Modeling and Execution of Event Stream Processing in Business Processes

Stefan Appel, Pascal Kleber, Sebastian Frischbier, Tobias Freudenreich, Alejandro Buchmann

TU Darmstadt, Databases and Distributed Systems Group, Darmstadt, Germany

Abstract

The Internet of Things and Cyber-physical Systems provide enormous amounts of real-time data in form of streams of events. Businesses can benefit from the integration of these real-world data; new services can be provided to customers, or existing business processes can be improved. Events are a well-known concept in business processes. However, there is no appropriate abstraction mechanism to encapsulate event stream processing in units that represent business functions in a coherent manner across the process modeling, process execution, and IT infrastructure layer. In this paper we present Event Stream Processing Units (SPUs) as such an abstraction mechanism. SPUs encapsulate application logic for event stream processing and enable a seamless transition between process models, executable process representations, and components at the IT layer. We derive requirements for SPUs and introduce EPC and BPMN extensions to model SPUs at the abstract and at the technical process layer. We introduce a transformation from SPUs in EPCs to SPUs in BPMN and implement our modeling notation extensions in Software AG ARIS. We present a runtime infrastructure that executes SPUs and supports implicit invocation and completion semantics. We illustrate our approach using a logistics process as running example.

Keywords:
Event-stream processing, Business process modeling, Business process execution, EPC extensions, BPMN extensions

1. Introduction

Business process modeling and execution is widely adopted in enterprises. Processes are modeled by business experts and translated into executable workflow representations. They are executed inside IT infrastructures, e.g., Service-oriented Architectures (SOAs) or workflow management systems. With the adoption of the Internet of Things and Cyber-physical Systems, huge amounts of information become available that reflect the state of the real world. The integration of this up-to-date information with business processes (BPs) allows quick reactions on unforeseen situations as well as offering new services to customers, e.g., monitoring of environmental conditions during transport of goods and handling exceeded temperature thresholds [1].

A common paradigm for the representation of information from sources like the Internet of Things or Cyber-physical Systems is streams of events. The notion of a *stream* illustrates that new events occur over time, e.g., continuous temperature sensor readings. In such event-based systems, event producers do not necessarily know the event consumers, or whether the events will be consumed at all. This independence is intrinsic to the event-based approach [2]. The decoupling of event producers and consumers as well as the arrival of an indefinite number of events over time requires an appropriate event dissemination mechanism. Commonly, publish/subscribe systems are used; they allow asynchronous m:n communication between fully decoupled participants. Event consumers specify their interest in events in form of *subscriptions*; event producers specify the type of events they may publish in *advertisements*.

While single events are a well-known and established concept in BP models [3, 4], event stream processing lacks an appropriate abstraction for the seamless integration across the process modeling, process execution, and IT infrastructure layer. In collaboration with Software AG[1] we developed *Event Stream Processing Units* (SPUs) as such an integration concept for event stream processing.

In this paper we present SPUs. SPUs provide a service-like abstraction to encapsulate event stream processing logic at the abstraction level of business functions. They hide implementation details and are suitable building blocks for the integration of event stream processing with BPs. We analyze BP modeling, BP execution, and the IT infrastructure, and derive requirements for SPUs at the modeling, execution, and IT infrastructure layer. We address the decoupled nature of event-based systems and provide process modelers with an appropriate representation of SPUs that can be mapped to executable workflow representations and the IT infrastructure seamlessly. At the IT layer, SPUs are manageable components that are conceptually equivalent to services in a SOA. SPUs contain, for example, complex event processing (CEP) functionality.

