

Model-Based Development of Executable Business Processes for Web Services

Reiko Heckel^{1,2} and Hendrik Voigt²

¹ Faculty of Computer Science, University of Dortmund, Germany

² Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn, Germany
{reiko,hvoigt}@upb.de

Abstract. In order to implement business processes, the composition of simpler services provided by different independent participants requires a high degree of standardization and flexibility. For this purpose, platform-independent XML-based languages like the *Business Process Execution Language for Web Services (BPEL4WS)* are suitable. XML documents are in fact human readable, but in general they are hard to produce and to understand by business experts which are, however, most qualified for defining business processes. We present a *model-based development method* based on an intuitive and adequate modelling notation, an automatic transformation of process models to their XML-based encoding, and techniques to analyze processes. In this context the *Unified Modelling Language (UML)* as standard notation for modelling software, *graph transformation* as meta language for defining model transformations, and a semantic interpretation of process models in terms of *Communicating Sequential Processes (CSP)* are used.

1 Introduction

A *Web service* is a software component that can be dynamically discovered, linked, and invoked by its clients via XML-based protocols. This software-oriented definition of the term can be contrasted with a business-oriented view, considering a Web service as a *business process*, implemented by the composition (and coordination) of simpler services provided by other businesses.

The composition of services provided by different independent parties, at both development time or runtime, requires a high degree of standardization and flexibility. Therefore, rather than hard-coding business processes in platform-specific programming languages which depend on certain compilers and runtime environments, platform-independent XML-based languages like the *Business Process Execution Language for Web Services (BPEL4WS)* [1] are advocated. Such processes in XML representation can, at least in theory, be adapted at runtime, exchanged between different services, and executed on different standardized interpreters.

However, even if XML documents are text files and therefore, in principle, human readable, the XML representation of a processes is hard to produce and to

understand even by an experienced programmer. It resembles, in a linear form, the abstract syntax tree of a program without providing the usual front-end notation. What is more, in their role as business processes, Web service processes should be defined by business experts which are not typically programmers.

Therefore a *model-based development method* is required based on

1. an intuitive and adequate *modelling notation*, to allow precise specifications of processes at the conceptual level
2. an *automatic transformation of process models* to their XML-based encoding, to avoid the costly and error-prone task of deriving the implementation manually
3. *techniques to analyze processes* at the model level for syntactic and semantic properties, to avoid “debugging” the XML code

These problems and requirements are prototypical for a wide variety of languages and platforms, in the Web services domain and elsewhere. Therefore, instead of defining and implementing languages, transformations, and analysis tools for every single problem, reusable solutions are required.

In this paper, we will present an approach based on the combination of three such solutions: the *Unified Modeling Language (UML)* [8] as standard notation for modelling software, *graph transformation* [12] as meta language for defining model transformations, and a semantic interpretation of process models in terms of *Communicating Sequential Processes (CSP)* [6] which offers a language to express semantic consistency properties and tool support for analysis.

In the following section we will discuss the complementary roles of these techniques in general and outline their application to the model-based development of Web service processes.

2 Defining a Model-Based Development Method

An outline of our approach can best be given in terms of the triangle in Fig. 1, whose vertices are the languages by which processes may be represented, and whose edges represent uni- or bi-directional transformations between these representations.

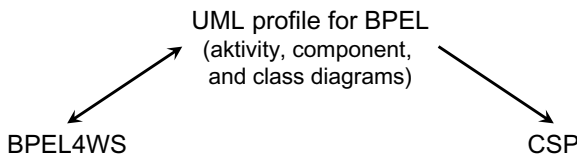


Fig. 1. Outline of the approach: languages and transformations

The UML, as a general-purpose modelling language, provides a rich set of concepts to model all kinds a software system. However, to address the more specific aspects of a particular application domain or implementation platform,

the language needs to be specialized and extended. For this purpose, the standard [8] foresees the extension mechanism of *profiles*, a compromise between desirable flexibility of the language and necessary compatibility with existing tools. We shall use a profile to tailor, in particular, UML activity diagrams to the specification of BPEL4WS processes. In conjunction with these, class diagrams and component diagrams shall be used to describe, respectively, data types and software architecture relevant to the process.

Besides the concrete visual representation of a UML model, an abstract representation is required to capture its *semantically relevant* structure. This *abstract syntax* of UML models is defined by means of a *meta model*, i.e., a class diagram with well-formedness constraints expressed in the Object Constraint Language (OCL) [7]. The meta model specifies the collection of all legal *abstract syntax graphs*—its instances—each of which represents a legal model. This graph-based internal representation of UML models, which is typical of visual languages in general, shall be needed when defining the transformation of models.

These transformations do, in fact, represent the core features of a model-based development approach. In our case, they occur in two places: the transformation into BPEL4WS, the implementation language, and into CSP, the language for behavioral analysis. In many situations, two-way transformations are required, e.g., to support a *round-trip engineering* approach, where not only models are transformed into implementations (forward engineering), but also vice versa (reverse engineering), thus allowing incremental changes at both levels.

Our tool for describing (potentially bi-directional) transformations between models and other (typically textual) languages is the approach of *pair grammars* [10], i.e., a coupling of context-free grammars which allows to generate a sentence in the target language after parsing a given sentence in the source language. Since at least one of the languages will be a graphical one, context-free *graph grammars* [5] shall be employed which generalize context free grammars on strings by describing languages whose sentences are graphs.

For a mapping specification to be manageable and reusable, a modular approach is important which is structured in terms of the fundamental concepts of the domain. In this case, whenever a concept is added or modified, the corresponding transformation rules can be exchanged without affecting the rest of the mapping specification. For the domain of executable business processes, or workflow models, a corresponding concept analysis has produced an established list of *workflow patterns* [16], a subset of which is supported by UML activity diagrams. In fact, it turns out that these workflow patterns, interpreted over either activity diagrams or BPEL4WS processes, provide us with the pairs of context-free rules making up the pair grammar that specifies the translation.

The model-based analysis of processes represents the final ingredient of our approach. Depending on the representation on which the analysis is performed, we distinguish between *syntactic* and *semantic* analysis. The former is often restricted to the evaluation of well-formedness constraints on (the abstract syntax graph of) the model which reveal inconsistencies in structural dependencies and

typing. However, syntactic analysis also includes the manual review of models based on semi-formal error patterns, a method that is quite successful in revealing behavioral problems and that may be the only method available in the presence of semi-formal models.

Formal analysis of behavioral properties, however, can hardly be done at the syntactic level, but requires a mapping of models into a semantic domain providing (1) a representation of the behavior to be analyzed, (2) means to express the desired properties, and (3) techniques and tools to check if these properties hold [4]. We have chosen the semantic domain of CSP [6] for this purpose, whose refinement relations are the basis for expressing properties over processes while tool support is provided by the FDR2 model checker [11].

The paper is structured according to the triangle in Fig. 1. The following section is devoted to the modelling of processes in the UML. Then, Sections 4 and 5 deal, respectively, with the mapping between UML and BPEL4WS, and the analysis of processes, including the mapping to CSP. Section 6 concludes the paper and summarizes the results.

3 Modelling BPEL4WS Processes in the UML

In this section, we describe how UML diagrams can be used to model Web service processes. We put special emphasis on the behavioral aspect given by BPEL4WS process interactions. As already discussed in [15], visual and more high-level modelling languages like UML have important advantages in comparison to low-level XML-based specification languages. Among others, they allow a better abstraction from implementation details and are therefore better understandable.

In particular UML use case-, component-, class-, and activity diagrams are suitable for modelling Web service processes in the context of business processes. In the following, we will demonstrate this by a sample model of an online shop.

