Analysis, Transformation and Improvements of ebXML Choreographies based on Workflow Patterns

Ja-Hee Kim* and Christian Huemer

Research Studios Austria Studio Digital Memory Engineering
ARC Seibersdorf research GmbH Thurngasse 8/20, A-1090 Wien
Department of Computer Science and Business Informatics, University of Vienna,
at Liebiggasse 4/3-4, 1010 Vienna, Austria.
kim@mminf.univie.ac.at christian.huemer@univie.ac.at

Abstract. In ebXML the choreography of a business process should be modeled by UMM (UN/CEFACT Modeling Methodology) and is finally expressed in BPSS (Business Process Specification Schema). Our analysis of UMM and BPSS by workflow patterns shows that their expression power is not always equivalent. We use the workflow patterns to specify the transformation from UMM to BPSS where possible. Furthermore, the workflow patterns help to show the limitations of UMM and BPSS and to propose improvements.

1 Introduction

The trend towards service-oriented architectures resulted in a growing interest in the choreography of B2B business processes, which is in the focus of this paper. The most prominent example of an service-oriented architecture is Web Services [1]. Web Services are defined as a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web Service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols [2]. The Web Services base standards are WSDL, UDDI and SOAP. However, Web Services are isolated and opaque. Business processes require collections of Web Services jointly used to realize more complex functionality [3]. This lead to the development of the Business Process Execution Language for Web Services (BPEL). BPEL's primarily focuses on the orchestration of executable business processes. In addition, BPEL supports so-called abstract processes for specifying a choreography of business protocols between business partners [4].

Apart from Web Services, the ebXML framework is another important approach. In contrast to Web Services, ebXML has been developed specifically for e-business. ebXML is also based on a service-oriented architecture. ebXML provides a set of loosely coupled specifications that enable so-called business service

* This work was partially supported by the Post-doctoral Fellowship Program of Korea Science & Engineering Foundation (KOSEF).

interfaces (BSI) of different business partners to interoperate. These specifications span over the topics of messaging, registries, profiles & agreements, business processes, and core (data) components. Accordingly, business serves interfaces are expected to carry out standardized business processes. The ebXML architecture specification recommends to use the UML-based UN/CEFACT Modeling Methodology (UMM) for analyzing and designing the inter-organizational business processes. Those aspects that are relevant for configuring the business service interfaces are mapped to the XML-based business process specification schema (BPSS). BPSS instances are stored in a registry and are referenced by the profiles of companies supporting the corresponding business process.

As mentioned above, interoperability requires that collaborating business partners implement a shared business logic. BPEL and UMM/BPSS provide languages describing a share business logic with respect to the choreography of a business process. Thus, it is important that these languages lead to unambiguous definitions of business processes. Furthermore, these languages must be able to capture choreography requirements that appear in any B2B business process. Inasmuch it is important to systematically evaluate the capabilities of these languages. There does not exist a special metric for evaluating B2B processes. However, a B2B business process might be considered as an inter-organizational workflow. Aalst et al. developed workflow patterns to analyze executable workflows [5]. An evaluation of BPEL according to these patterns is provided in Wohed et al. [6]. In our paper we use the same patterns to evaluate ebXML processes. In other words, we analyze UMM version 12 [7] and BPSS 1.1 [8]. Both UMM and BPSS describe a choreography rather than an executable process orchestration. An ebXML process flow consists of collaborative activities that are decomposed in a way that each of the two collaborating partners perform exactly one activity (c.f. Section 2). From a specific partner's view the process flow is still the same, but instead of the collaborative activities the flow consists only of the activities assigned to the corresponding partner. Thus, we feel that the patterns are relevant even to analyze a choreography.

We demonstrate how the workflow patterns are realized in UMM and BPSS. Patterns that cannot be realized usually indicate limitations of the current versions and give hints for improvements in future revisions. Showing how a pattern is expressed in both standards, helps to identify mapping rules between the standards. This is important for automatically deriving a BPSS specification from a UMM model, but also for reverse engineering.

The remainder of this paper is structured as follows. Section 2 gives an introduction into the two standards of interest: UMM and BPSS. In Section 3 we show how the 20 patterns, which are organized in 6 categories, are expressed in both UMM and BPSS. Each pattern supported by the standards is demonstrated by means of a practical example. The summary in Section 4 gives an overview of the patterns supported and of the derived mapping rules.

2 Overview of UMM and BPSS

Since 1997 the United Nationss Centre for Trade Facilitation and Electronic Business (UN/CEFACT) has been developing it modeling methodology UMM. UMM concentrates on the business operational view (BOV) of the Open-edi reference model [9]. The BOV is limited to those aspects regarding the making of business decisions and commitments among organizations. This means that UMM is independent of the technology - e.g. Web Services or ebXML - used to implement a B2B partnership. UMM is based on UML. It defines a UML profile for modeling the business aspects of inter-organizational business processes. The UMM methodology covers 4 views: The business domain view (BDV) is used to gather existing knowledge. It collects information about existing business processes and does not construct new ones. The goal of the business requirements view (BRV) is to identify possible business collaborations in the considered domain and to detail the requirements of these collaborations. The business transaction view (BTV) defines the choreography of the business collaboration and structures the business information exchanged. The fundamental principle of the business service view (BSV) is to describe the interactions between network components.

The most important view for our evaluation is the BTV, since it deals with the choreography of the inter-organizational business process called business collaboration in UMM. A business collaboration is performed by two (= binary collaboration) or more (multi-party collaboration) business partners. A business collaboration might be complex involving a lot of activities between business partners. However, the most basic business collaboration is a binary collaboration realized by a request from one side and an optional response from the other side. This simple collaboration is a unit of work that allows roll back to a defined state before it was initiated. Therefore, this special type of collaboration is called business transaction.

Consequently, a business transaction consists always of two collaborating activities. Each activity is performed by one business partner. The initiating business activity outputs information that is sent to the reacting business activity. In case of a simple information distribution or notification the reacting business activity processes the information and the transaction is completed. If a response is expected the reacting business activity outputs the business information and returns it to the initiating business activity. Note, that acknowledgments are not explicitly modeled in the BTV, but time values assigned to a business activity signify that they expect an acknowledgment from the collaborating activity in a given time frame.

In UMM a business transaction is modeled by an activity graph. Fig.1 shows the example of an *authorize payment* business transaction in UMM. Owing to the strict well-formedness rules described above, a business transaction follows always the same pattern as shown in Fig.1. In case of information distribution and notification the object flow returning a business document is omitted. Due to the strict choreography of the activities within a business transaction, our

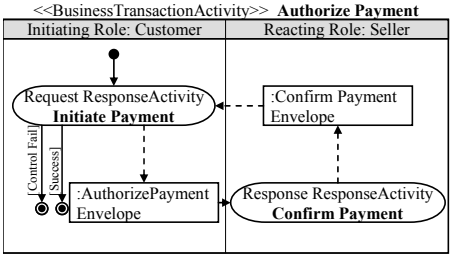


Fig. 1. An example of a business transaction.

pattern based analysis in the following section does not consider the activities within a business transaction.

