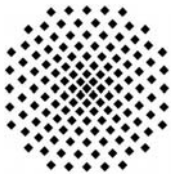


Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik

Events Make Workflows Really Useful

Matthias Wieland, Daniel Martin,
Oliver Kopp, Frank Leymann

Report 2008/09

Institut für Architektur von Anwendungssystemen

Universitätsstraße 38
70569 Stuttgart
Germany

CR: H.4.1

Contents

1 Introduction

Events are a widely used abstraction to facilitate asynchronous communication in IT systems. Although the terminology might vary slightly across different domains, the concept of an event and event communication are omnipresent. Events are used for disseminating information in e.g. mobile environments and sensor networks, application integration through messaging middleware and monitoring to control and govern large and diverse installations of IT infrastructure.

On an abstract level, an *event* is defined as "any happening of interest in a computer system" [MFP06], e.g. values reported by sensors, timers, caused by human interaction with the system or generally any detectable state change that can be described in a computer processable manner. The main characteristic of event-based systems is their inherent asynchronous nature, decoupling sender (producer) and receiver (consumer) of an event in the dimensions of time, reference and location [AADH05]. With the rise of RFID and the ubiquity of computing devices, event-based systems recently gained a lot of attention in the area of manufacturing and retail e.g. in form of the METRO Future Store[1].

Naturally, these industries also have a strong interest in Business Process Management (BPM) technology to align and support their business processes with IT infrastructure. *Business processes* are expressed using specialized business process languages. The standards commonly used in that area are the Web Service Business Process Execution Language [OAS07] (WS-BPEL or BPEL for short) and the Business Process Modeling Notation [Obj08] (BPMN). Both standards are implementations of the *orchestration layer* of a service-oriented architecture (SOA) and make heavy use of *services* as the level of individual business functionality.

The main problem this report deals with is the question how the service-oriented and event-driven architecture paradigm can be combined using BPM as the vehicle and driver for integration. Our proposal is centered around *Event-driven Process Chains* [STA05, KNS92] (EPC), an event-centric business process modeling language that treats events as "first class citizens", i.e. the occurrence of events are fundamental elements of the business process.

EPCs are part of the ARIS framework [Sch03], a "holistic modeling approach" to design and document architectures of integrated information systems from a business' perspective. In ARIS, EPCs are used in the "control view" to describe business processes, allowing for integration and reuse of elements from other views of a model. EPCs consist of four main elements: (i) events (depicted as hexagons), (ii) functions (depicted as rounded boxes), (iii) connectors (depicted as circles) and (iv) control-flow edges. Events in EPCs are *passive*, i.e. they represent a state change in the system, but do not cause

[1] `http://www.future-store.org/`

it (e.g. they do not provide decisions, but represent decisions taken). Events trigger functions, which are *active* elements that represent the actual work and again raise events upon completion. Connectors are used to *join and split* control flow, represented by edges in the EPC graph. An EPC starts and ends with one or more events, process control flow itself strictly follows an *alternating sequence of functions and events*, possibly with connectors specifying the kind of control flow join and split in between.

EPCs are in strong contrast to other established process languages such as BPEL or BPMN, which are rather *Web service centric* and do not enforce to use events as an integral part already at the modeling level. In this report, we therefore use EPCs as the basis for integration of process logic with event-based systems, building a common view that integrates process and event-logic.

Consequently, this report is organized as follows: in Section 2 we give an overview on the high-level architecture of the proposed system and explain how events and complex event processing can be added to the process level. In Section 3 we provide an overview of related work and discuss state-of-the-art in related research areas today. Section 4 follows, and introduces our scenario and the running example we use throughout the report. This is followed by an in-depth, stepwise discussion of our proposed methodology describing the usage of a service-oriented and event-driven architecture. Section 6 provides an complete walk-through over all steps and with an in-dept discussion of all source files included. Section 7 concludes this report and gives an outlook on future work.

2 Architecture

Figure 1 depicts the overall architecture of the proposed system: Workflow technology traditionally distinguishes between specification and execution layers, an approach we follow in our architecture. To specify the workflow, designers use EPCs and refine the events from the workflow definition using a complex event description language (right-hand side of Figure 1). Since the process itself is modeled on an abstract level, it must be possible to define complex events on the same abstract level in order to make it possible for the domain expert to define the process and the complex events himself. So the complete specification can be done by the domain expert who has the expert knowledge in the domain the process is defined for. That specification has a defined structure in order to allow a good automated support for transforming it to the execution layer. However the generated artifacts (the BPEL workflows for the workflow engine and the CEP statements for the CEP system) are not executable directly because some execution information about the IT infrastructure has to be added manually by an IT expert. After adding the execution information the system can start working by sending event notifications from the CEP engine to workflow instances running in the

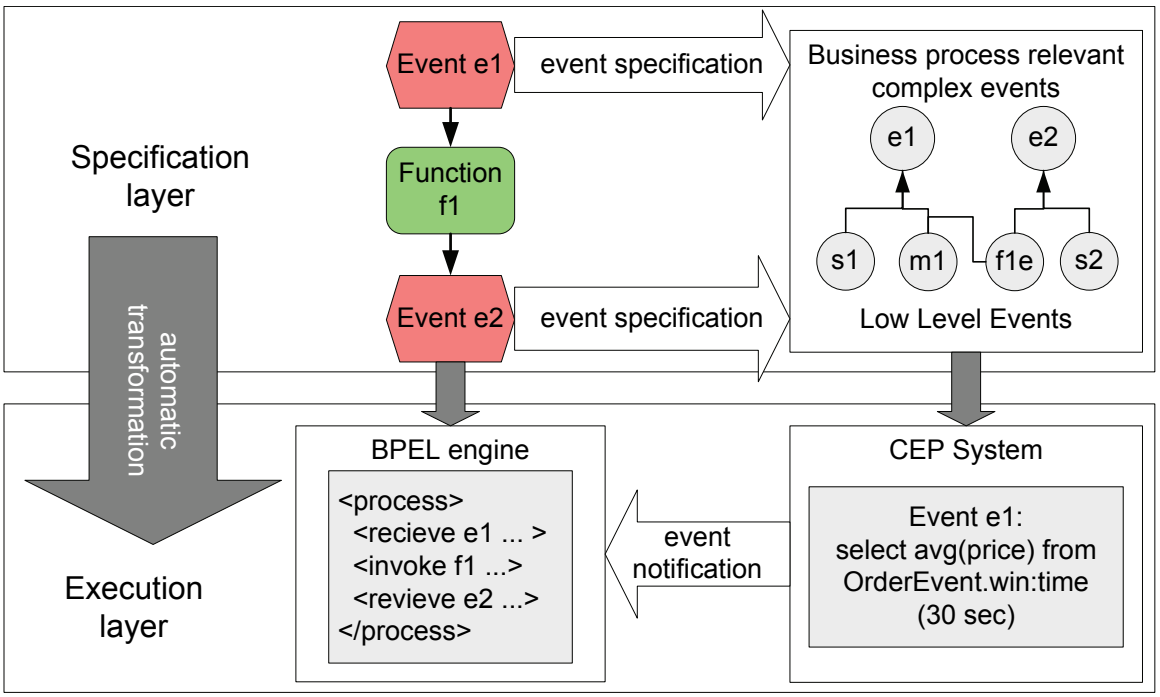


Figure 1: System overview

BPEL engine. An event can that trigger the instantiation of a new workflow or it can be consumed by an already running workflow.

3 Related Work

Common Base Event (CBE) [Bri04] is an XML-based standard that defines the structure and content of events coming from disparate sources of enterprize systems in a consistent and common format. Logging, tracing, management and business events are described consistently over different systems that emit them, allowing for easier automated situation handling and correlation by providing a common data structure to work with. In our scenario, CBE significantly lowers efforts at the level of the Event-Processing Engine; it reliefs us from the burden of writing data transformation logic for each different event source and format, effectively leading to a much cleaner design and implementation of the overall system. In cases were event sources do not produce events in CBE form, adapter logic needs to be written, but this logic does not transform the data to another proprietary format but to a standard format that may be reused and feed into other event-consuming systems as well.

Many works are available on combining Service-Oriented Architecture and Event-Driven

Architecture (e.g. IBM: [Mar06], HP: [WALD07], Academic: [LC08]). In summary, they state that it is important to combine these architecture styles. They also state that it is difficult to develop applications for such a combined architecture. As a step towards solving that problem this report we presents in our knowledge the first *methodology* (SOEDA) supporting the *development* of such service-oriented event-driven applications. Of course there exist many island solutions for parts of the methodology. Because of that SOEDA tried to build on available solutions where possible.

Event-driven Process Chains (EPCs) offer a superset of possible process instantiation patterns in comparison to BPEL processes [DM08]. Therefore, all approaches mapping EPCs to BPEL have to constrain the semantics of start events to the semantics offered by BPEL.

Workflow engines produce events to signal what currently happens in the process [IBM08]. We present in [KKS+06] a generalized BPEL event model, which abstracts from engine specific events and provides general events. These events in turn can be used to implement arbitrary applications as shown in [KKL07]. Using the events in a process model it is possible to react on state changes in the current process instance.

Patterns for evaluating event specification languages are presented in [BDG07]. In this report, we do not propose a new event definition language, but show how events can be used to drive control-flow of a process.

3.1 Data Stream Processing

There exist many systems on the market for handling data streams. All of the following systems could be used for realizing the example scenarios and for the processing of complex events:

- Aurora (Brandeis University, Brown University and MIT)
- Borealis (Brandeis University, Brown University and MIT)
- STREAM (Stanford University)
- Telegraph (UC Berkeley)
- SASE (UC Berkeley/UMass Amherst)

The disadvantage in using such a data stream system is that process control flow is scattered around the whole data steam processing system. That means that each services in the process chain knows themselves to which service they have to pass their output data to. This is the complete opposite of the idea of service-oriented business processes, where the process coordinates the services and the services themselves are “dumb” and do

not know anything about the control flow. This has the advantage of central monitoring, execution control of the ongoing processes.

3.2 Process Modeling

3.2.1 EPC

To execute an EPC model on a workflow machine, there are two general ways: (i) give the EPC an execution semantics or (ii) define a mapping to a workflow language with an execution semantics.

The inherent problem of giving a semantics to an EPC is the OR-join: it is hard to decide how long an OR-join should block the control flow. It is especially hard in processes containing cycles [ADK02, Kin06]. Most discussions debate whether this should be resolved through local knowledge, i.e. by introducing additional arcs in the model or "negative control tokens" that make it possible to unblock and evaluate the join condition when it is clear that no more tokens can arrive. On the other hand, execution engines have been proposed that decide—by looking at the global state of the process—if a join can be unblocked since no more tokens will arrive on the input arcs. An extensive presentation and comparison of the proposed semantics can be found in [Men07, MA07, AH05, Weh07, WEAH05].

By mapping the EPC to a workflow language with a defined execution semantics, the EPC is also given an execution semantics: the semantics of the workflow language used. The Web Services Business Process Execution Language (BPEL, [OAS07]) is the current de-facto standard for workflow execution. Thus, the current approach in SOEDA is to map EPCs to BPEL.

3.2.2 BPMN

The Business Process Modeling Notation (BPMN, [Obj08]) is a graphical notation to model business processes. It exists in parallel to the EPC notation, but offers different control flow constructs. In [DGB07] the Business Event Modeling Notation (BEMN) is introduced. It is a language to specify events using a graphical notation inspired by BPMN. However, the link to an event infrastructure is not given.