Use case diagrams describe the business segment of our example. As shown in Fig. 2, the use case itself symbolizes the business process, in this case the service provided by the shop. The participants in the use case represent the roles of the partners that interact with the process. In the example, a buyer, a delivery service, and an invoice service interact with the online shop service.

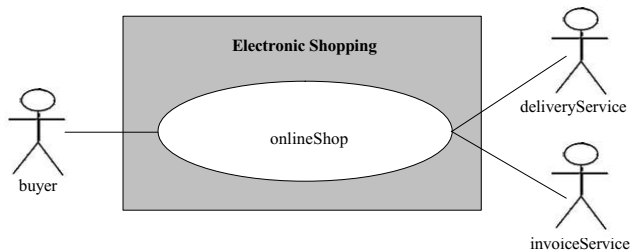


Fig. 2. Use Case Diagram

Component diagrams are used to refine the dependencies described in the use case diagram. In doing so, the component symbol is used for depicting a Web service. Both, the participants of the business process and the business process itself are modelled as Web services (see Fig. 3). In order to establish possible points of interactions, *port types* are added to the diagram as interfaces (the circular symbols, PT is used as abbreviation for port type). If a Web service *provides* a port type, the interface is connected to the component symbol by a solid line. If a Web service *requires* a port type, this is modelled by a dashed arrow (a UML *dependency*) to the corresponding interface.

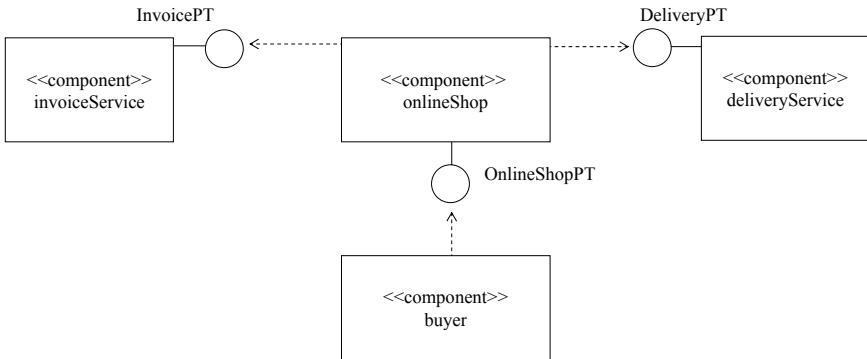


Fig. 3. Component Diagram

For example in Fig. 3, the online shop service provides exactly one port type `OnlineShopPT` and requires the port types `DeliveryPT` and `InvoicePT` for processing the corresponding data.

Class diagrams are used to provide further details of the different port types by defining their operations and involved parameters. In Fig. 4, the three port types from the component diagram are refined. In this simplified example, each port type provides only one operation which is mainly used to submit and receive the processing data to and from the partners. Likewise, the necessary messages and parameter types could be modelled by the class diagram.

Protocols and business processes for Web services are modelled with activity diagrams. Besides control-flow elements (decision, fork, join, etc.), the activity diagrams contain the necessary basic activities for interacting with the partner services. The activities are stereotyped like *receive*, *reply*, or *invoke* depending on their function as defined in the BPEL4WS specification [1]. A triplet consisting of partner, port type and operation follows the stereotype. This triplet assigns one of the available port type operations to each activity.

Fig. 5 shows the process of the online shop service. The first part is used to accept an order provided by the buyer through `OnlineShopPT`. After the data has been received, the online shop concurrently invokes an invoice and a delivery service with the required data. In order to simplify the example, we have

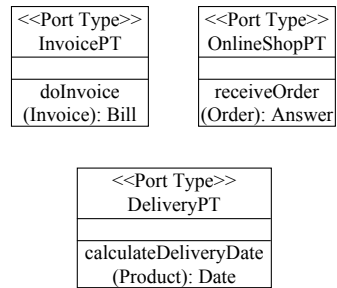


Fig. 4. Class Diagram

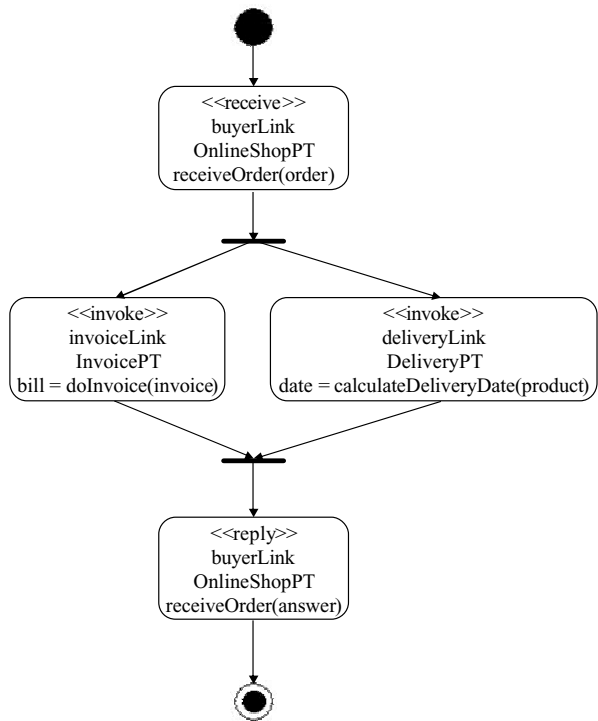


Fig. 5. Activity Diagram: Online Shop

omitted the details of assigning variables to establish the connection between the received order and further activities. In addition, the decision process for generating the answer for the buyer is hidden. The last activity in the row, which can be identified by the stereotype *receive* in Fig. 5, sends the corresponding answer to the buyer.

The protocols of the partners are modelled with activity diagrams, too. This step is essential for further analysis, because inconsistent behavior between the participants has to be discovered to ensure a correct execution of the interactions.

Therefore, the following three activity diagrams in Fig. 6 represent the processes on the Buyer, Invoice Service and Delivery Service part, respectively. In this example, a quick comparison of their activities with the activities of the Online Shop shows that the processes are behaviorally compatible.

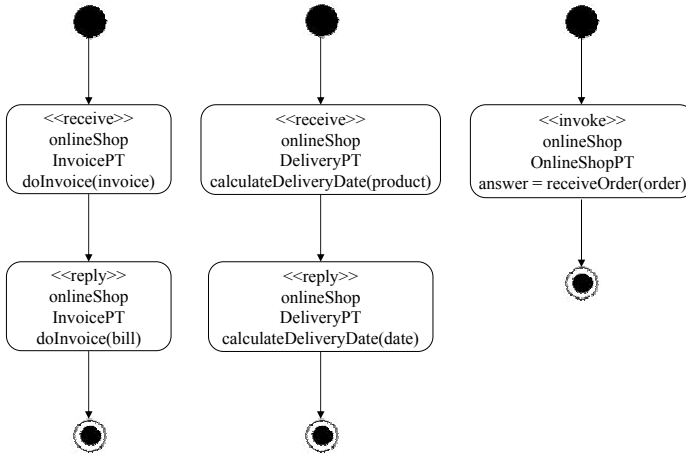


Fig. 6. Activity Diagram: Buyer, Invoice Service and Delivery Service

For more complex interactions, the consistency of all involved partner processes cannot be checked as easily as in this example. For this reason, Section 5 discusses more advanced methods to maintain certain consistency properties. Before, however, the transformation of process models into BPEL4WS processes by means of pair grammars is described.

4 Mappings between UML and BPEL4WS

A formal definition of the translation between UML models and an executable process language is important, not only to automate the translation, but also to define and ensure consistency at the model level (cf. Section 3).

