Nautilus Event-driven Process Chains: Syntax, Semantics, and their mapping to BPEL

Oliver Kopp, Tobias Unger, Frank Leymann

Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
70569 Stuttgart
{oliver.kopp, tobias.unger, frank.leymann}@iaas.uni-stuttgart.de

Abstract: Nautilus Event-driven Process Chains (N-EPCs) are a variant of Event-driven process chains allowing multiple events between functions. This allows events to be used as transition conditions in a mapping to the Business Process Execution Language for Web Services (BPEL). We will give a formal definition of N-EPCs and show how they can be mapped to BPEL. A close look will be taken how connectors can be eliminated while preserving their semantics.

1 Introduction

Event Driven Process Chains (EPCs) were introduced in 1992 as an intuitive metamodel for process modeling [KNS92]. The business process modeling tool Nautilus [Ge06a] is using a slightly modified version of EPCs, which we call Nautilus Event-Driven Process Chains (N-EPCs). The main difference is that in N-EPCs functions and events need not alternate, allowing multiple events between functions. This enables a more detailed modeling of the control flow by allowing nested conditions such as “amount > 1 million euros and premium customer”. Figure 1 illustrates this using a simplified process for loan application processing. Surely, this can also be modeled using the traditional EPCs of [KNS92] by adding functions that execute nothing. However, the N-EPC approach makes this addition obsolete and eases reading the modeled EPCs.

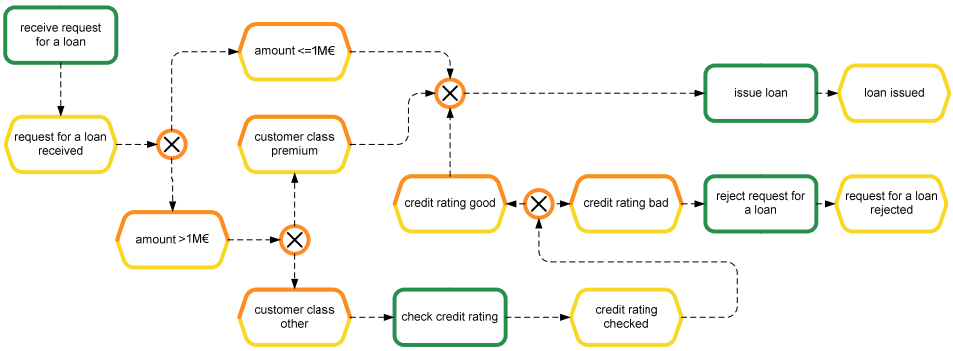

Figure 1: N-EPC describing a simplified process for loan application processing

Published in: M. Nüttgens, F. J. Rump, J. Mendling, eds: Proc. of the 5th GI Workshop on Event-Driven Process Chains (EPK 2006), Vienna, Austria, Dec. 2006. CEUR Workshop Proceedings Volume 224. ISSN-1613-0073.

Enabling multiple events between functions allows a detailed modeling of conditions between functions. N-EPCs are neither directly executable in BPEL compliant workflow systems nor do BPEL modeling tools support the import of N-EPCs and EPCs. On the other hand, a large number of workflow systems and modeling tools support the Business Process Execution Language for Web Services (BPEL, [An03]). Thus, we map N-EPCs to BPEL to enable both, their execution and import into BPEL tools.

In the following, we first describe the basics of N-EPCs, including their syntax and a discussion of their semantics. The main sections of this paper are devoted to the mapping to BPEL, covering in detail how events, connectors and functions get mapped.

2 Event-driven Process Chains in Nautilus

Nautilus Event-Driven Process Chains (N-EPCs) build the basis of the metamodel of Nautilus. An N-EPC consists of three different types of nodes: events, functions, and connectors. An event describes a state of the modeled process. A state is the result of a previous step, can trigger next steps or both. A function describes a step in the process. Connectors join or fork the control flow. A connector is either an AND, an OR, or an XOR connector. For the labeling of functions and events, the concept of objects, verbs, and states is introduced. A function is doing something with an object. After the processing, the object is in a new state. Therefore, functions are labeled with a verb and an object and events are labeled with a state and an object. Events and functions need not to be alternating. Transition from one function to another may be via multiple events. Figure 2 shows the elements of an N-EPC.

Figure 2: Elements of an N-EPC

An event directly after a function is called “trivial event”, because it always occurs after a function has been executed. The object of a trivial event is the same as in the preceding function. It is a modeling convention to use the participle of the verb as the new state of the object. For a detailed discussion of the convention see [Ge06b]. An event labeled with (o,s) occurs, if the state of the object o changes to the state s.

2.1 Syntax

There are several approaches for formalizing the syntax of EPCs. [NR02] and [Ki04] provide one of them. Since the metamodel of N-EPCs is different from the one introduced in [KNS92], we have to introduce a new formal definition. This formalism stays close to the one found in [Ki04].

Notation 1 (Set of all sequences). Let T be any set. By T^+ we denote the set of all finite sequences over elements of the set T, e.g. T={0,1}, then T^+={0,1,00,01,10,…}.

Definition 1 (N-EPC). An N-EPC is a tuple M=(E,F,C,A,l,T,V,O,S) satisfying:

- E,F,C are disjoint sets
- E is a set of events
- F is a set of functions
- C is a set of connectors
- A is a set of arcs, A⊂(E∪F∪C)×(E∪F∪C).
- T is a set of terms
- V is a set of verbs
- O is a set of objects
- S is a set of states
- l is the labeling function: l: (E∪F∪C∪V∪O∪S) → (V∪O∪S∪T^+),

$$l(x)=y,\ y\in \begin{cases} O \times S & x \in E \\ V \times O & x \in F \\ \{\oslash, \odot, \otimes\} & x \in C \\ T^+ & x \in V \cup O \cup S \end{cases}$$

To ease reading, we introduce following notations:

Notation 2 (Predecessors and successors). Let N be a set of nodes and let A⊆N×N be the set of arcs. For each node n∈N, $adj^+(n)=\{m|(n,m)\in A\}$ denotes the successors of the node and $adj^-(n)=\{m|(m,n)\in A\}$ denotes the predecessors of the node. If there is only one successor, $adj_1^+(n)$ returns that successor and $adj_1^-(n)$ that predecessor:

$$adj_1^-(n) = \begin{cases} m, m \in adj^-(n) & |adj^-(n)| = 1 \\ \perp & \text{otherwise} \end{cases}, \qquad adj_1^+(n) = \begin{cases} m, m \in adj^+(n) & |adj^+(n)| = 1 \\ \perp & \text{otherwise} \end{cases}$$

"⊥" indicates "undefined". It is included in the definition to let $adj_1^+(n)$ and $adj_1^-(n)$ return a value for all n∈N.

