Transformation of BPEL Processes to EPCs

Jan Mendling[1], Jörg Ziemann[2]
[1]Vienna University of Economics and Business Administration, Austria
jan.mendling@wu-wien.ac.at
[2]Institute for Information Systems, University of Saarland, Germany
ziemann@iwi.uni-sb.de

Abstract: The Business Process Execution Language for Web Services (BPEL) is frequently used to implement business processes on a technical level. Yet, as BPEL is also very much related to business logic, BPEL process definitions have to be communicated to business analysts, whether for approval or for process re-engineering. As BPEL does not offer a graphical notation, an automatic transformation to a graphical language like Event-Driven Process Chains (EPCs) is required. In this paper, we present a transformation of BPEL to EPCs. We first define a conceptual mapping from BPEL to EPCs which provides the foundation for a transformation program from BPEL to EPML. Furthermore, we present the concepts used in our transformation program which are also applicable for transformations from block-oriented BPEL to other graph-based process languages.

1 Introduction

Various languages have been proposed for business process modelling focusing on different aspects including business documentation, formal analysis, service composition or choreography [MNN04]. Recently, Web Service composition is gaining increasing attention as a technology to define business processes building on Web Services. The Business Process Execution Language for Web Services (BPEL4WS or BPEL) [ACD+03] is such a language for the definition of executable processes composed of Web Services. Yet, BPEL does not define a graphical notation for its modelling primitives. Accordingly, it is a problem to communicate BPEL process definitions to business analysts when their approval is needed. Basically, this problem stems from a difference between suitable presentations of business processes to business and technical staff.

There is several research available that advocates a transformation between process modelling languages of these two different levels (see e.g. [zMR04, LGB05]). Such an approach is also applicable in order to communicate BPEL processes as Event-Driven Process Chains (EPC) [KNS92]. EPCs are especially well suited to serve as a target for a mapping from BPEL. Firstly, the graphical notation of EPCs is standardized which facilitates understandability. Secondly, as EPCs are well understood by business analysts, because they are frequently used to represent business requirements, e.g. in the context of SAP with the SAP Reference Model [KT98]. Furthermore, there is extensive tool support

for modelling with EPCs. This allows for a simple reengineering of BPEL processes that have been made available as EPC models. Finally, there is a standardized interchange format for EPCs available called EPC Markup Language (EPML) [MN05] that can serve as the target format of a transformation program. In this paper, we present a transformation of BPEL to EPCs. After an introduction into both languages in Section 2, we define a conceptual mapping from BPEL to EPCs (Section 3). This mapping builds the foundation for a transformation program from BPEL to EPML. We continue with a discussion of implementational issues that arose while writing the transformation program, in particular how block-oriented BPEL control flow can be mapped to a graph-based EPC representation (Section 4). Furthermore, we present related research in Section 5 and conclude the paper with an outlook on future research.

2 BPEL and EPCs - An Introduction

BPEL is an executable language to specify Web Service composition. That means that BPEL builds on a set of elementary Web Services to define a more complex process that is also accessible as a Web Service. BPEL offers several concepts of which we briefly sketch those that are relevant for the proposed mapping to EPCs. More details on activities and handlers will be explained in the context of the mapping. For a comprehensive overview refer to the specification [ACD+03].

- *Variables:* In BPEL variables are used to store workflow data and messages that are exchanged with Web Services. Variables have to be declared in the header part of a BPEL process.
- *PartnerLinks:* Partner links represent a bilateral message exchange between two parties. Via a reference to a `partnerLinkType` the `partnerLink` defines the mutual required `portTypes` of a message exchange: it holds a `myRole` and a `partnerRole` attribute to define who is playing which role. `PartnerLinks` are relevant for *basic activities* that involve Web Service requests.
- *Basic Activities:* Basic activities define the operations which are performed in a process. These include operations involving Web Services like the `invoke`, the `receive`, and the `reply` activity. There are further activities for assigning data values to variables (`assign`) or `wait` to halt the process for a certain time interval. Figure 1 shows a code fragment from the example given in the BPEL spec [ACD+03] which includes `invoke`, `receive`, `reply`, `wait` activities.
- *Structured Activities:* BPEL offers *structured activities* for the definition of control flow, e.g. to specify concurrency of activities (using `flow`), alternative branches (e.g. via `switch`), or sequential execution (`sequence`). These structured activities can be nested. Beyond that, `links` can be used to specify synchronization constraints similar to control flow arcs. In Figure 1 sequence activities are nested in a flow activity to define the control flow.