3.2.3 BPEL

A BPEL workflow defines an orchestration of Web services and consists of structured and basic activities. The actual business functions are not implemented by BPEL itself,

but by Web services, where the business data is sent to and received from using messages (events are represented as messages too). Hence, the most important basic activities are `invoke` and `receive`. An `invoke` activity is used to send a message to a Web service. A `receive` activity is used to receive a message. The structured activity `pick` realizes a one-out-of-m choice of messages to receive: the first arrived message wins and the other messages are ignored at that activity. Control flow itself is either modeled block-structured using `if` and `sequence` activities or using graph-based constructs realized by the `flow` activity. In a `flow` activity, activities are connected using links. Each link carries a transition condition. If the transition condition evaluates to true, the link status is true and false in the other case. Each activity carries a join condition. As soon as the status of all incoming links is clear, the join condition is evaluated. If it evaluates to true, the activity is executed and subsequently the transition conditions of all outgoing links evaluated. If the join condition evaluates to false, the activity is skipped and the status of all outgoing links is set to false. In all cases, the successors are visited. This behavior is called "Dead-path Elimination", which is formally defined in [LR00], specified for BPEL in [OAS07] and explained in detail in [CKLW03].

A BPEL process does not need to be executable by itself. The BPEL specification offers to model *abstract processes*, which may hide operational details. So called opaque activities can be used to model left-out behavior. Each abstract process is assigned to an abstract process profile. There are two abstract process profiles defined in the BPEL specification: one for specifying *observable behavior* and one for specifying *templates*. The allowed changes to go from an abstract BPEL process to an executable BPEL process are described by the so called *executable completion*. For example, new activities may only be added at marked places in the control flow and not anywhere in the process.

3.3 Transformation of EPC to BPEL

In [MLZ08], different transformation strategies to transform EPCs to BPEL are presented. The presented transformations consider start events, functions, connectors and end events only. The transformations only distinguishes between `empty`, `terminate` and other basic activities. An `empty` activity does nothing by itself and the `terminate` activity terminates the whole process regardless of any activities running in parallel. In all transformations following properties hold: Intermediate events are ignored. Each start event is mapped to a general basic activity, end events are mapped to a `terminate` activity. Functions are mapped to general basic activities. The different transformation strategies differ in the transformation of connectors. In the strategy "Element-Preservation", each element of the EPC is mapped to a BPEL element: Connectors get `empty` activities and the control-flow edges are mapped to control links. As pre-condition, the input EPC has to be acyclic. In the strategy "Element-Minimization", the joining and split behavior of connectors is put in the respective preceding or subsequent function. Thus, the number

of activities is reduced, but the join and split behavior is preserved. In the strategy "Structure-Identification", the structures `if`, `sequence`, `while` and parallel (modeled by a `flow` activity without any links) are identified. As pre-condition, the input EPC has to be structured as defined in [KHB00]. In the strategy "Structure-Maximization", as much as possible structured are identified. If that is not possible, the remaining control-flow connectors are transformed to control links. The input EPC may only have loops with one entry and one exit. In the strategy "Event-Condition-Action-Rules", arbitrary EPCs are transformed: first, as much as possible structured are identified. If that is not possible, event-condition-action rules (ECA rules) are applied as presented in [ODBH06].

A taxonomy for model transformations is provided in [MG06]. The most important criteria is the distinction between horizontal and vertical transformation. In a horizontal transformation, the model is transformed to another model on the same abstraction level. In a vertical transformation, the target model resides on a different level of abstraction. In general, transformations of unmarked EPCs are horizontal transformations, whereas transformations of marked EPCs are vertical transformations [SKI08]. A general overview of all available transformations from EPCs to BPEL and their classification using the taxonomy of [MG06] is given in [SKI08].

The transformation implemented in the ARIS SOA Designer follows the "Structure-Identification" strategy [SI07]. Functions are transformed to `invoke` activities. Intermediate events are dropped. A single start event is mapped to a `receive`. If there are multiple start events, they are mapped depending on the subsequent connector c: If c is of type *and*, the start events are mapped to `receive` activities nested in a `flow`. If c is of type *xor*, the start events are mapped to a `pick` activity. Connector type *or* is not supported. End events are transformed to `invoke` activities.

The work of [VVK08] applies the "Structure-Maximization" strategy to graphs with nodes and edges. The main characteristic of the algorithm is that a local change of the graph also leads to a local change in the resulting BPEL process. Since the source meta model is different from EPCs and does not distinguish functions and events, the mapping does not deal with events. The work presented in [ODBH06] presents a mapping of BPMN to BPEL. It follows the "Event-Condition-Action-Rules" strategy. The works of [VVL08] and [VVK08] present how to detect single-entry-single-exit (SESE) fragments in a business process and how to form a process structure tree out of this information. The work of [GB08] extends the work and presents a technique for determining the type of a SESE region. Sample types of a SESE region are among others "sequence" or "repeat-until". Using the work of [VVK08] and [GB08] together, each arbitrary graph can be decomposed into SESE regions annotated with their type.

In [ZM05], the transformation from EPCs to BPEL is based on the Element-Preservation strategy, but does not ignore intermediate events. The work distinguishes between data-based XOR-splits and event-based XOR-splits. If a XOR-split is data-based, the

each subsequent event is transformed to a transition conditions on the respective link. If a XOR-split is event-based, the split is transformed to a `pick` activity, having each event as one branch. For OR-splits, they only support data-based splits. In summary, they explicitly distinguish between data-based events and incoming events: data-based events are triggered by a change of process-internal data. If a function has changed something externally, an event happened which in turn triggered the incoming event.

In contrast to other approaches, both [ZM05] and [KUL06] generated BPEL processes capturing more information than the control-flow of the input EPC: variables, partner links and operations are also generated. However, the generated processes still do not contain enough information to be fully executable: they are still abstract (and not executable) BPEL processes.

The work presented in [LLSG08] focuses on user-interface generation out of processes described in EPCs. Functions of EPCs are marked with a type to distinguish between functions executed by Web services and functions executed by humans. Each function can be decomposed into a sub-process modeled by an EPC again. Out of a EPC describing human tasks only, a user-interface is generated. The start and end events are used to trigger the display of the UI and to signal the finish of the task. The synchronization with the main process is done by a server emitting and receiving these events. The EPC itself is also executed by this server. To model asynchronous communication, markers are introduced to distinguish Web service reception, UI events and other EPC events. The work leaves open how these Web service events are produced. Furthermore, it is not described how the server deals with "other EPC events".

Handling unstructured loops is an important aspect when transforming business processes. In [ZHB+06] it is shown how arbitrary cycles can be transformed to structured cycles "with controlled code complexity" without applying ECA rules. Current transformation approaches do not use this idea, but use Event-Condition-Action-Rules to transform unstructured loops into BPEL.

4 Scenarios and Use-Cases

Throughout the report following two very contrary scenarios are used. The first scenario is from one of the main fields of application of EPCs: order processing. Fig. 2 shows an excerpt of the order processing EPC presented in [STA05].

The second scenario comes from pervasive environments. Here many sensors and actors are deployed in the environment and can be accessed for observing or manipulating the state of the environment. Sensors produce big amount of data (in form of data streams) due to their sampling rate and their quantity. Fig. 3 shows as examples what can be done with that kind of sensor data three simple processes controlling the room temperature

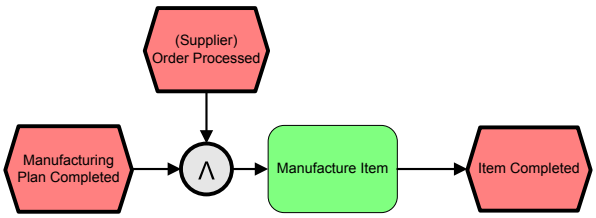


Figure 2: Detailed excerpt of the business process "order processing" [STA05]

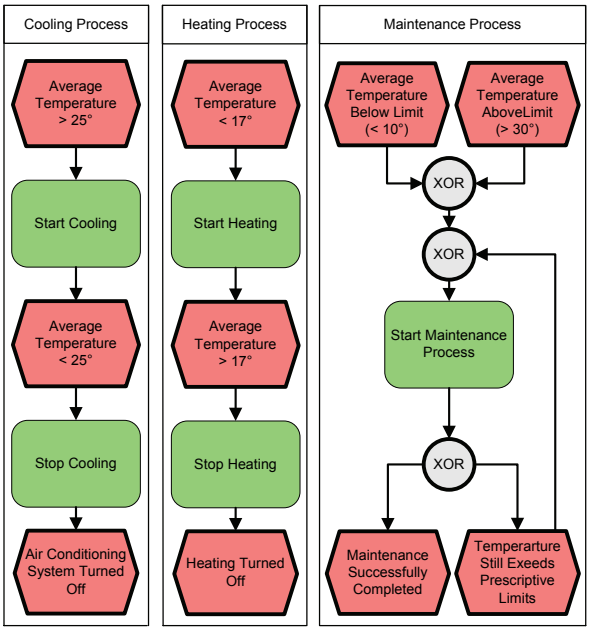


Figure 3: Exemplary temperature control processes based on sensor observation

to keep it between optimal values. With normal workflow systems such control loops systems cannot be managed due to the amount of messages per second and the data stream handling possibilities. Using the concepts described in this report BPM is enabled to manage also processes like these. The stream handling and message reduction is done by complex event processing and so the workflow system can concentrate on handling the control flow and only receives single process relevant messages. The example shown here is of course not of practical use, it is only a simple example that would be realized in practice with a normal climate control unit. However for example in business processes the cold chain could be observed in order to meet rules defined for the handling of food or a production could be observed for quality prediction of the produced products.

4.1 Modeling Use-Cases

The EPC specification can be applied in following two different use-cases. The first use-case is valid for newly modeled processes. The second one can be applied if already a lot processes are modeled based on the methodology presented in that report.

1. The standard use-case: A domain expert describes the processes he wants to implement as EPCs. Then using that EPCs the complex events needed for the process execution are derived from the EPC events. How the complex events are observed has to be implemented by an IT expert.

2. The second use-case builds up on the first. If already many EPCs have been defined in an company there exist as direct consequence a lot of complex event definitions. That definitions can be collected in an repository and then if the domain expert is modeling an new process he can use that already existing definitions. Which means that the modeling of new processes is done as follows: When the domain expert needs an event in his EPC he searches in the event-repository if already an event is defined for that purpose and than uses that event in the process. That has the advantage that the processes are waved together through the events. For example the order processing EPC has the last event "item completed". Now the process modeler wants to specify the item shipping then he can use the "item completed" event as start event of the new process. Furthermore in the shipping process the temperature observation can be reused e.g. for monitoring of a cooling chain. In this case only the values of the average temperature have to be changed in the event definitions. All other work can be reused.

5 Concept of Mapping Specification to Execution Artifacts

The goal of the concept is to provide a automatic transformation from the EPC specification to the needed artifacts for execution. The EPC models are usually not detailed enough for a direct execution. Because of that EPC modeling is called semantic business process modeling. The execution details are not modeled for purpose that the processes are better understandable. They are used mainly for discussion between people with is comparable to the specification of an software system. Therefore, the automatic transformation cannot produce directly executable artifacts. The artifacts have to be refined later by an IT-expert for execution.