Notation 3 (Connected nodes). Let N be a set of nodes and let A⊆N×N be the set of arcs. For each node n∈N, adj*(n) denotes the transitive closure, i.e. the set of all directly or indirectly connected nodes including the node n itself.

M is a valid N-EPC if it satisfies the following conditions:

1. Each function has exactly one successor. That successor is an event[1]:
   $\forall f \in F: |adj^+(f)|=1$ and $adj_1^+(f) \in E$
2. Each function has at most one predecessor:
   $\forall f \in F: |adj^-(f)| \leq 1$
3. Each event has at most one predecessor and at most one successor:
   $\forall e \in E: |adj^-(e)| \leq 1$ and $|adj^+(e)| \leq 1$
4. Events are connected by connectors, are triggered by functions or trigger functions:
   $\forall e \in E: (|adj^-(e)|=1 \Rightarrow adj_1^-(e) \in F \cup C)$ and $(|adj^+(e)|=1 \Rightarrow adj_1^+(e) \in F \cup C)$
5. Events and connectors are always directly or indirectly connected to a function:
   $\forall n \in E \cup C: \exists f \in adj^*(n): f \in F$.
6. There are no self loops[2]:
   $\forall c \in C: (c,c) \notin A$
7. There has to be at least one event between an OR connector with more than one successor and a function:
   $\forall w \in \{((n_0,n_1),(n_1,n_2),\ldots,(n_{j-1},n_j))| (n_i,n_{i+1}) \in A, 0 \leq i < j, n_0 \in C, l(n_0)=\bigcirc\!\!\!\!\vee, |adj^+(n_0)|>1, n_j \in F, n_k \in E \cup C, 0<k<j\}: \exists n_k: n_k \in E$
8. There has to be at least one event between an XOR connector with more than one successor and a function:
   $\forall w \in \{((n_0,n_1),(n_1,n_2),\ldots,(n_{j-1},n_j))| (n_i,n_{i+1}) \in A, 0 \leq i < j, n_0 \in C, l(n_0)=\otimes, |adj^+(n_0)|>1, n_j \in F, n_k \in E \cup C, 0<k<j\}: \exists n_k: n_k \in E$

We call a connector with more than one successor a “fork” and a connector with more than one predecessor a “join”. A connector can be a fork and a join if it has more than one successor and more than one predecessor. In addition, we call a connector with $\bigcirc\!\!\!\!\wedge$ as label “AND connector”, with $\bigcirc\!\!\!\!\vee$ as label “OR connector”, and with $\otimes$ as label “XOR connector”.

2.2 Semantics

[Ge06b] does not contain a formal semantics of N-EPCs, leaving room for mulitple interpretations. For explaining the intended semantics, we use the process folders from [vdADK02]. Process folders are comparable to tokens in a Petri net. Process folders mark the current active functions, events, connectors, and arcs in an N-EPC. An N-EPC

---

[1] The metamodel does not distinguish between trivial events and other events, since a trivial event is defined as the event directly following a function.

[2] It is sufficient to add a constraint for $c \in C$, because rules 1 to 4 ensure that events and functions cannot be connected to themselves.

does not contain explicit start- or end-nodes. The initial state of an N-EPC is formed by a non-empty sub-set of nodes without incoming arcs that are marked with a process folder. The end of the processing of an N-EPC is formed by process folders that are only at nodes without outgoing arcs. During execution, the process folder is passed by the nodes of the N-EPC. A function takes the process folder on its incoming arc, executes and puts the process folder on its outgoing arc. An event takes the process folder on its incoming arc and owns the process folder. As soon as it occurs, it puts the process folder on its outgoing arc. The arcs themselves do nothing with the process folder. Connectors fork and join process folders. We will first explain the semantics of forks and afterwards the semantics of joins.

Generally speaking, a fork connector ensures that the incoming process folder gets propagated to the next join or function. If a fork connector has multiple incoming edges, it is first treated as a join connector, which is explained below. If a fork connector has one incoming arc, the process folder on the incoming arc is taken and put on its outgoing arcs according to the label of the fork. If the connector is an AND fork, the folder is put on all of its outgoing arcs. If the connector is an OR fork, the folder is put on at least one of its outgoing arcs. If the connector is a XOR fork, the folder is put on one of its outgoing arcs. For XOR and OR forks, the decision, which arc is taken depends on the succeeding events. Therefore the semantics of XOR and OR forks is non-local. See Figure 3 as example: Arc 1 is active. Since the XOR connector has only one predecessor, it takes the process. If event e1 occurs, the XOR connector puts its process folder onto arc 2. If event e2, event e3 or both events occur, the process folder is via arc 3. If e1 and e2 occur, the behavior is non-deterministic: The following things can happen: An error is raised to the outside[3], the XOR connector doesn't pass the process folder at all, the process folder is passed at random to arc 2 or to arc 3 or even to both arcs.

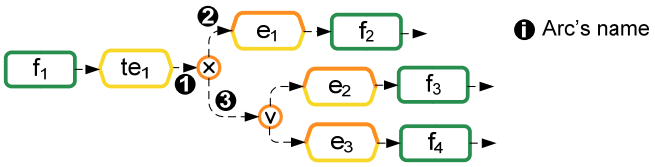

Figure 3: N-EPC[4] illustrating multiple events between functions

If the events are chained, the evaluation gets more complicated. Figure 4 contains an example. There, e1 and e2 can both occur. If e3 and e4 occur, the process folder gets passed to arc 2, even if both event e1 and e2 occurred. If e3 and e4 do not occur, e5 or e6 will occur so that the process folder is passed to arc 3. Therefore, the decision cannot be taken by solely looking at the first reachable events. All events on the path from the XOR connector to each function have to be regarded. This is specific to N-EPCs, since there possibly is more than one event between an XOR connector and a function.