- *Handlers:* There are different handlers in order to respond to the occurrence of a fault, an event, or if a compensation has been triggered. Handlers are declared in the header part of a BPEL process (not shown in Figure 1).

```
001 <process name="purchaseOrderProcess"
002         targetNamespace="..."
003         xmlns="..."
004         xmlns:lns="...">
...
044   <sequence>
045     <receive partnerLink="purchasing"
046              portType="lns:purchaseOrderPT"
047              operation="sendPurchaseOrder"
048              variable="PO">
049     </receive>
050     <flow>
051       <links>
052         <link name="ship-to-invoice"/>
053         <link name="ship-to-scheduling"/>
054       </links>
055       <sequence>
056         <assign>
057           <copy>
058             <from variable="PO" part="customerInfo"/>
059             <to variable="shippingRequest"
060                 part="customerInfo"/>
061           </copy>
062         </assign>
063         <invoke  partnerLink="shipping"
064                  portType="lns:shippingPT"
065                  operation="requestShipping"
066                  inputVariable="shippingRequest"
067                  outputVariable="shippingInfo">
068           <source linkName="ship-to-invoice"/>
069         </invoke>
070         <receive partnerLink="shipping"
071                  portType="lns:shippingCallbackPT"
072                  operation="sendSchedule"
073                  variable="shippingSchedule">
074           <source linkName="ship-to-scheduling"/>
075         </receive>
076       </sequence>
077       <sequence>
078         <invoke  partnerLink="invoicing"
079                  portType="lns:computePricePT"
080                  operation="initiatePriceCalculation"
081                  inputVariable="PO">
082         </invoke>
083         <invoke  partnerLink="invoicing"
084                  portType="lns:computePricePT"
085                  operation="sendShippingPrice"
086                  inputVariable="shippingInfo">
087           <target linkName="ship-to-invoice"/>
088         </invoke>
089         <receive partnerLink="invoicing"
090                  portType="lns:invoiceCallbackPT"
091                  operation="sendInvoice"
092                  variable="Invoice"/>
093       </sequence>
094       <sequence>
095         <invoke  partnerLink="scheduling"
096                  portType="lns:schedulingPT"
097                  operation="requestProductionScheduling"
098                  inputVariable="PO">
099         </invoke>
100         <invoke  partnerLink="scheduling"
101                  portType="lns:schedulingPT"
102                  operation="sendShippingSchedule"
103                  inputVariable="shippingSchedule">
104           <target linkName="ship-to-scheduling"/>
105         </invoke>
106       </sequence>
107     </flow>
108     <reply partnerLink="purchasing"
109            portType="lns:purchaseOrderPT"
110            operation="sendPurchaseOrder"
111            variable="Invoice"/>
112   </sequence>
113 </process>
```

Figure 1: Code snippet of the example given in the BPEL spec [ACD+03]

We present EPCs as captured by the EPML format [MN05] that also serves as the target of our transformation program. EPML offers the traditional EPC elements (see Figure 2): *functions* for modelling activities, *events* to represent pre- and post-conditions of functions, *connectors* to describe different joins and splits, e.g. for concurrent or alternative branches of a process, *hierarchical functions* and *process interfaces* to specify sub-processes. These elements can be connected with control flow arcs. Furthermore, we will use *participant* elements and *data fields* in our mapping from BPEL. The first one captures organizational or human resources involved in the process, the second describe data elements. Both these elements can be connected to function elements via so-called relations. For more details on EPCs and EPML refer to [MN05].

Figure 3 illustrates the mapping from BPEL to EPML by giving the example purchase order process of the BPEL specification (see Figure 1 and [ACD+03]) as an EPC business

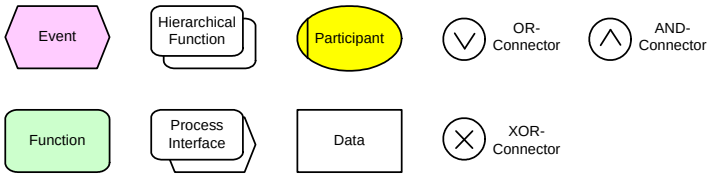

Figure 2: EPC symbols used in the mapping to BPEL

process model. The grey columns highlight the concurrent sequences that are nested within a `flow` activity in BPEL. The process part on the right-hand side captures the fault handler that has been modelled for the process. Yet, it needs to be mentioned that EPCs are not able to express explicit termination semantics - this would be required in order to correctly map fault handlers to EPCs. As you can see in Figure 3, there is an OR join waiting for the fault handler to complete. Standard EPCs do not offer a cancellation concept which could be used to represent termination semantics more appropriately. For more details on this topic refer to [MNN05]. Basic activities map to function-event blocks, that may have relationships with data fields (capturing BPEL variables) or participants (representing partnerRoles of a partnerLink). The example shows that BPEL defines complex business semantics that need to be understood by a process owner or a business analyst. This illustrates the need for an automatic transformation from BPEL to EPCs.