A business collaboration is built by more business transactions. It is important that the business collaboration defines an execution order for the business transactions. In UMM, this choreography is defined by an activity graph called business collaboration protocol. In the current version 12 of UMM all activities of the business collaboration protocol must be stereotyped as business transaction activities. A business transaction activity must be refined by the activity graph of a business transaction. This means that recursively nesting business collaborations is not possible in UMM. A business collaboration protocol is able to model a multi-party collaboration. However, each business transaction involves exactly two partners by definition. Our pattern-based analysis evaluates the choreography of the business collaboration protocol. It checks whether a certain pattern is supported by the business collaboration protocol or not. All the examples illustrated in Fig.3 to Fig.10 are business collaboration protocols. It is important to note that UMM is based on UML 1.4. Therefore some limitations are a result of limitations of UML 1.4. We will point out if they are solved by UML 2.0.

The work on BPSS was based on the UMM meta model. However, it is not mandatory to use UMM in order to create an BPSS instance. The goal of the BPSS is to provide the bridge between e-business process modeling and specification of e-business software components [8]. It provides an XML schema to specify a collaboration between business partners, and to provide configuration parameters for the partners' runtime systems in order to execute that collaboration between a set of e-business software components. BPSS identified those UMM modeling elements that are relevant for the runtime systems and discarded the rest. The relevant modeling elements have been expressed in XML schema.

The UMM artefacts that are considered by BPSS are more or less the business transaction and the business collaboration protocol. Nevertheless, the mapping is not always straight forward as we will recognize in the next section. Again, our analysis will not evaluate the activities within a business transaction due to its strict choreography. Therefore, the analysis considers the business collaboration protocol equivalent called binary collaboration. As the name indicates, BPSS supports only the definition of collaborations between two partners. Multi-party

collaborations were deprecated in BPSS 1.1. In contrast to UMM, the activities within a collaboration might not only refer to business transactions, but also to other collaborations. Therefore, a recursive nesting of binary collaborations is possible. In order to align with the UMM examples we will not use this concept in our analysis. Fig.2 presents the XML schema definition for a binary collaboration in BPSS.

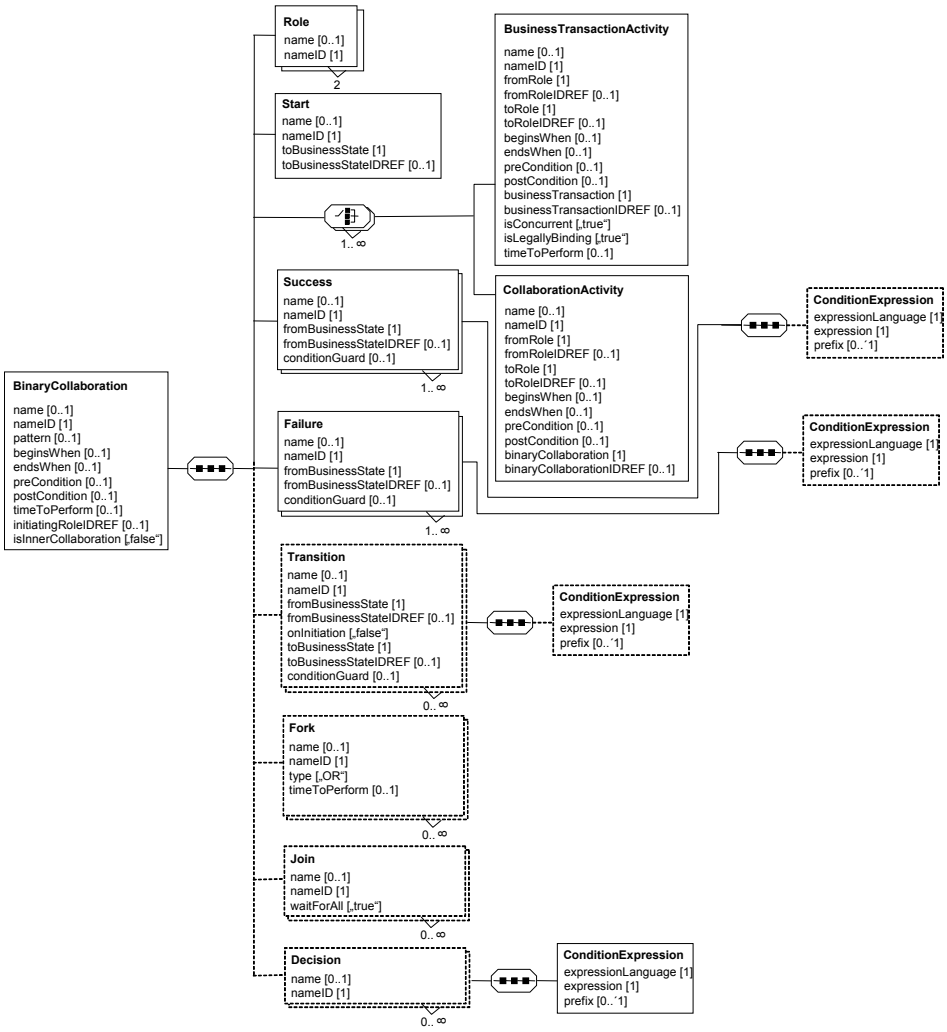


Fig. 2. BPSS 1.1 binary collaboration element.

3 Workflow pattern based transformation

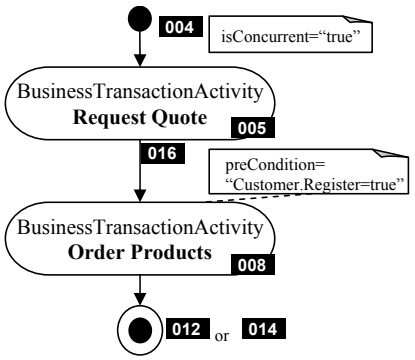
In this section we analyze UMM and BPSS based on well-known workflow patterns proposed by Aalst *et al.* [5]. Since UMM is based on UML, analyzing UMM is very similar to UML [10]. However, UMM's meta model defines B2B-specific tagged values. Sometimes a pattern is realized by these tagged values - which is marked 't' in Table 1 at the end of the paper. Furthermore, UMM does not use all features of UML activity graphs due to a more restrictive meta model. These workflow patterns are categorized into six classes - basic control patterns, advanced branching and synchronization patterns, structural patterns, patterns involving multiple instances, state-based patterns, and cancelation patterns. The UMM and BPSS analysis for each class of patterns is presented in a separate subsection. This analysis shows the expression power and the limitations both of UMM and BPSS. It gives hints to improve the expression power of UMM and BPSS. Furthermore, the analysis helps to derive the transformation rules between UMM model and BPSS.

3.1 Basic control flow patterns

Aalst *et al* categorize basic control patterns into *sequence, parallel split, synchronization, exclusive choice*, and *simple merge*. They are similar to definitions of elementary control flow concepts provided by WfMC [11]. Both of UMM and BPSS support all theses patterns.

Sequence. A sequence pattern means all activities are executed one by one. Each subactivity state of UMM represents *business transaction activity* or *collaboration transaction activity* of BPSS and the state is connected to other state by *transition*. Fig.3 illustrates a very simple *binary collaboration* for ordering products. In this example, a request quote transaction is followed by the order products transaction.