For the explanation, we have to define the syntax of EPCs, BPEL and complex events. Definition 1 presents the syntax of EPCs. The definition is based on [Kin06]. We extended

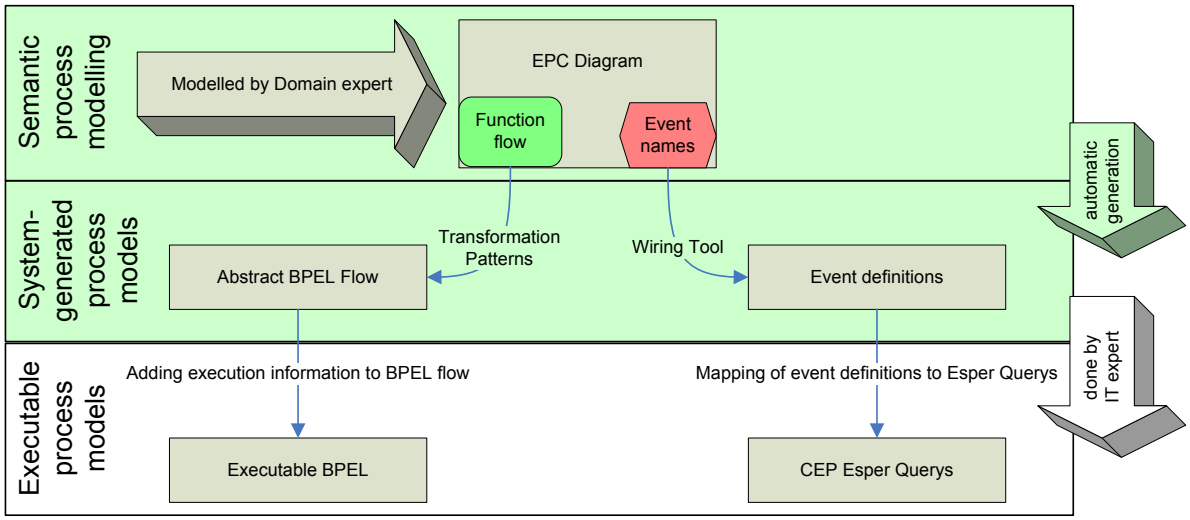


Figure 4: System overview

the labeling function ι to also label the events and functions. Formal definitions of the properties of EPCs have been omitted and can be found in [Kin06].

Definition 1: EPC Syntax
An EPC M is a tuple $(E_{EPC}, F_{EPC}, C_{EPC}, A_{EPC}, \Sigma, \iota)$ having following properties:

- E_{EPC} *denotes the set of events.*
- F_{EPC} *denotes the set of functions.*
- C_{EPC} *denotes the set of connectors.*
- *The sets E_{EPC}, F_{EPC} and C_{EPC} are disjoint.*
- $A_{EPC} \subset (E_{EPC} \cup F_{EPC} \cup C_{EPC}) \times (E_{EPC} \cup F_{EPC} \cup C_{EPC})$ *is a binary relation denoting the control flow arcs connecting the events, functions and connectors.*
- $(E_{EPC} \cup F_{EPC} \cup C_{EPC}, A_{EPC})$ *forms a graph which at least one source and at least one sink. A source is a node without incoming arcs and a sink is a node without outgoing arcs.*
- *There is no control flow cycle consisting of connector nodes only.*
- *All events have at most one incoming and one outgoing control flow arc.*
- *All functions have to take one incoming and one outgoing control flow arc.*
- *Functions and events have to alternate.*
- Σ *denotes a set of labels.*
- $\iota : (E_{EPC} \cup F_{EPC} \cup C_{EPC}) \rightarrow \{and, or, xor\} \cup \Sigma$ *is the labeling function. A label of a connector is either and, or or xor. A label of an event or a function is an element out of Σ. A connector labeled with and having more than one outgoing control flow arc is called* AND-split*. A connector labeled with or having more than one outgoing control flow arc is called* OR-split*. A connector labeled with xor having more than one outgoing control flow arc is called* XOR-split*. A connector labeled*

with and having more than one incoming control flow arc is called AND-join. *A connector labeled with or having more than one incoming control flow arc is called* OR-join. *A connector labeled with xor having more than one incoming control flow arc is called* XOR-join.

The BPEL specification specifies the serialization format of a BPEL process as well as the semantics of this serialization. The serialization is block-structured, but BPEL itself offers both, block-structured and graph-based specification of workflows [KMWL08]. A complete abstract syntax for BPEL is provided in [KML08]. Since the transformation maps to a subset of the BPEL syntax, we present this subset in Definition 2. Besides formalizing the required activities only, the simplified BPEL syntax skips the declaration of variables, operations and port types.

Definition 2: Simplified BPEL Syntax
A BPEL workflow B is a tuple ($\mathcal{A}$, $\mathsf{type}_{\mathcal{A}}$, $\mathcal{N}$, $\mathsf{name}_{\mathcal{A}}$, $\mathcal{E}$, HR, $\mathcal{L}$, tc, LR, jc, $\mathsf{assignCopy}$, $\mathcal{V}$, $\mathsf{outputVar}$, $\mathcal{PL}$, $\mathsf{partnerLink}_{\mathcal{CO}}$, $\mathcal{O}$, $\mathsf{name}_{\mathcal{O}}$, $\mathsf{operation}_{\mathcal{CO}}$) *having following properties:*

- $\mathcal{A}$ *denotes the set of all activities.*
- *The function* $\mathsf{type}_{\mathcal{A}} : \mathcal{A} \rightarrow \{scope, flow, while, repeatUntil, pick, receive, empty, assign, opaqueActivity\}$ *assigns a type to an activity.*
- *For simplicity,* $\mathcal{A}_t$ *denotes all activities of type* t. *For example,* $\mathcal{A}_{empty}$ *contains all activities* $a \in \mathcal{A}$, *where* $\mathsf{type}_{\mathcal{A}}(a) = empty$.
- $\mathcal{N}$ *denotes the set of all NCNames.*
- *The function* $\mathsf{name}_{\mathcal{A}} : \mathcal{A} \rightarrow \mathcal{N}$ *assigns a name to an activity.*
- $\mathcal{E}$ *denotes the set of message events. A message event is used to denote the incoming message for each branch of a* `pick` *activity.*
- $\mathcal{CO}_{receiving} = \mathcal{A}_{receive} \cup \mathcal{E}$ *is the set of all constructs for receiving messages. These are* `invoke`*-activities,* `receive` *activities and* `onMessage` *constructs.* `onMessage` *constructs are children of* `pick` *activities.*
- $\mathcal{CO}_{message} = \mathcal{CO}_{receiving} \cup \mathcal{A}_{invoke}$ *is the set of all constructs for sending and receiving messages.*
- *The nesting of the activities is denoted by the hierarchy relation* $\mathsf{HR} \subseteq \mathcal{A} \times (\mathcal{E} \cup \{\bot\}) \times \mathcal{A}$. (a_1, x, a_2) *denotes that* a_1 *is connected to* a_2 *by the label* x. *In case of branches of a* `pick` *activity, an event out of* $\mathcal{E}$ *is used as label. Hence,* x *denotes the received message. In case of a* `flow`*, the empty label (*$\bot$*) is used.*
- *The set* $\mathcal{L}$ *is the set of links. A link is used in a* `flow` *activity to define the control flow graph.*
- *The set of* $\mathcal{C}$ *is the set of all expressions returning a Boolean value.*
- *The function* $\mathsf{tc} : \mathcal{L} \rightarrow \mathcal{C} \cup \{\bot\}$ *assigns a transition condition to a link.* $\bot$ *denotes the default transition condition* `true`.
- *The relation* $\mathsf{LR} \subseteq \mathcal{A} \times \mathcal{L} \times \mathcal{A}$ *denotes the link relation including the transition condition.*

- *The function* $\mathsf{jc} : \mathcal{A} \rightarrow \text{Ex} \cup \{\bot\}$ *assigns a join condition to an activity.* $\bot$ *denotes the default join condition, which is a logical OR over all incoming links.*
- *The function* $\mathsf{assignCopy} : \mathcal{A}_{assign} \rightarrow \text{Ex}$ *assigns an assignment expressions to an* `assign` *activity. This is a simplification of the whole capabilities of BPEL's* `assign` *activity, which can contain multiple* `copy` *statements and other types of assignment.* Ex *denotes the set of all expressions. We use an E4X-based syntax. E4X provides XML processing for JavaScript [Ecm05].*
- $\mathcal{V}$ *denotes the set of all variables*
- *The function* $\mathsf{outputVar} : \mathcal{CO}_{receiving} \rightarrow \mathcal{V}$ *assigns an output variable to a receiving construct.*
- $\mathcal{PL}$ *denotes the set of partner links. A partner link is used to establish the relation between a BPEL process and the partner Web service. In the case of the* `receive` *activity, the partner link is used to denote the WSDL port type to be used to receive the message*
- *The function* $\mathsf{partnerLink}_{\mathcal{CO}} : \mathcal{CO}_{message} \rightarrow \mathcal{PL}$ *is a function assigning a partner link to a message construct.*
- *The set* $\mathcal{O}$ *is the set of WSDL operations.*
- *The function* $\mathsf{name}_{\mathcal{O}} : \mathcal{O} \rightarrow \mathcal{N}$ *assigns a name to an operation.*
- *The function* $\mathsf{operation}_{\mathcal{CO}} : \mathcal{CO}_{message} \rightarrow \mathcal{O}$ *assigns an operation to a message construct.*

Definition 3: Event Sets

- *Set of complex event names* $C\text{Æ}_N$ *consists of all names of complex events available in the system. They are identical to the names of the EPC events.*
- *Set of complex event definitions* $C\text{Æ}_D$ *consists of descriptions for the deduction of all complex events out of event adapters and conditions.*
- *Set of complex event queries* $C\text{Æ}_Q(Esper)$ *contains concrete implementations of all* $\text{æ}_d \in C\text{Æ}_D$ *for a specific complex event processing system, e.g. Esper.*
- *Set of event sources* E_S *contains all sensors or other data producers of raw source events.*
- *Set of event adapters* E_A *contains all services that make the event sources technically accessible. An event adapter* $e_a \in E_A$ *is similar to a web-service that is the other way round which means it produced output only. Is used for getting messages from an event source awaited by the receive of a BPEL flow.*

The complete methodology how to map the specification to the execution artifacts is presented in Fig. 4. Following subsections describe each step of the methodology in detail. The first and second step have to be done in sequential order. The third and fourth step can be done in parallel. The fifth and last step can be done after all other steps have been finished.

5.1 Step 1: Process Definition

The first step is the creation of the EPC specification by the domain experts. We propose EPC as high level process definition language in that case because it provides events as a integral part. That has the big advantage in contrast to other modeling languages like for example BPMN that a alternation between event based processing and workflow based control flow is prescribed by the process model. Of course the EPC process definitions are not executable as workflows directly because they require a lot of refinement for that. In a very short summary all the EPCs have to specify is the control flow between called services (EPC functions) and how to observe when the functions are completed or new environmental states appear that have to be treated (EPC events).

5.2 Step 2: Complex Event Extraction

The second step is the extraction of all event names from the EPC specification. That event names are inserted as complex event expression names to $C\text{Æ}_N$, $C\text{Æ}_N{=}E_{EPC}$. That events are the complex events that have to be deducted from the available event sources. Also that events are the only events the process has to be notified of later in the execution phase.

5.3 Step 3: Mapping Process Definition to Workflow Template

The third step is the automatic EPC function flow to abstract BPEL flow transformation. The algorithm and patterns needed for that are described in Section 5.3.1.

The basic transformation idea is that EPC functions are mapped to `opaque` activities[2] in BPEL and are later possibly refined to `invoke` activities calling the specified EPC modules. The EPC events are mapped to BPEL `receive` activities that wait for messages notifying the occurrence of the specific event.