Translations between string and graph representations of programs and data may be formally defined by means of *pair grammars* [10]. A pair grammar consists of two context-free grammars describing, respectively, the source and the target language, together with a correspondence between their rules and non-terminals. In this way, it defines a correspondence between source and target sentences which represents a mapping between the two languages. By virtue of their symmetric nature, pair grammars allow the definition of bi-directional mappings between UML activity diagrams and BPEL4WS processes.

To support the translation of *graphical* languages, pair grammars are based on context-free *graph* grammars, i.e., formal grammars similar to ordinary context-free grammars, except that the language defined is a set of graphs rather

than a set of strings. Context-free grammars, in fact, form a subclass of context-free graph grammars. In this paper, an informal introduction to the most important concepts of pair grammars is given. For an exhaustive and formal treatment we refer to [10].

Our presentation is based on *edge replacement graph grammars*, one of the simplest forms of context-free graph grammars which form, in their generalized form of *hyperedge replacement (HR) graph grammars* [5], one of the two major approaches in the literature. Here, context-freeness means that the left-hand side of a rule is given by a single edge (or hyperedge, i.e., an edge attached to an arbitrary number of vertices) representing a nonterminal which is replaced by the graph that forms the right-hand side of the rule. The obvious alternative consists in replacing a nonterminal node, leading to the family of *node-replacement* approaches [2].

4.1 Pair Grammars

First, we introduce the basic notions of graphs and context-free graph grammars. A *graph* consists of vertices and edges such that each edge has a source and a target vertex in the graph, respectively. In accordance with [5], in our graphs *labelled edges* carry the relevant information, while nodes just represent the points where the edges are attached.

Definition 1 (edge-labelled graphs). *Let C be a fixed set of edge labels. A (directed edge-labelled) graph $G = \langle G_V, G_E, \text{src}^G, \text{tar}^G, \text{lab} \rangle$ over C has a set of vertices G_V , a set of edges G_E , two functions $\text{src}^G : G_E \rightarrow G_V$ and $\text{tar}^G : G_E \rightarrow G_V$ associating to each edge its source and target vertex, and a labelling function $\text{lab} : G_E \rightarrow C$ associating with every edge its label.*

Definition 2 (edge replacement (ER) graph grammars). *Let $N \subseteq C$ be a set of nonterminal labels. A production rule over N is a pair $p = A \xrightarrow{x,y} R$ where $A \in N$ and R is a graph over C with distinguished vertices $x, y \in R_V$.*

An edge replacement (ER) graph grammar $\mathcal{G} = \langle C, N, P, S \rangle$ consists of the sets of labels and nonterminals introduced above, a set P of productions over N , and a start symbol $S \in N$.

Edge replacement graph grammars subsume context free grammars by representing strings of terminal and nonterminal symbols as chains of correspondingly labelled edges. Every application of a rule derived in this way from a context-free grammar rule takes out one nonterminal edge and replace it with a path, gluing the source vertex of the path to the (former) source vertex of the edge, and analogously for the target.

In the general case of ER grammar rules, an edge in graph is replaced by a graph with two attachment points x, y (that we think of as “source” and “target” vertices) which are glued to the source and target vertex of the replaced edge, respectively.

Full hyperedge replacement grammars generalize this by allowing an arbitrary number of attachment points for the graphs to be inserted and, consequently, for the edge to be replaced. We have limited ourselves to ER graph

grammars here for simplicity, and because they are sufficient to illustrate the concept of grammar-based translation of graphical languages.

The definition of pair grammars, originally formulated for a simple kind of node-replacement graph grammars, has been tailored to our purposes.

Definition 3 (pair grammar). *A pair grammar is a quadruple $\mathcal{Q} = \langle C, N, PP, S \rangle$ where C and N are sets of labels and nonterminals as before, $S \in N$ is a start symbol, and PP is a finite set of triples (p_1, h, p_2) , where*

1. $p_1 = A_1 \xrightarrow{x_1, y_1} R_1$ and $p_2 = A_2 \xrightarrow{x_2, y_2} R_2$ are ER rules over C and N as above such that $A_1 = A_2$,
2. h is a nonterminal edge pairing of R_1 and R_2 , i.e., a bijection between their nonterminally labelled edges such that $e_1 h e_2$ implies $\text{label}(e_1) = \text{label}(e_2)$.

The language defined by a pair grammar \mathcal{Q} consists of ordered pairs of graphs from the left and right language, respectively, of \mathcal{Q} . The pair grammar defines how these graph pairs may be generated in parallel from the same start symbol. At each intermediate stage in the generation we have a pair of graphs, each containing some nonterminal nodes, and a correspondence between these nonterminal nodes. At each rewriting, a corresponding pair of nonterminal nodes, one in each graph, is rewritten according to a rule of the pair grammar, and a new correspondence is set up between nonterminal nodes in the resulting graphs using the nonterminal pairing of the grammar rule.

4.2 UML–BPEL4WS Mapping

Let us illustrate the notions introduced so far by means of a mapping between UML activity diagrams and BPEL4WS processes, specified by the pair grammar whose rules are shown in Fig. 7 through 10.


Left production rule of the pair grammar		Right production rule of the pair grammar	
Act ::=	$\langle \text{process} \rangle$ $AI:Act$ $\langle / \text{process} \rangle$	Act ::=	

Fig. 7. Pattern 0: Start and End

Production rules of this pair grammar combine ordinary context-free grammar rules for BPEL4WS processes and truly graphical rules for (a subset of) UML activity diagrams. In order to regard these diagrams as graphs that can be generated by edge rewriting, activities shapes as well as fork / join bars are interpreted as terminal edges. Nodes (presented as little circles) are introduced whenever two activities are connected by a transition. As the only nonterminal label, *Act* stands for an arbitrary diagram with one entry and one exit transition. Thus our sets of labels are defined by $C = \{activity, bar, Act\}$ and $N = \{Act\}$, such that *Act* is the only possible start symbol.

Vertices have no labels, but we distinguish attachment points as filled circles with name x or y . Nonterminal edges (i.e., with label *Act*) are denoted as boxes connected to their source and pointing to their target vertex.

The bijection h between nonterminal edges of the right-hand sides of left and right rules is given by identical names for corresponding edges. According to Def. 3, corresponding edges as well as left hand sides must be of the same label. For example, in the rule of Fig. 8, symbols $A1 : Act$ through $A1 : Act$ in the upper BPEL4WS production correspond to the edges with the same name in the lower UML production rule.

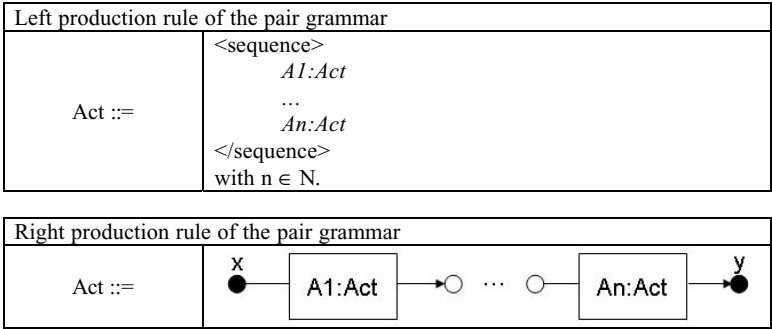


Fig. 8. Pattern 1: Sequence

It is interesting to note how the definition of the individual rules is inspired by the workflow patterns [16]. For example, in the case of *sequence* or *parallel split and synchronization*, the interpretation of each pattern in both BPWL4WS and activity diagrams yields the right hand sides of the two corresponding rules.

The mapping specified by the pair grammar shall be applied to our online shop example. Recall Fig. 5 specifying a business process of the shop. In order to execute this process it shall be translated into the BPEL4WS. The operational idea is to start parsing the sentence of the source language (UML) and to generate the sentence in the target language (BPEL4WS) along the resulting derivation tree.