The UMM *business collaboration protocol* is based on a UML 1.4 activity graph. BPSS was developed by mapping the UMM meta model of the *business collaboration protocol* into an XML representation. However, not all UMM concepts are represented one-to-one in BPSS. Therefore, a transformation from UMM models to BPSS is not straightforward. A very significant difference between UMM and BPSS is the handling of final states. A *final* state of UMM should be transformed to either *success* or *failure* element of BPSS. A *single* UMM final state representing both a successful and an unsuccessful result must be mapped to both a *success* and a *failure* element in BPSS. User input or naming convention of *final* state of UMM may be able to help the decision. Moreover, UMM needs two concepts for a transition to a *final* state: The transition and the state. However, the two concepts are merged into a single BPSS element, representing both a transition and a state. The same concept applies to initial states.



(a) UMM

```
001 <BinaryCollaboration nameID="BC1" timeToPerform="P2D">
002   <Role name="Seller" nameID="Role1" />
003   <Role name="Customer" nameID="Role2"/>
004   <Start toBusinessState="Request Quote" nameID="SS1" />
005   <BusinessTransactionActivity name="Request Quote"
006     nameID="BTA1" fromRole="Customer" toRole="Seller"
007     businessTransaction="Request Quote"isConcurrent="true" />
008   <BusinessTransactionActivity name="Order Products"
009     nameID="BTA2" fromRole="Customer" toRole="Seller"
010     preCondition="Customer.Register=true"
011     businessTransaction="Order Products" />
012   <Success nameID="FS1" fromBusinessState="Order Products"
013     conditionGuard="Success" />
014   <Failure nameID="FS2" fromBusinessState="Order Products"
015     conditionGuard="Failure" />
016   <Transition nameID="T1" fromBusinessState="Request Quote"/>
017 </BinaryCollaboration>
```

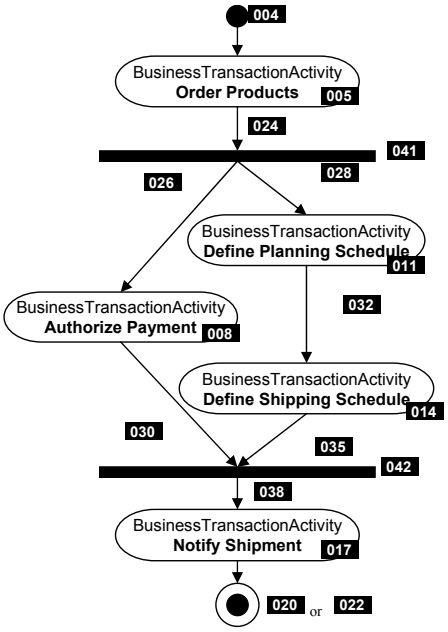
(b) BPSS

Fig. 3. An example of a sequence pattern.

Parallel split and synchronization. A parallel split pattern is a kind of AND-fork, after which multiple succeeding threads are executed in parallel. For example, after *ordering products*, the customer should *authorize the payment*. In parallel to this authorization, the *planning schedule* and a subsequent *shipping schedule is defined*. In UMM this parallel split pattern is modeled using pseudo state *fork* depicted by a bar (c.f. in Fig. 4a). BPSS uses a*fork* element of type *or* (see line 041 in Fig. 4b). This means that its attribute *type* is set to *or*. This is in opposite to *xor* which is discussed in the *deferred choice* pattern.

A synchronization pattern, a synonym for an AND-join, forms an antithesis to the parallel split pattern. A successor of a synchronization pattern starts if all its predecessors are completed. In Fig. 4a the seller ships the products after the completion of both activities *authorized payment* and *define shipping schedule*. This means that the *notify shipment* transaction must wait for the completion of both preceding activities. In UMM the synchronization pattern is realized by a *synchronization* pseudo state. Similarly to a *fork* state, the synchronization is depicted as a bar. BPSS realizes this pattern using a *join* element whose *wait for all* attribute is *true* (line 042 in Fig. 4b). This is in opposite to an OR-join where the the *wait for all* attribute is set to *false*.

Exclusive choice and simple merge. After an exclusive choice pattern one execution path is chosen from many alternative branches based on a decision. UMM uses a *decision* pseudo state which is depicted as a diamond. Usually, the decision is based on the state of a business object. For example, after *requesting quote* the customer may want to *order products*. If the customer is registered, the customer can *order products* right away. Otherwise the customer should register himself before ordering. In 5a the decision is based on the state of the *customer information*. If it is confirmed the next transaction is *order products*, and *register customers* otherwise. BPSS realizes the exclusive choice pattern by a *decision* element (line 028 in Fig. 5b). The decision element specifies a *condition*



(a) UMM

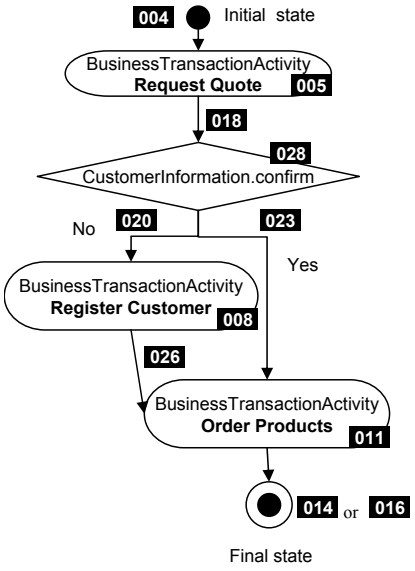
```
001 <BinaryCollaboration nameID="BC2">
002   <Role name="Seller" nameID="Role1" />
003   <Role name="Customer" nameID="Role2"/>
004 <Start toBusinessState="Order Products" nameID="SS2" />
005 <BusinessTransactionActivity name="Order Products"
006   nameID="BTA2" fromRole="Customer"
007   toRole="Seller" businessTransaction="Order Products"/>
008 <BusinessTransactionActivity name="Authorize Payment"
009    nameID="BTA3" fromRole="Customer"
010    toRole="Seller" businessTransaction="Authorize Payment"/>
011 <BusinessTransactionActivity name="Define Planning Schedule"
012   nameID="BTA4"  fromRole="Customer" toRole="Seller"
013   businessTransaction="Define Planning Schedule" />
014 <BusinessTransactionActivity name="Define Shipping Schedule"
015   nameID="BTA5" fromRole="Customer" toRole="Seller"
016   businessTransaction="Define Shipping Schedule"/>
017 <BusinessTransactionActivity name="Notify Shipment"
018   nameID="BTA6"  fromRole="Seller" toRole="Customer"
019   businessTransaction="Notify Shipment"/>
020 <Success nameID="FS3" fromBusinessState="Notify Shipment"
021    conditionGuard="Success"/>
022 <Failure nameID="FS4" fromBusinessState="Notify Shipment"
023    conditionGuard="Failure"/>
024 <Transition nameID="T2" fromBusinessState="Order Products"
025    toBusinessState="Fork"/>
026 <Transition nameID="T3" fromBusinessState="Fork"
027    toBusinessState="Authorize Payment"/>
028 <Transition nameID="T4" fromBusinessState="Fork"
029    toBusinessState="Define Planning Schedule"/>
030 <Transition nameID="T5" fromBusinessState="Authorize Payment"
031    toBusinessState="Join" />
032 <Transition nameID="T6"
033   fromBusinessState="Define Planning Schedule"
034   toBusinessState="Define Shipping Schedule"/>
035 <Transition nameID="T7"
036   fromBusinessState="Define Shipping Schedule"
037   toBusinessState="Join"/>
038 <Transition nameID="T8"
039   fromBusinessState="Join before Notify Shipment"
040   toBusinessState="Notify Shipment"/>
041 <Fork name="Fork" nameID="F1" type="OR" />
042 <Join name="Join" nameID="J1" waitForAll="true" />
043 </BinaryCollaboration>
```

(b) BPSS

Fig. 4. An example of a parallel split and a synchronization pattern

expression (line 020). All transitions starting from the decision (line 020 and 023) carry mutually exclusive condition guards with respect to the *condition expression*. Another realization of an exclusive choice is using the result of *binary transaction activity*. We detail this realization in arbitrary cycle.