An main goal of that transformation is that both flows have a similar structure to enable domain expert understand the the executed workflows due to the similarity to the EPC they modeled as specification. This allows business activity monitoring to be done without additional efforts directly in the BPEL flow. To achieve that goal, we follow the "Element-Preservation" strategy [MLZ08]. The difference of our transformation to the presented strategy is the treatment of events: Instead of ignoring events, we map EPC events to message receiving constructs. Functions are mapped to `opaque` activities. Each

[2]The BPEL activity representing an opaque activity is named `opaqueActivty`. While writing "`opaqueActivity` activity" is consistent to write "`invoke` activity", we prefer to write "`opaque` activity" to increase readability.

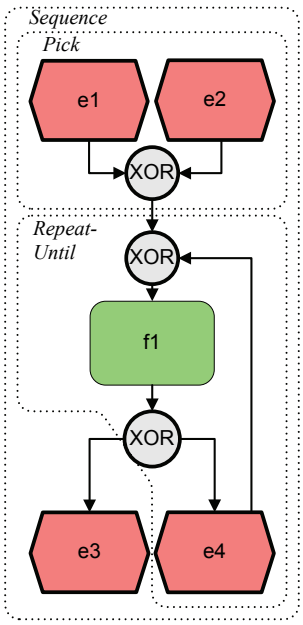


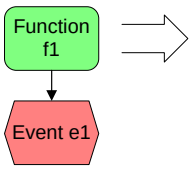
Figure 5: Process Structure Tree of the Maintenance Process

`opaque` activity has to manually be refined in the executable completion. The message receiving constructs are directly executable.

5.3.1 Transformation Algorithm

Graphs are enforced to be acyclic in BPEL workflows. To express loops, the `repeatUntil` and `while` activities are offered. In addition, a choice between multiple incoming messages has to be modeled by a `pick` activity. In contrast to a BPEL workflow, an EPCs is allowed to be a cyclic graph. To map EPCs to BPEL workflows, we keep the original graph structure as much as possible in the resulting BPEL workflow. To detect the structures which have to be mapped to the block-structures offered by BPEL, we combine the techniques presented in [VVK08] and [GB08]. [VVK08] presents a technique to identify the structure of a graph, called "Process Structure Tree" (PST). The structures of the tree can be classified as types such as repeat-until-loop by using the technique presented in [GB08]. Fig. 5 presents the process structure tree for the maintenance process. In the case a structure cannot directly be expressed using BPEL constructs, we map this structure into an `opaque` activity. We use transformation patterns to transform identified structures. A transformation pattern consists of the source EPC structure and target BPEL structure.

The general idea is presented in Fig. 6: the function is transformed to an `opaque` activity and the event is transformed to a `receive`. Note that functions such as $\mathsf{type}_{\mathcal{A}}$ are



BPEL workflow $B =$
$(\mathcal{A}, \mathsf{type}_{\mathcal{A}}, \mathcal{N}, \mathsf{name}_{\mathcal{A}}, \emptyset, \mathsf{HR}, \mathcal{L}, \mathsf{tc}, \mathsf{LR}, \emptyset, \emptyset, \mathcal{V}, \mathsf{outputVar}, \mathcal{PL}, \mathsf{partnerLink}_{\mathcal{CO}}, \mathcal{O}, \mathsf{name}_{\mathcal{O}}, \mathsf{operation}_{\mathcal{CO}})$

$$
\begin{array}{lcl}
\mathcal{A} & = & \{\mathsf{process}, \mathsf{flow}, f1, e1\} \\
\mathsf{type}_{\mathcal{A}} & = & \{(f1, \mathit{opaqueActivity}), (e1, \mathit{receive}), \\
 & & (\mathsf{process}, \mathit{scope}), (\mathsf{flow}, \mathit{mathitflow})\} \\
\mathcal{N} & = & \{\mathit{FunctionF1}\} \\
\mathsf{name}_{\mathcal{A}} & = & \{(f1, \mathit{FunctionF1})\} \\
\mathsf{HR} & = & \{(\mathsf{process}, \bot, \mathsf{flow}), \\
 & & (\mathsf{flow}, \bot, f1), (\mathsf{flow}, \bot, e1)\} \\
\mathcal{L} & = & \{l1\}
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{tc} & = & \{(l1, \bot)\} \\
\mathsf{LR} & = & \{(f1, l1, e1)\} \\
\mathcal{V} & = & \{\mathit{Event1}\} \\
\mathsf{outputVar} & = & (e1, \mathit{Event1}) \\
\mathcal{PL} & = & \{\mathit{EventsPLT}\} \\
\mathsf{partnerLink}_{\mathcal{CO}} & = & \{(e1, \mathit{EventsPLT})\} \\
\mathcal{O} & = & \{\mathit{EventE1}\} \\
\mathsf{name}_{\mathcal{O}} & = & \{(\mathit{EventE1}, \mathit{EventE1})\} \\
\mathsf{operation}_{\mathcal{CO}} & = & \{(e1, \mathit{EventE1})\}
\end{array}
$$

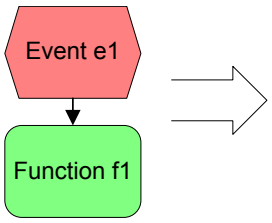
Figure 6: General idea of the transformation. The syntax of BPEL is expressed in using the simplified BPEL syntax

represented as a relation. The `opaque` activity and the event are connected using a link $l1$, which takes no transition condition ($\mathsf{tc}(l1) = \bot$). The name of the link is made unique throughout the BPEL process. The name of the BPEL `opaque` activity is the camel case version of the name of the EPC function ($\mathsf{name}_{\mathcal{A}}(a) = \mathsf{camelCase}(\iota(f))$). The `receive` does not get a name assigned. The name of the partner link is always `EventsPLT`. The name of the operation is the camel case version of the event ($\mathsf{name}_{\mathcal{O}}(o) = \mathsf{camelCase}(\iota(e))$). The name of the output variable is `Event<i>`, where `<i>` is replaced by an unique number ($\mathsf{operation}_{\mathcal{CO}}(v) = \mathsf{uniqueEventVariableName}(\mathsf{U})$). In the case, `onMessage` branches of `pick` activities are generated, the `partnerLink`, `operation` and `variable` fields are generated the same way as in the case of the `receive` activity.

In the following, we present the transformation patterns used to transform the EPC. In case a structure cannot be mapped by a pattern, the structure is collapsed to an `opaque` activity, which has to be defined manually after the mapping.

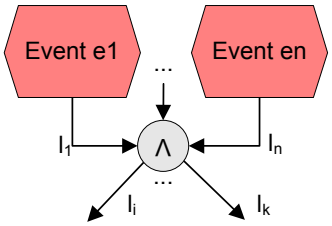
Fig. 7 presents the transformation for the most likely case of one start event. The start event is mapped to a `receive` activity. Fig. 8 presents the transformation of multiple start events which are joined via an AND join. The semantics of that fragment is that all of the events have to occur. Therefore, we map each event to a `receive` activity without any incoming links. Thus, the `receive`s are executed concurrently. Fig. 9 presents the transformation of multiple start events which are joined via an XOR join. The semantics of that fragment is that one of the events have to occur. Thus, we map that fragment to a `pick` activity, where each event is put as one `onMessage` branch. Each `onMessage` branch contains one empty activity which is connected to an `empty` activity. That `empty` activity is the mapped XOR connector and contains the mapped outgoing control-flow arcs of the XOR connector.

Regarding the mapping of intermediate events, we demand that they are immediately preceded by a function, an AND split or a XOR split. If an event is preceded by a



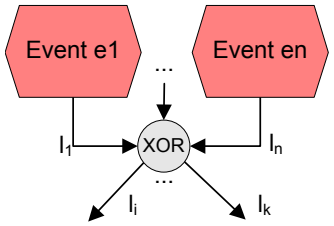

```
<receive partnerLink="EventsPL"
  operation="EventE1" variable="Event1" >
  <sources><source linkName="l"/></sources>
</receive>
<opaqueActivity name="FunctionF1">
  <targets><target linkName="l"/></targets>
</invoke>
```

Figure 7: Mapping of a single start event



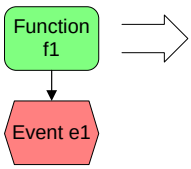

```
<receive partnerLink="EventsPL"
    operation="EventE1" variable="Event1">
    <sources><source linkName="l1"/></sources>
</receive>
...
<receive partnerLink="EventsPL"
    operation="EventEn" variable="Eventn">
    <sources><source linkName="ln"/></sources>
</receive>
<empty>
  <sources>
    <source linkName="li">
    ...
    <source linkName="lk">
  </sources>
</empty>
```

Figure 8: Mapping of multiple start events




```
<pick>
  <sources><source linkName="l"/></sources>
  <onMessage partnerLink="EventsPL"
    operation="EventE1" variable="Event1">
    <empty />
  </onMessage>
  ...
  <onMessage artnerLink="EventsPL"
    operation="EventEn" variable="Eventn">
    <empty />
  </onMessage>
</pick>
<empty>
  <targets><target name="l" /></targets>
  <sources>
    <source linkName="li">
    ...
    <source linkName="lk">
  </sources>
</empty>
```

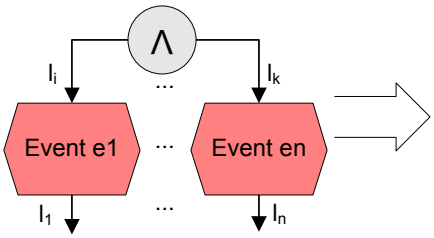
Figure 9: Mapping of exclusive start events



BPEL workflow $B =$
$(\mathcal{A}, \mathsf{type}_{\mathcal{A}}, \mathcal{N}, \mathsf{name}_{\mathcal{A}}, \emptyset, \mathsf{HR}, \mathcal{L}, \mathsf{tc}, \mathsf{LR}, \emptyset, \emptyset, \mathcal{V}, \mathsf{outputVar}, \mathcal{PL}, \mathsf{partnerLink}_{\mathcal{CO}}, \mathcal{O}, \mathsf{name}_{\mathcal{O}}, \mathsf{operation}_{\mathcal{CO}})$

$$
\begin{array}{lcl}
\mathcal{A} & = & \{\mathsf{process}, \mathsf{flow}, f1, e1\} \\
\mathsf{type}_{\mathcal{A}} & = & \{(f1, \mathit{opaqueActivity}), (e1, \mathit{receive}), \\
 & & (\mathsf{process}, \mathit{scope}), (\mathsf{flow}, \mathit{mathitflow})\} \\
\mathcal{N} & = & \{\mathit{FunctionF1}\} \\
\mathsf{name}_{\mathcal{A}} & = & \{(f1, \mathit{FunctionF1})\} \\
\mathsf{HR} & = & \{(\mathsf{process}, \bot, \mathsf{flow}), \\
 & & (\mathsf{flow}, \bot, f1), (\mathsf{flow}, \bot, e1)\} \\
\mathcal{L} & = & \{l1\}
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{tc} & = & \{(l1, \bot)\} \\
\mathsf{LR} & = & \{(f1, l1, e1)\} \\
\mathcal{V} & = & \{\mathit{Event1}\} \\
\mathsf{outputVar} & = & (e1, \mathit{Event1}) \\
\mathcal{PL} & = & \{\mathit{EventsPLT}\} \\
\mathsf{partnerLink}_{\mathcal{CO}} & = & \{(e1, \mathit{EventsPLT})\} \\
\mathcal{O} & = & \{\mathit{EventE1}\} \\
\mathsf{name}_{\mathcal{O}} & = & \{(\mathit{EventE1}, \mathit{EventE1})\} \\
\mathsf{operation}_{\mathcal{CO}} & = & \{(e1, \mathit{EventE1})\}
\end{array}
$$

