Development of Transformations from Business Process Models to Implementations by Reuse[1]

Teduh Dirgahayu, Dick Quartel, and Marten van Sinderen

Centre for Telematics and Information Technology
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands
{t.dirgahayu, d.a.c.quartel, m.j.vansinderen}@utwente.nl

Abstract. This paper presents an approach for developing transformations from business process models to implementations that facilitates reuse. A transformation is developed as a composition of three smaller tasks: pattern recognition, pattern realization and activity transformation. The approach allows one to reuse the definition and implementation of pattern recognition and pattern realization in the development of transformations targeting different business process modeling and implementation languages. In order to decouple pattern recognition and pattern realization, the approach includes a pattern language to represent the output of the pattern recognition task, which forms the input of the pattern realization task.

1 Introduction

The behaviour represented in a business process model is not only determined by its activities, but also by its structure. This structure defines how the activities are related, which determines amongst others the order in which the activities are executed. In general, the structure is composed of generic structures representing well-known and frequently-used relationships, such as sequence, choice and concurrency. We use the term *pattern* to denote these generic structures. A pattern is an example of a structure concept, as described in [16]. It represents some type of relationship between activities without determining what activities are related. Patterns are typically nested to form the structure of a model.

In a transformation from business process models to implementations, two successive tasks can be identified in transforming patterns: pattern recognition and pattern realization. Pattern recognition identifies the patterns that form the structure of a model. Pattern realization translates these patterns into constructs of an implementation language.

Pattern recognition is not a trivial task. The definition and implementation of pattern recognition is expensive in terms of effort, cost and time. When one develops transformations from a modeling language to a set of implementation languages, it would be efficient if pattern recognition can be developed once and be reused with

[1] This work is part of the Freeband A-MUSE project (http://a-muse.freeband.nl), which is sponsored by the Dutch government under contract BSIK 03025.

different pattern realizations. This is possible because pattern recognition is specific to the modeling language and independent from implementation languages. If a new pattern must be recognized, it is only the pattern recognition that needs to be extended. In our proposal, patterns identified by pattern recognition are documented in an intermediate model that is defined in a kind of pattern language.

Pattern realization translates elements in the intermediate model to constructs of an implementation language. When one develops transformations from a set of modeling languages to an implementation language, it would be efficient if the same pattern realization can be reused with different pattern recognition. This is possible because pattern realization is specific to the implementation language and independent form modeling languages.

To allow reuse, pattern recognition and pattern realization should be decoupled from each other. This can be achieved by defining the intermediate model in a language that is independent from any modeling and implementation languages. This pattern language is used as a standard way for representing patterns found in a model. Pattern recognition and pattern realization are developed as individual transformations. In order to keep the definition and implementation of pattern realization simple, it is preferable that a pattern is represented as a single concept in an intermediate model.

The objective of this paper is to present an approach that facilitates the reuse of pattern recognition and pattern realization in transformations from business process models to implementations. The approach aims at giving transformation development the benefits of development by reuse, i.e. less development cost, shorter time-to-market and correctness [9, 19]. The approach includes a pattern language for defining intermediate models between pattern recognition and pattern realization.

The structure of this paper is as follows. Section 2 discusses our approach in more detail. Section 3 presents a pattern language for defining intermediate models. Section 4 gives examples of transformations. Section 5 relates our work with other research activities. Finally, section 6 presents our conclusions and future work.

2 Approach

The idea of our approach is illustrated in Figure 1. Suppose that we develop a UML-to-BPEL transformation. The transformation is developed as a composition of three smaller tasks: T_A for transforming UML activities into BPEL basic activities, T_{P1} for recognizing patterns in UML and T_{P2} for realizing these patterns into BPEL structured activities. Patterns recognized in the source model M_{UML} are documented in the intermediate model M_{int}. The separation of pattern transformations T_{P1} and T_{P2} from activity transformation T_A is to let the intermediate model M_{int} focus on patterns and be independent from UML and BPEL. Each activity in the source model is associated with a unique identifier that is maintained in all the transformations. The results of transformations T_{P2} and T_A are combined to produce a BPEL model. The identifier of an activity is used to correctly put a BPEL basic activity associated with that identifier within a BPEL structured activity in which the BPEL basic activity should be contained. This example is presented in more detail in Section 4.