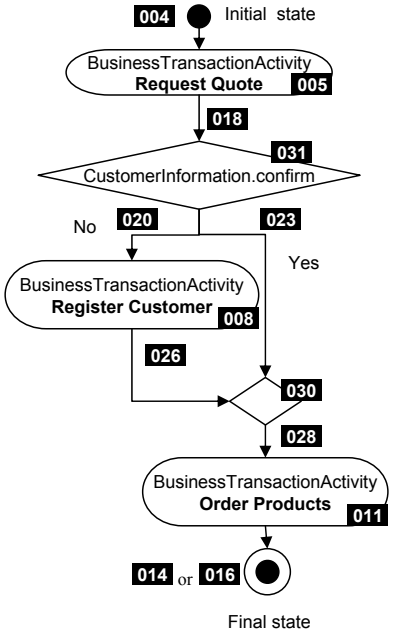
A simple merge pattern, an antithesis of the exclusive choice pattern, merges several alternative branches. For a simple merge pattern, neither any special pseudo state nor any element is mandatory. Multiple transitions leading to one state (*business transaction activity order products*) represent the pattern like Fig.5a (line 023 and 026 in Fig. 5b). However, UMM also supports a *merge* state depicted by a diamond as illustrated in Fig.5c. In this case, a *merge* state is transformed to a *join* element (line 030 in Fig.5d) whose attribute *wait for all* is *false*.



(a) UMM without *merge* state

```
001 <BinaryCollaboration nameID="BS3">
002  <Role name="Seller" nameID="Role1" />
003  <Role name="Customer" nameID="Role2"/>
004  <Start toBusinessState="Request Quote" nameID="SS1" />
005  <BusinessTransactionActivity name="Request Quote"
006   nameID="BTA1" fromRole="Customer"
007   toRole="Seller" businessTransaction="Request Quote"/>
008  <BusinessTransactionActivity name="Register Customer"
009   nameID="BTA7" fromRole="Customer"
010   toRole="Seller" businessTransaction="Register Customer"/>
011  <BusinessTransactionActivity name="Order Products"
012   nameID="BTA2" fromRole="Customer"
013   toRole="Seller" businessTransaction="Order Products"/>
014  <Success nameID="FS5" fromBusinessState="Order Products"
015   conditionGuard="Success" />
016  <Failure nameID="FS6" fromBusinessState="Order Products"
017   conditionGuard="Failure" />
018  <Transition nameID="T9" fromBusinessState="Request Quote"
019   toBusinessState="Test Customer Information"/>
020  <Transition nameID="T10"
021   fromBusinessState="Test Customer Information"
022    toBusinessState="Register Customer"conditionGuard="Failure"/>
023  <Transition nameID="T11"
024   fromBusinessState="Test Customer Information"
025    toBusinessState="Order Products" conditionGuard="Success" />
026  <Transition nameID="T12" fromBusinessState="Register Customer"
027    toBusinessState="Order Products"/>
028  <Decision nameID="D1" name="Test Customer Information">
029   <ConditionExpression expressionLanguage="DocumentEnvelopeNotation"
030    expression="CustomerInformation.confirm" />
031  </Decision>
032 </BinaryCollaboration>
```

(b) BPSS without *merge* state



(c) UMM with *merge* state

```
001 <BinaryCollaboration nameID="BS3">
002  <Role name="Seller" nameID="Role1" />
003  <Role name="Customer" nameID="Role2"/>
004  <Start toBusinessState="Request Quote" nameID="SS1" />
005  <BusinessTransactionActivity name="Request Quote"
006   nameID="BTA1" fromRole="Customer"
007   toRole="Seller" businessTransaction="Request Quote"/>
008  <BusinessTransactionActivity name="Register Customer"
009   nameID="BTA7" fromRole="Customer"
010   toRole="Seller" businessTransaction="Register Customer"/>
011  <BusinessTransactionActivity name="Order Products"
012   nameID="BTA2" fromRole="Customer"
013   toRole="Seller" businessTransaction="Order Products"/>
014  <Success nameID="FS5" fromBusinessState="Order Products"
015   conditionGuard="Success" />
016  <Failure nameID="FS6" fromBusinessState="Order Products"
017   conditionGuard="Failure" />
018  <Transition nameID="T9" fromBusinessState="Request Quote"
019   toBusinessState="Test Customer Information"/>
020  <Transition nameID="T10"
021   fromBusinessState="Test Customer Information"
022    toBusinessState="Register Customer"conditionGuard="Failure"/>
023  <Transition nameID="T11"
024   fromBusinessState="Test Customer Information"
025    toBusinessState="Simple Merge" conditionGuard="Success" />
026  <Transition nameID="T12" fromBusinessState="Register Customer"
027    toBusinessState="Simple Merge"/>
028  <Transition nameID="T30" fromBusinessState="Simple Merge"
029   toBusinessState="Order Products"/>
030  <Join nameID="J3" name="Simple Merge" waitForAll="false" />
031  <Decision nameID="D1" name="Test Customer Information">
032   <ConditionExpression expressionLanguage="DocumentEnvelopeNotation"
033    expression="CustomerInformation.confirm" />
034  </Decision>
035 </BinaryCollaboration>
```

(d) BPSS with *merge* state

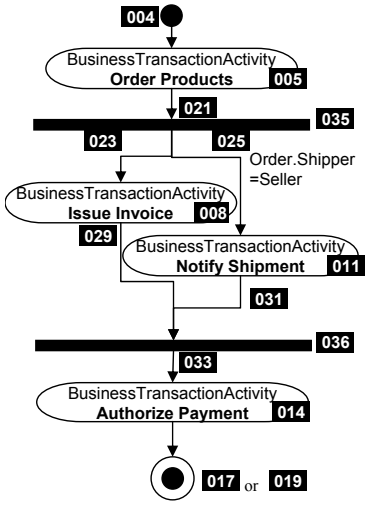
Fig. 5. An example of an exclusive choice and a simple merge pattern.

3.2 Advanced branching and synchronization patterns

In this subsection we examine more advanced patterns for branching and merge. This category includes *multi choice, synchronizing merge, multi merge*, and *discriminator*.

Multi choice and synchronizing merge. After a multi choice pattern several execution paths are chosen from many alternative threads based on a decision. For example, after *ordering products*, the seller usually initiates both the *issue invoice* transaction and the *notify shipment* transaction. Both must be completed in order to *authorize payment*. However, notify shipment makes only sense if the seller ships the products. If the customer collects the products this transaction is not necessary. Therefore, its execution is based on the party accomplishing the shipment. UMM supports this pattern by placing guards on the outgoing transitions from a *conditional fork*. In Fig.6a the *transition* from a *fork* pseudo state to *activity* state, *notify shipment*, is guarded by the decision on whether *the shipper is the seller*or not. In BPSS the transitions (line 025 in Fig.6b) from the *fork* element (line 011) with condition expressions (line 027) realize this pattern. A *fork* pseudo state may have several guarded outgoing transitions. All decisions must be evaluated before the first business transaction preceding the multi choice is executed. The order of evaluating these condition expressions is not important.