3 Mapping from BPEL to EPCs

As non-control flow elements of EPCs do not have a formal semantics and BPEL still includes some ambiguities (see e.g. [MSW+04]), it is important to explicitly define the purpose of the mapping and to explicate the resulting design principles. Our focus is to provide for a graphical representation of BPEL processes as EPCs in order to communicate the process dynamics to business analysts. This leads to the design principles that are applicable for the proposed mapping:

1. There should be no restriction of the constructs used in the BPEL models. We only assume the BPEL process models to be compliant with the BPEL specification in order to make the mapping work.

2. The EPC visualization focuses on the dynamic behavior of the BPEL model. Elements of a BPEL model that represent static information are represented in the EPC only if they help business analysts to understand the process logic. This implies that partnerLink declarations, variable declarations, and correlation sets as such are not addressed by the mapping. Still variables and partnerRoles of partnerLinks are represented when they are involved with the execution of an activity, e.g. when a receive activity writes an incoming message to a variable.

3. In order to present EPCs in a way that business analysts are familiar with, the in-

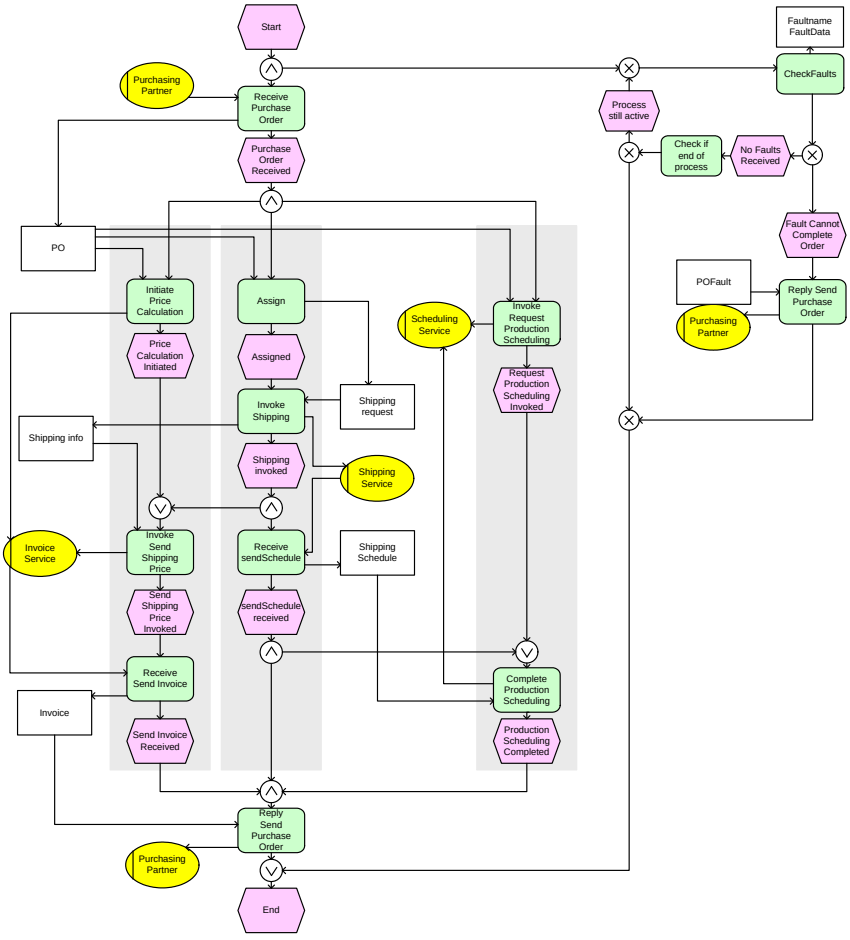

Figure 3: Example Process of BPEL Specification as EPC, compare Figure 1. The grey area highlights the sequences nested in the flow activity [ACD$^+$03].

vention of BPEL specific EPC constructs is not intended here. Yet, BPEL specific parameters may be contained in the resulting EPML file, as long as these attributes do not affect the graphical EPC model. Such parameters can be written to EPML attributes which may be annotated to most elements of an EPML file.

