A dialogue (Figure 10) is presented which requires definition of values of the additional parameters specified in the subtransformation.

Figure 10 Values of additional parameters are defined during the model transformation (see online version for colours)



Transform model

Define additional parameters

Define additional parameters for required tasks from the source model

Process tasks
- CreateLoans
- GetCustomerInfo
- CheckHouse
- AssessRisk
- SendConfirmation
- CCheckCreditRefAgency
- SendToManager
- CheckCredit
- Sendreject

VoAP for type:Bundle

serviceName: CREATELOAN
Enter a service name to invoke

serviceVersion: 01
Enter a service version to invoke

Note: An architect has specified the service name and version for a Bundle task type as defined in Figure 8

The architect or developer presses the Finish button after entering values of additional parameters. ADTransformer then executes the transformation definition through the subtransformations and generates the architect or the developer model.

6.4 *Generating code*

The generated developer model contains all necessary information to be executable. Hence, BPEL code can be generated directly from it. We have therefore developed a utility tool, which generates BPEL code from the developers model. For this purpose we used SiTra (Akehurst et al., 2006) which is a simple, Java based transformation framework.

7 Related works

Recently, the introduction of SOA and its popularity within the industry (Erl, 2005) has caused much attention in discovery of methods for modelling, analysis, specification and implementation of business processes. Our research aims to support such methods by providing a flexible framework feasible for tool implementation.

MDD and the MDA (MDA, 2007; Stahl et al., 2006) have been used extensively in transforming business process models to implementations, particularly from UML activity diagrams to service composition languages. Bézivin et al. (2004) uses the ATL transformation languages to transform UML models into three different target platforms; Java, web services and Java web service developer pack. Bordbar and Staikopoulos (2004) studies transformation of activity diagrams to BPEL and based on MOF compliant meta models. Skogan et al. (2004) proposes a method that uses activity diagrams to design web service compositions and transform them into different service composition languages. The method also builds on transforming WSDL descriptions into UML, which can then be used to build the service compositions. Koehler et al. (2003, 2005) has worked on model driven generation of BPEL implementations based on activity diagrams using techniques originating from compiler theory and declarations of rules in the Object Constraint Language. The BMNM specification contains a chapter that specifies how BPMN models can be mapped to BPEL. Using BPMN ensures using a language which is specifically designed for business process modelling. Moreover, the transformation to BPEL allows implementation of the business process.

Pokraev et al. (2007) focus on the use of MDD for application integration. Starting from high-level models of existing applications, the business expert models the interaction in a constraint-oriented style with the help of State machines. Then applying MDD, the proposed interaction is mapped and implemented to realise interaction of the services. Dirgahayu et al. (2008) extend this approach by allowing definition of multiple level of abstraction and allowing the business analysis to check if the applications can be integrated prior to the integration.

Dirgahayu et al. (2007) make use of pattern to map business process models into their implementations. In the context of this paper, a pattern is a representation of structure depicting the relationship between the activities. The paper decouples the process of transformation into pattern recognition and pattern realisation. By creating an intermediate model between the two task of recognising and realisation, the presented approach ensure reusing patterns and their realisations in different model transformations resulting in lower development cost, shorter time-to-market and better quality of implementation with fewer bugs.

Shishkov et al. (2007) present an application design process for refactoring of business models and mapping them to platform specific models in order to create loosely coupled service oriented applications.

8 Conclusions and future work

This paper has described a methodology for a smoother implementation of business processes. The methodology provides a framework for bridging the gaps between models at different abstraction levels of a business process. The main idea of the methodology is to capture domain or enterprise, specific knowledge of the coherence between one abstraction level and another as parameterised patterns consisting of pattern templates, additional transformation parameters and transformation rules.

We have used an example of a mortgage approval process to illustrate the challenges of creating models at different abstraction levels and to describe our approach which simplifies and semi-automates the transformation from the business level to the architectural level and from the architectural level to the development level. The presented framework can be implemented as a software tool to allow

1 shorter software development cycle
2 better synchronisation of models at different levels of abstraction
3 higher quality of code and design through reuse
4 improved development process.

A pre-requisite for applying the methodology is the use of different modelling languages for different abstraction levels. A business analyst, an architect and a developer for a specific enterprise each needs his or her own language to create models with required precision and information details.

The methodology and prototype tool has not yet been evaluated and validated. As future work, we therefore plan to make an empirical evaluation through case studies. The methodology and prototype tool should be applied in three different domains. A number of languages and tools at different abstraction levels are build for each domain. Successively, the languages and tools are used to model and implement two different kinds of business processes in each of the domain. The empirical evaluation should also include a number of analysts, architects and developers that use the languages and tools.

We will further explore transformation rules in the global transformation and in the subtransformations and we will examine approaches to model refactoring which will allow survival of manually introduced changes in generated models. The concept of additional parameters will be extended to include separate information models. This will result in a reduction of the complexity of modelling and the possibility to represent the business from multiple viewpoints.

References

Akehurst, D., Bordbar, B., Evans, M., Howells, W. and McDonald-Maier, K. (2006) 'SiTra: simple transformations in Java', in *ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, Vol. 4199, pp.351–364.