Figure 10: Mapping of an event preceded by a function

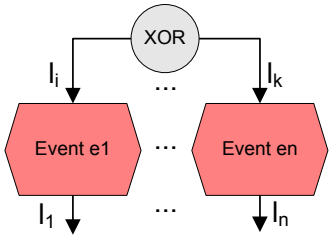



```
<empty name="AND">
  <sources>
    <source linkName="l_i">
    ...
    <source linkName="l_k">
  </sources>
</empty>
<receive partnerLink="EventsPL"
  operation="EventE1" variable="Event1">
  <targets><target linkName="l_i"></targets>
  <sources><source linkName="l_1"/></sources>
</receive>
...
<receive partnerLink="EventsPL"
  operation="EventEn" variable="Eventn">
  <targets><target linkName="l_k"></targets>
  <sources><source linkName="l_n"/></sources>
</receive>
```

Figure 11: Mapping of an event preceded by an AND split

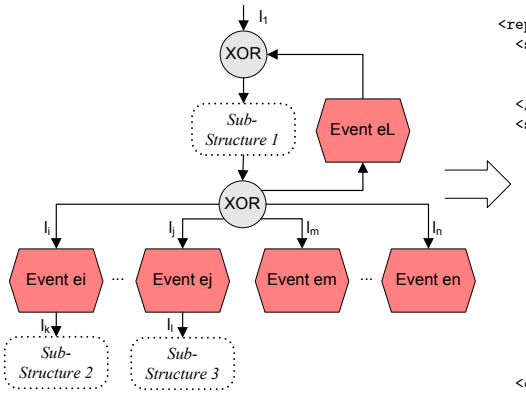
single function, the event is transformed to a `receive` activity as shown in Fig. 10. If the events are preceded by an AND split, the fragment is transformed as shown in Fig. 11: the AND is transformed to an `empty` activity, the events are transformed to `receive` activities and connected via links. If the events are preceded by an XOR split, the fragment is transformed to a `pick` activity as shown in Fig. 12: each event gets an `onMessage` branch in the `pick` activity. Each `onMessage` branch consists of an `empty` activity, where the outgoing links of the respective events are connected to.

Repeat-until loops are transformed as shown in Fig. 13. To indicate whether the loop has to be run, the indicator variable `doRepeatUntilLoop1` is used. Three types of loop events are distinguished: (i) the loop event, (ii) events without successors and (iii) events with successors. Since these events are mutually exclusive, they can be catched in one `pick` activity. If the event is the loop event, the indicator variable is set to `true`. If the




```
<pick>
  <onMessage partnerLink="EventsPLT" operation="EventE1"
   variable="EventN">
    <empty>
      <sources><source linkName="l1"/></sources>
    </empty>
  </onMessage>
  ...
  <onMessage partnerLink="EventsPLT" operation="EventEn"
   variable="EventN'">
    <empty>
      <sources><source linkName="ln"/></sources>
    </empty>
  </onMessage>
</pick>
```

Figure 12: Mapping of exclusive events

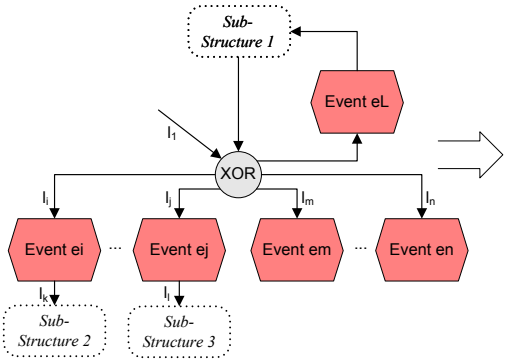



```
<repeatUntil> <targets><target name="l1"/></targets>
  <sources>
    <source name="lk" transitionCondition="EventEiHappened" />
    ...
  </sources>
  <sequence>
    ...mapping of sub-structure 1...
    <pick>
      <onMessage partnerLink="EventsPLT" operation="EventEL" variable="EventL">
        <assign>doRepeatUntil = true</assign>
      </onMessage>
      <onMessage partnerLink="EventsPLT" operation="EventEi" variable="Event1">
        <assign>doRepeatUntil1 = false; EventEiHappend=true</assign>
      </onMessage>
      ...
      <onMessage partnerLink="EventsPLT" operation="EventEn" variable="Eventn">
        <assign>doRepeatUntil1 = false</assign>
      </onMessage> </pick> </sequence>
  <condition>doRepeatUntil1</condition> </repeatUntil>
```

Figure 13: Mapping of a repeat-until loop

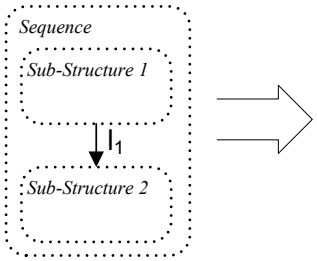
event is an event is an event without successors, the indicator variable is set to `false`. If the event is an event is an event with successors, the indicator variable is set to `false` and a Boolean variable is set to `true`. This variable is used as transition condition on the `link` connecting the `repeatUntil` with the mapped sub-structure belonging to the event.

Fig. 14 presents how a while loop is mapped. The transformation idea is the same as for repeat-until loops: the loop events are catched by a single `pick` activity. The difference to the repeat-until transformation is the placement of the `pick` activity. We do not want to duplicate code. Therefore, the looping variable is set to `true` and the first activity in the while loop is the `pick` activity. Here, the decision whether the loop should really run or the loop should be exited is taken. The subsequent activity is an `if` activity checking whether the loop body should be run. The concrete loop body is nested in that `if` activity.




```
<while>
  <targets><target linkName="l1"/></targets>
  <sources>
    <source name="lk" transitionCondition="EventEiHappened" />
    ...
  </sources>
  <condition>doWhile1</condition>
  <sequence>
    <pick>
      <onMessage partnerLink="EventsPLT" operation="EventEL" variable="EventL">
        <assign>
          doWhile1 = true
        </assign>
      </onMessage>
      <onMessage partnerLink="EventsPLT" operation="EventEi" variable="Event1">
        <assign>doWhile1 = false; EventEiHappend=true</assign>
      </onMessage>
      ...
      <onMessage partnerLink="EventsPLT" operation="EventEn" variable="Eventn">
        <assign>doWhile1 = false</assign>
      </onMessage>
    </pick>
    <if>
      <condition>doWhile1</condition>
      ...mapping of sub-structure 1...
    </if>
  </sequence>
</while>
```

Figure 14: Mapping of a while loop




```
<flow>
  <links><link name="l1"/></links>
  <sub-structure1>
    <sources><source name="l1"/></sources>
    ...mapping of first sub-structure...
  </sub-structure1>
  <sub-structure2>
    <targets><target name="l1"/></targets>
    ...mapping of second sub-structure...
  </sub-structure2>
</flow>
```

Figure 15: Transformation of a sequence

The transformation of sequences is presented in Fig. 15: the sub-structures in a sequence are connected using `links` carrying the default transition condition `true`.

If an event is used by a transformation pattern, the mapping of that event is defined by that pattern. For example, this is the case at the event "Maintenance Successfully Completed" in the maintenance process. That event is transformed by the repeat-until pattern and not any other pattern. To ensure correct processing order, the process structure tree is processed from the leaves to the source and each element of the EPC is marked whether it has been transformed.

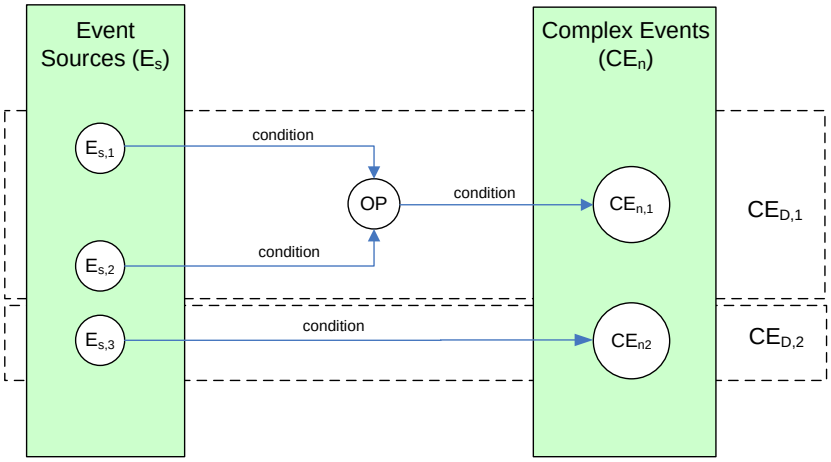


Figure 16: Wiring of Source-Events to Complex-Events

5.4 Step 4: Specify the Complex Event Processing Rules

The fourth step is the wiring between the low level event sources and all complex events with an wiring tool or other existing supporting software. This software writes out the event definitions $C\!E_D$ which are the names and the dependencies to the event sources for evaluation.

Wiring of source-events to complex events, i.e. the specification of the rules how complex events are created, is defined using a dedicated event wiring tool. The resulting specification is targeted to communicate these complex event generation rules to the actual developer who implements code for the event system that aggregates basic events to the complex events with the specified semantics.

Fig. 16 shows an example of how the graphical wiring for the complex event "AverageTemperatureAboveLimit" may look like. Generally, the event definition is always a tree with the complex event as the root node and the source events or other complex events as leave nodes. Intermediary nodes are operators. Up to now a language such as BEMN [DGB07] could be used. However, the event wiring tool also has to specify how correlation is initiated in the case of a start event and how events and process instances are correlated using e.g. a correlation ID.

Please note that it is not necessary to have a dedicated modelling tool at hand in order to be able to carry out this task. Any graphical modelling tool that is able to draw simple graphs and annotate them with textual description suffices. Since natural language is used for the definition of all $C\!E_D$:

1. Each E_S needs to be identified, i.e. it must be clear which actual source of events (a sensor for instance) should be used.

2. Each arc has to denote a condition that is used to filter events flowing through.
3. OP may be any kind of functionality, e.g. a logical operator or set operation that performs actions on all incoming events.
4. Each complex event CE_N therefore is specified through basic events and a combination of operators. All information necessary to deduct the complex event must be included.

5.5 Step 5: Make Executable

The last and fifth step is a technical step. This step is done by IT experts and only adds technical execution information. On the one hand an IT-expert has to map all CE_D definitions to queries for the used CEP system for execution CE_Q. Also event adapters E_A for all needed event sources E_S have to be provided (installed or implemented). On the other hand the generated BPEL flow is abstract that means it cannot be executed directly. The missing execution information and refinements needed for execution have to be added by the IT-exert. For example variables and assigns for the process internal data flow and also selecting the web service interfaces (WSDL) that should be used in the invokes activities. That runtime aspects are described in detail in the following.

The information as provided until now is not enough for the whole system to be automatically executed. In this section, we therefore list each area where information is missing and propose ways to fill the gap towards a fully machine executable system. We start by looking at the event sources and discuss the missing pieces of the systems that process them on their way to the final destination, the business process.

Modify Event Sources In order to be able to be notified of state changes in the environment, various resources need to be able to communicate source events to the event engine that processes and aggregates them to complex events via rules. Event sources include, amongst others:

- Sensors of any kind that report state changes in either a continuous stream of events (like a temperature sensor) or single event per state change
- Triggers in a database system, e.g. when data was modified or inserted
- Human interaction with systems, e.g. pressing a button or editing an electronic form
- State changes in application programs such as Electronic Resource Planning (ERP) or Electronic Manufacturing Systems (EMS).
- The process engine itself that is running the business processes in a factory. Events include state changes in the lifecycle of activities, start and completions

of processes, etc.