Parsing the source language. As result of parsing the activity diagram, its syntactic structure is represented by the derivation tree in Fig. 11. In the first step of the construction of this diagram, the basic activities (rectangles with rounded corners) are replaced with nonterminal edges. It follows the detection of structured activities based on the patterns of sequence (the outermost box $a6 : Act$) and parallelism (box $a5 : Act$).

Thus, the parsing yields a hierarchical decomposition of the diagram which can be seen as a tree with the innermost boxes (terminal edges) as leafs and the outermost box (the start symbol) as root. This structure resembles derivation trees of ordinary context-free grammars. In particular, it abstracts from the

ordering of independent derivation steps, i.e., boxes that are not nested in one another, like a_1, a_2, a_3, a_4 in Figure 11, can be processed in any order or in parallel.

When the graph representing the activity diagram is reduced to the start symbol, the parsing process is finished successfully.

Generating the target language. The next step is to generate the corresponding BPEL4WS sentence, invoking the rules of the right grammar following the structure of the derivation tree. This second phase begins with the start symbol, i.e., the tree is computed bottom up, but evaluated top down.

After the first step, one derives the box with the *iprocess* tag (see the left production rule in Figure 7). Since the correspondence between the rules of the two languages is fixed by the pairing, there is no other option but to start with this pattern, which represented the last step in the generation of the tree.

Next, the structure of the process is refined by introducing sequence and flow instructions (compare Fig. 8 and 9). First, the sequence is inserted because this is the next outermost structure. Subsequently, the parallel part is generated. Alternatively, one could have substituted the first activity in the flow of the sequence, because this step is independent of the generation of the parallel activities.

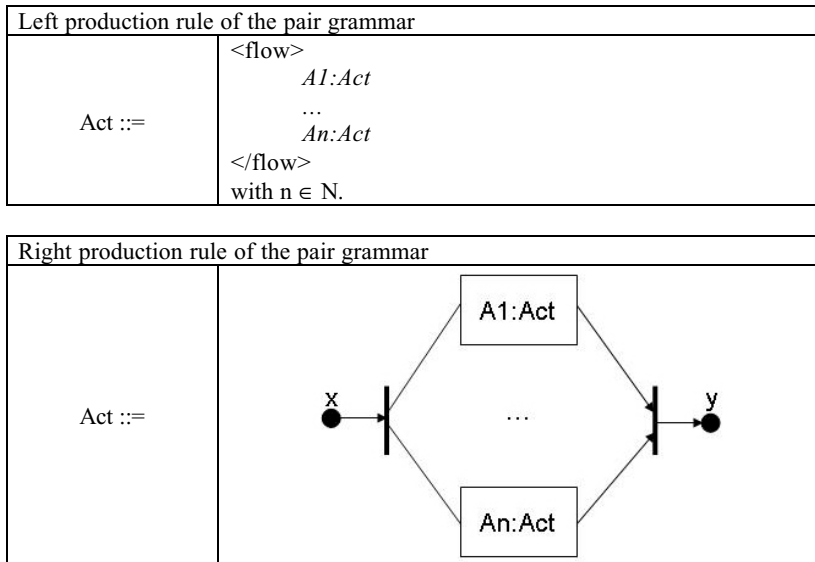


Fig. 9. Pattern 2: Parallel Split and Synchronization

We emphasize the role of the correspondence between the nonterminals on the right sides. Symbols $A1 : Act$ to $An : Act$ are associated with the edges $A1 : Act$ to $An : Act$, so that the “content” of edge A_i can determine the replacement of the corresponding nonterminal.

At this stage, the considered BPEL-process looks like this:

```
<process>
  <sequence>
    A1:Act
    <flow>
      A2:Act
      A3:Act
    </flow>
    A4:Act
  </sequence>
</process>
```

In the last step, all nonterminals are substituted by terminals. As shown in Figure 10, all variables of the right production rule must be replaced with the corresponding terminals for partner, port type, and so on. After generating all basic activities, the transformation is completed. The resulting BPEL4WS process is listed below.

```
<process>
  <sequence>
    <receive name="receiveOrder"
      partnerLink="ns:buyerLink"
      portType="ns:onlineShopPT"
      operation="ns:receiveOrder"
      variable="order"
      createInstance="yes"/>
    <flow>
      <invoke name="invokeBank"
        partnerLink="ns:invoiceLink"
        portType="ns:InvoicePT"
        operation="ns:doInvoice"
        inputVariable="invoice"
        outputVariable="bill"/>
      <invoke name="invokeDeliverer"
        partnerLink="ns:deliveryLink"
        portType="ns:DeliveryPT"
        operation="ns:calculateDeliveryDate"
        inputVariable="product"
        outputVariable="date"/>
    </flow>
    <receive name="replyOrder"
      partnerLink="ns:buyerLink"
      portType="ns:onlineShopPT"
      operation="ns:receiveOrder" variable="answer"/>
  </sequence>
</process>
```

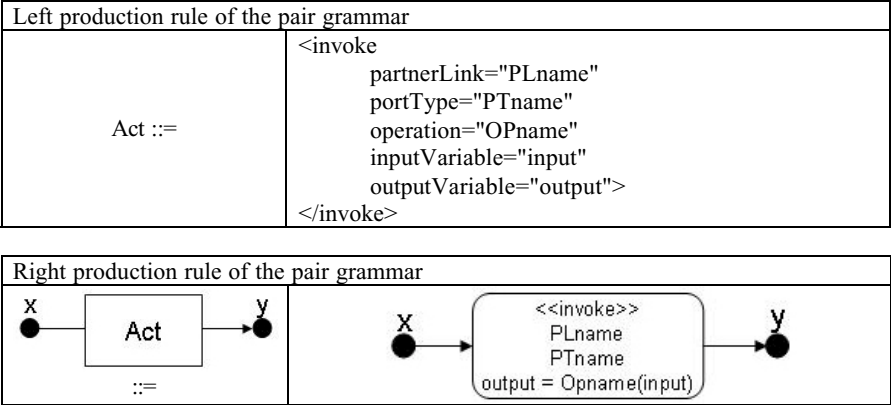


Fig. 10. Pattern: Invoke

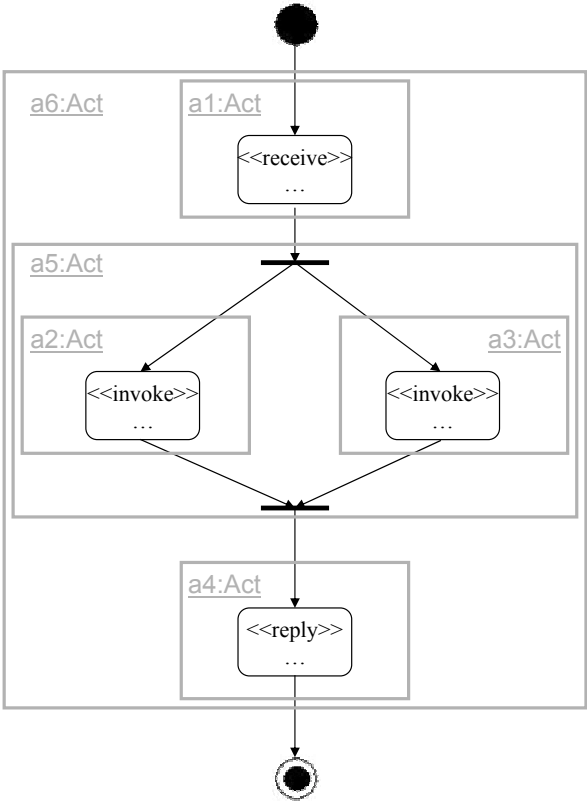


Fig. 11. Decomposition of the activity diagram

4.3 Properties of the Mapping

A basic requirement for an automated translation, even before semantic considerations are made, is that there should be a unique sentence of the target language generated for every given sentence of the source language. Uniqueness may be relaxed by some notion of semantic equivalence, but in many cases such notion is not readily available, or very hard to verify.