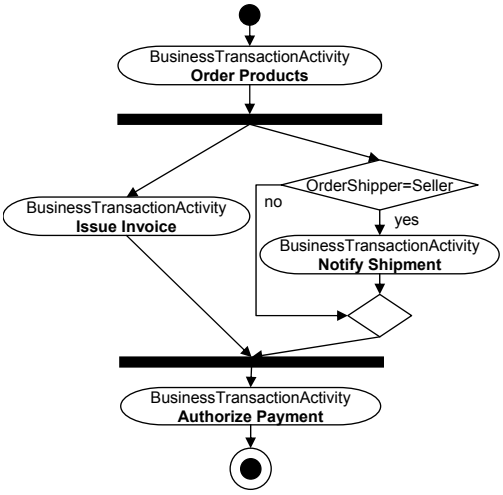
A synchronizing merge, an antithesis of the multiple choice, converges into one continuing activity. UMM realizes this pattern in exactly the same way as the *synchronization* pattern. By definition, the synchronization pseudo state does not wait for threads that have not been started. Since BPSS does not support this pattern directly, we need a work around to realize this pattern. This work around uses a *join* element whose attribute *wait for all* is *true* like in the case of the synchronization pattern (line 036 in Fig.6b). However the *join* element cannot be executed since *wait for all* attribute indicates that the *join* element must wait for all incoming threads to finish. Hence, the *fork* element (line 035) includes an attribute *time to perform*. After the specified time is exceeded, all the not executed transactions are skipped and the collaboration continues from the corresponding join. Although a *time to perform* attribute makes this pattern possible, this realization wastes time. Moreover, it is dangerous since we can ignore not only pruned threads by guarded transitions but also *binary transaction activities* that must be executed. For example, we assume the customer has a responsibility of collecting the products in Fig.6. Even if the invoice is issued in one hour, the *authorize payment* should wait for two days. More dangerous is the case of not issuing the invoice for two days after *ordering products*, because the customer should *authorize payment* anyway. For avoiding this problem, UML 2.0 recommends a *decision* node instead of a guarded transition like Fig.6c and Fig.6d. A circle with a cross in Fig.6d is a *flow final* node. This node is supported by UML 2.0 and means the termination of only a thread. The representation of Fig.6c can be directly transformed to BPSS. If a *type* of a *fork* is *xor* and the cor-



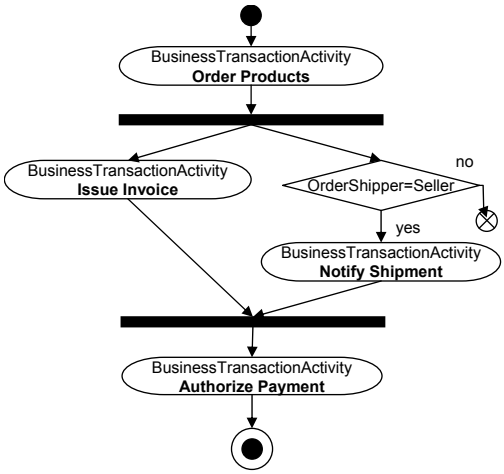
(a) UMM

```
001 <BinaryCollaboration nameID="BC2">
002  <Role name="Seller" nameID="Role1" />
003  <Role name="Customer" nameID="Role2"/>
004  <Start toBusinessState="Order Products" nameID="SS2" />
005  <BusinessTransactionActivity name="Order Products" nameID="BTA2"
006   fromRole="Customer" toRole="Seller"
007   businessTransaction="Order Products"/>
008  <BusinessTransactionActivity name="Issue Invoice" nameID="BTA8"
009   fromRole="Seller" toRole="Customer"
010   businessTransaction="Inssue Invoice"/>
011  <BusinessTransactionActivity name="Notify Shipment" nameID="BTA6"
012   fromRole="Seller" toRole="Customer"
013   businessTransaction="Notify Shipment"/>
014  <BusinessTransactionActivity name="Authorize Payment"nameID="BTA3"
015   fromRole="Customer" toRole="Seller"
016   businessTransaction="Authorize Payment"/>
017  <Success nameID="FS20" fromBusinessState="Authorize Payment"
018   conditionGuard="Success" />
019  <Failure nameID="FS21" fromBusinessState="Authorize Payment"
020   conditionGuard="Failure" />
021  <Transition nameID="T31" fromBusinessState="Order Products"
022   toBusinessState="Multi Chice"/>
023  <Transition nameID="T32" fromBusinessState="Multi Chice"
024   toBusinessState="Issue Invoice"/>
025  <Transition nameID="T33" fromBusinessState="Multi Chice"
026   toBusinessState="Notify Shipment" conditionGuard="Success">
027    <ConditionExpression expressionLanguage="OCL"
028     expression="Order.Shipper = Seller" /> </Transition>
029  <Transition nameID="T34" fromBusinessState="Issue Invoice"
030   toBusinessState="Synchronize Merge"/>
031  <Transition nameID="T35" fromBusinessState="Notify Shipment"
032   toBusinessState="Synchronize Merge"/>
033  <Transition nameID="T36" fromBusinessState="Synchronize Merge"
034   toBusinessState="Authorize Payment"/>
035  <Fork name="Multi Choice"nameID="F3"type="OR"timeToPerform="P2D"/>
036  <Join name="Synchronize Merge" nameID="J3" waitForAll="true" />
037  </BinaryCollaboration>
```

(b) BPSS



(c) UML using a *decision* state

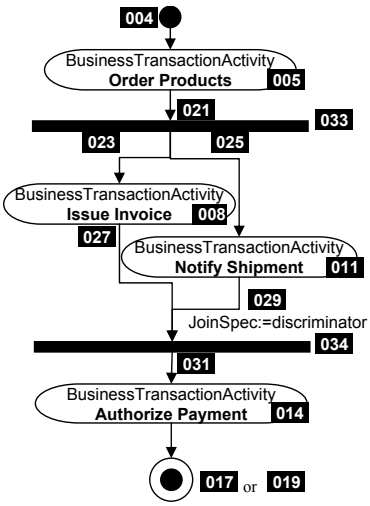


(d) UML using a *flow final* node

Fig. 6. An example of a multi choice and a synchronizing merge pattern.

responding *join* element is not reached in *time to perform*, a timeout exception is generated.

Discriminator. A discriminator pattern is similar to the synchronization pattern since multiple threads converge to one thread and the following thread is executed only once. However, the continuing activity starts after the first preceding thread finishes. We consider an example similar to the one used for explaining multi choice and a synchronizing merge. The seller is always responsible for the shipment. The customer authorizes the payment either if the invoice is issued or if the seller notifies the shipment. UMM does not support this pattern, since there is no semantically equivalent pseudo state. In UML 2.0, a join specification might be assigned to a join node. This join specification decides when the continuing thread is started (see Fig.7a). BPSS realizes the pattern by a *join* element (line 034 in Fig.7b), whose *wait for all*'s value is *false*.