All of those sources need to be modified or adapted in such a way that they emit their information in a canonical and standardized format. While this may sound like a tremendous amount of work, much is done already in this direction through the availability of Common Base Events (CBE) [Bri04] and similar standards. Furthermore, a generic adapter to industrial standards such as CAN / CANOpen[3] can solves this problem in a generic way, so that sensors itself do not even have to be touched at all. Moreover, large industrial infrastructures building upon those bus systems are transformed into event sources using a single adapter.

Map source events to complex events In Step 4 (Section 5.4) we proposed a simple graphical language to aggregate source events to complex events using various operators. For this step however we do not enforce the use of a particular graphical language. The only requirement we have is that the graphical representation can be transformed to rules understood by the event engine, so that complex events according to the graphical specification are generated.

This step might as well use another graphical event definition language such as the one proposed in [DGB07] or even the graphical representation of event rules such as the Event Processing Language (EPL)[4].

Communicate Complex Events to the Process Engine Since BPEL relies upon WSDL for communication with services and also provides its service interfaces in WSDL form, complex events from the event engine to the BPEL engine are communicated using Web service mechanisms. The BPEL process offers one `portType` for all events (in the form of `receive` operations) of the original EPC. Each complex event is represented by a different `operation` with the name of the event as the name of the operation and the payload of the event as payload. The WSDL/SOAP binding style [CCMW01] we propose to use is document/literal.

Provide executable completions for the BPEL process The transformation from EPCs to BPEL results in an abstract BPEL file that resembles the structure and control-flow semantics of the original EPC model, but is not yet executable. The EPC model does not contain all the necessary details to be able to directly generate executable BPEL. Information that is missing includes:

- PartnerLink, PortType and Operation definitions for Web service interaction activities such as `invoke`, `receive` or `reply`. Generally, all respective WSDL files for the Web service definitions of the process need to be created and linked to process activities.
- Definitions of process variables

[3] http://www.canopen.org/

[4] See `http://esper.codehaus.org/`

- *Copy expressions* in `assign` activities
- Transition- and join conditions

Another part of the executable completion is message and event correlation: Since multiple complex events are targeted at the same process instance, a mechanism is needed to dispatch each event to the right process instance. e.g. if a new complex event arrives at the process engine that is already running two process instances, it must be clear which process instance finally receives the event or if a new instance must be created to handle it. Through correlation identifiers in the event message, e.g. a sensor id, correlation sets can be created in the BPEL definition that allows the event to be correctly dispatched. Another solution is to inject artificial correlation information (such as a unique correlation id shared across complex events that belong to the same incident) into the complex directly at the event processing engine. In both cases, the process engine has enough information at hand to decide whether a new process instance needs to be created or the event is targeted at an already running instance.

6 Complete Example with All Mapping Steps

For a complete understanding how the described methodology is used this chapter presents a step by step example. For every step the results produced based on the concepts in Section 5 are shown.

6.1 Step 1: Define Process as EPC

For this task any EPC modeling tool can be used. There are many available in different scale. Examples are the ARIS Toolset, the Oryx framework [KDW08] and Microsoft Visio EPC stencil sets. For later automation purposes it is important that the EPC can be stored in an exchangeable file format.

The EPCs processes for the examples can be found in Section 4. The "Order Processing" (see Fig. 2) and the "Maintenance Process" (see third EPC in Fig. 3) process are used as example to show the steps of the methodology.

6.2 Step 2: Identify the Needed Complex Events

The Events extracted from the "Order Processing" are: "Order Processed", "Manufacturing Plan Completed", "Item Competed". The events extracted from "Maintenance Process" are: "Average Temperature Below Limit", "Average Temperature Above

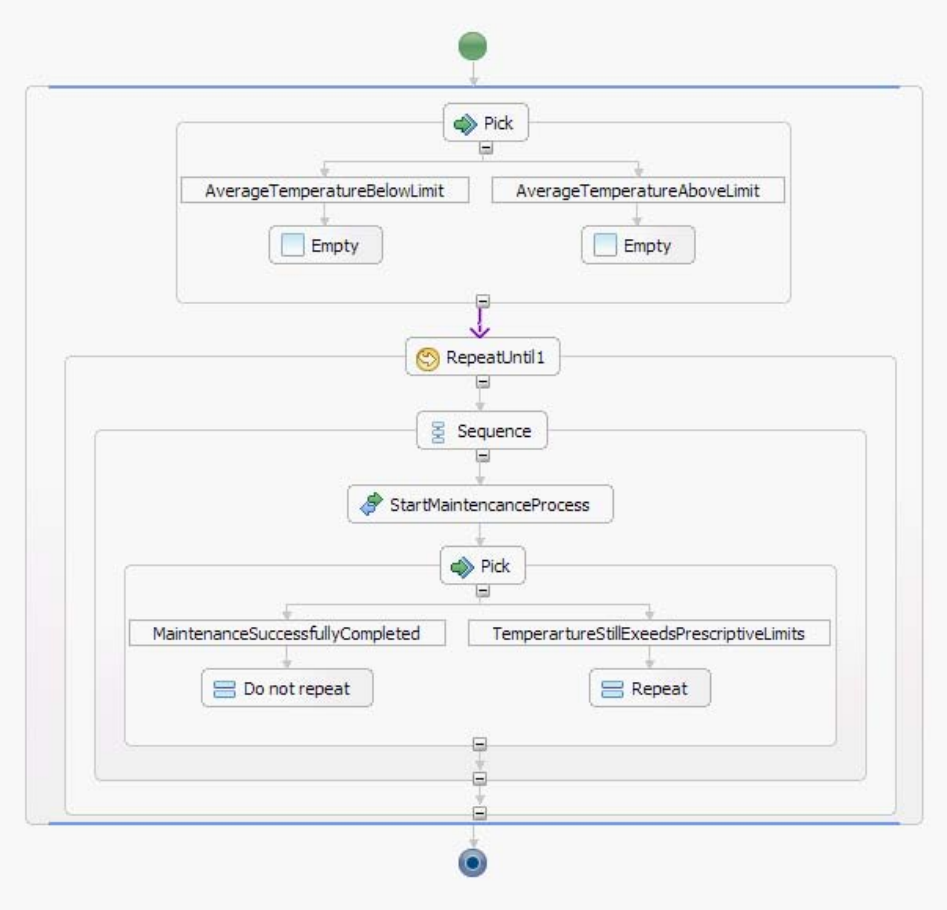


Figure 17: Eclipse BPEL editor view of "Template-MaintenanceProcess.bpel"

Limit", "Temperature Still Exceeds Prescriptive Limits", "Maintenance Successfully Completed". The result of this step is shown in Listing 1.

Listing 1 ...

$C\text{Æ}_N$={OrderProcessed, ManufacturingPlanCompleted, ItemCompeted, AverageTemperatureBelowLimit, AverageTemperatureAboveLimit, TemperartureStillExeedsPrescriptiveLimits, MaintenanceSuccessfullyCompleted}

6.3 Step 3: Transform the Process Definitions to Workflow Templates

Using the PST and patterns defined in Section 5.3.1 the EPC process definition can be transformed automatically to an BPEL workflow template. A BPEL template is not

executable but contains the complete control flow structure. The additional information needed for execution is added in step 5 (Section 6.5). The process structure tree for the maintenance process is shown in Fig. 5. The identified sequence is transformed to a `flow` activity, where the contained activities are connected using `links`. The pick-block at the beginning is transformed to a `pick` activity. The repeat-until loop is transformed as described above. The generated BPEL template for the "Maintenance Process" is shown in Fig. 17, which is a screenshot of the BPEL file opened in the Eclipse BPEL designer[5]. It is important to note that the transformation is a vertical transformation, in the sense that not only the control flow structure is mapped, but also the events are enriched by a connection to Web services. For the two examples the result of the EPC to BPEL transformation is presented in Listing 6.3 and Listing 6.3.

Listing 2 BPEL Process Template for "Order Processing"

```
<process xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/"
    name="Order Processing" targetNamespace="http://www.example.com/company/"/>
  <flow name="Flow">
    <links>
      <link name="Link1"/>
      <link name="Link2"/>
      <link name="Link3"/>
      <link name="Link4"/>
    </links>
    <receive createInstance="yes" partnerLink="EventsPLT"
      operation="SupplierOrderProcessed" variable="Event1">
      <sources>
        <source linkName="Link1"/>
      </sources>
    </receive>
    <receive createInstance="yes" partnerLink="EventsPLT"
      operation="ManufacturingPlanCompleted" variable="Event2">
      <sources>
        <source linkName="Link2"/>
      </sources>
    </receive>
    <empty name="And">
      <targets>
        <joinCondition>$Link1 and $Link2</joinCondition>
        <target linkName="Link1"/>
        <target linkName="Link2"/>
      </targets>
      <sources>
        <source linkName="Link3"/>
      </sources>
    </empty>
```

[5] `http://www.eclipse.org/bpel/`

```
32      <opaqueActivity name="ManufactureItem">
33        <targets>
34          <target linkName="Link3"/>
35        </targets>
36        <sources>
37          <source linkName="Link4"/>
38        </sources>
39      </opaqueActivity>
40      <receive partnerLink="EventsPLT" operation="ItemCompleted" variable="Event3">
41        <targets>
42          <target linkName="Link4"/>
43        </targets>
44      </receive>
45    </flow>
46  </process>
```

Listing 3 BPEL Process Template for "Maintenance Process"

```
1   <process xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/"
2       exitOnStandardFault="yes" name="RepeatMaintenanceProcess"
3       targetNamespace="http://www.example.com/company/">
4     <flow name="Flow">
5       <links>
6         <link name="Link1"/>
7       </links>
8       <pick createInstance="yes">
9         <sources>
10          <source linkName="Link1"/>
11        </sources>
12        <onMessage partnerLink="EventsPLT"
13          operation="AverageTemperatureBelowLimit" variable="Event1">
14          <empty />
15        </onMessage>
16        <onMessage partnerLink="EventsPLT"
17          operation="AverageTemperatureAboveLimit" variable="Event2">
18          <empty />
19          </onMessage>
20      </pick>
21      <repeatUntil>
22        <targets>
23          <target linkName="Link1"/>
24        </targets>
25        <sequence>
26          <opaqueActivity name="StartMaintencanceProcess"/>
27          <pick>
28            <onMessage partnerLink="EventsPLT"
```

```
29              operation="TemperartureStillExeedsPrescriptiveLimits" variable="Event3">
30              <assign>
31                doRepeatUntilLoop1 = true
32              </assign>
33            </onMessage>
34            <onMessage partnerLink="EventsPLT"
35              operation="MaintenanceSuccessfullyCompleted" variable="Event4">
36              <assign>
37                doRepeatUntilLoop1 = false
38              </assign>
39            </onMessage>
40          </pick>
41        </sequence>
42        <condition>not(doRepeatUntilLoop1)</condition>
43          <!-- repeats until condition equals true -->
44      </repeatUntil>
45    </flow>
46  </process>
```

For a better understanding of the generated BPEL "Maintenance Process" a screen-shot of the BPEL file opened in the Eclipse BPEL designer is shown in 17 on page 27. The two start events of the EPC are transformed to a BPEL `pick`, since they are connected by an "exclusive or". If the exclusive or had been an "and" another pattern would be used. The loop is represented in BPEL as `repeatUntil`, since the loop starts with an activity. If the loop started with an event if would be a `while` activity.