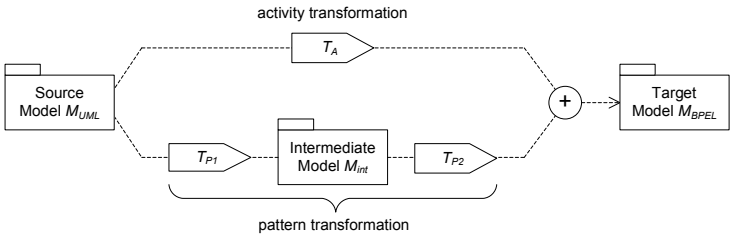

Fig. 1. Separating pattern transformations T_{P1} and T_{P2} from activity transformation T_A

Figure 2 shows how this approach facilitates the reuse of pattern recognition and pattern realization. Activity transformations between source and target models are omitted for brevity. We have transformations T_{P1} for recognizing patterns in UML and T_{P2} for realizing patterns in BPEL. To have a UML-to-Java transformation, we reuse transformation T_{P1} and develop transformation T_{P4} for realizing patterns in Java. To have a BPMN-to-Java, we develop transformation T_{P3} for recognizing patterns in BPMN and reuse transformation T_{P4}. To have a BPMN-to-BPEL, we reuse transformations T_{P3} and T_{P2}.

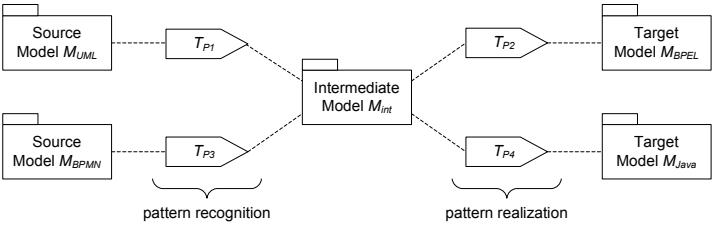

Fig. 2. Reusing pattern recognition and pattern realization

An activity transformation is not reusable because it is specific to the languages of the source and target models. Consider, for example, the mapping of a UML AcceptEventAction to a BPEL receive activity. The UML action may require BPEL-specific information to be supplied to the BPEL activity, e.g. partnerLink, portType and operation. A UML-to-BPEL activity transformation must handle this BPEL-specific information. This makes the transformation is not reusable for transforming a UML activity to other implementation languages. Obviously the transformation cannot be used to transform, for instance, BPMN activities to BPEL activities. Therefore, an activity transformation has to be developed for each pair of modeling and implementation languages.

2.1 Requirements for a Pattern Language

To facilitate the reuse of pattern recognition, an intermediate model should be defined in a pattern language that is independent from (i.e., abstracts from distinguishing features of) any implementation language. An intermediate model that is specific to an implementation language can only be realized in that implementation language. In that case, pattern recognition cannot be reused with other implementation languages.

To facilitate the reuse of pattern realization, a pattern language should be independent from any modeling language. A pattern language that is specific to a modeling language may not be able to document patterns defined in other modeling languages. In that case, the pattern realization is useful for that modeling language only.

In order to define a pattern language, we need a limited set of basic patterns. Patterns in a business process model will be documented in an intermediate model in terms of those basic patterns. We investigate patterns that are commonly found in business processes models and in implementations. We focus on realizing business process models in an imperative language or in the imperative part of an implementation language.

We consider workflow patterns as patterns that are commonly found in business processes. In [1], twenty different workflow patterns have been identified. Our approach uses workflow patterns as modeling constraints, i.e. any composition in the model should conform to one of these patterns. Pattern recognition should be able to identify workflow patterns in a model.

Control patterns [7, 20] are patterns that are found in implementations. A control pattern defines a certain execution ordering of activities in a structured program. The patterns are *sequence*, *selection* and *iteration*. Concurrent programming [3] adds another control pattern called *concurrence*.

We have three alternatives to determining a limited set of basic patterns for a pattern language. The alternatives are as the following.