This paper extends previous work: In [5] we introduce Business Process Model and Notation 2.0 (BPMN) extensions to

Email addresses: appel@dvs.tu-darmstadt.de (Stefan Appel), kleber@dvs.tu-darmstadt.de (Pascal Kleber), frischbier@dvs.tu-darmstadt.de (Sebastian Frischbier), freudenreich@dvs.tu-darmstadt.de (Tobias Freudenreich), buchmann@dvs.tu-darmstadt.de (Alejandro Buchmann)

[1] www.softwareag.com

model SPUs. SPUs, however, are a generic concept not limited to a specific modeling notation. Thus, we present an additional implementation of SPUs in this paper: we introduce extensions to model SPUs in Event-driven Process Chains (EPCs) as an alternative BP modeling notation with focus on abstract process models from a business perspective. We also present a mapping between SPUs in EPCs and SPUs in BPMN to illustrate the generic applicability of our SPU concept; it can be applied to different modeling notations to support, for example, a holistic BP modeling approach. As further extensions to [5] we provide more examples on modeling SPUs with BPMN and additional details on workflow execution semantics. We also present an implementation for modeling and execution of SPU-containing process models.

The paper is structured as follows: we introduce a logistics scenario as running example; we then derive requirements for the integration of event stream processing with BPs at the modeling, execution, and IT infrastructure layer. In Section 3, we introduce Event Stream Processing Services (ESPSs) as an extension to EPCs to model SPUs. We introduce Event Stream Processing Tasks (ESPTs) as BPMN 2.0 extension to model SPUs. We present a mapping of SPU-containing process models from EPCs to BPMN and from BPMN to BPEL as well as a runtime environment for SPUs. In Section 4 we present the implementation of our EPC and BPMN extensions in Software AG ARIS; we sketch the model-to-execute workflow that brings SPU-containing business process models to execution. In Section 5 we discuss related work; we summarize our findings in Section 6.

Scenario

We illustrate our concept of SPUs by means of an order-to-delivery process. The processing of an order consists of multiple process steps: an order is received, the invoice for the order is prepared and the payment is processed. With SPUs, data generated during the physical transport can now be integrated with this process. An event stream that provides monitoring data related to the shipment can be used to detect, e.g., temperature threshold violations. An SPU can represent such a monitoring task and integrate it at the BP modeling, BP execution, and IT infrastructure layer. A shipment monitoring SPU is *instantiated* with the shipment of an order. The SPU *completes* after delivery. Throughout the paper, we illustrate our approach on the basis of such a monitoring SPU.

2. Event Stream Integration Requirements

Business process models describe workflows in companies in a standardized way. They document established business procedures with the goal of making complex company structures manageable. This encompasses the business perspective as well as the IT perspective. For the modeling and execution of processes, an appropriate level of abstraction is crucial to hide irrelevant details to the process modeler. Building blocks for BP modeling, BP execution, and IT infrastructure should encapsulate business functions in a self-contained way, e.g., like services in a SOA [6]. The BP model describes interactions between these building blocks.

The implementation of BPs in enterprises involves three layers: the modeling layer, the execution layer, and the IT infrastructure layer (see Figure 1). During design time, business experts create models from a business perspective, e.g., using Event-driven Process Chains (EPCs) [7, 3] or the Business Process Model and Notation 2.0 (BPMN) [4]. The models are then transformed into executable workflows represented in, e.g., the Business Process Execution Language (BPEL) [8]. Typically, the workflow execution requires IT support, which is provided by a SOA and workflow management systems.

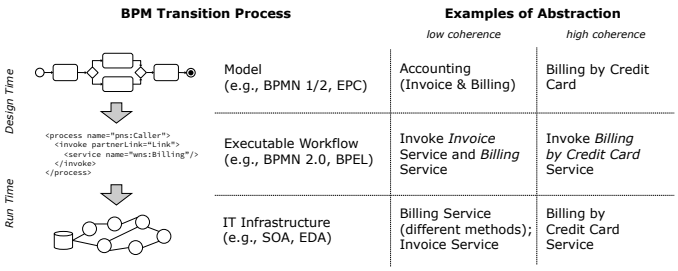

Figure 1: Transition steps between process modeling, process execution, and IT infrastructure layer.