(a) UMM

```
001 <BinaryCollaboration nameID="BC4">
002  <Role name="Seller" nameID="Role1" />
003  <Role name="Customer" nameID="Role2"/>
004  <Start toBusinessState="Order Products"nameID="SS2" />
005  <BusinessTransactionActivity name="Order Products"nameID="BTA2"
006   fromRole="Customer"  toRole="Seller"
007   businessTransaction="Order Products"/>
008  <BusinessTransactionActivity name="Issue Invoice" nameID="BTA8"
009   fromRole="Seller"  toRole="Customer"
010   businessTransaction="Inssue Invoice"/>
011  <BusinessTransactionActivity name="Notify Shipment"nameID="BTA6"
012   fromRole="Seller"  toRole="Customer"
013   businessTransaction="Notify Shipment"/>
014  <BusinessTransactionActivity name="Authorize Payment"
015   nameID="BTA3" fromRole="Customer"
016    toRole="Seller" businessTransaction="Authorize Payment"/>
017  <Success nameID="FS7" fromBusinessState="Authorize Payment"
018   conditionGuard="Success" />
019  <Failure nameID="FS8" fromBusinessState="Authorize Payment"
020   conditionGuard="AnyProtocolFailure" />
021  <Transition nameID="T13" fromBusinessState="Order Products"
022   toBusinessState="Parallel Split"/>
023  <Transition nameID="T14" fromBusinessState="Parallel Split"
024   toBusinessState="Issue Invoice"/>
025  <Transition nameID="T15" fromBusinessState="Parallel Split"
026   toBusinessState="Notify Shipment"/>
027  <Transition nameID="T16" fromBusinessState="Issue Invoice"
028   toBusinessState="Discriminator"/>
029  <Transition nameID="T17" fromBusinessState="Notify Shipment"
030   toBusinessState="Discriminator"/>
031  <Transition nameID="T18" fromBusinessState="Discriminator"
032   toBusinessState="Authorize Payment"/>
033  <Fork name="Parallel Split" nameID="F2" />
034  <Join name="Discriminator" nameID="J2" waitForAll="false" />
035 </BinaryCollaboration>
```

(b) BPSS

Fig. 7. An example of a discriminator pattern.

Multi merge. After a multi merge pattern multiple threads are merged into one continuing activity, which is executed whenever a precedence thread reaches the multi merge pattern. UMM does not support it now, since the current UMM is based on UML 1.4, which forces forks and joins to be well-nested. However, in UML 2.0 this constraint disappears and a *merge* node following a *fork* node realizes this pattern. BPSS does not support the multi merge pattern either. Future versions of BPSS need improvements to support this pattern. We recommend to adopt the concept of UML 2.0. In UML 2.0 there exist both a pseudo state merge node - depicted as a diamond - and a synchronization node - depicted as bar. Currently, in BPSS there exists a single element *join* to realize simple merge, synchronization and discriminator. We prefer a new element similar to the UML diamond to realize simple merge and multi merge.

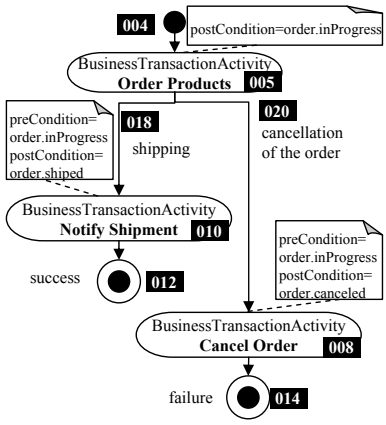
3.3 State-based patterns

If the execution of one activity depends on the state of another activity, the pattern is categorized into the class of state-based patterns. *Deferred choice, interleaved parallel routing* and *milestone* are categorized into this class of patterns.

Deferred choice. A deferred choice pattern selects only one continuing activity from several candidates like an exclusive choice, but the decision is implicit. For example, the seller ships the products after *order products* unless the customer *cancels the products* before the shipment. In this example, we do not know whether the next *business transaction activity* is *notify shipment* or *cancel order* before one of them starts. In UMM the deferred choice is realized by events, e.g. the shipment of the products or the decision to cancel the order. Furthermore, the post condition of each activity in the deferred choice must be in contradiction to the pre conditions of the other activities in the deferred choice. In BPSS, a corresponding element for the deferred choice exists. BPSS realizes this pattern using the element *fork* whose *type* attribute is *xor* (line 022 in Fig.8b). As soon as a succeeding *business transaction activity* starts, the others become unavailable.

Interleaved parallel routing. An interleaved parallel routing pattern defines the execution of a set of activities in an arbitrary order. Each activity of the set is executed once. At a given point in time only one activity is executed. The execution order is fixed at run time. Neither UMM nor BPSS supports the interleaved parallel routing pattern.

Milestone. A milestone pattern is the start of an activity depends on the state of one or more other activities. For example, *order products* in Fig.3 can be only initiated after *register customer* and before *unregistered customer* in Fig.9. UMM uses tagged values, *pre condition* and *post condition* for this pattern. The *pre condition* and *post condition* are transformed as attributes of a *business transaction activity* in BPSS. Before initializing *order products*, (line 008 in Fig.3b),



(a) UMM

```
001 <BinaryCollaboration nameID="BC6" name="Deferred Choice">
002  <Role name="Seller" nameID="Role1" />
003  <Role name="Customer" nameID="Role2"/>
004  <Start toBusinessState="Order Products" nameID="SS2" />
005  <BusinessTransactionActivity name="Order Products"
006   nameID="BTA2" fromRole="Customer"
007   toRole="Seller" businessTransaction="Order Products" />
008  <BusinessTransactionActivity name="Cancel Order" nameID="BTA10"
009   fromRole="Customer" toRole="Seller"businessTransaction="Cancel Order"/>
010  <BusinessTransactionActivity name="Notify Shipment" nameID="BTA6"
011   fromRole="Seller"toRole="Customer"businessTransaction="Notify Shipment"/>
012  <Success nameID="FS11" fromBusinessState="Notify Shipment"
013   conditionGuard="Success" />
014  <Failure nameID="FS12" fromBusinessState="Cancle Order"
015   conditionGuard="Failure" />
016  <Transition nameID="T21" fromBusinessState="Order Products"
017   toBusinessState="Deferred Choice"/>
018  <Transition nameID="T22" fromBusinessState="Deferred Choice"
019   toBusinessState="Cancel Order"/>
020  <Transition nameID="T23" fromBusinessState="Deferred Choice"
021   toBusinessState="Notify Shipment"/>
022  <Fork name="Deferred Choice" nameID="F4" type="XOR" />
023 </BinaryCollaboration>
```

(b) BPSS

Fig. 8. An example of a deferred choice pattern.

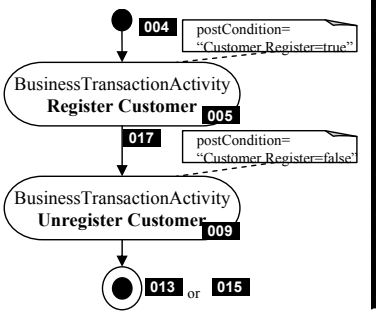
its *pre condition* "Customer.Register=true" is checked (line 010 in Fig.3b). This value is modified in *register customer* and *unregister customer* of another *binary collaboration* using *post condition* (line 008 and 011 in Fig.9b).

3.4 Structural patterns

In this subsection we examine the structural patterns *arbitrary cycle* and *implicit termination*. Preventing these patterns improves the readability and makes the interpretation easier. However, neither UMM nor BPSS imposes structural restrictions on the model.