4. BPEL constructs should be transformed to blocks of EPC elements that offer equivalent semantics. EPC elements get names that are generated from the names of the corresponding BPEL elements.

5. It is a point of discussion whether BPEL handlers should be included in generated EPC models. We include the mapping in this paper being aware that business analysts might not be interested in them, and EPC models would be more compact leaving them out. Still, the handler related fragment can easily be deleted from the EPC model.

Mapping of basic activities: There are two aspects that have to be considered for most of the mappings. First, all basic and structured activities may be target or source of links. The BPEL `flow` activity may define multiple links that represent synchronization constraints between a source and a target activity. We will discuss this concept in the context of the `flow`. Second, basic activities map to an EPC function-event block in the general case. Structured activities determine the control flow between these function-event blocks. Beyond such a block, the mapping may generate additional data fields and participants in the EPC representation. We illustrate the mappings in the following.

Invoke, Receive, and Reply: All these three activities are related to Web Service interaction. All of them specify the attributes `partnerLink`, `portType` and `operation`. In the example of Figure 3 the process is instantiated when a purchase order is received from a purchasing partner via a `receive` activity. At run-time the process engine maps `partnerLink` and `portType` to actual endpoints that can be used in the message exchange. All these three activity types map to a function-event block whose names are built from the type of the activity and its `operation` attribute. Accordingly, a receive with `operation` *Purchase Order* yields the EPC function name *Receive Purchase Order*. The three activities involve messages that are read from input and written to output variables. These are mapped to EPC data fields that are connected via a directed relation to the function element. The relation points to the data field if the variable is written, and to the function in the other case. The name of the data field holds the name of the variable involved. Furthermore, a participant element is generated whose name is taken from the `partnerRole` of the `partnerLink`. Figure 4 illustrates the mapping. For the `invoke` activity two cases have to be distinguished. A **synchronous invoke** is similar to the execution of a remote function with in- and out-parameters: the control flow is continued only after the result of the Web Service invocation is received. This implies that synchronous invocations are connected with two data fields representing the input and the output variable. In contrast, the **asynchronous invoke** does not wait for the answer of the remote Web Service, accordingly there is only an input variable to be represented in the EPC model. A optional correlation element inside an `invoke` activity is not transformed, compensation and fault handler can be attached to the activity and will be described in

detail later. The representation of **receive** is similar to `invoke`. Each `receive` is connected to only one data field which receives the incoming message. **reply** on the other hand has a data field for the outgoing message.

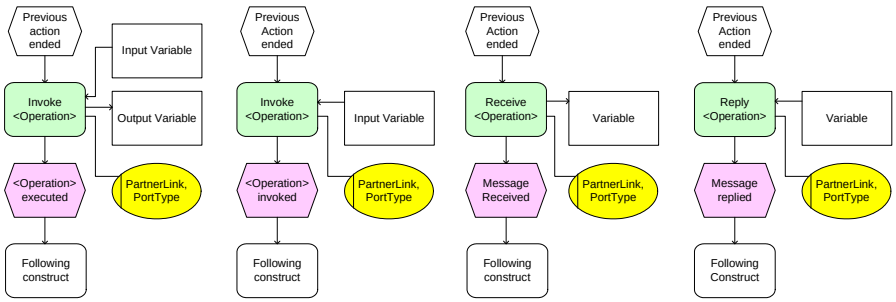

Figure 4: BPEL Web Service related basic activities mapped to EPCs

Other basic activities: Figure 5 shows the EPC representations of the other basic activities. All of them, with `empty` as an exception, are mapped to an function-event block. The **wait** activity is connected to a data field that specifies the time to be waited. Time may be specified either as a duration (e.g. for 1 hour) or by a point in time (e.g. until 2:00pm). The subsequent event occurs when the specified time has arrived. The **terminate** activity immediately terminates all activities of a business process and is followed by an end event of the process.[1] The **assign** activity is used to set the value of variables. The new value can stem either from another variable, from an expressions or from constants. Inside of an `assign` activity one or more `copy` operations specify which values are assigned. Each variable involved is represented in EPCs by a data field. The EPC function is followed by an event that represents the end of the `copy` operation. Another basic BPEL activity, the **empty** activity, does not yield a function-event block. Yet, it has to be included in the mapping if there are `source` and `target` links connected to it (see `flow` activity). The **throw** activity writes a fault to the fault variable of the current scope signalling an exception. This triggers the fault handler (explained in the subsection on handlers). Furthermore, the **compensate** activity is represented by a function-event block. It starts compensation which is also discussed in the context of BPEL handlers.