The transition process from a BP model to, e.g., SOA service interactions is not trivial and requires expertise from the business perspective as well as from the IT perspective. To enable the seamless implementation of modeled processes, the abstraction of business functions should have the same granularity at each layer; a coherent abstraction across the layers minimizes the transition effort [9]. The example in Figure 1 illustrates this: the *low coherence* case requires a refinement with each transition step (a single BPMN task maps to multiple services) while the *high coherence* case allows a one-to-one transition between the business function representations available at each layer (e.g., BPMN tasks, BPEL invocations, and SOA services). In the following, we derive requirements for SPUs as business function abstractions. With the encapsulation of event stream processing in SPUs, a high coherence between the different layers is achieved; this supports a seamless transition between process model, executable workflow, and IT infrastructure. Table 1 shows the requirements we derive in the following sections in a summarized form to simplify later reference.

2.1. Business Process Modeling Layer

Process models are typically created by business experts that have a good knowledge about the company structure and established workflows. These process models describe interactions between business functions [6]. For a clear separation of concerns between the business perspective and the IT perspective, it is necessary to encapsulate event stream processing logic in SPUs that hide technical details at the modeling layer. SPUs are the abstract representation of business functions that process event streams. SPUs require at least one event stream as input and may output event streams or single events. An important

characteristic of SPUs is the demand for continuous processing of event streams; rather than in single request/reply interactions, SPUs process new events as they arrive, e.g., a shipment monitoring SPU receives new monitoring data continuously.

Requirements

For the integration of event streams, the modeling notation has to provide *elements or patterns to express SPUs (R_1)* [10]. While the actual event-processing functionality is encapsulated inside SPUs, event streams – as they represent a core data asset – should be accessible by the modeler [11]. Further, integrating event streams during modeling simplifies the transition to an executable workflow [12]. Thus, the modeling notation has to provide *means to express event streams as input/output to/from SPUs (R_2)*. Finally, the modeling notation must allow *SPUs to run continuously and in parallel to other tasks (R_3)*. This includes *appropriate execution semantics adapted to event-based characteristics (R_4)* [13].

2.2. Workflow Execution Layer

The execution of BP models requires a transition from the, often graphical, model notation to a formal process representation. The interactions between the different process tasks are formalized in a workflow description, e.g., using BPEL. This workflow description contains, e.g., service invocations and defines the input data for services. Like traditional BP tasks can be mapped to human tasks or service invocations, SPUs need to be mapped from the model to the IT infrastructure.

Requirements

To support SPUs at the workflow execution layer, the execution notation has to *support the instantiation of the SPU containers provided by the IT infrastructure (R_5)* [14]. It further needs means to *define streams of events as input and output of SPUs (R_6)*. The *instantiation and completion of SPUs needs to be configurable with respect to event-based characteristics (R_7)* [13].

2.3. IT Infrastructure Layer

The IT infrastructure holds the technical representations of SPUs. It is responsible for the execution of the encapsulated event stream processing logic. In contrast to SOA services, SPUs follow the event-based paradigm. While services are invoked explicitly, SPUs react on streams of events. Services encapsulate business functions in a pull manner (reply is requested); SPUs encapsulate reactive business functions that are defined on event streams pushed into the system.

Requirements

The IT infrastructure has to *provide a runtime environment for SPUs that respects event-based characteristics, e.g., implicit instantiation (R_8)* [13]. It must *provide containers for SPUs that represent business functions (R_9)* [10]. Just like services, these *SPU containers must be manageable and capable of receiving the required data in form of event streams (R_{10})* [15].

R_1	Model notation support for SPUs
R_2	Model notation support for event streams
R_3	Continuous and parallel execution of SPUs
R_4	Event-based-compliant SPU completion semantics
R_5	SPU instantiation support
R_6	Support input/output to/from SPUs in form of event streams
R_7	Control over SPU instantiation and completion behavior
R_8	Runtime environment for SPUs
R_9	SPU container model
R_{10}	Support for SPU management

Table 1: Combined requirements at business process modeling, business process execution, and IT infrastructure layer