---

[3] That means, the human reader (and executer) of the N-EPC is stopping executing the process and should talk to his manager.

[4] If the labeling with tuples is not important for the description, we omit that labeling and just use unique labels

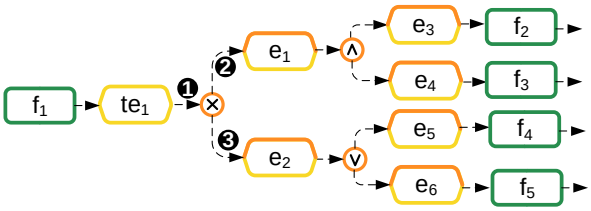

Figure 4: N-EPC illustrating multiple events between functions

We first describe the semantics informally, and then formalize it: An active XOR connector triggers[5] if exactly one of its outgoing arcs can trigger. An activated OR connector triggers if at least one of its outgoing arcs can trigger. An activated AND connector triggers if all of its outgoing arcs can trigger. An arc triggers if its target element can trigger. A function can always trigger. Finally, an event can trigger if it has occurred and its successor can trigger. It is important that besides the occurrence of the event its successor can trigger, since all the events on the way to the next function have to be regarded. It is not always known in advance when a certain event occurs. A state change of an object can happen because the object was modified by a function or because the modification of the object was not modeled by the N-EPC. In the latter case the event is called "external event". An external event can be "temperature below 25°C". The change of the object "temperature" can be modeled by a function "get temperature", but the definition of N-EPC does not force that an object's state can only be modified by functions. This fact makes the evaluation of a function "canTrigger" time-dependent. The formalization of this time-dependency is the subject of our current research. The formal definition of canTrigger without considering the time-dependency is as follows:

Definition 2 (Function canTrigger): The function canTrigger(x) returns true if the element x can trigger, false otherwise. The function *eo* returns true, if exactly one of its parameters is true. An event e is true if it has occurred.

$$\text{canTrigger}(x)=\begin{cases} eo(\text{canTrigger}(n_1),\ldots,\text{canTrigger}(n_j)) & x\in C \text{ and } l(x)=\otimes, \text{adj}^+(x)=\{n_1,\ldots,n_j\} \\ \bigvee_i(\text{canTrigger}(n_i)) & x\in C \text{ and } l(x)=\bigcirc, \text{adj}^+(x)=\{n_i\} \\ \bigwedge_i(\text{canTrigger}(n_i)) & x\in C \text{ and } l(x)=\oslash, \text{adj}^+(x)=\{n_i\} \\ x\wedge(\text{canTrigger}(\text{adj}_1^+(x))) & x\in E \\ \text{true} & x\in F \end{cases}$$

The time-dependency has no influence on the verification of the N-EPCs, since every possible execution path in an EPC is considered for the verification (see [Me06] for an example of verification of EPCs). In the verification, a fork connector can always trigger. Therefore, the connectors pass the process folders randomly to their successors, according to the connector's semantics. I.e. the XOR connector passes the process folder to one of its successors, the OR connector passes the process folder to either one, two,

[5] We use the term "trigger" as a description for the situation that the element of the EPC is active and passes the process folder to one or more outgoing arcs. An arc "triggers" if it is active and it passes the process folder to its target element.

…, or all of its successors and the AND connector passes to process folder to all of its successors without evaluating canTrigger().

It is important to note, that there is a modeling convention that the first fork following a function can always trigger. The semantics shown in [NR02] and [Ki04] share this convention since these semantics assume that a fork will always trigger.

In EPCs, the semantics of XOR and OR joins is non-local [NR02], which also applies for N-EPCs. Non-locality means that means that the XOR connector triggers if one incoming arc is active and all other incoming arcs: (a) not active, and (b) cannot become active if other elements of the EPC are triggering. The OR connector triggers if at least one incoming arc is active and all other arcs cannot become active. This informal description has been formalized in [NR02] using a transition relation. The given transition relation leads to problems in certain EPCs, first discussed in [vdADK02] and solved in [Ki04]. Figure 5 shows the sample from [vdADK02] in N-EPC notation.

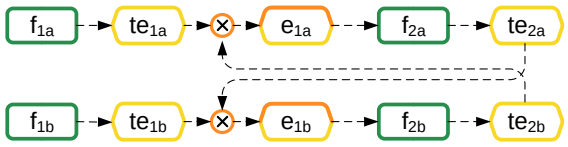

Figure 5: N-EPC showing a vicious circle

According to the semantics in [NR02], if the two XOR connectors are active, they will never propagate the process folder to their successors [vdADK02]. The mapping to BPEL will restrict the N-EPCs, preventing the occurring of any vicious circles.

Besides vicious circles, joins can be interpreted in two ways. We take the AND join as an example to illustrate them. One interpretation can be that as soon as one incoming arc becomes active, it is sure that all other incoming arcs will become active. If not, it is a modeling error. The semantics introduced in [LSW97] uses this understanding. The second interpretation of an AND join is, that it propagates the process folders if all incoming arcs are active. In all other cases it does not propagate the process folder. In particular, this case is not a modeling error. If a connector does not propagate a process folder, all of its subsequent activities that are not reachable in other ways will not be executed, too. This interpretation is shared by the semantics presented in [NR02] and [Ki04], and the informal semantics of N-EPCs. Note that the interpretation in [Ge06b] is equivalent to the semantics of the dead path elimination (DPE) if the N-EPC is acyclic. DPE forces the process graph to be acyclic and eliminates activities that cannot be reached during the execution of a BPEL process. The semantics of the DPE includes that a join synchronizes on the incoming arcs. Therefore, an XOR join directly following an AND fork will never propagate the process folders on its incoming arcs. We will give a more detailed explanation of DPE in section 3.