A purely syntactic criterion is the notion of *(un)-ambiguity* of the pair grammar which is based on corresponding notions for the underlying grammars.

Definition 4 (ambiguity). *An ER graph grammar \mathcal{G} is ambiguous iff there exists a graph G in the generated language which has two distinct derivation trees.*

That means, a grammar is *unambiguous* if every sentence can be parsed in essentially one way, up to the ordering of independent steps which are abstracted from in the parse tree.

It is difficult to prove unambiguity in general, but there exists a simpler sufficient condition based on the idea of critical pairs in rewriting: Consider the grammar as a reduction system, applying its rules from right to left in order to reduce the given graph to the start symbol. Now, unambiguity is ensured if there is never a true conflict between the application of two reduction rules. A conflict is evident in an overlapping of the left-hand sides of two reduction rules (i.e., the right-hand sides of two production rules) if (cf. [9])

- they intersect in anything else than attachment vertices, and
- both rules are applicable to reduce the graph formed by the overlapping

The second condition is violated if one of the reduction rules attempts to delete a vertex that is connected to an edge originating from the other rule. In this case, the resulting structure is no longer a graph since the edge misses its source or target vertex. Hence, in a critical pair, an overlap in a non-attachment node entails that all edges connected to this node in both rules are also in the intersection. In turn, an overlap of an edge obviously entails an overlap of its source and target node.

That means, the intersection includes all nodes and edges of both rules indirectly reachable from a non-attachment node or edge in the intersection. Since the right-hand sides of our rules are connected, this implies that the overlap is complete whenever a non-attachment node or an edge is involved.

For a pair grammar we consider unambiguity for both directions of the translation. Generally, it requires that the source grammar is unambiguous, so the parse tree is uniquely determined, and that for every source production there is exactly one target production, so the tree uniquely induces a derivation in the target. Depending on who is source and who is target, this results in the notions of left and right unambiguity.

Definition 5 (unambiguity of pair grammars). *A pair grammar \mathcal{Q} is left (right) unambiguous if the left (right) grammar of \mathcal{Q} is unambiguous, and \mathcal{Q} contains no two distinct rules with identical left (right) rules.*

\mathcal{Q} is unambiguous if it is both left and right unambiguous.

The pair grammar presented in Section 4.2 is unambiguous. However, there exists another possibility for describing sequences in BPEL4WS, besides the one shown in Fig. 8. The alternative shown in Fig. 12 uses the *flow* construct, specifying the desired temporal dependencies by means of links between activities.

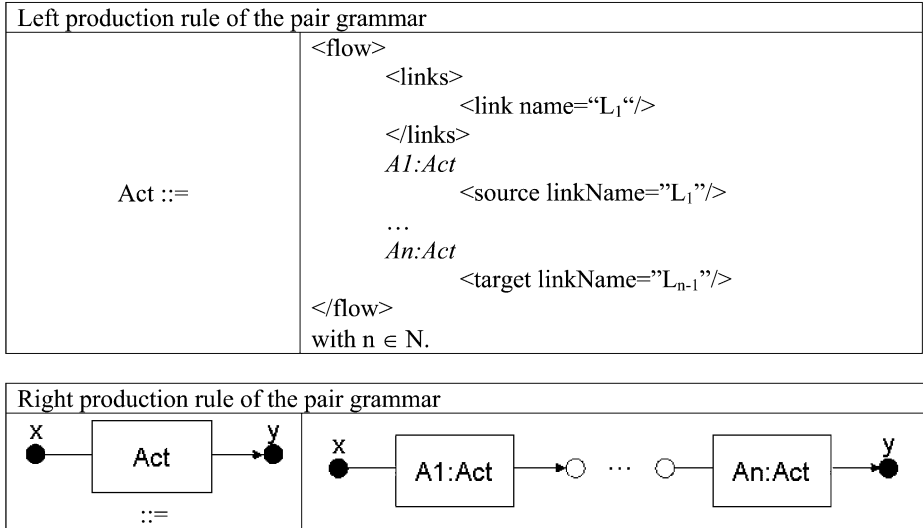


Fig. 12. Pattern 1: Sequence (via BPEL4WS flow construct)

The resulting pair grammar is no longer right unambiguous because there are two rule pairs sharing the same right rule. Indeed, since we have two choices to implement a sequence, the result of the translation is no longer unique. Hence, for right-to-left mappings, one of the two rules should be disregarded.

On the other hand, the alternative sequence rule is useful for the left-to-right from BPEL4WS to UML because, when reverse engineering a process we cannot assume a certain style of implementation, but have to handle the full spectrum of language constructs. Fortunately, the extended pair grammar is still left unambiguous.

This example shows that, while in potentially bi-directional, it could be necessary to tailor the mapping description to one or the other direction in order to achieve unambiguity.

5 Model-Based Analysis

Model-based development tends to create a variety of artifacts that describe the system to be built from different viewpoints and at different levels of abstraction. This allows developers to concentrate on the concern of present interest, reducing complexity by hiding other not so relevant concerns, but it also creates consistency problems between the different descriptions.

Besides this general, domain and language-independent cause for consistency problems, there are more specific reasons resulting from the choice of a specific development method, target language, or application domain. Typical for Web services is, for example, the consistency between descriptions of *required* and *provided* services, which need to be matched before services can be composed. This includes, e.g., the compatibility of their signatures to ensure type safety, and of their interaction protocols to avoid deadlocks.

Other consistency problems are inherited from the target language of development, in our case BPEL4WS, which entails particular restrictions for processes formulated in that language. Type checking rules for BPEL4WS require, for example, that all operations used in activities of the process must be declared in the appropriate port types. This induces a dependency between the class diagram containing the interfaces from which the port types are derived and the activity diagram where the process is modelled. Thus, the precise notion of consistency that needs to be applied at the model level depends on both the restrictions at the implementation level and the mapping of models to implementations as described in Section 4.

In this section, we are dealing with model-based analysis of consistency problems. Both subject and result of the analysis are given at the model level because, obviously, it would limit the applicability of a method if developers were forced to work on different representations of processes, e.g., to eliminate a fault directly in an XML-based business process language, or to analyze a process in a process algebra. This, again, emphasizes the need for automated mappings between different representations.

For dealing with consistency in UML-based development processes, we apply a general methodology for specifying and analyzing consistency [4]. In short, the methodology consists of the following steps.

1. Consistency problems must be identified in a given UML-based development process, documented and categorized into
 - problems of *syntactic* (e.g., structural) nature that can be formulated and solved at the level of models;
 - problems of *semantic* (e.g., behavioral) nature that require a separate semantic representation.
2. For each semantic consistency problem, a suitable semantic domain must be chosen and a semantic mapping of models into this domain must be designed.
3. For both syntactic and semantic problems, consistency conditions must be stated as constraints, either over the abstract syntax or the semantic representation of models.

In the following two subsections we consider, in turn, the syntactic and the semantic case.

5.1 Syntactic Analysis

In our sample process, for illustration we identify syntactic consistency problems between *component diagrams* and *class diagrams*, as well as between *activity diagrams* and *class diagrams*.

For component and class diagrams a consistency problem occurs whenever an interface is used in a component diagram that is not declared in the class diagram. This consistency problem can be considered syntactic because a syntactic condition can be formulated using OCL, requiring that each interface used is declared in the class diagram.