3. Event Stream Processing Units

To support SPUs at the BP modeling, BP execution, and IT infrastructure layer, we suggest mechanisms at each layer. At the modeling layer, we introduce *Event Stream Processing Services* (ESPSs) to represent SPUs in EPCs and *Event Stream Processing Tasks* (ESPTs) to represent SPUs in BPMN process models. Our SPU concept is applicable in the context of different modeling notations as we show by means of a mapping between ESPSs and ESPTs. At the IT infrastructure layer, we adapt *Eventlets* [14] for the implementation of SPUs. The execution layer is responsible for the mapping between ESPTs and Eventlets. This is shown in Figure 2: like services are the basic building blocks of a SOA, SPUs are the basic building blocks of an event-driven architecture (EDA). At the execution layer, service tasks in a model are mapped to, e.g., web services. Equally, ESPTs are mapped to Eventlets.

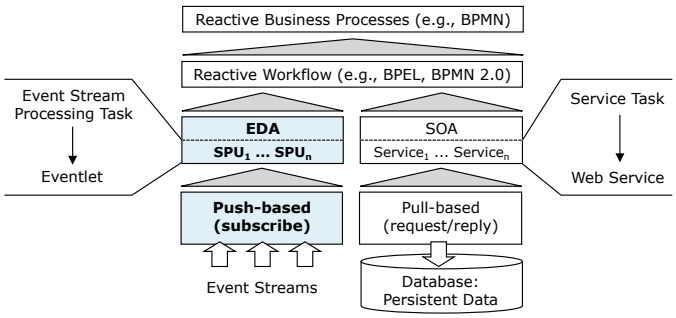

Figure 2: Stream Processing Units (SPUs) as building blocks of an event-driven architecture (EDA)

3.1. Modeling Layer

Different business perspectives need to be incorporated during BP modeling. Business analysts provide knowledge about processes from an abstract business perspective. The resulting process models are then refined iteratively, e.g., by process engineers, and evolved to more technical process representations. EPCs, for example, follow a strict scheme with business events followed by functions. They are widely adopted in industry and

well suited for the abstract modeling from the business perspective [16]. BPMN is a newer and more powerful process modeling notation. It supports abstract as well as technical process models and is typically more compact than EPCs. Both notations can be combined to support a holistic BP modeling process. This is, for example, the case in Software AG's model-to-execute process: EPCs are used for abstract, business-oriented models; BPMN is used for technical process models. A model transformation process is specified for the transition between EPCs and BPMN.

The integration of SPUs with EPCs and BPMN requires extensions to the modeling notations that address the characteristics derived from the streaming nature of event data. SPUs exhibit the following specific properties that cannot be expressed completely with existing EPC and BPMN elements:

- *Execution semantics:* After the instantiation, SPUs can run indefinitely; events arrive and are processed continuously, e.g., temperature measurements during the shipment transport. The completion semantics differ from service-like request/reply interactions where the reply triggers the process control flow to proceed. In contrast, completion of SPUs has to be triggered - either implicitly or explicitly. In either case, the completion indicates a clean shutdown. *Implicit completion* requires the specification of a condition that determines when the SPU should complete. Examples are a timeout in case no new events arrive, the detection of a certain event pattern, or dedicated events, e.g., shipment arrival. *Explicit completion* triggers the completion of an SPU externally. For example, when a process reaches a point where the processing of an event stream is not required anymore, e.g., shipment arrival has been confirmed.

- *Signaling:* The continuous processing inside of SPUs requires support to trigger concurrent actions, e.g., triggering exception handling in case of a temperature threshold violation without stopping the shipment monitoring SPU.

- *Event stream input and output:* The inputs for SPUs are event streams. An event stream is specified by a subscription to future events, e.g., temperature measurements for a certain shipment. The output is specified by an advertisement that describes the events producible by an SPU.