2.3 Extended Nautilus Event-driven Process Chains (N-eEPCs)

We have so far introduced the Nautilus Event-driven Process Chain, which contains events, functions, connectors, the arcs connecting them, and the labels on them. Nautilus also provides the possibility to model what information is sent and received by a function and which tool performs a function. The general concept is based on the Entity-Relationship Model [Ch76]. An entity can be a function, an information item, and a tool[6]. "Relations" define the relationships between entities. Nautilus allows the modeler to choose from a provided set of relation types. Table 1 lists the relation types important for the mapping to BPEL.

Entity	Relation	Entity
function	sends	information item
function	receives	information item
function	is executed by	tool

Table 1: Relation types important for the mapping to BPEL

Relating a function to another entity is optional. Thus, a function may receive something but not executed by a tool. The existence of the relation "is executed by" implies that the function is not executed by the process, but by an executer. Figuratively speaking, the function is not part of the process, but is instead called by the process. Therefore, "f sends i" means that the function f sends the information i to the process. "f receives i" means, that the process sends the information item i to the function f. If a function receives and sends information items, the order should be interpreted as "first receive, then send" [Ge06b]. It is important to note that there is no possibility for events to send or receive data.

Definition 3 (N-eEPCs). An N-eEPC is a tuple Mx=(E,F,C,A,l,T,V,O,S,I,P,s,r,x) and has to satisfy following properties:

- M=(E,F,C,A,l',T,V,O,S) forms a valid N-EPC. l' is derived from l by restricting the preimage of l to (E∪F∪C∪V∪O∪S). The image remains unchanged.
- l is the labeling function:
  l: (E∪F∪C∪V∪O∪S∪I∪P) → (V∪O∪S∪T^+),
  $$l(x)=y,\ y\in \begin{cases} \text{as defined in definition 1} & x \in E \cup F \cup C \cup V \cup O \cup S \\ T^+ & x \in I \cup P \end{cases}$$
- I is the list of information items
- P is the list of tools

[6] Nautilus contains 25 more elements in addition to these. We concentrate on the ones that are important for the mapping to BPEL. See [Ge06b] for a complete list of entities.

- s is the function assigning the sent information items to functions[7]:
  $s: F \rightarrow 2^I$
- r is the function assigning the received information items to functions:
  $r: F \rightarrow 2^I$
- x is the function assigning the executing tools to functions:
  $x: F \rightarrow 2^P$

3 The Business Process Execution Language for Web Services

BPEL is currently the widest-spread language to support orchestration of Web Services. It is implemented by several companies, including IBM, Microsoft, and Oracle. The current version is 1.1. In this section, a brief overview will be presented of the subset of BPEL 1.1 used by the mapping. A complete explanation of the language and its elements can be found in [An03]. Additionally, [LR05] illustrates the main concepts, and [Ou05] and [HSS05] provide a formal description.

A BPEL process make use of Web Service(s) offered by partners and is itself offered as one or more Web Service(s). These Web services are specified as port types[8]. The connection with a partner is modeled using a partner link, typed by a partner link type having one or two roles. Each role refers to a port type. If a BPEL process uses the partner's port type and does not offer a port type to the partner, the partner link type contains a single role element and vice versa.

A BPEL process model may use different kinds of activities. An `invoke` activity is used to invoke a partner's Web Service operation, and consists of input/output variables, a partner link and the name of the operation. If the operation is one-way, then the output variable is not specified. A `receive` activity is used to handle incoming calls to the process's Web Service operations, and consists of a variable, a partner link, and the name of the process's operation. Variables are typed, usually with WSDL message types [An03]. Such simple activities can be grouped into complex activities like `flow`,.

Activities inside a flow can be connected by links that have transition conditions. Every activity with incoming links has a join condition, which as a logical operation on the status of those links. When a flow starts, it enables all immediately enclosed activities with no incoming links. Once an activity completes, the transition conditions on the outgoing links get evaluated and the status of each link is set to the result of the evaluation. The join condition of an activity is evaluated once all the incoming links have fired. If it is true, then the activity executes. If it is false, the activity is skipped and the status of its entire outgoing links is set to "false". This procedure, formally defined in [LR00], is called "dead-path-elimination" (DPE).

---

[7] 2^S is used to refer to the power set of a set S

[8] A port type is an interface defined with a WSDL file. For a complete description of WSDL see [Ch01].

Besides modeling control flow using flat-graph process definition approach, control flow can be modeled with a structural constructs approach or a combination of the two approaches. Structural constructs are `switch` and `sequence`. A `sequence` executes the containing activities sequentially in lexical order. A `switch` contains one or more branches with conditions assigned, and optionally a default branch. The conditions are evaluated in lexical order. As soon as a condition evaluates to true, the respective branch is taken. If no condition evaluates to true, the default branch, if existing, is executed.

BPEL Version 2.0 has just reached the public review stage of standardization [Al06]. There are no major conceptual changes regarding the navigation from 1.1 to 2.0. Thus this explanation and the presented mapping will remain valid with regards to BPEL 2.0.

4 Mapping N-eEPCs to BPEL

We will use the extended Nautilus Event-driven Process Chains for mapping to BPEL, since they contain annotations showing which tool executes a function and what data is needed and returned by each function. To get an idea of the mapping of N-eEPCs to BPEL we give a rough overview at first. In the subsequent paragraphs, we will give a detailed explanation of each step and a justification why the transformation was chosen as such.

The elements of the N-eEPC are used as follows: The information about the executing tools and sent and received information items is used to generate the partner's WSDL files[9] and the corresponding partner links. Each function becomes an operation in a port type. In the generated BPEL process the generated activities are nested in a `flow` activity to naturally reflect the structure of the structure of the EPC. Each function is transformed into a `receive`, `invoke` or `empty` activity, depending on the sent and received information and whether an executing tool is assigned. Connectors get `empty` activities having their label transformed into a join condition. The incoming and outgoing arc of an event is mapped to a link connecting the mapped predecessor and mapped successor using the event as transition condition. Events with no successor get mapped to an `empty` activity.

Having provided the high level description of the mapping, we now introduce the restrictions on the N-eEPCs which can be mapped to BPEL.