1. *Workflow patterns as basic patterns*. Many basic patterns will have no direct (1-to-1) mapping to constructs of an implementation language. Pattern realization must translate such a basic pattern into a composition of constructs of an implementation language. A drawback of this alternative is that if a new workflow pattern must be recognized, the pattern language, pattern recognition and pattern realization must be extended.
2. *Control patterns as basic patterns*. Every basic pattern will be able to be mapped directly to a construct of an implementation language. A workflow pattern must be represented as a composition of one or more basic patterns in an intermediate model. If a new workflow pattern must be recognized, only pattern recognition must be extended.
3. *Workflow and control patterns as basic patterns*. This alternative combines the first and second alternatives. It brings the drawback of first alternative.

We took the second alternative to define a pattern language that we call *Common Behavioural Patterns Language* (CBPL). We present the language in Section 3. CBPL can be considered as an abstract platform [2] that offers a large set of implementation languages to realize an intermediate model. Each CBPL pattern can be realized in many implementation languages, such as BPEL, Java, C/C++ or VB.

2.2 Documenting Patterns

We can distinguish between workflow patterns that are comparable to CBPL patterns and workflow patterns that are not comparable (i.e., have no 1-to-1 mappings; see Table 1). The parallel split and exclusive choice of workflow patterns are comparable

to CBPL concurrence and selection, respectively. A synchronization pattern can be composed with a parallel split pattern to allow an activity be performed after a CBPL concurrence. A simple merge pattern can be composed with an exclusive choice pattern to allow an activity be performed after a CBPL selection.

Table 1. Comparison of workflow patterns and CBPL patterns

Workflow patterns	**CBPL Patterns**
Comparable patterns	
Sequence	Sequence
Parallel split (+ Synchronization)	Concurrence
Exclusive choice (+ Simple merge)	Selection
Arbitrary cycles[a], Multiple instances (MI)[b]	Iteration
Incomparable patterns	
Implicit termination	-
Deferred choice	-
Interleaved parallel routing	-
Milestone	-
Multi-choice	-
Synchronizing merge	-
Multi-merge	-
Discriminator	-
Cancel activity	-
Cancel case	-

[a] in special cases called *structured cycles*

[b] consists of four patterns: MI without synchronization, MI with a priori design knowledge, MI with a priori runtime knowledge, and MI without a priori runtime knowledge.

The arbitrary cycles pattern indicates that an activity can be performed repeatedly without imposing some structure. A special case of this pattern, which is called structured cycle [1], is comparable to CBPL iteration.

The multiple instances pattern indicates that an activity can have multiple instances at a same time. This pattern may not be directly supported in some implementation languages [1]. A work-around solution [21] performs multiple instances of the activity at different times. This solution is comparable to CBPL iteration.

A workflow pattern that is incomparable to a CBPL pattern should be documented as a composition of CBPL patterns. When the precise semantics of this workflow pattern cannot be documented as a composition of CBPL patterns, a work-around solution [21] can be developed to closely represent the semantics.

3 Common Behavioural Patterns Language

Figure 3 presents the CBPL metamodel. An *activity* is an abstract concept that represents an activity with a certain purpose. An activity can be a *structure activity* or an *action*. A structure activity represents an activity whose purpose is to determine the

execution ordering of other activities. This activity is used to define CBPL patterns. An action represents an activity whose purpose is to perform a certain task. It contains an identity to relate the action to an activity in activity transformation.

A pattern is a kind of a structure activity. CBPL patterns are represented as *sequence*, *concurrence*, *selection* and *iteration*. A sequence contains one or more activities to be executed in succession. A concurrence contains two or more activities that can be executed independently. An iteration contains an activity to be executed repeatedly as long as its condition holds. A selection contains one or more *cases* to be selected. A case contains an activity to be executed when its condition holds. A *default case* is selected when no other case can be selected.