3.1.1. SPU Integration with EPCs

EPCs are a notation for computation independent models (CIM), a concept in model driven architectures [17]. CIMs, also referred to as business or domain models, are created from a business viewpoint and contain only few technical details. EPCs became popular as process modeling notation for SAP R/3 as well as notation in the context of ARIS (Architecture of Integrated Information Systems) [18]. ARIS is an approach for holistic business process modeling and management of enterprise information systems. EPCs consist of functions (e.g., confirm order) preceded and followed by business events (e.g., order arrived and order confirmed). Event stream processing can be modeled as such EPC functions; an EPC function refers to an SPU. In ARIS, EPC functions are supported by services with attached capabilities, e.g., an order confirmation service has the capability to send out confirmations [19]. To model SPUs in EPCs we specify an appropriate supporting service type for EPC functions that represents event stream processing. This involves extensions to EPCs on the basis of service type objects with capabilities.

Details of a service type object are modeled in a service allocation diagram; it describes a service from an abstract point of view. In the service allocation diagram arbitrary objects are connected to the service type object via associations. Connected objects are, for example, descriptions of the organization (organizational unit, responsible person) or data objects used as input or as output of the service.

We introduce *Event Stream Processing Services* (ESPSs) to support EPC functions that represent event stream processing. ESPSs are a distinct service category based upon service type objects [19]; they are represented with a custom symbol type (see Figure 3, center). Since event objects in EPCs are not sufficient to model event streams – they rather represent a state of a process than the continuous nature of event streams – we introduce *Event Stream Specifications* (ESSs) that reflect input data and output data in form of event streams. We adapt a cluster model object to represent these event streams (see Figure 3, left). The Event Stream Processing Unit type (see Figure 3, right) is used to refer to the technical realization of SPUs.

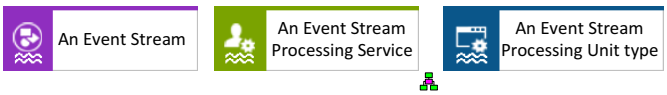

Figure 3: Extensions to EPCs: Event Stream Specification (ESS), Event Stream Processing Service (ESPS), and Event Stream Processing Unit Type

Definition 1. *In EPCs an **Event Stream Specification (ESS)** ($\rightarrow R_2$) references a stream of events. An ESS is a subtype of an abstract business object. The object type can be used as input or as output of functions, ESPSs, or Event Stream Processing Unit types. The attached connection type specifies whether the ESS is input to or output from other objects. An ESS used as input determines the subscription an ESPS has to issue. An ESS used as output determines the advertisement that describes the event output stream of an ESPS.*

Definition 2. *A function in an EPC that refers to an SPU is supported by an **Event Stream Processing Service (ESPS)** ($\rightarrow R_1, R_3, R_4$). An ESPS requires at least one ESS as input. It may have output ESSs. When the control flow reaches a function supported by an ESPS, this ESPS is activated with the specified ESS as input. The completion of the ESPS is triggered implicitly or explicitly ($\rightarrow R_5$).*

Implicit and explicit completion of ESPSs is expressed with different instantiation capabilities: Start Processing with Completion Condition and Start Processing. Explicit completion is also expressed as a distinct capability of the ESPS. Upon completion, either implicitly or explicitly, the ESPS stops processing, performs a clean shutdown, and passes on the control flow.

To trigger concurrent actions, ESPSs can send events; this is modeled as a loop. An ESPS can be modeled with combined explicit and implicit completion. An ESPS has an associated Event Stream Processing Unit type: it provides the link to technical process model representations. The Service Allocation Diagram for an ESPS is shown in Figure 4.

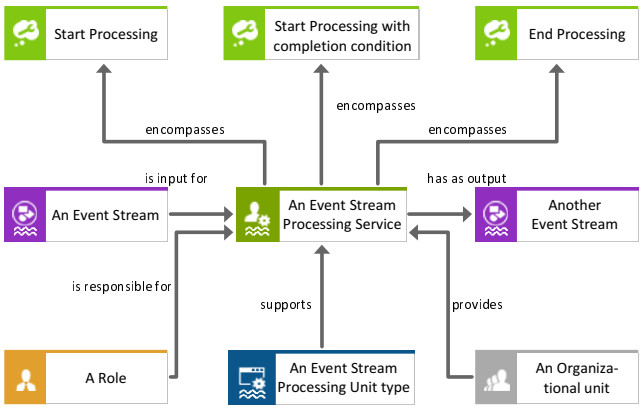

Figure 4: A Service Allocation diagram provides the abstract configuration of an Event Stream Processing Service. Objects, like the organizational unit, the responsible person, or input/output data, are connected to the service type object via associations.