Mapping of structured activities: BPEL contains five structured activities to define the control flow; those include `sequence`, `switch`, `pick`, `while` and `flow`. A **sequence** contains one or more activities as nested child elements that are executed sequentially in the order they are listed. This sequence is mapped to a sequence of EPC elements corresponding to the elements nested in the BPEL sequence. The **while** activity is used to repeat a basic or structured actitity as long as a specified condition holds true. This is mapped to an XOR join followed by a function to check the condition. An XOR split leads to two events: if the *condition is true* event is triggered, the nested branch is executed another time. Otherwise, navigation continues with the activity subsequent to the `while`. The

[1] This mapping is basically a work-around as end events have implicit termination semantics in EPCs and a cancellation concept is missing in standard EPCs.

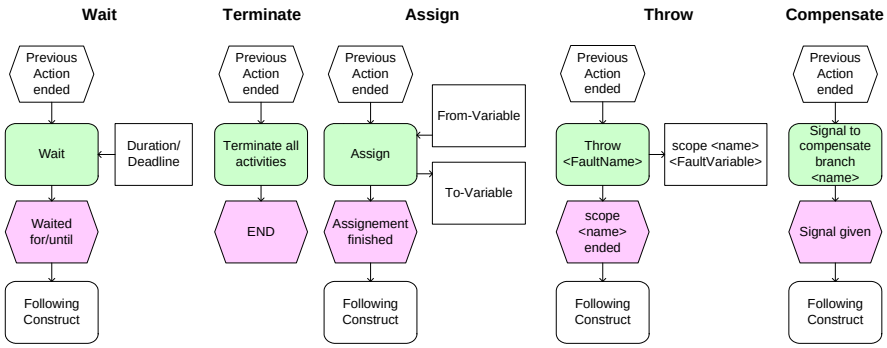

Figure 5: Further BPEL basic activities mapped to EPCs

switch activity consists of one function that evaluates an expression and, depending on the result, one of alternative branches is activated. The EPC representation of `switch` consists of a function for checking the condition followed by a block of alternative branches between an XOR split and an XOR join. The `pick` activity has some similarities to the `switch`. Yet, instead of evaluating an expression it waits for the occurrence of one out of a set of events and executes the associated activities. These events may be related to time or to message receipts. Syntactically, the `pick` maps to the same control flow elements as the `switch`. In the case of `OnMessage` conditions the message is specified with non-control flow elements similar to a `receive` activity. In the case of an `OnAlarm` event the time is modelled similar to the `wait` activity. Each alternative event is followed by nested activities merged with an XOR join. The **flow** construct enables modelling of concurrent activity branches. In EPCs concurrency is modelled by a block of parallel branches started with an AND split and synchronized with an AND join. There may be further synchronization conditions between activities specified by so-called links: each activity nested in a `flow` can be source and target of multiple links. This means that the target activity has to synchronize with the completion of the source activity. In general, each target link maps to an arc that enters an OR join prior to the target activity. Each source link maps to an AND split after the completion event of the source activity. This mapping was applied in the purchase order process of Figure 3. Additionally, the source activities may contain a transition condition (compare left part of Figure 6). If this condition yields true the subsequent AND split activates the following activity of the own branch as well as the link to the target activity. In the other case the own branch is also continues, but without activating the target link. The non-local semantics of EPC's OR join perfectly match to represent the death-path-elimination specified by BPEL for links with transition conditions. For more on this topic refer to [Ki04].

Mapping of handlers: In BPEL so-called **scopes** are used to declare areas of a process that share correlation sets, fault handlers, event handlers, compensation handlers or variables. Handlers will be mapped to EPCs without the need to generate explicit scope constructs. Variable declarations and correlations sets are not transformed. **Handlers** can be associated with whole BPEL processes, scopes, or single invoke activities. In the fol-