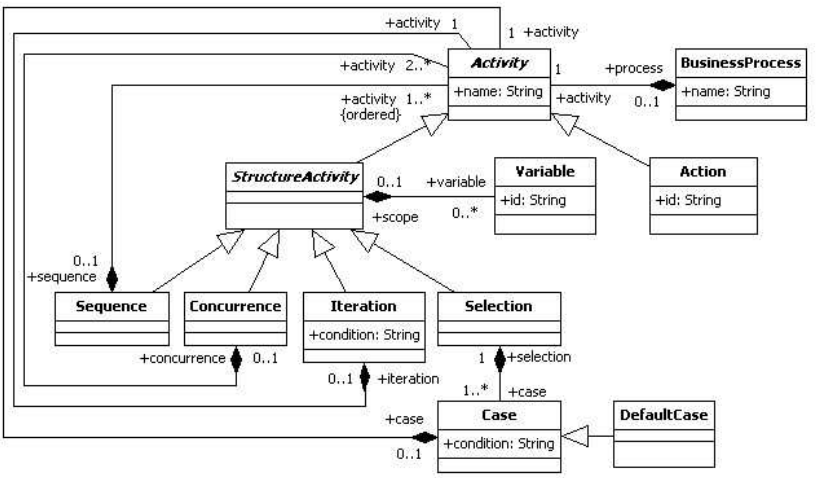

Fig. 3. CBPL metamodel

A structure activity may define *variables*. A variable holds an information value that is required by the execution of the structure activity or the activities contained in the structure activity. A variable is transformed by activity transformations but its declaration must be in a proper scope, i.e. a structure activity. A variable contains an identity to relate the variable to a variable in an activity transformation.

A *business process* defines the behaviour of a business process in order to provide its functionality. It contains an activity to be performed. When a business process consists of many activities, the activity of a business process is a structured activity containing those activities.

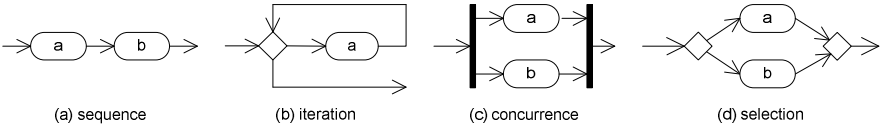

Fig. 4. Representation of CBPL patterns in UML

We do not provide a concrete syntax for CBPL. CBPL patterns should be expressed using elements of the modeling language of a business process model.

Figure 4 shows the representation of CBPL patterns in UML activities diagram. Join and merge nodes of the concurrence and the selection patterns, respectively, are optional.

4 Example

We illustrate the use of our approach in the development of a UML-to-BPEL transformation. The transformation is developed as a composition of a UML-to-CBPL transformation as pattern recognition, a CBPL-to-BPEL transformation as pattern realization and a UML-to-BPEL activity transformation. The example focuses on pattern recognition and pattern realization.

Figure 5 shows an example of a business process model of an insurance application. The business process receives an application and determines whether the application type is collective or individual. The business process then calls another service according to the application type. When the business process receives a confirmation from the service, it forwards the confirmation to the applicant.

The structure of the model is formed from two CBPL patterns: sequence and selection. A CBPL sequence pattern is formed from an AcceptEventAction `receive Application`, a composition that conforms to the CBPL selection pattern and a SendSignalAction `replyConfirmation`. A CBPL selection pattern is formed from two CallOperationActions: `applyForCollective` and `applyForIndividual`. Information that is not relevant to pattern transformation is omitted for brevity.

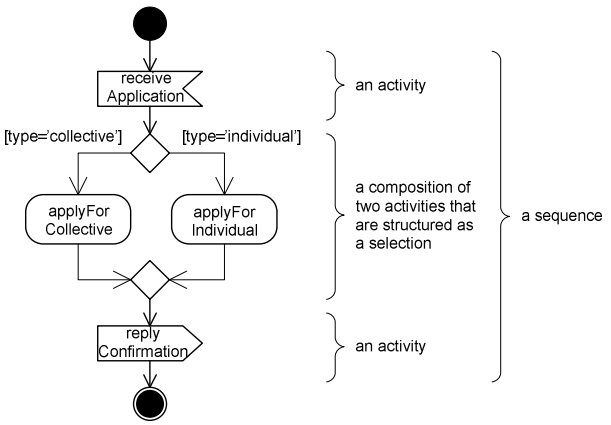

Fig. 5. A business process model of insurance application

The UML-to-CBPL transforms the UML model to an intermediate model depicted in Figure 6(a). It is a good practice to put the only action of a structure activity in a sequence pattern. If the pattern realization then transforms the action into a succession of activity constructs, the activity constructs are already in a sequence. The CBPL-to-BPEL transformation transforms the intermediate model to a BPEL model depicted in Figure 6(b). The figure shows only constructs that are relevant to patterns. The

similarity between the models and between their metamodels (see [6, 8] for the BPEL metamodel) eases the development of a CBPL-to-BPEL transformation.