BPEL supports natural loops[10] having the entry node as the only exit node and BPEL does not support arbitrary cycles. On the other hand, N-eEPCs model loops implicitly. Mendling et al. [MLZ06] showed what kind of loops can be made explicit and be transformed to BPEL. Since loop detection is out of the focus of this paper, we require that the N-eEPC used in the mapping be acyclic.

---

[9] Every tool becomes a partner in the BPEL process. For a complete description of WSDL and it's relation to BPEL see [Ch01] and [An03]

[10] Natural loops are loops with a single entry node [Mu97].

Definition 4 (Acyclicity of an N-eEPC). A N-eEPC M_e=(E,F,C,A,l,T,V,O,S,I,P,s,r,x) is acyclic, if the N-EPC M=(E,F,C,A,l,T,V,O,S) is acyclic.

Definition 5 (Acyclicity of an N-EPC). A N-EPC M=(E,F,C,A,l,T,V,O,S) is acyclic, if there is no non-empty path from a connector c to the same connector:
$\nexists w: w=((n_0,n_1),(n_1,n_2),\ldots,(n_{j-1},n_j)), n_0,n_j \in C, n_j=n_0, (n_i,n_{i+1})\in A, 0\leq i<j.$

In addition to the acyclicity we demand that the N-eEPC contains a single root and that this root is a function. Handling multiple roots introduces special handling – e.g. the detection of pick[11] – and other syntactical restrictions, which are out of scope of this paper.

Definition 6 (Roots of an N-eEPC). A N-eEPC M_e=(E,F,C,A,l,T,V,O,S,I,P,s,r,x) has the same roots as the N-EPC M=(E,F,C,A,l,T,V,O,S): roots(M_e)=roots(M).

Definition 7 (Roots of an N-EPC). Roots of a N-EPC M=(E,F,C,A,l,T,V,O,S) are the functions, events and connectors having no incoming arc: roots(M)={n | $n\in E\cup F\cup C$, $|adj^-(n)|=0$}

Information items are not annotated with a type. Therefore a fixed type has to be taken. We chose `string` as type, because most types can be serialized to a string. This leads to incompatibility if existing Web Services should be wired to the exported process. In that case, we suggest using mediation. Interface maps and data maps are one way to do mediation [IBM06]. Interface maps can be used if the name and the signature of an operation do not match. With an interface map, two different operations can be mapped to each other. For mapping data types, the data mapping can be used. A data map maps two data types to each other.

N-eEPCs allow multiple executers be related to a function. To get a clear mapping, we demand that at most one executing tool is assigned for each function.

To formally explain the mapping, we need the definition of an intermediary N-eEPC (IN-eEPC) that stores the current state of the transformation:

Definition 8 (Intermediary N-eEPC, IN-eEPC). An IN-eEPC I_M is a tuple I_M = (E,F,C, A,l,T,V,O,S,I,P,s,r,x,jc,tc), where M_e=(E,F,C,A,l,T,V,O,S,I, P,s,r,x) is a N-eEPC, where no syntactical restrictions apply. E.g. a function may have several successors. In addition jc and tc are defined as follows:

- jc is a function assigning the join condition to a function and a connector:
  $jc: F\cup C \rightarrow \{\owedge, \ovee, \otimes, \perp\}$
- tc is a function assigning an event (interpreted as transition condition) to an arc:
  $tc: A\rightarrow E\cup\{\perp\}$

[11] See [An03] for an explanation

We will first present the elementary steps of the mapping and combine them in a complete algorithm at the end. First, we will explain the mapping of events, then the mapping of connectors, and finally the mapping of functions from an N-eEPC to an IN-eEPC. Then, we map the IN-eEPC to BPEL. We will take an arbitrary N-eEPC $M_e=(E_e,F_e,C_e,A_e,l_e,T_e,V_e,O_e,S_e,I_e,P_e,s_e,r_e,x_e)$ satisfying the conditions above as the starting point and map it to an IN-eEPC $I_M = (E,F,C,A,l,T,V,O,S,I,P,s,r,x,jc,tc)$. I_M is initialized as follows:

$E:=E_e$, $F:=F_e$, $C:=C_e$, $A:=A_e$, $l:=l_e$, $T:=T_e$, $V:=V_e$, $O:=O_e$, $S:=S_e$, $I:=I_e$, $P:=P_e$, $s:=s_e$, $r:=r_e$, $x:=x_e$, $tc(x):=\perp\ \forall x\in A$, $jc(x)=\begin{cases} l(x) & x \in C \\ \perp & \text{otherwise} \end{cases}$

4.1 Mapping of events

Having an initial I_M, we can start with the transformation of events. Generally, events are mapped to transition conditions. There are two exceptions: Trivial events and events having no outgoing arcs[12]. Trivial events always occur after the preceding function has finished. Therefore, they do not bring additional semantics and can be removed from the N-eEPC as shown in Algorithm 1.

Algorithm 1 Removal of trivial events

```
procedure RemoveTrivialEvents(I_M)
    for all f∈F do
        e ← adj1+(f)
        if |adj+(e)| > 0 then
            A ← A \ {(f,e), (e,adj1+(e))} ∪ {(f,adj1+(e))}
        else
            A ← A \ {(f,e)}
        end if
        E ← E \ {e}
    end for
end procedure
```

Events having no outgoing arcs may be handled in two ways: Either remove them from the graph or transform them to `empty` activities. We decided to keep them in the resulting BPEL process instead of removing them. `Empty` activities do not lead to new partner interaction, but take execution time. On the other hand, a form of documentation is lost if they are removed. Thus it is an open discussion if removal is preferable to keep them. During the modification of I_M events having no outgoing arcs are transformed to functions with no executer and no information items assigned. They will get mapped to `empty` activities during the transformation of functions.

[12] Events with no incoming arcs do not occur, since we demanded that M_e has a single root which is a function.