Arbitrary cycles. A structural cycle pattern is a loop that has only one entry and only one exit point. The *while* and *for* statements of C language are examples of structural cycles. Contrary to a structural cycle, an arbitrary cycle pattern has no restriction on the number of entry and exit point. Some arbitrary cycles are constructed by the combination of multiple *decisions*, *xor-typed forks* and *transitions*. In this case UMM and BPSS are able to realize the arbitrary cycle. However, arbitrary cycles might involve forks and joins as well. Since each fork has a corresponding join, transitions can not cross the boundary of the fork-join-block, UMM does not fully support the arbitrary cycle pattern. Since BPSS does not include a similar well-formedness rule it fully supports the arbitrary cycle.

Fig.10 offers an example of an arbitrary cycle pattern. An undesirable situation, such as a lack of raw material, a natural disaster, or a strike, can prevent the seller from shipping the exact number of products in time. In this case, the seller should inform the customer about the situation and *request for purchase*

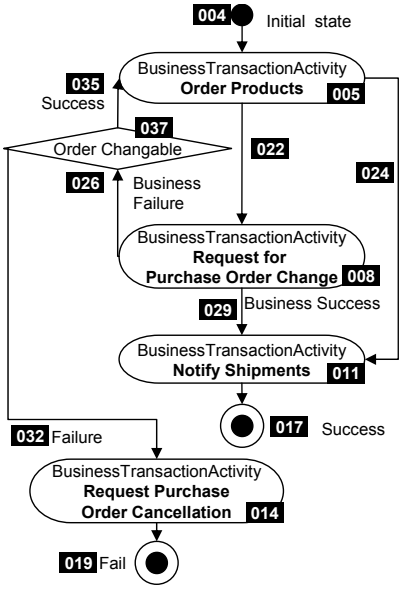


(a) UMM

```
001 <BinaryCollaboration nameID="BC30" >
002  <Role name="Seller" nameID="Role1" />
003  <Role name="Customer" nameID="Role2"/>
004  <Start toBusinessState="Registering" nameID="SS30" />
005  <BusinessTransactionActivity name="Register Customer"
006   nameID="BTA30" fromRole="Customer" toRole="Seller"
007   businessTransaction="Register Customer"isConcurrent="true"
008   postCondition="Customer.Register=true"/>
009  <BusinessTransactionActivity name="Unregister Customer"
010   nameID="BTA31" fromRole="Customer" toRole="Seller"
011   postCondition="Customer.Register=false"
012   businessTransaction="Unregister Customer" />
013  <Success nameID="FS1" fromBusinessState="Order Products"
014   conditionGuard="Success" />
015  <Failure nameID="FS2" fromBusinessState="Order Products"
016   conditionGuard="Failure" />
017  <Transition nameID="T1" fromBusinessState="Request Quote"/>
018 </BinaryCollaboration>
```

(b) BPSS

Fig. 9. An example of a milestone pattern.



(a) UMM

```
001 <BinaryCollaboration nameID="BC5">
002  <Role name="Seller" nameID="Role1" />
003  <Role name="Customer" nameID="Role2"/>
004  <Start toBusinessState="Order Products" nameID="SS2" />
005  <BusinessTransactionActivity name="Order Products"
006   nameID="BTA2" fromRole="Customer"
007   toRole="Seller" businessTransaction="Order Products"/>
008  <BusinessTransactionActivity name="Request Purchase Order Change"
009   nameID="BTA15" fromRole="Seller" toRole="Customer"
010   businessTransaction="RequestPurchase OrderChange"/>
011  <BusinessTransactionActivity name="Notify Shipment"
012   nameID="BTA6" fromRole="Seller"
013    toRole="Customer" businessTransaction="Notify Shipment"/>
014  <BusinessTransactionActivity name="Request Purchase Order Cancellation"
015   nameID="BTA16" fromRole="Customer" toRole="Seller"
016   businessTransaction="RequestPurchaseOrderCancellation"/>
017  <Success nameID="FS3" fromBusinessState="Notify Shipment"
018   conditionGuard="Success" />
019  <Failure nameID="FS16"
020   fromBusinessState="Request Purchase Order Cancellation"
021   conditionGuard="Failure" />
022  <Transition nameID="T19" fromBusinessState="Order Products"
023   toBusinessState="Request Purchase Order Change"/>
024  <Transition nameID="T37" fromBusinessState="Order Products"
025   toBusinessState="Notify Shipment" />
026  <Transition nameID="T38"
027   fromBusinessState="Request Purchase Order Change"
028   toBusinessState="OrderChange" conditionGuard="BusinessFailure" />
029  <Transition nameID="T39"
030   fromBusinessState="Request Purchase Order Change"
031   toBusinessState="Notify Shipment" conditionGuard="BusinessSuccess" />
032  <Transition nameID="T40" fromBusinessState="OrderChange"
033    toBusinessState="Request Purchase Order Cancellation"
034    conditionGuard="Failure" />
035  <Transition nameID="T41" fromBusinessState="OrderChange"
036    toBusinessState="Order Products" conditionGuard="Success" />
037 <Decision name="OrderChange" nameID="D6">
038   <ConditionExpression expressionLanguage="DocumentEnvelopeNotation"
039     expression="OrderChangable" />
040    </Decision>
041 </BinaryCollaboration>
```

(b) BPSS

Fig. 10. An example of an arbitrary cycle pattern.

order change. The choice between *request for purchase order change* and *notify shipment* is realized by a deferred choice pattern. If the customer accepts the request, the seller ships the products. Otherwise, the customer decides whether he changes the order or cancels the order. If he decides to change the order, the *binary collaboration* restarts from *order products*. Since the cycle (*order products* → *request for purchase order change* → *order changable* → *order products*) has three exit points, this example includes an arbitrary cycle pattern.

Implicit termination. Both UMM and BPSS have an explicit final state (a *final* state of UMM and a *success* and a *failure* element of BPSS) but they also support a special kind of implicit termination pattern. An implicit termination pattern means a situation where there is no activity to be done even if a final state is not reached and at the same time the workflow is not in deadlock. For example, *binary collaboration* has an attribute, *time to perform* (line 001 in Fig.3b). That is, the *binary collaboration* is forced to terminate in two days although the final state has not been reached.

3.5 Patterns involving multiple instance

We examine patterns involving multiple instances in this subsection. These patterns are categorized by the ability to launch multiple instances of activities and synchronization among the instances. The patterns are *multiple instances without synchronization*, *multiple instances with a priori design time knowledge*, *multiple instances with a priori run time knowledge*, and *multiple instances without a priori run time knowledge*.

Multiple instances with a priori design time knowledge and *multiple instances with a priori run time knowledge* restrict the number of instances at design time and run time, respectively. In contrast, *multiple instances without synchronization* and *multiple instances without a priori run time knowledge* have no limitation on the number of instances. While UML supports *multiple instances with a priori design time knowledge* and *multiple instances with a priori run time knowledge* [10], UMM does not use this feature.

Multiple instances without a priori run time knowledge can manage the relationship among instances such as synchronization differently from *multiple instances without synchronization*. BPSS supports only the *multiple instances without synchronization* by assigning *true* to a *business transaction activity*'s attribute *is concurrent* (line 007 in Fig.3b). Since the activity diagram of UML does not support this pattern, *is concurrent* is expressed as a tag value of an activity in UMM.