3.1.2. Example: Modeling Shipment Monitoring with EPCs

To illustrate the application of our EPC extensions, we model the monitoring of environmental conditions in the order process introduced in Section 1 with EPCs. An SPU is used to process an event stream that contains environmental data events, e.g., temperature measurements. Figure 5 shows the service allocation diagram for the ESPS that represents this shipment monitoring SPU. The shipment monitoring ESPS requires shipment-monitoring events as input; it has capabilities to initialize the SPU with implicit and explicit completion.

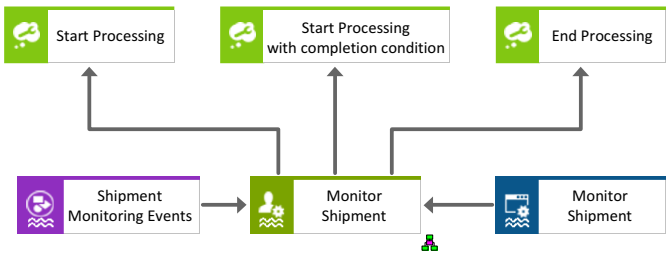

Figure 5: The Shipment Monitoring example Service Allocation Diagram

Figure 6 shows the EPC process model with implicit completion of the SPU. The monitor shipment function receives shipment-monitoring events as input event stream. The function is supported by the monitor shipment ESPS (see Figure 5); it uses a capability of the ESPS to initialize an SPU with an implicit completion condition. The implicit completion condition is assigned to the connection between the EPC function and the ESPS *start processing with completion condition* capability. Implicit completion is triggered when the shipment arrives at its destination address.

To report an environmental condition violation, the SPU is followed by an XOR operator: the process control flow triggers the environmental condition violation event and directly returns to the monitoring function. This control flow loop is instantaneous so that the SPU continues processing; the event stream processing is not interrupted at the technical layer. This SPU-specific loop pattern is used to model asynchronous messaging in EPCs as required in event stream processing scenarios. The completion of the SPU is triggered as soon as the completion condition is fulfilled, i.e., when the shipment reaches its destination. With completion of the SPU, the control flow moves on to the XOR operator and results in the shipment arrived event, which completes the process.

Figure 7 shows the order process with explicit completion of the SPU. The monitor shipment function is supported by the ESPS capability for initialization with explicit completion. An additional process step is added after the arrival confirmation to trigger the explicit completion. The *complete shipment monitoring* function uses the *end processing* capability of the shipment monitoring ESPS to trigger the completion of the shipment monitoring.

It is also possible to model SPUs with implicit and explicit completion in parallel. The monitor shipment function shown in Figure 6 is then combined with the complete shipment function shown in Figure 7.

3.1.3. SPU Integration with BPMN

BPMN 2.0 is widely adopted in industry and has a broad tool support. From a technological perspective, processes can be modeled in different granularities with BPMN. From a semantical perspective, the single building blocks (BPMN tasks) of a process model should reflect business functions and hide technical details. Our extensions to BPMN are shown in Figure 8. As for EPCs, we define *Event Stream Specifications* (ESSs) that reflect input data and output data in form of event streams. Further, we introduce *Event Stream Processing Tasks* (ESPTs) to model SPUs.