6.4 Step 4: Specify How the Complex Events Should be Evaluated

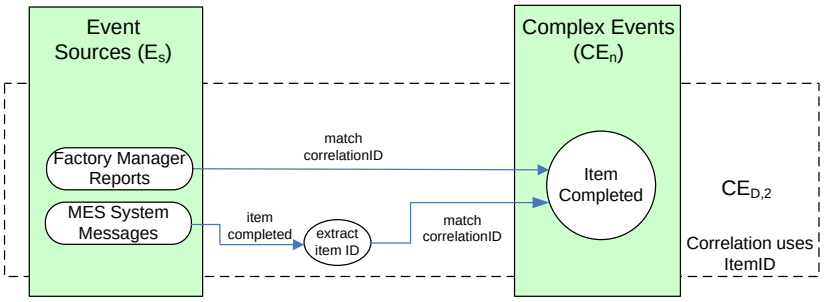


Figure 18: Event Wiring Example for "AverageTemperatureAboveLimit"'

In this step the definition of the CE_D have to be made. The wire descriptions specifies how event sources E_S are accessed through event adapters E_A to evaluate a complex

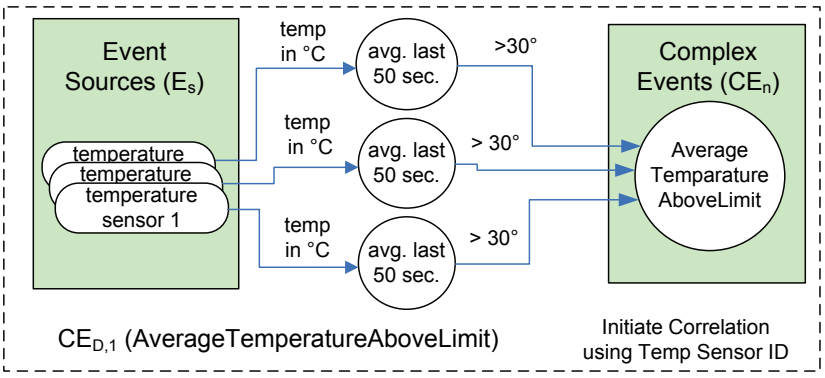


Figure 19: Event Wiring Example for "ItemCompleted"

event *CE*. Fig. 18 shows the wire description for *AverageTemperatureAboveLimit*. Here all sensors in a building are monitored and the average temperature in 50 seconds are taken as raw events. That raw events are in forehand filtered for degree Celsius and afterwards for the value bigger than 30. The correlation works on the sensor ID. Fig. 19 shows the wire description for *ItemCompleted'*. Here a factory manager can directly report a item as completed or a message from a MES (Manufacturing Execution System) is used as raw event.

6.5 Step 5: Do the Runtime Configuration and Deployment

To make the process template executable following further steps are required.

- First a WSDL file can be automatically generated containing the PortType and all Operations for the events in the process.
- Then a further WSDL file is generated containing the PartnerLinkTypes for the Complex Event System using the previously defined PortType.
- Afterwards, the BPEL template can be extended to an executable BPEL file by: importing the WSDL definitions, adding variables for handling the incoming events, define PartnerLinks for the PartnerLinkType with myRole as EventConsumer, describing all receives and onMessages where a event is received with the corresponding operation.
- Until now all this is generic and can be done automatically. The last step has to be done by hand: Mapping the `opaque` activities to something executable. In the example the `opaque` activity is mapped to a invoke activity that starts the MaintenanceSystem.

Now the BPEL process can be deployed on a BPEL engine. The CEP system has to be set up to send the event messages to that engine. The executable BPEL files and WSDL listings can be found in Listing 8. The needed PartnerLink Types integrating both are generated as follows:

Listing 4 PartnerLinkType definitions

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <definitions xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
      xmlns:tns="http://www.example.com/company/Artifacts"
      xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
      xmlns:wsdl="test.com" name="ExecutableMaintenanceProcessArtifacts"
      targetNamespace="http://www.example.com/company/Artifacts"
      xmlns="http://schemas.xmlsoap.org/wsdl/">
3   <plnk:partnerLinkType name="ComplexEventSystemPTL">
4     <plnk:role name="EventConsumer"
          portType="wsdl:MaintenanceProcess_IncomingMessages"/>
5     <plnk:role name="EventProvider"/>
6   </plnk:partnerLinkType>
7   <import location="ExecutableMaintenanceProcess.wsdl" namespace="test.com"/>
8 </definitions>
```

7 Conclusion and Future Work

We presented a new methodology that helps to cope with the yet unsolved challenge how to specify complex events and their effects on workflows. The prove for applicableness of the introduced method is shown in the report by presenting a complete walk-through of all steps based on two examples of contrary application areas. Regarding the mapping of EPC to BPEL, that work combine the work of [VVK08] and [GB08] to provide a complete mapping including support of loops and workflow instantiating start events.

Future work is to do research on how provide good tool support for the presented methodology. There are different areas of tools that would help a domain expert to use the proposed methodology: First an integrated modeling tool that allows to model EPC processes, BPEL processes and transformation patterns. Secondly a complex event wiring tool that allows the specification of the complex events based on the available sensors.

8 The Complete Listings

Listing 5 Executable BPEL Process for the “Maintenance Process”

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<bpws:process xmlns:bpws="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" exitOnStandardFault="yes"
  name="MaintenanceProcess" targetNamespace="http://www.example.com/company/"
  xmlns:ns1="http://www.example.com/company/Artifacts" xmlns:ns="test.com">
    <bpws:import namespace="test.com" location="TestArtifacts.wsdl"
        importType="http://schemas.xmlsoap.org/wsdl/"></bpws:import>
    <bpws:import namespace="http://www.example.com/company/Artifacts"
    location="ExecutableMaintenanceProcessArtifacts.wsdl"
        importType="http://schemas.xmlsoap.org/wsdl/"></bpws:import>
  <bpws:import namespace="test.com" location="ExecutableMaintenanceProcess.wsdl"
    importType="http://schemas.xmlsoap.org/wsdl/"></bpws:import>
  <bpws:partnerLinks>
    <bpws:partnerLink name="EventsPLT" partnerLinkType="ns1:ComplexEventSystemPTL"
      myRole="EventConsumer" partnerRole="EventProvider"></bpws:partnerLink>
        <bpws:partnerLink name="MaintenanceSystem" partnerLinkType="ns:Test"
        partnerRole="TestProvider" myRole="TestRequester"></bpws:partnerLink>
    </bpws:partnerLinks>
  <bpws:variables>
    <bpws:variable name="doLoopRepeatUntil1" type="xsd:bool" />
    <bpws:variable name="Event1" element="ns:EventMessage"></bpws:variable>
    <bpws:variable name="Event2" element="ns:EventMessage"></bpws:variable>
    <bpws:variable name="Event4" element="ns:EventMessage"></bpws:variable>
    <bpws:variable name="Event3" element="ns:EventMessage"></bpws:variable>
        <bpws:variable name="MaintenanceSystemRequest"
            messageType="ns:TestRequestMessage"></bpws:variable>
        <bpws:variable name="MaintenanceSystemResponse"
            messageType="ns:TestResponseMessage"></bpws:variable>
    </bpws:variables>
  <bpws:flow name="Flow">
    <bpws:links>
      <bpws:link name="Link1" />
    </bpws:links>
    <bpws:pick createInstance="yes" name="Pick">
      <bpws:sources>
        <bpws:source linkName="Link1" />
      </bpws:sources>
      <bpws:onMessage partnerLink="EventsPLT"
        operation="AverageTemperatureBelowLimit"
                portType="ns:MaintenanceProcess_IncomingMessages" variable="Event1">
        <bpws:empty/>
            </bpws:onMessage>
      <bpws:onMessage partnerLink="EventsPLT"
        operation="AverageTemperatureAboveLimit"
                portType="ns:MaintenanceProcess_IncomingMessages" variable="Event2">
        <bpws:empty />
      </bpws:onMessage>
    </bpws:pick>
    <bpws:repeatUntil name="RepeatUntil1">
      <bpws:targets>
```

```
51          <bpws:target linkName="Link1" />
52        </bpws:targets>
53        <bpws:sequence>
54                  <bpws:invoke name="StartMaintencanceProcess"
55                  partnerLink="MaintenanceSystem" operation="initiate"
56                  portType="ns:Test"
57                  inputVariable="MaintenanceSystemRequest"
58                      outputVariable="MaintenanceSystemResponse"></bpws:invoke>
59                  <bpws:pick>
60             <bpws:onMessage partnerLink="EventsPLT"
61               operation="MaintenanceSuccessfullyCompleted"
62                           portType="ns:MaintenanceProcess_IncomingMessages"
63                           variable="Event3">
64               <bpws:assign name="Do not repeat"
65                 validate="no">
66                 <bpws:copy>
67                   <bpws:from>false</bpws:from>
68                   <bpws:to variable="doLoopRepeatUntil1" />
69                 </bpws:copy>
70               </bpws:assign>
71             </bpws:onMessage>
72             <bpws:onMessage partnerLink="EventsPLT"
73               operation="TemperartureStillExeedsPrescriptiveLimits"
74                           portType="ns:MaintenanceProcess_IncomingMessages"
75                           variable="Event4">
76               <bpws:assign name="Repeat"
77                 validate="no">
78                 <bpws:copy>
79                   <bpws:from>true</bpws:from>
80                   <bpws:to variable="doLoopRepeatUntil1" />
81                 </bpws:copy>
82               </bpws:assign>
83             </bpws:onMessage>
84           </bpws:pick>
85        </bpws:sequence>
86        <bpws:condition><![CDATA[not(doLoopRepeatUntil1)]]></bpws:condition>
87      </bpws:repeatUntil>
88    </bpws:flow>
89  </bpws:process>
```

Listing 6 WSDL Definitions for the “Maintenance Process”

```
1  <?xml version="1.0"?>
2  <definitions name="Test"
3          targetNamespace="test.com"
4          xmlns:tns="test.com"
```

```
5           xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
6           xmlns="http://schemas.xmlsoap.org/wsdl/"
7           xmlns:p="http://www.w3.org/2001/XMLSchema"
8           xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
9
10      <types>
11          <schema attributeFormDefault="unqualified"
12                  elementFormDefault="qualified"
13                  targetNamespace="test.com"
14                  xmlns="http://www.w3.org/2001/XMLSchema">
15              <element name="EventMessage" type="xsd:anyType"></element>
16          </schema>
17      </types>
18
19      <message name="AverageTemperatureBelowLimitRequest">
20        <part name="payload" element="tns:EventMessage"></part>
21      </message>
22      <message name="AverageTemperatureAboveLimitRequest">
23        <part name="payload" element="tns:EventMessage"></part>
24      </message>
25      <message name="MaintenanceSuccessfullyCompletedRequest">
26        <part name="payload" element="tns:EventMessage"></part>
27      </message>
28      <message name=" TemperartureStillExeedsPrescriptiveLimitsRequest">
29        <part name="payload" element="tns:EventMessage"></part>
30      </message>
31
32      <portType name="MaintenanceProcess_IncomingMessages">
33        <operation name="AverageTemperatureBelowLimit">
34          <input message="tns:AverageTemperatureBelowLimitRequest">
35              </input>
36        </operation>
37        <operation name="AverageTemperatureAboveLimit">
38          <input message="tns:AverageTemperatureAboveLimitRequest">
39              </input>
40        </operation>
41        <operation name="MaintenanceSuccessfullyCompleted">
42          <input message="tns:MaintenanceSuccessfullyCompletedRequest">
43              </input>
44        </operation>
45        <operation name="TemperartureStillExeedsPrescriptiveLimits">
46          <input message="tns:TemperartureStillExeedsPrescriptiveLimitsRequest">
47              </input>
48        </operation>
49      </portType>
50
51      <binding name="BPELengineBinding"
52        type="tns:MaintenanceProcess_IncomingMessages">
53        <soap:binding style="document"
54          transport="http://schemas.xmlsoap.org/soap/http" />
```

```
55        <operation name="AverageTemperatureBelowLimit">
56          <soap:operation
57            soapAction="test.com/AverageTemperatureBelowLimit" />
58          <input>
59            <soap:body use="literal" />
60          </input>
61        </operation>
62        <operation name="AverageTemperatureAboveLimit">
63          <soap:operation
64            soapAction="test.com/AverageTemperatureAboveLimit" />
65          <input>
66            <soap:body use="literal" />
67          </input>
68        </operation>
69        <operation name="MaintenanceSuccessfullyCompleted">
70          <soap:operation
71            soapAction="test.com/MaintenanceSuccessfullyCompleted" />
72          <input>
73            <soap:body use="literal" />
74          </input>
75        </operation>
76        <operation name="TemperartureStillExeedsPrescriptiveLimits">
77          <soap:operation
78            soapAction="test.com/TemperartureStillExeedsPrescriptiveLimits" />
79          <input>
80            <soap:body use="literal" />
81          </input>
82        </operation>
83      </binding>
84
85      <service name="BPELengine">
86        <port name="EnginePort" binding="tns:BPELengineBinding">
87          <soap:address location="http://www.example.org/" />
88        </port>
89      </service>
90  </definitions>
```