Algorithm 2 Transformation of events without outgoing arcs

procedure TRANSFORMLEAFEVENTS(I_M)
  E' ← E
  for all e∈E', $|adj^+(e)|=0$ **do**
    E ← E \ {e}
    F ← F ∪ {e}
  end for
end procedure

The remaining events have one incoming and one outgoing arc. They get transformed into transition conditions. With this interpretation of events, only events occurring as a result of a function can be mapped. We chose this mapping, because the metamodel of N-eEPCs does not allow annotating events with "sends", "receives", or "executed by" relations. The information items in N-eEPCs are not related to each other, too. E.g. "address consists of street, postal code and city"[13] cannot be modeled. Assume a function sending "address" and a succeeding event "(London, city)". Without additional information, an algorithm cannot decide whether the event belongs to the function. Furthermore, we did not want to extend the metamodel, because events that receive information are a completely new concept to business analysts that know the EPC metamodel.

Algorithm 3 Transformation of events to transition conditions

procedure TRANSFORMEVENTSTOTRANSITIONCONDITIONS(I_M)
  E' ← E
  for all e∈E', $|adj^-(e)|=1$, $|adj^+(e)|=1$ **do**
    $a \leftarrow \{(adj_1^-(e), adj_1^+(e))\}$
    $A \leftarrow A \setminus \{(adj_1^-(e), e), (e, adj_1^+(e))\} \cup \{a\}$
    E ← E \ {e}
    tc' ← tc
    $tc(x) \leftarrow \begin{cases} tc'(x) & x \neq a \\ e & x = a \end{cases}$
  end for
end procedure

The transition condition will be `object = 'state'` in the resulting BPEL file. This is sufficient for cases where the object and the information item can be identified and the effective data type of the information item is `string`. One modeling convention used is that an object and an information item describe the same entity if the label of the object equals the label of the information item. If an information item cannot be identified with an object, the transition conditions have to be edited manually after the transformation to get a valid executable BPEL process.

[13] We are aware of the fact that there are numerous variations of an address. E.g. an address can alternatively contain the state, consist of a post-office box, etc.

4.2 Mapping of connectors

Now all cases of the events are handled so we continue stating the handling of the connectors. As explained in the beginning of this section, each connector gets mapped to an `empty` activity. We will show that this is a straight forward mapping that can be improved to reduce the amount of activities in the generated BPEL process.

First of all, connectors having a single incoming edge and a single outgoing edge can safely be removed from I_M. As soon as their incoming arc is active, they can pass the process folders to the outgoing arc. They do not have to wait for other incoming arcs or to evaluate canTrigger() on the outgoing arc. If they are asked for canTrigger(), they can just return the value of their outgoing arc. Therefore, we remove them[14].

Algorithm 4 Removal of connectors not joining and not forking

procedure RemoveConnectorsNotJoiningAndNotForking(I_M)
  while $\exists c \in C: |adj^-(c)|=1 \wedge |adj^+(c)|=1$ **do**
    $A \leftarrow A \setminus \{(adj_1^-(c),c), (c,adj_1^+(c))\} \cup \{(adj_1^-(c),adj_1^+(c))\}$
    $C \leftarrow C \setminus \{c\}$
  end while
end procedure

BPEL does not provide any construct for “fork conditions”. A connector in EPCs has such an implicit fork condition. E.g. an XOR fork states that only one outgoing arc can be active. The only construct similar to a fork condition is the `switch` activity which could be represented in N-EPCs using an XOR block[15]. See Figure 6 for an illustration.

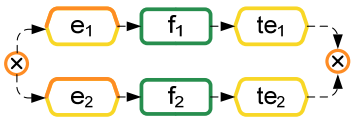

Figure 6: N-EPC modeling a switch

Instead of f_1 and f_2 there can be other EPC constructs allowing switches to be nested. This is the only case where BPEL supports fork conditions. There is no similar BPEL construct for AND or OR blocks. Furthermore, regarding arbitrary EPCs[16] the XOR block is just one of many cases. To avoid dealing with special cases, we do not use switch. As a result, the semantics of fork connectors cannot be directly mapped to BPEL. The semantics of a fork includes the assurance that an active fork will trigger at some time. This implies that the events succeeding the fork will occur in a combination allowing the fork to trigger. As a result, no fork condition is needed. Thus forks mapped to `empty` activities do not bring additional semantics to the BPEL process. They can be removed instead if following conditions apply:

[14] The removal is done before any events get mapped. Otherwise, transition conditions get lost.
[15] XOR blocks are formally defined in [Ru99]
[16] Called “unstructured process graph” in [MLZ06]

1. The fork c has a single incoming arc
2. Either
    a. The incoming arc does not contain a transition condition and all outgoing arcs do not contain a transition condition
    b. The incoming arc contains a transition condition and all outgoing arcs do not contain a transition condition
    c. The incoming arc does not contain a transition condition and some of the outgoing arcs contain a transition condition
3. If 2a does not apply, there is no arc connecting the predecessor to one of the successors of c having a transition condition.

The predecessor of the fork will be connected with all successors of the fork. Assume $\{g_i\}$ is the set of generated arcs. If condition 2b is fulfilled, the transition condition of the incoming arc is copied to each g_i. If condition 2c is fulfilled, the transition condition of each outgoing arc is copied to the corresponding arc g_i. Because of these copying rules the condition 3 is needed. BPEL does not allow multiple links between two activities. This is assured in I_M by the definition of $A \subset (E \cup F \cup C) \times (E \cup F \cup C)$ (cp. definitions 1,3,8). There is the possibility to merge the transition condition of two arcs but we think the rule "at most one event forms a transition condition" eases the understanding of the generated BPEL process. The procedure is shown in algorithm 5.

BPEL supports join conditions on any activities with incoming links. Therefore, the label of the connector is used as join condition, which is transformed to a XPATH statement representing the join condition in BPEL. Using BPEL's dead path elimination, activities not reachable will not be executed during the execution of the BPEL process.

Similar to the handling of forks, joins with a single outgoing edge can be eliminated by connecting the predecessors to the successor. The difference to the handling of the forks is the join condition. The join condition of the join connector replaces the join condition of the successor. Since the algorithm can be derived from algorithm 5, we give an example of the transformation of joins in figure 7.