With regard to activity diagrams and class diagrams, a similar kind of consistency problem is illustrated in Fig. 13: An activity contains references to the partner component, interface and operation which must be in a certain relation (e.g., the interface declares the operation and is implemented or used by the component playing the partner role). Considering the class diagram in Fig. 4, it is obvious, that the **OnlineShopPT** does not support a **receiveInstruction** operation.

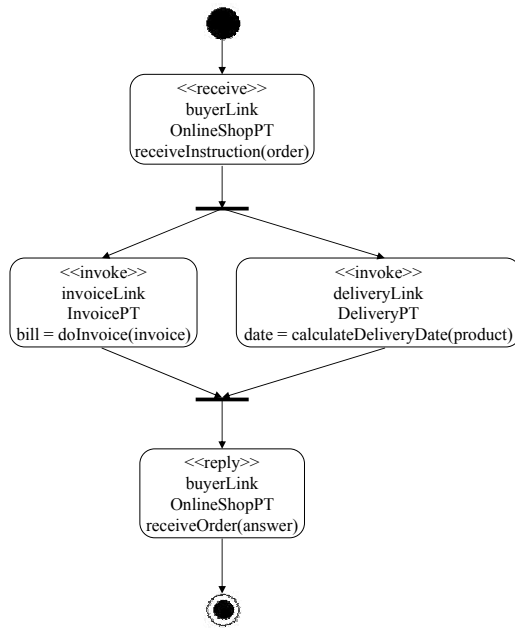


Fig. 13. Activity Diagram: Syntactic consistency problem

When modelling Web service processes, we also have to take into account language-specific consistency properties, as for instance: In a BPEL4WS process a process instance must not simultaneously enable two *receive* actions for the same partner, port type, and operation. If two receive actions for the same partner, port type, and operation are, in fact, simultaneously enabled, e.g., in two concurrent threads of the process, then a standard fault must be thrown by the process interpreter that complies with the BPEL4WS specification. In such a case, the processing of the current scope is terminated. Possible results are the invocation of compensation handlers (if defined) or the abortion of the

entire process. In any case, the chance for a successful completion of the process decreases.

In order to avoid such conflicts, we can provide sufficient static conditions at the model level, e.g., by means of a semi-formal error pattern as illustrated in Fig. 14. *ActivityA* and *ActivityB* are place holders for sub-processes of arbitrary structure. Such a pattern, which must not occur in a process, can serve as a guideline for the developer or as a documentation for a formal analysis of this property, if available.

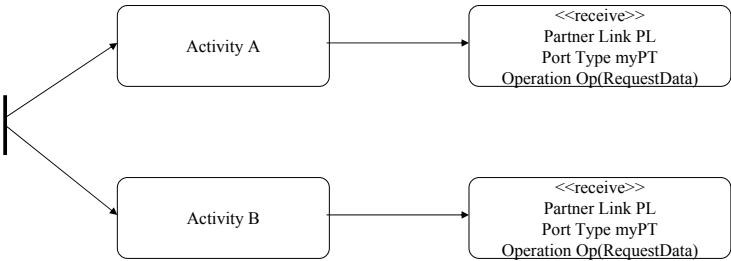


Fig. 14. Conflict potentials in parallel sections

Another example of an error pattern is shown in Fig. 15. It reflects, at the level of models, the fact that BPEL4WS requires control flows that are based on links to be acyclic.



Fig. 15. Cycles

5.2 Semantical Analysis

Next, we deal with consistency conditions that are formulated and analyzed in a separate semantic domain. In our example, different interaction protocols of the participants are combined and conclusions about their compatibility are given. Thus, we focus on the behavioral aspect. In particular, if the business processes have a complex control flow, the concrete or abstract syntax of models is not suitable for such problems. Therefore we choose the process algebra CSP [6] as semantic domain for analysis.

Since we cannot assume that developers are familiar with CSP, and in order to avoid mistakes in the translation, an automated mapping of models into CSP is required, as well as a mapping of analysis results back into UML models.

Translation from UML to CSP. In principle, the translation from UML activity diagrams into the semantic domain CSP is based on the same concepts as the translation shown in Section 4. However, in this case a pair grammar is not fully satisfactory, because one has to generate complex *Pre* and *Post* processes for managing events accurately in CSP. These are interconnected by an *Environment*. In order to control these processes, the concept of a *Global Scheduler* is adopted. Furthermore, the *Control* process guarantees the correct termination of the whole system. Both basic and structured activities (i.e., nodes like split and join that describe the control flow) are coded as separate CSP processes denoted as *Activity processes*. Thus Activity processes emulate the proper control flow as well as the sending and receiving of events. In order to execute the Activity processes independently, they are combined by interleaving.

Processes may be composed by operators which require synchronization on some events. Each component must be willing to participate in a given event before the whole can make the transition. In this regard, suitable communication and synchronization *alphabets* consisting of events must be defined. The composition of processes is itself a process, allowing a hierarchical description of a system. The process structure can be represented as a tree. The root node represents the process as a whole (cf. *System_Control*). According to the number of sub-components of the node branches are added. Fig. 16 visualizes the process structure, whereby rectangles indicate the parallel composition symbol including the alphabet in question and ovals display CSP processes. The several atomic Pre, Post and Activity processes are hidden, because their occurrence strongly depends on the underlying example.

In the following, we focus on the generation of the Activity processes. As already mentioned, the concept of pair grammars is suitable for demonstrating the basic idea. Now the grammar for UML activity diagrams is paired with the one for CSP processes. This is shown in Fig. 17.

On each left-hand side of the considered rules the type of the expected non-terminal is used. In this way, these nonterminals are paired. For the grammar for UML activity diagrams we refer to Section 4, because it has already been discussed. The right-hand sides of the rules for the CSP grammar include terminals like *if* and *else*, and nonterminals like *ActivityA₁*, whereby *Activity* indicates the type and *A₁* indicates the variable. We have chosen this alternative representation for a better readability of the CSP process. Below we explain the *Activity process* in short without deepening the language CSP too much.

ActivityA₁ is the name of the considered *Activity process*. The first event in the flow activates this process (compare *act_ActivityA₁*). Afterwards *t* reads out the channel *transition_x*, whereas the value 1 respectively 0 indicates a positive respectively negative precondition. If *t* has in fact the value 1, then the corresponding transition process is instructed to reset itself. After this the transition for the following Activity process is set and *ActivityA₁* is disabled and ends with a recursive invocation on itself, so it is available in the next step of the Global Scheduler.