Listing 7 Message Definitions for the “Maintenance Process”

```
1   <?xml version="1.0"?>
2   <definitions name="Test"
3           targetNamespace="test.com"
4           xmlns:tns="test.com"
5           xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
6           xmlns="http://schemas.xmlsoap.org/wsdl/"
7           xmlns:p="http://www.w3.org/2001/XMLSchema"
```

```
8           xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
9
10      <types>
11          <schema attributeFormDefault="unqualified"
12                  elementFormDefault="qualified"
13                  targetNamespace="test.com"
14                  xmlns="http://www.w3.org/2001/XMLSchema">
15              <element name="EventMessage" type="xsd:anyType"></element>
16          </schema>
17      </types>
18
19  </definitions>
```

References

[AADH05] L. Aldred, W. M. van der Aalst, M. Dumas, A. H. ter Hofstede. On the Notion of Coupling in Communication Middleware. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, volume 3761/2005 of *LNCS*, pp. 1015–1033. Springer, 2005. doi:10.1007/11575801_6. URL `http://www.springerlink.com/content/8m702qh8xvkxvr5r/`.

[ADK02] W. M. P. van der Aalst, J. Desel, E. Kindler. On the Semantics of EPCs: A Vicious Circle. In *EPK 2002*. 2002.

[AH05] W. M. P. van der Aalst, A. H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[BDG07] A. Barros, G. Decker, A. Grosskopf. Complex Events in Business Processes. In *Business Information Systems*, volume 4439/2007 of *LNCS*, pp. 29–40. Springer, 2007. doi:10.1007/978-3-540-72035-5_3. URL `http://www.springerlink.com/content/f00777104222w515/`.

[Bri04] D. Bridgewater. Standardize messages with the Common Base Event model. *IBM DeveloperWorks*, 2004.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. *Web Services Description Language (WSDL) 1.1*, 2001. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.

[CKLW03] F. Curbera, R. Khalaf, F. Leymann, S. Weerawarana. Exception Handling in the BPEL4WS Language. In *International Conference on Business Process Management*, volume 2678 of *LNCS*, pp. 276–290. 2003.

[DGB07] G. Decker, A. Grosskopf, A. Barros. A Graphical Notation for Modeling Complex Events in Business Processes. In *EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, p. 27. IEEE Computer Society, Washington, DC, USA, 2007.

[DM08] G. Decker, J. Mendling. Instantiation Semantics for Process Models. In *Proceedings of the 6th International Conference on Business Process Management (BPM)*, LNCS, pp. 164–179. 2008.

[Ecm05] Ecma International. *Standard ECMA-357 - ECMAScript for XML (E4X) Specification*, 2nd edition edition, 2005.

[GB08] L. García-Bañuelos. Pattern Identification and Classification in the Translation from BPMN to BPEL. In *On the Move to Meaningful Internet Systems: OTM 2008*, volume 5331/2008 of *LNCS*, pp. 436–444. Springer, 2008. doi:10.1007/978-3-540-88871-0_30. URL `http://www.springerlink.com/content/415211u321845582/`.

[IBM08] IBM. WebSphere Process Server - Extension names for business process events, 2008. URL `http://publib.boulder.ibm.com/infocenter/dmndhelp/v6r1mx/index.jsp?topic=/com.ibm.websphere.bpc.610.doc/doc/bpc/rmonitor_process_extensionnames.html`.

[KDW08] S. Krumnow, G. Decker, M. Weske. Modellierung von EPKs im Web mit Oryx. In *Proceedings of the 7th GI-Workshop Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (EPK)*. 2008.

[KHB00] B. Kiepuszewski, A. H. M. ter Hofstede, C. Bussler. On structured workflow modelling. In *CAiSE*. Springer, 2000.

[Kin06] E. Kindler. On the Semantics of EPCs: A Framework for Resolving the Vicious Circle. *Data Knowl. Eng*, 56(1):23–40, 2006. doi:10.1016/j.datak.2005.02.005.

[KKL07] R. Khalaf, D. Karastoyanova, F. Leymann. Pluggable Framework for Enabling the Execution of Extended BPEL Behavior. In *Proceedings of the 3rd International Workshop on Engineering Service-Oriented Application (WESOA 2007)*. Springer-Verlag, 2007.

[KKS+06] D. Karastoyanova, R. Khalaf, R. Schroth, M. Paluszek, F. Leymann. BPEL Event Model. Technical Report Computer Science 2006/10, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, University of Stuttgart, Institute of Architecture of Application Systems, 2006. URL `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2006-10&engl=1`.

[KML08] O. Kopp, R. Mietzner, F. Leymann. Abstract Syntax of WS-BPEL 2.0. Technical Report Computer Science 2008/06, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, University of Stuttgart, Institute of Architecture of Application Systems, 2008. URL `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2008-06&engl=1`.

[KMWL08] O. Kopp, D. Martin, D. Wutke, F. Leymann. On the Choice Between Graph-Based and Block-Structured Business Process Modeling Languages. In *Modellierung betrieblicher Informationssysteme (MobIS 2008)*, CEUR Workshop Proceedings. CEUR, 2008. URL `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2008-92&engl=1`.

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

[KUL06] O. Kopp, T. Unger, F. Leymann. Nautilus Event-driven Process Chains: Syntax, Semantics, and their mapping to BPEL. In *Proceedings of the 5th GI Workshop on Event-Driven Process Chains (EPK 2006)*. 2006.

[LC08] Z. Laliwala, S. Chaudhary. Event-driven Service-Oriented Architecture. In *International Conference on Service Systems and Service Management*. 2008. doi:10.1109/ICSSSM.2008.4598452.

[LLSG08] D. Lübke, T. Luecke, K. Schneider, J. M. Gomez. Using event-driven process chains for model-driven development of business applications. *International Journal of Business Process Integration and Management*, 3(2):109–117, 2008.

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

[MA07] J. Mendling, W. M. P. van der Aalst. Formalization and Verification of EPCs with OR-Joins Based on State and Context. In *CAiSE*. 2007.

[Mar06] J.-L. Maréchaux. Combining Service-Oriented Architecture and Event-Driven Architecture using an Enterprise Service Bus. http://www.ibm.com/developerworks/library/ws-soa-eda-esb/, 2006.

[Men07] J. Mendling. *Detection and Prediction of Errors in EPC Business Process Models*. Ph.D. thesis, Vienna University of Economics and Business Administration, 2007.

[MFP06] G. Muehl, L. Fiege, P. R. Pietzuch. *Distributed Event-Based Systems*. Springer, 2006.

[MG06] T. Mens, P. V. Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006. doi:10.1016/j.entcs.2005.10.021.

[MLZ08] J. Mendling, K. B. Lassen, U. Zdun. On the Transformation of Control Flow between Block-Oriented and Graph-Oriented ProcessModeling Languages. *Int. J. Business Process Integration and Management (IJBPIM)*, 3(2):96–108, 2008.

[OAS07] OASIS. Web Services Business Process Execution Language Version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`, 2007.

[Obj08] Object Management Group. *Business Process Modeling Notation, V1.1*, 2008. URL `http://www.omg.org/spec/BPMN/1.1/PDF`.

[ODBH06] C. Ouyang, M. Dumas, S. Breutel, A. H. M. ter Hofstede. Translating Standard Process Models to BPEL. In *CAiSE*. Springer, 2006.

[Sch03] A.-W. Scheer. *ARIS-Modellierungs-Methoden, Metamodelle, Anwendungen*. Springer, 2003.

[SI07] S. Stein, K. Ivanov. EPK nach BPEL Transformation als Voraussetzung für praktische Umsetzung einer SOA. In *Software Engineering 2007*, volume 105 of *LNI*, pp. 75–80. GI, Hamburg, Germany, 2007.

[SKI08] S. Stein, S. Kühne, K. Ivanov. Business to IT Transformations Revisited. In *MDE4BPM*. 2008.

[STA05] A.-W. Scheer, O. Thomas, O. Adam. *Process-Aware Information Systems: Bridging People and Software Through Process Technology*, chapter Process Modeling Using Event-Driven Process Chains, pp. 119–146. Wiley & Sons, 2005.

[VVK08] J. Vanhatalo, H. Völzer, J. Koehler. The Refined Process Structure Tree. In *BPM*. Springer, 2008.

[VVL08] J. Vanhatalo, H. Völzer, F. Leymann. Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In *Service-Oriented Computing - ICSOC 2007*, volume 4749/2008 of *LNCS*, pp. 43–55. Springer, 2008. doi:10.1007/978-3-540-74974-5_4. URL `http://www.springerlink.com/content/7w6170q3t6q56861/`.

[WALD07] M. Wei, I. Ari, J. Li, M. Dekhil. ReCEPtor: Sensing Complex Events in Data Streams for Service-Oriented Architectures. Technical Report HPL-2007-176 20071102, HP, 2007.

[WEAH05] M. T. Wynn, D. Edmond, W. M. P. van der Aalst, A. H. M. ter Hofstede. Achieving a General, Formal and Decidable Approach to the OR-Join in

Workflow Using Reset Nets. In *Applications and Theory of Petri Nets*. 2005. doi:10.1007/11494744_24. URL `http://www.springerlink.com/content/g41avg6f8xfmdyph/`.

[Weh07] J. Wehler. Boolean and free-choice semantics of Event-driven Process Chains. In *EPK*. 2007.

[ZHB+06] W. Zhao, R. Hauser, K. Bhattacharya, B. R. Bryant, F. Cao. Compiling business processes: untangling unstructured loops in irreducible flow graphs. *International Journal of Web and Grid Services*, 2:68–91, 2006. doi:10.1504/06.8880. URL `http://www.inderscience.com/link.php?id=8880`.

[ZM05] J. Ziemann, J. Mendling. EPC-Based Modelling of BPEL Processes: a Pragmatic Transformation Approach. In *Proceedings of the 7th International Conference Modern Information Technology in the Innovation Processes of the Industrial Enterprises (MITIP 2005)*. Genova, Italy, 2005.

All links were last followed on 2008-11-21.