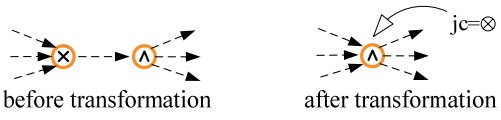

Figure 7: Illustration of the elimination of join connectors

Algorithm 5 Transformation of forks

function ForkSatisfiesConditions(c)
  $tc_i \leftarrow tc((adj_1^-(c),c)) \neq \perp$ ▷ For completeness: $tc(\perp, c) := \perp$
  $tc_o \leftarrow \exists s \in adj^+(c)$: $tc(c,s) \neq \perp$
  $arc \leftarrow \exists s \in adj^+(c): \exists a \in A, a = (adj_1^-(c), s)$: $tc(a) \neq \perp$
  $c_1 \leftarrow |adj^-|(c) = 1$
  $c_{2a} \leftarrow (\neg tc_i \wedge \neg tc_o)$
  $c_{2b} \leftarrow (tc_i \wedge \neg tc_o)$
  $c_{2c} \leftarrow (\neg tc_i \wedge tc_o)$
  $c_3 \leftarrow (c_{2a} \vee \neg arc)$
  return $c_1 \wedge (c_{2a} \vee c_{2b} \vee c_{2c}) \wedge c_3$
end function

procedure TransformForks(I_M)
  while $\exists c \in C$: $(|adj^+(c)| > 1) \wedge$ ForkSatisfiesConditions(c) **do**
    $C \leftarrow C \setminus \{c\}$
    $p \leftarrow adj_1^-(c)$
    $succ \leftarrow adj^+(c)$
    $A \leftarrow A \setminus \{(p,c)\}$
    for all $s \in succ$ **do**
      $a \leftarrow (p,s)$
      $A \leftarrow A \setminus \{(c,s)\} \cup a$
      if tc_i **then**
        $tc' \leftarrow tc$
        $$tc(x) \leftarrow \begin{cases} tc'(x) & x \neq a \\ tc'((p,c)) & x = a \end{cases}$$
      else if tc_o **then**
        $tc' \leftarrow tc$
        $$tc(x) \leftarrow \begin{cases} tc'(x) & x \neq a \\ tc'((c,s)) & x = a \end{cases}$$
      end if
    end for
  end while
end procedure

4.3 Mapping of functions

The mapping of the functions is the last step. We will also present the mapping of arcs, information items and executers since all of them are closely related to functions.

Functions describe what is done by whom and which information is sent and received by the executer. Therefore, functions sending or receiving something and being executed by someone or something are mapped to `receive` and `invoke` activities. In all other cases, functions are mapped to an `empty` activity. The incoming and outgoing arcs are mapped to incoming and outgoing links. The outgoing link is additionally assigned a transition condition if the belonging arc has a transition condition assigned. The join condition of the generated activity depends on the value of jc. Assume f is the current function to be

mapped. If jc(f)=▽ then no join condition is generated, since the OR join is the default join behavior in BPEL. If jc(f)=△ then a logical `and` over all incoming links is generated. If jc(f)=⊗, "exactly one" over all incoming links is expressed by logical `and`, `or` and `not` connectors. Assume l_1, l_2 and l_3 are the incoming links. Then the join condition expressed in XPATH representing XOR looks as follows:

```
((($l1) and (not $l2) and (not $l3)) or
((not $l1) and ($l2) and (not $l3)) or
((not $l1) and (not $l2) and ($l3))
```

l_i is gets expanded to `bpws:getLinkStatus('li')` as soon as it is written into the BPEL file.

Let f be a function having an executer and incoming and outgoing information assigned. f is mapped to a `receive` if sends something and does not receive anything. The reason is that the function is not executed by the process itself but the executer assigned to the function. If someone external sends something to the process but does not get something from the process, the process has just to receive it. In all other cases, an `invoke` is generated. Assume the function f sending information i_1 and receiving information i_2. Receive and send should be interpreted as "first receive, then send"[17]. Thus mapping to single `invoke` activity sending i_1 (using the input variable) and afterwards receiving i_2 (using the output variable) is enough: The executer first receives i_1 and then the BPEL process is ready to receive i_2. If the function receives i and does not send anything, it is mapped to an `invoke` activity sending i. The sent and received information is used to generate the variables and message types. Assume $r(f)=\{i_1,i_2,i_3\}$. With this information, a message type named i1_i2_i3 and a variable named i1_i2_i3 of the type i1_i2_i3 is generated[18]. The message type contains three parts named i1, i2, and i3 each of them having the type `string`. The variable i1_i2_i3 is used as the input variable. The generation of variables for s(f) follows the these rules, too.

The label of the function is used as the name for the operation: The concatenated verb and object form the name of the operation. For the generation of the WSDL files, the partner link types, the partner links and the port types, the tool executing a function is used. Each generated port type is put in a separate WSDL file. If a function was mapped to a `receive` activity, the operation is put into a port type named after the executer with the suffix `input`. The partner link and the partner link type are named `<tool>-to-process`. The generated port type is assigned to the partner link type and the partner link as `myRole`. If a function was mapped to an `invoke`, the operation is put into a port type named after the executer and assigned to the partner link `<tool>-to-process` as `partnerRole`. The partner link and its partner link type are generated, if it they had not already been generated by the mapping to an `receive` activity.

---

[17] cp. section 2.3

[18] If the message type has already been generated in a previous step, the message type and variable are not generated again. For the ordering in the name alphabetical ordering is used.

4.4 The complete algorithm

Using the previous descriptions, the complete algorithm is straight-forward. The algorithm first initializes I_M with M. Connectors not joining and not forking are removed afterwards. Then the events are transformed according to section 4.1 “Mapping of events”. After this transformation, the connectors are transformed as presented in section 4.2 “Mapping of connectors”. In the last step, a depth-first search (DFS) is started from the root of I_M. During the DFS functions and connectors can be found. Events are transformed to transition conditions or to functions during the step “Mapping of events”. A connector is mapped to an `empty` activity, where the links and the join conditions are generated as stated in “Mapping of functions”. Each function gets mapped to an `empty`, `receive`, or `invoke` activity as descriped in section 4.3 “Mapping of functions”.