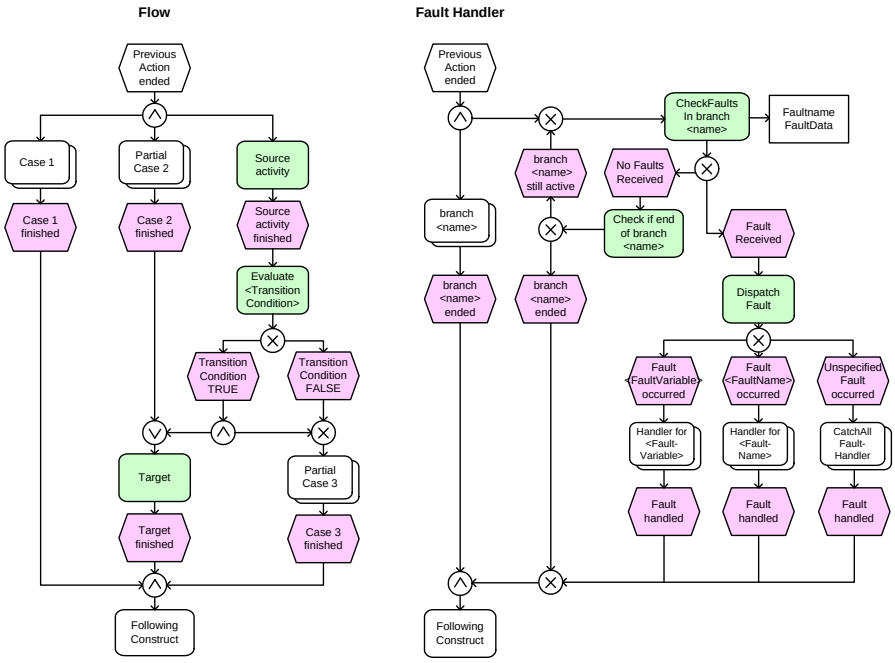

Figure 6: BPEL flow and fault handler mapped to EPCs

lowing we will refer to these three concepts using the term *branch*. Handlers wait for the occurrence of specified events. As a response certain activities are executed which are specified as sub-elements of the corresponding handler. There are two kinds of **event handlers** available. The **onMessage** event handler is mapped to an AND split and AND join that separates the main process from the event handling. The actual handling is mapped to a loop that waits as long for incoming messages as the concurrent main process has not ended. The `OnMessage` handler is associated to non-control flow elements similar to the `receive` activity. Each time a matching message arrives corresponding activities of the handler are executed. If the main process is still active, the loop is re-entered to wait for new matching messages. The `onAlarm` event handler is related to time events. Its mapping is similar to `onMessage`, but the activities to handle the event are executed once at most. Accordingly, there is no loop needed. The time event maps to a function similar to the `wait` activity.

Fault handlers are activated in response to a `throw` basic activity. Throwing a fault stops all processing of the current branch. The fault name and fault data enable the fault handler to identify the fault thrown. Because the borders of scopes are not shown in the EPC, the faults that have to be caught by the attached handler are marked with the name of the branch in which they occur (compare right part of Figure 6). Like the handlers described before, the fault handler waits until either a fault event occurs or the execution of the main branch ends. In the first case a function evaluates which kind of fault occurred and chooses

the corresponding activity. Therefore all BPEL `catch` constructs map to events subsequent to an XOR split. According to the BPEL specification, the event description contains either the name and variable, only the name, or only the variable of a fault. After the fault was handled and not re-thrown to the enclosing scope, the control flow is continued after the branch in which the fault occurred. If a branch does not specify a `catchAll` handler, the so-called implicit fault handler is included. This is mandatory if the scope contains throw activities that have no corresponding handler. The implicit fault handler triggers compensation of all child scopes and re-throws the fault in the parent scope.

Unlike the fault handler a **compensation handler** is only available for invocation if the corresponding activity has completed normally. After this, it can be invoked until the process ends. Therefore, a compensation handler is mapped to a separate EPC process in the EPML file that consists of a loop starting after the activity to be compensated has ended. Inside the loop a function checks if the corresponding compensation signal is thrown, in which case the compensation function is executed. Such signalling can only be represented descriptive in EPCs. Implicit compensation handlers only have to be mapped to EPCs if they contain a child or descendent scope or invoke activity that has a compensation handler.

4 The BPEL2EPML Transformation Program

Building on this conceptual mapping we have started implementing a transformation program called BPEL2EPML in the object-oriented scripting language XOTcl [NZ00], which is an extension of Tcl, using the tDOM package. So far, BPEL processes made up of flow and sequence activities as well as Web Service basic activities can be transformed automatically to EPML. The transformation follows a Flattening strategy[2] as reported in [MLZ05]. For the implementation the following three issues had to be solved: transformation of basic activities, transformation of structured activities, and compliance with EPC syntax rules.