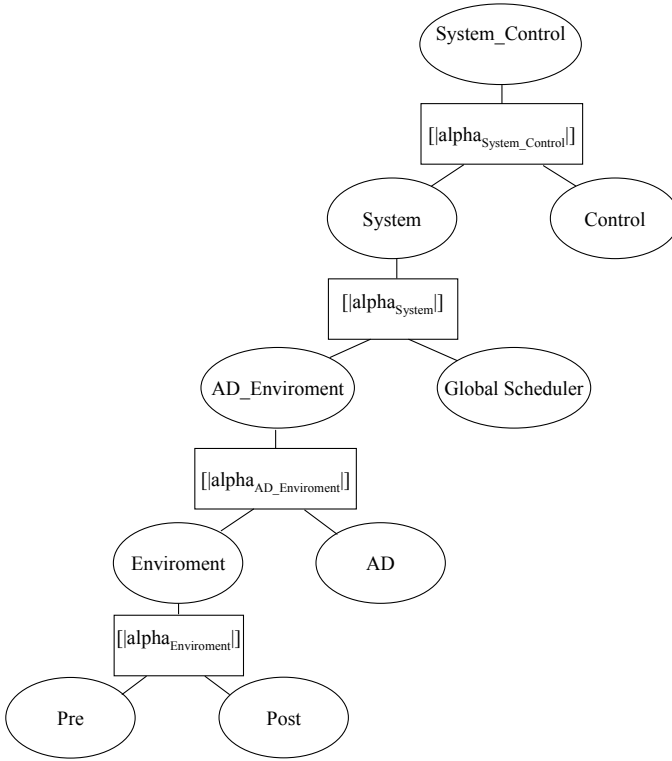


Fig. 16. Combing processes in CSP

For UML activity diagrams and CSP, we do not require a bi-directional translation, because we do not assume that business processes are formulated as CSP processes. For analyzing CSP processes we use the model checker FDR2 [11]. As results of a check one obtains a trace, which the process is, or is not, willing to execute. These traces can be transformed into a sequence diagram. Hence, a developer is able to work solely at the model level. Moreover, the complex structure of the established CSP process is the reason for customizing the concept of pair grammars in this regard. For completely describing this translation, we would have to upgrade from pair grammars to so-called triple graph grammars [13]. Beside lifting the restriction to context-free grammars, triple graph grammars allow to store auxiliary data, accumulated during the translation, inside a third intermediate graph, which also keeps track of the relation between source and target. This feature is important to determine the alphabet of a process by collecting data on operations and partners occurring in the process. However, in this paper we stick to the pair grammar representation which is still sufficient to convey the basic ideas.

After defining a translation from UML activity diagrams into CSP, we can finally turn to the actual semantical analysis. Basically, we distinguish between

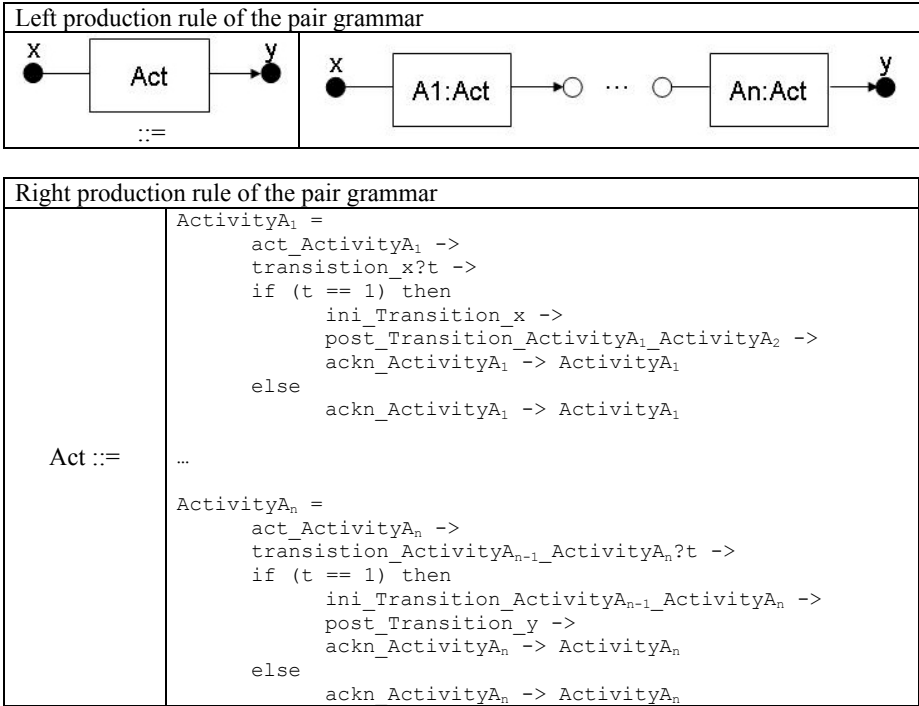


Fig. 17. Pattern 1: Sequence

classical requirements for concurrent processes and requirements for business processes.

Classical concurrency properties. At first we consider classical requirements like *deadlock* or *livelock*. Concerning the property of *deadlock freedom*, we need to provide a consistency concept for activity diagrams. At first we give a definition for deadlock.

Definition 6 (deadlock). *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.*

As an example, consider the modified activity diagram of the Buyer shown in Fig. 18 and the one of the Online Shop Service, illustrated in Fig. 5. Both processes expect a signal from each other which gives rise to a deadlock. This circumstance is independent of changes in their interfaces.

In general, whenever a deadlock occurs, processes can not be completed successfully. Hence, we take into account suitable measures to avoid such conflicts. The tool FDR2 supports the detection of deadlocks. However, due to the complex structure of the CSP process implementing a business process, CSPs definition of a deadlock, which requires that a process does not communicate at all, is not applicable here. Instead, we have to check for a livelock in order to capture the notion of a business process deadlock in CSP, see [17] for details.

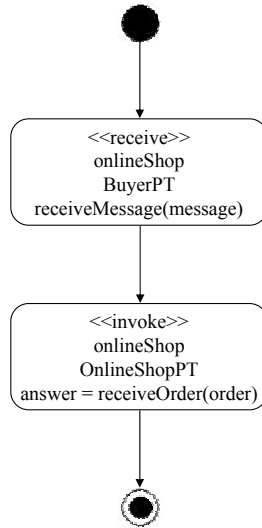


Fig. 18. A deadlock situation

Requirements for business processes. Now we turn to requirements which must be formulated depending on the context of a business process. The verification of so called security properties is based on the comparison of CSP processes to a set of traces. This set defines sequences of events and by doing so secure states are specified. In this sense, a CSP process is in fact secure if its provided traces are in the set of traces of the security property. In addition we want to check, if a concrete activity diagram covers several scenarios. These scenarios can be formalized as UML diagrams. In this context UML behavior diagrams are of special importance (compare UML sequence and statechart diagrams). In order to compare all these different diagram types, a sufficient transformation into the semantic domain CSP must be established. In [14], Stehr picks the translation of sequence and statechart diagrams out as a central theme. In this survey we have already demonstrated, how activity diagrams can be translated into CSP.

Such a scenario can be derived from different use cases.

- By modelling sequence diagrams one has the opportunity to define permitted respectively prohibited examples. This means that a developer identifies concrete scenarios, which are checked regarding a concrete business process.
- Concrete executions of existing business process instances can be monitored. By doing so sequence diagrams can be formulated by assigning exchanged messages to objects.

In both cases, one has to establish the relationship between the different message and event names, respectively, used in the diagrams. In general, sequence diagrams are much less formalized than activity diagrams. This task can only be handled by the developer himself. Agreeing on the same name space is a precondition for meaningful analysis.

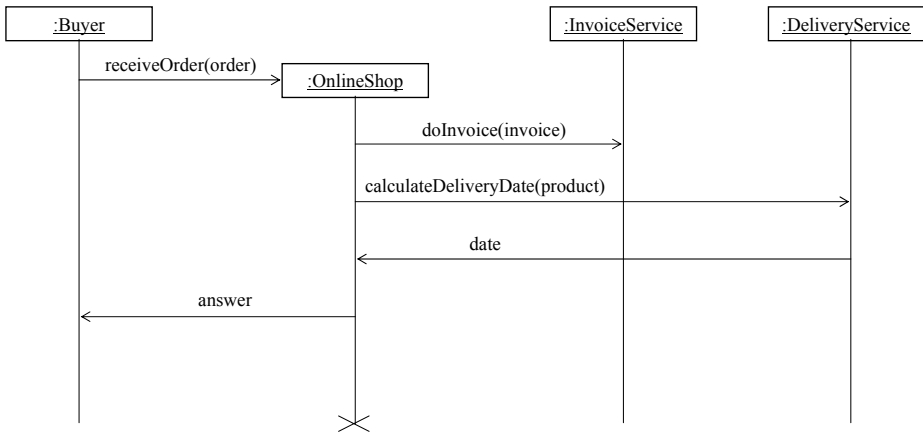


Fig. 19. Property formulated as a UML sequence diagram

Messages	Events
receiveOrder(order)	post_Event_buyerLink_OnlineShopPT_receiveOrder_Request
doInvoice(invoice)	post_Event_invoiceLink_InvoicePT_doInvoice_Request
	post_Event_invoiceLink_InvoicePT_doInvoice_Response
calculateDeliveryDate (product)	post_Event_deliveryLink_DeliveryPT_calculateDeliveryDate_Request
date	post_Event_deliveryLink_DeliveryPT_calculateDeliveryDate_Response
answer	post_Event_buyerLink_OnlineShopPT_receiveOrder_Response

Fig. 20. Assigning messages of the sequence diagram to the events of the CSP process

An example of such an assignment is shown in Fig. 20. We emphasize that the events of the CSP process are not introduced in this context, because the underlying example consists of over 1400 lines of code. For the complete example we again refer to [17].