3.6 Cancelation patterns

Both *cancel activity* pattern and *cancel case* pattern are cancelation patterns. By performing activities of the cancelation patterns other activities are withdrawn.

Cancel activity. A cancel activity pattern cancels an enabled activity. UML supports through transition with triggers. However, UMM does not use this feature and BPSS does not support this pattern directly, either. This pattern can be supported by a deferred choice pattern [5]. However, a *business transaction activity* is composed of other activities in the *business transaction* view and each activity of *business transaction* corresponds to other workflow in the company. Moreover, we cannot interrupt the *business transaction activity* even using a deferred choice. In Fig. 11 if *define planning schedule* fails, we do not need to *authorize payment* any more. However, even if *define planning schedule* fails before the customer sends the *authorize payment envelope*, BPSS does not provide a pattern to cancel the authorize payment transaction. The only workaround is a milestone pattern using pre- and post-conditions.

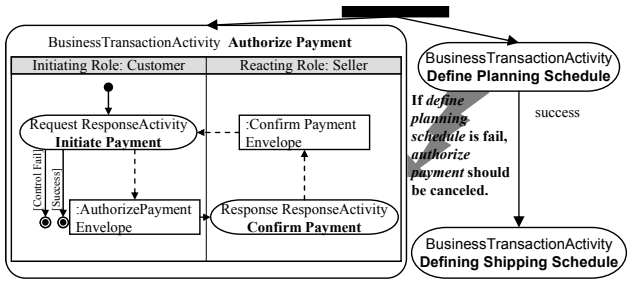


Fig. 11. A problem of a cancel activity pattern.

Cancel case. A cancel case patterns terminates a *binary collaboration*. In UMM (BPSS), as soon as a *final* state (a *success* or a *failure* element) is reached, the *binary collaboration* is terminated. Even if other *business transaction activities* remain, they do not open any more. In this case, a timeout exception can be generated. Therefore, although a *binary collaboration* has several *final* states, they should be mutually exclusive.

4 Conclusion

In this paper, we analyze the expression power of UMM and BPSS by workflow patterns. We summarize the analysis in Table 1. A '+' and a '-' in a cell of the table refer to direct support and no support, respectively. Even if a pattern is rated as a '-', we can realize the pattern partially by the combination of other patterns [5]. A 't' means that the pattern is realized as a tag value in UMM not by a feature of an activity graph in UML. A '2' indicates that the pattern can be supported if UMM will adapt UML 2.0.

According to the presentations of each pattern in both UMM and BPSS, we are able to derive the transformation rules listed below. This list covers all known

	UMM v. 12	BPSS v. 1.10
Sequence	+	+
Parallel Split	+	+
Synchronization	+	+
Exclusive Choice	+	+
Simple Merge	+	+
Multi Choice	+	+
Synchronizing Merge	+	-
Multi Merge	2	-
Discriminator	2	+
Arbitrary Cycles	-	+
Implicit Termination	t	+
MI without Synchronization	t	+
MI with a Priori Design Time Knowledge	-	-
MI with a Priori Runtime Knowledge	-	-
MI without a Priori Runtime Knowledge	-	-
Deferred Choice	+	+
Interleaved Parallel Routing	-	-
Milestone	t	+
Cancel Activity	-	-
Cancel Case	+	+

Table 1. Comparison of UMM and BPSS

rules necessary to transform a UMM business collaboration protocol to a BPSS binary collaboration. Our future work includes representation of these rules in a formal syntax and an implementation of the mapping from UMM business processes represented in XMI [12] to BPSS.

- An *initial* state and an *activity* state are transformed to a *start* element and a *business transaction activity*, respectively.
- A *final* state is transformed to a *success* or a *failure* element. We need some convention for deciding whether *final* state is transformed to a *success* or a *failure* element.
- A *transition* of UMM is transformed to a *transition* of BPSS. However, if the transition leads to a *final* state, the transition becomes an attribute of a *success* or a *failure* element. The same exception applies to transitions from the initial state.
- A *synchronization* state with multiple incoming transitions is transformed to a *fork* element whose *type* attribute is *or*. If some outgoing transitions are guarded, the *fork* element needs a *time to perform* attribute.
- A *synchronization* state with multiple outgoing transitions is transformed to a *join* element whose *wait for all* attribute is *true*.
- A *decision* state with multiple incoming transitions is transformed to a *decision* element.
- A *decision* state with multiple outgoing transitions is transformed to a *join* element whose *wait for all* attribute is *false*.

– If an *business transaction activity* has multiple outgoing transitions and each transition is triggered by event, a *fork* element is inserted between *business transaction activity* and its outgoing transitions. The *type* of the *fork* is *xor* if triggering events are not concurrent.
– Tagged values, *is concurrent* and *time to perform*, are transformed to the same named attributes of *business transaction activity* and *binary collaboration*, respectively.

Our research resulted in the need for improvements for both UMM and BPSS. Some patterns like an implicit termination and an arbitrary cycle are supported by both. For reasons of readability these patterns should be avoided. Therefore, we need to study well-formedness rules for prohibiting these patterns.

In spite of their similarity, the transformation between UMM and BPSS is not straightforward since the sets of workflow patterns that the two languages support are not exactly the same. For example, while BPSS realizes a discriminator pattern, a BPSS instance derived from UMM will never use this pattern since UMM cannot support the pattern yet. If a new UMM version adopts UML 2.0, the gap between UMM model and BPSS is reduced since the new UMM supports more workflow patterns including a discriminator.

We can also apply the workflow pattern to transformations between other heterogeneous business process modeling languages such as BPEL4WS and XPDL.

References

1. Ferris, C., Farrell, J.: What are web services. Communications of the ACM **46** (2003) 31 – 35
2. W3C Web Services Architecture Working Group: Web services architecture requirements; W3C working draft (2002) http://www.w3.org/TR/2002/WD-wsa-reqs-20021114.
3. Leymann, F., Roller, D.: Modeling Business Processes with BPEL4WS. In: Proceedings of the 1st GI Workshop XML4BPM. (2004) 7–24
4. Peltz, C.: Web services orchestration and choreography. IEEE Computer **36** (2003) 46 – 52
5. Van der Aalst, A., Hofsteded, A.T., Kiepuszewski, B., Barros, A.: Workflow patterns. Distributed and Parallel Databases **14** (2003) 5 – 51
6. Wohed, P., van der Aalst, W.M., Dumas, M., ter Hofstede, A.H.: Analysis of web services composition languages: The case of bpel4ws. Lecture notes in computer science **2813** (2003) 200–215
7. UN/CEFACT TMG: UN/CEFACT modeling methodology, revision 12 (2003) http://www.untmg.org.
8. UN/CEFACT TMG: ebXML business process specification version 1.10 (2003)
9. ISO: Open-edi Reference Model. ISO/IEC JTC 1/SC30 ISO Standard 14662. ISO (1995)
10. Dumas, M., ter Hofstede, A.H.: UML activity diagrams as a workflow specification language. Lecture notes in computer science **2185** (2001) 76 – 90
11. WfMC: Workflow management coalition terminology & glossary. Technical Report WFMC-TC-1011, WfMC (1999)
12. Jeckle, M.: OMG's XML Metadata Interchange XMI. In: Proceedings of the 1st GI Workshop XML4BPM. (2004) 25–42