If we want to develop an UML-to-Java transformation, we can reuse the pattern recognition that was developed in the UML-to-BPEL transformation. For the example in Figure 5, the pattern recognition will give the same output as Figure 6(a). The structure of the intermediate model resembles the structure of a Java method. This makes the development of a CBPL-to-Java transformation relatively easy. A sequence is transformed to a scope indicated with a pair of curly brackets `{}`; a selection is transformed to a keyword `if`; and a case within the selection becomes the condition of the `if`.

```
<sequence>
 <action id="1" />
  <selection>
   <case collective>
    <sequence>
     <action id="2" />
    </sequence>
   </case>
   <case individual>
    <sequence>
     <action id="3" />
    </sequence>
   </case>
  </selection>
 <action id="4" />
</sequence>
```

(a) intermediate model in CBPL

```
<sequence>
 <receive .../>
 <switch>
  <case collective>
   <sequence>
    <assign .../>
    <invoke .../>
    <assign .../>
   </sequence>
  </case>
  <case individual>
   <sequence>
    <assign .../>
    <invoke .../>
    <assign .../>
   </sequence>
  </case>
 </switch>
 <reply .../>
</sequence>
```

(b) target model in BPEL

Fig. 6. The example in CBPL and BPEL

5 Related Work

The need for transformation (de)composition is studied in [10]. The study addresses issues that should be considered in transformation decomposition and composition, such as order of rule execution, tangling and scattering concerns, and additive changes. The study focuses on the development of a transformation language that can handle those issues.

(De)composition of a transformation are also studied in the area of aspect orientation [5, 12, 17]. A transformation is decomposed according to concerns, e.g. logging, security and transaction. Aspect orientation does not consider the structure and activities of a business process model as concerns and, hence, does not decompose a transformation according to them.

Transformations from business process models to implementations can be found in [6, 11, 15, 18]. None of them indicates how the transformations are (de)composed. Each develops a transformation directly based on a transformation specification.

Pattern transformation can be considered as an implementation of pattern application approach [13]. Our approach extends the idea of pattern application such that pattern application can be developed by reuse.

The UML 2.1.1 StructuredActivities package [14] is to support traditional structured programming constructs. It provides the concepts of sequence, conditional and loop nodes, which are similar, respectively, to sequence, selection and iteration patterns in CBPL. UML has no single concept for concurrence. CBPL allows concurrence be represented as a single concept.

6 Conclusions and Future Work

In this paper, we present an approach for developing a transformation from business process models to implementations that facilitates reuse. The approach separates pattern transformation from activity transformation. Pattern transformation is developed as a composition of two transformations that are performed in succession: pattern recognition and pattern realization. Pattern recognition identifies patterns that form the structure of a model. Pattern realization translates these patterns into constructs of an implementation language. The results of pattern realization and activity transformation are combined to produce the implementation of the business process model. The approach facilitates the reuse of pattern recognition and pattern realization in transformations from different modeling languages to different implementation languages.

The approach takes workflow patterns as knowledge for pattern recognition. Thus, workflow patterns act as modeling constraints. Patterns recognized are documented in an intermediate model. The approach includes a pattern language, which we call CBPL, to define this intermediate model. CBPL documents workflow patterns in terms of CBPL patterns.

We have developed a transformation from business process models in ISDL [4] to BPEL as a composition of pattern recognition and pattern realization. Activity transformation is not developed as an individual transformation yet. The activity transformation is embedded in the pattern recognition and pattern realization. Currently we are refactoring the pattern recognition and pattern realization such that the activity transformation part stands as an individual transformation. We will then compose the pattern recognition, pattern realization and activity transformation as suggested by the approach described in this paper.