The result of a check shows whether the trace of the sequence diagram is in the set of traces of the activity diagram. In this example, a required event does not have any correspondence in the sequence diagram. And in fact, this event must be executed in the underlying business process. Hence, the CSP processes (and thus the business processes visualized as activity diagrams) do not conform to the defined property.

Further on, complete subsystems can be compared to each other. In doing so, we want to check, if a subsystem might be replaced by another. This analysis is only based on CSP processes, which are representations of UML activity diagrams. For this purpose the FDR2 trace refinement checker is suitable.

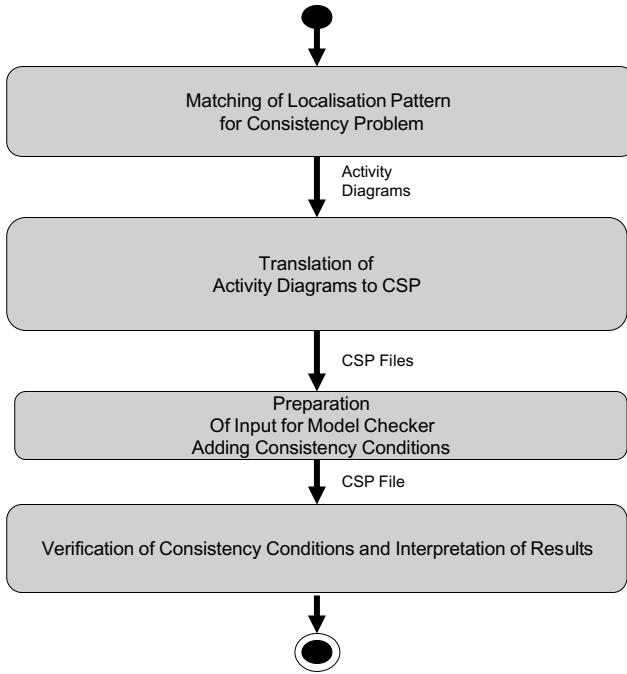


Fig. 21. A sample consistency check

Defining consistency checks. On the basis of a consistency concept, consistency checks can be defined in order to validate that a model is consistent. A consistency check must therefore validate the consistency conditions of a consistency concept. Within our approach, such a check may involve the translation of a model into a semantic domain, the verification of consistency conditions by a model checker, and an interpretation of the results.

Informally, the specification of such a consistency check can be visualized by an activity diagram extended by mechanisms for modelling object flow. In the following, we will sketch the definition of a consistency check for the consistency problem type of activity diagrams, ensuring their deadlock freedom.

In Fig. 21, the consistency check for activity diagrams is shown (with object flow visualized by arrows). Within the first activity, a UML localization pattern is used for locating and identifying, within a larger UML model, those activity diagrams relevant for the consistency check. These are then given to the translation activities. Within the translation activities, the translation to CSP is performed. Resulting CSP files are then assembled to a single file which can be handed over directly to the model checker.

This concept is implemented in the ConWork tool developed at the university of Paderborn, which allows to define flexible consistency checks based on rule-based translations of UML diagrams into CSP [3].

6 Conclusions

With the wide integration of Web services into software development, modelling of Web service processes is gaining increasing importance. In order to be beneficial, a modelling approach should take into account the characteristics of Web service processes. Currently, the Unified Modeling Language is the accepted industrial standard for modelling object-oriented systems. In this paper, we have discussed how the UML can be applied for modelling Web service processes. Furthermore we have introduced several UML models for suitable abstractions of both structure and behavior of Web service processes. Then we have focused on a bi-directional translation between UML activity diagrams and BPEL4WS. Thus we provide a framework for forward and reverse engineering based on the considered translation concept. As consistency is not established by the language definition of UML, it must be ensured by the software engineer applying UML for modelling Web service processes. In order to prove consistency conditions in regard to the UML models, we categorized possible inconsistency types into syntactical and semantical problems depending on the language that is suitable for solving these. In this context the analysis of different interaction protocols participating in a given business process is of particular interest. For this task we chose CSP as semantic domain for further analysis. On this account we provide a translation from UML into CSP and propose an approach, how results of the model checker FDR2 can be visualized as UML models. The visual modelling language UML facilitates an adequate abstraction of implementation details and supports a better understanding of consistency analysis.

Future work includes the definition of a generic development process for Web service processes and the elaboration of consistency management within this development process. For that purpose, we must automate the translation between UML activity diagrams and BPEL4WS as well as UML activity diagrams into a semantic domain such as CSP. Currently, we are developing tool support for this task based on the Consistency Workbench [3]. This tool allows the software engineer to define translations of UML models into a semantic domain and define consistency checks as workflows, like visualized in Fig. 21.

References

1. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1, May 2003.
<http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/>.
2. J. Engelfriet and G. Rozenberg. Node replacement graph grammars. In Rozenberg [12], pages 1 – 94.
3. G. Engels, R. Heckel, and J. M. Küster. The consistency workbench: A tool for consistency management in uml-based development. In *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications. 6th International Conference, San Francisco, USA*, LNCS. Springer, 2003.

4. G. Engels, J.M. Küster, L. Groenewegen, and R. Heckel. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In V. Gruhn, editor, *Proc. European Software Engineering Conference (ESEC/FSE 01)*, Vienna, Austria, volume 1301 of *LNCS*, pages 327–343. Springer Verlag, 2001.
5. A. Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1992.
6. C. Hoare. Communicating sequential processes. *Communicat. Associat. Comput. Mach.*, 21(8):666–677, 1978.
7. Object Management Group. Object constraint language (OCL) 2.0, 2003. <http://www.omg.org/uml>.
8. Object Management Group. Unified modelling language(UML) 2.0, 2003. <http://www.omg.org/uml>.
9. D. Plump. Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence. In M. Plasmeijer and M.C. van Eekelen, editors, *Term Graph Rewriting*, pages 201–214. Wiley, 1993.
10. T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5:560–595, 1971.
11. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
12. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.
13. A. Schür. Specification of graph translators with triple graph grammars. In Tinhofer, editor, *Proceedings WG'94 International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163. LNCS 903, Springer-Verlag, 1994.
14. J. Stehr. *Semantical Consistency Check of UML Behavior Diagrams for Modelling Embedded Systems [in German]*. Diploma thesis, University of Paderborn, 2003.
15. S.Thöne, R.Depke, and G.Engels. Process-Oriented, Flexible Composition of Web Services with UML. In *Proc. of ER-Workshop on Conceptual Modeling Approaches for e-Business (eCOMO 2002)*; Tampere, Finland. LNCS, 2002.
16. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. *Distributed and Parallel Databases*. 2003.
17. H. Voigt. *Model-based Analysis of Executable Business Processes for Web Services [in German]*. Diploma thesis, University of Paderborn, 2003.