The program processes the structure of nested BPEL activities hierarchically, starting from the top process element. In order to derive a graph-based EPC model, unique IDs have to be generated for each EPC element plus corresponding arcs and relations that reference the correct IDs. The BPEL2EPML program defines a transformation class that holds the $nextId$ of type integer as a class variable. For each activity type, there is a specialized method to generate EPML output. Each time a new EPC element is added to the EPML output, its ID is set to the current $nextId$ which is incremented afterwards. Using this mechanism allows to map each **basic activity** to a function-event block with accompanying data fields and participant elements without a clash of unique ID elements. Yet, there is another mechanism needed to ensure that the function-event block is correctly connected to other function-event blocks via arcs.

The transformation methods for the **structured activities** provide for correct connections between the function-event blocks. Each structured activity method generates the corresponding control structure in which blocks for its child activities can be placed. For each

[2] Pseudo code for the control flow transformation can also be found in [MLZ05].

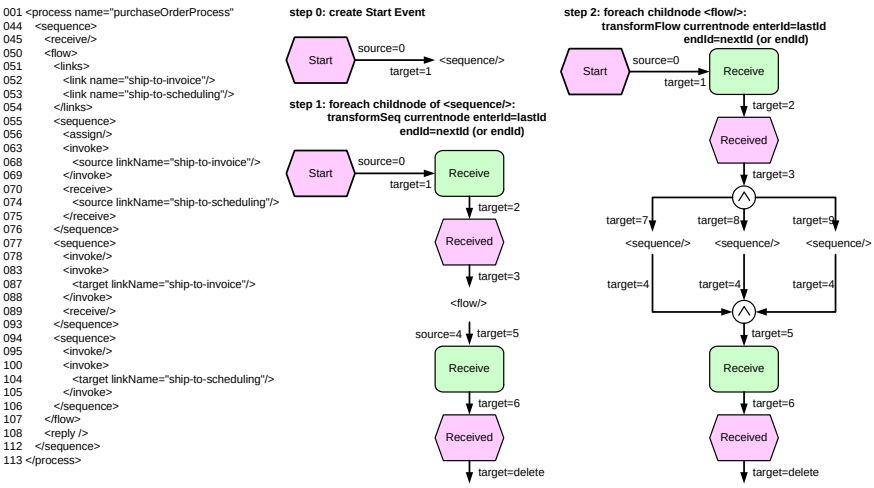

Figure 7: Id generation for BPEL activities

child activity its specialized transformation method is called, whether it is a nested structured or basic activity. There are three parameters that drive the specialized transformation methods: a DOM object holding the BPEL code that still needs to be transformed by the method; an $enterId$ that holds the ID the first EPC element of the nested block must have; and an $exitId$ holding the ID of the subsequent EPC element the nested block must generate an arc to. Consider the BPEL example of the second section which is given in an abbreviated way in the left part of Figure 7. The transformation is started by generating an EPC start event and an arc from it; the $nextId$ is initialized with 1, the $exitId$ is set to $delete$ (step 0). In the next step (step 1) the `sequence` element as the current XML node is transformed by generating EPC fragments for each child of `sequence`. For the last child element the $exitId$ is used as a parameter that the `sequence` received at its invocation. Yet, there is still a problem, because the reply of the example is transformed to a function-event block with an arc from the event to the $exitId$ given. This means an arc is pointing from the *received event* to an $id = delete$. In the next step (step 2) the `flow` is transformed. The IDs of the AND connectors have already been defined as $enterId$ and $exitId$ in the previous step in order to get a coherent EPC graph. This procedure continues until all nested BPEL activities have been processed.

The final arc with target $delete$ illustrates that additional rules have to be encoded to generate EPCs that comply with **EPC syntax rules**. A simple rule is to delete arcs that point to an $id = delete$. Furthermore, a more complex operation is needed to merge e.g. the final events of concurrent branches of a top level `flow` activity to one single end event. Without this operation, the AND join would not have a successor node which is not allowed for an EPC. The last events of the concurrent branches have to be deleted and an end event has to be added after the AND join of the `flow`. This implies that also arcs have to be redirected.

5 Related Research

There are several publications that define transformations between different business process modelling languages, e.g. from UML to BPEL [Ga03], from BPMN to BPEL [Wh04], from EPML to AML [MN04], from BPEL to Petri nets [HSS05] to name but a few. An overview and a comparison of different transformation strategies involving BPEL is reported in [MLZ05]. The merit of our approach is two-fold. First, several of these transformations take a graph-based modelling language as input to generate BPEL. Our contribution is to offer a transformation that facilitates the communication and re-engineering of BPEL process by giving a transformation from BPEL to EPCs as a graph-based language. Furthermore, our contribution is also technical by sketching a transformation for BPEL to graphs that can be also used to generate other graph-based output.