Our future work will investigate how to map workflow patterns that are not comparable to CBPL patterns onto CBPL. The mapping gives us an opportunity to evaluate the stability of the CBPL metamodel with regard to addition of new workflow patterns. To prove the idea of transformation development by reuse, a transformation from ISDL to Java will be developed by reusing pattern recognition that has been developed in our ISDL-to-BPEL transformation.

References

1. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P. Workflow Patterns. Distributed and Parallel Databases Volume 14 No. 3. Springer (2003) 5-51
2. Almeida, J.P.A. On the Notion of Abstract Platform in MDA Development. Proc. of 8th IEEE Intl. Enterprise Distributed Object Computing Conference (2004) 253-263
3. Andrews, G.R., Schneider, F.B. Concepts and Notations for Concurrent Programming. ACM Computing Survey Volume 15 Issue 1 (1983) 3-43
4. ASNA. ISDL Home. http://isdl.ctit.utwente.nl
5. Barbosa, P.A.A, Contreras, C.F.G., Rodriguez, J.M.M. MDA and Separation of Aspects: An Approach based on Multiple Views and Subject Oriented Design. Proc. of 6th Intl. Workshop on Aspect Oriented Modeling (2005)
6. Bordbar, B., Staikopoulos, A. On Behavioural Model Transformation in Web Services. Proc. of ER Workshops 2004. LNCS Volume 3289. Springer, Berlin (2004) 667-67
7. Chapin, N., Denniston, S.P. Characteristic of a Structured Program. SIGPLAN Notices, Volume 13 Issue 5 (1978) 36-45
8. Dirgahayu, T. Model-Driven Engineering of Web Services Composition: A Transformation from ISDL to BPEL. MSc thesis. University of Twente, Enschede (2005)
9. Hallsteinsen, S., Paci, M. (eds.) Experiences in Software Evolution and Reuse. Twelve Real World Projects. Springer, Berlin (1997)
10. Kurtev, I. Adaptability of Model Transformations. PhD thesis. University of Twente, Enschede (2005)
11. Koehler, J., Hauser, R., Kapoor, S., Wu, F.Y., Kumaran, S. A Model-Driven Transformation Method. Proc. of 7th IEEE Intl. Enterprise Distributed Object Computing Conference (2003) 186-197
12. Kulkarni, V., Reddy, S., Separation of Concerns in Model-Driven Development. IEEE Software Volume 20 Issue 5 (2003) 64-69
13. OMG. MDA Guide version 1.0.1. omg/2003-06-01 (2003)
14. OMG. Unified Modeling Language: Superstructure version 2.1.1. formal/2007-02-03 (2007)
15. Patrascoiu, O. Mapping EDOC to Web Services using YATL. Proc. of 8th IEEE Intl. Enterprise Distributed Object Computing Conference (2004) 286-297
16. Quartel, D., Dijkman, R., van Sinderen, M. Extending profiles with stereotypes for composite concepts. Proc. of 8th ACM/IEEE Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS). LNCS Volume 3713. Springer, Berlin (2005) 232-247
17. Solberg, A., Simmonds, D., Reddy, R., Ghosh, S., France, R. Using Aspect Oriented Techniques to Support Separation of Concerns in Model Driven Development. Proc. of 29th Annual Intl. Computer Software and Applications Conf. (2005) 121-126
18. Specht, T., Drawehn, J., Thranert, M., Kuhne, S. Modeling Cooperative Business Processes and Transformation to a Service Oriented Architecture. Proc. of 7th IEEE Intl. Conf. on E-Commerce Technology. (2005) 249-256
19. Sutcliffe, A. The Domain Theory: Patterns for Knowledge and Software Reuse. Lawrence Erlbaum Associates, Inc. (2002)
20. Williams, M.H. Generating Structured Flow Diagrams: The Nature of Unstructuredness. The Computer Journal Volume 20 No. 1 (1977) 45-50
21. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M. Analysis of Web Services Composition Languages: The Case of BPEL4WS. LNCS Volume 2813. Springer, Berlin (2003) 200-215