The algorithm presented in this paper was implemented in the BPEL module in Nautilus [Ged06a] and is now commercially available. The BPEL file and WSDL files it generates can be imported into a BPEL development environment like the IBM WebSphere Integration Developer. Once in such an environment, the process can be tweaked (transition conditions modified, interface and data maps added) to enable its integration into an existing infrastructure.

5 Related work

Mendling et al. [MLZ06] categorized the approaches of mapping EPCs to BPEL. The categorization includes a high level representation of the mapping, leaving out the generation of the type of the basic activity and leaving out the generation of port types, partner links and WSDL files. Additionally, they do not address allowing multiple events between functions because the metamodel of EPCs does not allow that. Nevertheless, our mapping falls into their “Element-Minimization” category.

Ziemann and Mendling [ZM05] presented an approach using the labels of the functions to generate `receives` and `invokes`. Our mapping derives the type of the activity from the information sent and received.

6 Summary and outlook

We presented the syntax and semantics of Nautilus extended Event-driven Process Chains (N-eEPCs). N-eEPCs in contrast to EPCs introduced in [KNS92], support multiple events between functions. This allows a detailed modeling of conditions between functions. We showed how multiple events between functions can be used to generate transition conditions in BPEL. The class of external events was completely dropped from the mapping, because the metamodel of N-EPCs does not offer assigning sent and received information to an event. Special care was taken to eliminate connectors to get a readable BPEL file. Another contribution of this paper is the use of “send”, “receive” and “executed by” relations to generate partner links and port types. A

commercially available implementation has been created. It produces BPEL files that can be used in known BPEL tools and systems.

Future research directions include the investigating of possibilities to support external events by N-eEPCs and the usage of function repositories to ease integration in an existing infrastructure.

Acknowledgement

This work was supported by the German Federal Ministry of Education and Research (project number 01ISE08B).

References

All links were followed on 2006-09-22.

[Al06] Alves, A. et al.: Web Services Business Process Execution Language Version 2.0. Public Review Draft, 23rd August, 2006. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf

[An03] Andrews, T. et al.: Business Process Execution Language for Web Services version 1.1, 2003. http://www-128.ibm.com/developerworks/library/specification/wsbpel/

[Ch01] Christensen, E. et al.: Web Services Description Language (WSDL) 1.1. W3C Note 15 March 2001. http://www.w3.org/TR/wsdl

[Ch76] Chen, Peter P.: The Entity-Relationship Model - Toward a Unified View of Data. ACM Transactions on Database Systems. Volume 1. Number 1. pp. 9-36. 1976.

[Ge06a] Gedilan Consulting GmbH: Homepage. http://www.gedilan.com

[Ge06b] Gedilan Technologies GmbH: Nautilus Benutzerhandbuch. Version 4.5. 2006.

[HSS05] Hinz, S.; Schmidt, K.; Stahl, C.: Transforming BPEL to Petri Nets. In (W.M.P. van der Aalst, W.M.P., et al.): Proceedings of the 3rd International Conference on Business Process Management (BPM 2005), volume 3649 of Lecture Notes in Computer Science, Springer, 2005; pages 220-235.

[IBM06] IBM Cooperation. IBM WebSphere Business Process Management Version 6.0 information center. http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/

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

[KNS92] Keller, G.; Nüttgens, N.; Scheer, A.-W.: Semantische Prozessmodellierung auf der Grundlage Ereignisgesteuerter Prozessketten (EPK). In: Technical Report, Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi), Heft 89, Universität des Saarlandes, 1992.

[LR00] Leymann, F.; Roller, D.: Production Workflow - Concepts and Techniques, PTR Prentice Hall, 2000.

[LR05] Leymann, F.; Roller, D.: Modeling business processes with BPEL4WS. In: Information Systems and E-Business Management, Volume 4, Number 3. Springer, 2005; pp. 265-284.

[LSW97] Langner, P.; Schneider, C.; Wehler, J.: Prozeßmodellierung mit ereignisgesteuerten Prozeßketten (EPKs) und Petri-Netzen. In: Wirtschaftsinformatik, Volume 5, Number 39, 1997; pp. 479-489.

[Me06] Mendling, J.; et al.: Faulty EPCs in the SAP Reference Model. In (Dustdar, S.; et al., eds.): Proceedings of the 4th International Conference Business Process Management (BPM 2006). Lecture Notes in Computer Science Volume 4102, Springer, 2006.

[MLZ06] Mendling, J.; Lassen, K. B.; Zdun, U.: Transformation Strategies between Block-Oriented and Graph-Oriented Process Modelling Languages. In (Lehner, F.; Nösekabel, H.; Kleinschmidt, P., eds.): Multikonferenz Wirtschaftsinformatik 2006 (MKWI 2006), Band 2, XML4BPM Track. GITO-Verlag Berlin, 2006; pp. 297-312.

[Mu97] Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997.

[NR02] Nüttgens, M.; Rump, F. J.: Syntax und Semantik Ereignisgesteuerter Prozessketten (EPK). In: PROMISE 2002, Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen, GI Lecture Notes in Informatics. P-21. Gesellschaft für Informatik, 2002; pp. 64–77.

[Ou05] Ouyang, C., et al.: Formal Semantics and Analysis of Control Flow in WS-BPEL (Revised Version). BPM Center Report BPM-05-15, BPMcenter.org, 2005.

[Ru99] Rump, F. J.: Geschäftsprozeßmanagement auf der Basis ereignisgesteuerter Prozeßketten : Formalisierung, Analyse und Ausführung von EPKs. Dissertation. Stuttgart; Leipzig. Teubner, 1999.

[vdADK02] Van der Aalst, W.; Desel, J.; Kindler, E.: On the semantics of EPCs: A vicious circle. In (Nüttgens, M.; Rump, F.J., eds): EPK 2002, Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, November 2002; pp. 71–79.

[ZM05] Ziemann, J.; Mendling, J.: EPC-Based Modelling of BPEL Processes: a Pragmatic Transformation Approach. In: Proceedings of MITIP 2005, Italy, 2005.