Alexander, C. (1964) *Notes on the Synthesis of Form*, Harvard University Press. Harvard University Press, Cambridge, Massachusetts.

Alonso, G., Casati, F., Kuno, H., and Machiraju, V. (2004) *Web Services: Concepts Architectures and Applications*, Springer-Verlag.

Bézivin, J., Hammoudi, S., Lopes, D. and Jouault, F. (2004) 'An experiment in mapping web services to implementation platforms', Technical report, LINA, University of Nantes.

Bordbar, B. and Staikopoulos, A. (2004) 'On behavioural model transformation in web services', in *Conceptual Modelling for Advanced Application Domain (eCOMO)*, pp.667–678, Shanghai, China.

BPEL (2003) *Business Process Execution Language for Web Services (BPEL4WS), Version 1.1*, available at http://www-128.ibm.com/developerworks/library/specification/wsbpel/.

Brahe, S. and Østerbye, K. (2006) 'Business process modeling: defining domain specific modeling languages by use of UML profiles', in Rensink, A. and Warmer, J., (Eds.), *ECMDA-FA 2006*, LNCS, Vol. 4066, pp.241–255, Springer, Heidelberg.

Dirgahayu, T., Quartel, D. and Sinderen, M. (2007) 'Development of transformations from business process models to implementations by reuse', in *3rd International Workshop on Model-Driven Enterprise Information Systems*, MDEIS 2007, pp.41–50.

Dirgahayu, T., Quartel, D. and Sinderen, M. (2008) 'Designing interaction behaviour in service-oriented enterprise application integration' in *ACM Symposium on Applied Computing*, pp.1048–1054.

Eriksson, H. and Penker, M. (2000) *Business Modeling with UML. Business Patterns at Work*, John Wiley & Sons, Inc.

Erl, T. (2005) *Service Oriented Architecture: Concepts, Technology and Design*, Prentice Hall.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Jouault, F. and Kurtev, I. (2005) 'Transforming models with ATL', in *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego Bay, Jamaica.

Kleppe, A., Warmer, J. and Bast, W. (2003) *MDA Explained: The Model Driven Architecture–Practice and Promise*, The Addison-Wesley Object Technology Series. Addison-Wesley.

Koehler, J., Hauser, R., Kapoor, S., Wu, F. Y. and Kumaran, S. (2003) 'A model-driven transformation method', in *7th International Enterprise Distributed Object Computing Conference (EDOC 2003)*, pp.186–197.

Koehler, J., Hauser, R., Sendall, S. and Wahler, M. (2005) 'Declarative techniques for model-driven business process integration', *IBM Systems Journal*, Vol. 44, No. 1, pp.47–65.

Kroll, P. and Kruchten, P. (2003) *The Rational Unified Process Made Easy. A Practitioner's Guide to the RUP*, Addison Wesley.

MacDonald, S., Szafron, D., Schaeffer, J., Anvik, J., Bromling, S. and Tan, K. (2002) 'Generative design patterns', in *IEEE International Conference on Automated Software Engineering*, pp.23–34.

Mitra, T. (2005) 'Business-driven development', IBM developer works article, available at http://www.ibm.com/developerworks/webservices/library/ws-bdd.

Model Driven Architecture (MDA) (2007) Object Management Group, available at www.omg.org/mda/.

Pokraev, S., Quartel, D., Steen, M., Wombacher, A. and Reichert, M. (2007) 'Business level service-oriented enterprise application integration', in *3rd International Conference on Interoperability for Enterprise Software and Applications*, pp.507–518.

QVT (2007). *Object Management Group: MOF 2.0 Query/Views/Transformations*, Final adopted specification ptc/07-07-07, available at Object Management Group at www.omg.org.

Shishkov, B., Sinderen, M. and Tekinerdogan, B. (2007) 'Model-driven specification of software services', in *ICBE '07: Proceedings of the IEEE International Conference on E-business Engineering*, pp.13–21.

Skogan, D., Grønmo, R. and Solheim, I. (2004) 'Web service composition in UML', in *Eighth IEEE International Enterprise Distributed Object Computing Conference (EDOC'04)*, pp.47–57.

Stahl, T., Völter, M., Bettin, J., Haase, A. and Helsen, S. (2006) *Model-Driven Software Development: Technology, Engineering, Management*, Wiley.

UML2.0 (2004) 'UML 2.0 superstructure specification, final adopted specification', available at http://www.omg.org/docs/formal/05-07-04.pdf.

van der Aalst, W.M.P., Hofstede, A.H.M., Kiepuszewski, B. and Barros, A.P. (2003) 'Workflow patterns', *Distributed and Parallel Databases*, Vol. 14, No. 1, pp.5–51.

Wagner, H-T., Beimborn, D., Franke, J. and Weitzel, T. (2006) 'IT business alignment and IT usage in operational processes: a retail banking case', in *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, Vol. 8, pp.172–194.

White, S. (2006) *Business Process Modeling Notation, Version 1.0*, Final adopted version, available at http://www.bpmn.org/Documents/OMG-02-01.pdf.