6 Conclusion and Future Work

In this paper, we have introduced a transformation from BPEL to EPCs. Building on a conceptual mapping, we presented a transformation program that is able to generate EPC models as EPML files from BPEL process definitions automatically. Such a transformation helps to communicate BPEL processes to business analysts that are often involved in the approval of business logic. Furthermore, the program can be used for re-engineering of BPEL processes. Finally, the transformation concept is general in such a way that it can be easily adapted to generate output of another graph-based process language that is encoded in XML. In future research we aim to define a profile for EPCs that offers the semantics to be mapped to BPEL. We plan to implement this mapping in a transformation program as well. The transformation from BPEL to EPML defines a starting point for such an endeavor.

References

[ACD+03] Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., und Weerawarana, S.: Business Process Execution Language for Web Services, Version 1.1. Specification. BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems. 2003.

[Ga03] Gardner, T.: UML Modelling of Automated Business Processes with a Mapping to BPEL4WS. In: *Proceedings of the First European Workshop on Object Orientation and Web Services at ECOOP 2003*. 2003.

[HSS05] Hinz, S., Schmidt, K., und Stahl, C.: Transforming BPEL to Petri Nets. In: *Proceedings of BPM 2005*. LNCS 3649. S. 220–235. 2005.

[Ki04] Kindler, E.: On the semantics of EPCs: Resolving the vicious circle. In: J. Desel and B. Pernici and M. Weske (Hrsg.), *Business Process Management, 2nd International Conference, BPM 2004*. volume 3080 of *Lecture Notes in Computer Science*. S. 82–97. Springer Verlag. 2004.

[KNS92] Keller, G., Nüttgens, M., und Scheer, A. W.: Semantische Prozessmodellierung auf der Grundlage “Ereignisgesteuerter Prozessketten (EPK)”. Technical Report 89. Institut für Wirtschaftsinformatik Saarbrücken. Saarbrücken, Germany. 1992.

[KT98] Keller, G. und Teufel, T.: *SAP(R) R/3 Process Oriented Implementation: Iterative Process Prototyping*. Addison-Wesley. 1998.

[LGB05] Lippe, S., Greiner, U., und Barros, A.: A Survey on State of the Art to Facilitate Modelling of Cross-Organisational Business Processes. In: *Proceedings of the 2nd GI-Workshop XML4BPM 2005, Karlsruhe, Germany*. 2005.

[MLZ05] Mendling, J., Lassen, K., und Zdun, U.: Transformation strategies between block-oriented and graph-oriented process modelling languages. Technical Report JM-2005-10-10. WU Vienna. October 2005.

[MN04] Mendling, J. und Nüttgens, M.: Transformation of ARIS Markup Language to EPML. In: *Proceedings of the 3rd GI Workshop on Business Process Management with Event-Driven Process Chains (EPK 2004)*. S. 27–38. 2004.

[MN05] Mendling, J. und Nüttgens, M.: EPC Markup Language (EPML) - An XML-Based Interchange Format for Event-Driven Process Chains (EPC). Technical Report JM-2005-03-10. WU Vienna, Austria. 2005.

[MNN04] Mendling, J., Nüttgens, M., und Neumann, G.: A Comparison of XML Interchange Formats for Business Process Modelling. In: *Proceedings of EMISA 2004 - Information Systems in E-Business and E-Government*. LNI. 2004.

[MNN05] Mendling, J., Neumann, G., und Nüttgens, M.: Towards Workflow Pattern Support of Event-Driven Process Chains (EPC). In: *Proceedings of the 2nd GI-Workshop XML4BPM 2005, Karlsruhe, Germany*. 2005.

[MSW+04] Martens, A., Stahl, C., Weinberg, D., Fahland, D., und Heidinger, T.: Business Process Execution Language for Web services - Semantik, Analyse und Visualisierung. Informatik-Berichte 169. Humboldt-Universität zu Berlin. 2004.

[NZ00] Neumann, G. und Zdun, U.: XOTcl, an Object-Oriented Scripting Language. In: *Proc. of Tcl2k: The 7th USENIX Tcl/Tk Conference, Austin, Texas, USA*. 2000.

[Wh04] White, S. A.: Business Process Modeling Notation. Specification. BPMI. 2004.

[zMR04] zur Muehlen, M. und Rosemann, M.: Multi-Paradigm Process Management. In: *Proc. of the Fifth Workshop on Business Process Modeling, Development, and Support - CAiSE Workshops*. 2004.