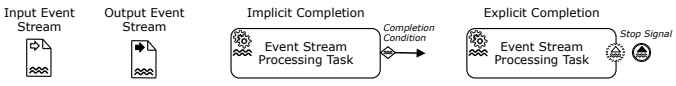

Figure 8: Extensions to BPMN: Event Stream Specifications (ESSs) and Event Stream Processing Tasks (ESPTs)

Definition 3. *In BPMN an **Event Stream Specification (ESS)** ($\rightarrow R_2$) references a stream of events and their parameters. ESSs can be used as input and output of ESPTs. An ESS used as input determines the subscription an ESPT has to issue. An ESS used as output determines the advertisement that describes the event output stream of an ESPT.*

Definition 4. *In BPMN an SPU is modeled as **Event Stream Processing Task (ESPT)** ($\rightarrow R_1, R_3, R_4$). An ESPT requires at least one ESS as input. It may have output ESSs. When the control flow reaches an ESPT, it is activated with the specified ESS as input. The transition from the active state to the completing state (see BPMN task lifecycle [4, p. 428]) is triggered implicitly or explicitly ($\rightarrow R_5$).*

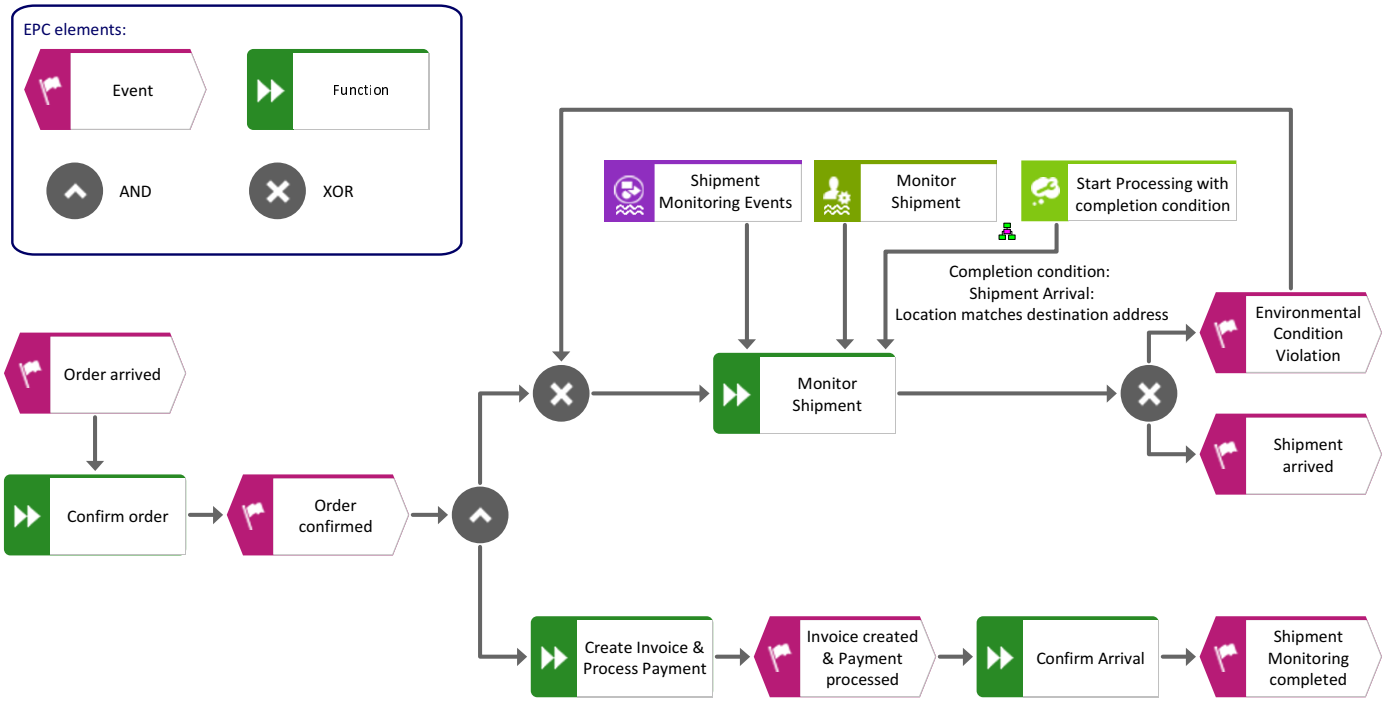

Figure 6: The Shipment Monitoring example BP as EPC utilizing an SPU with implicit completion. The previously defined EPC extension elements are used.

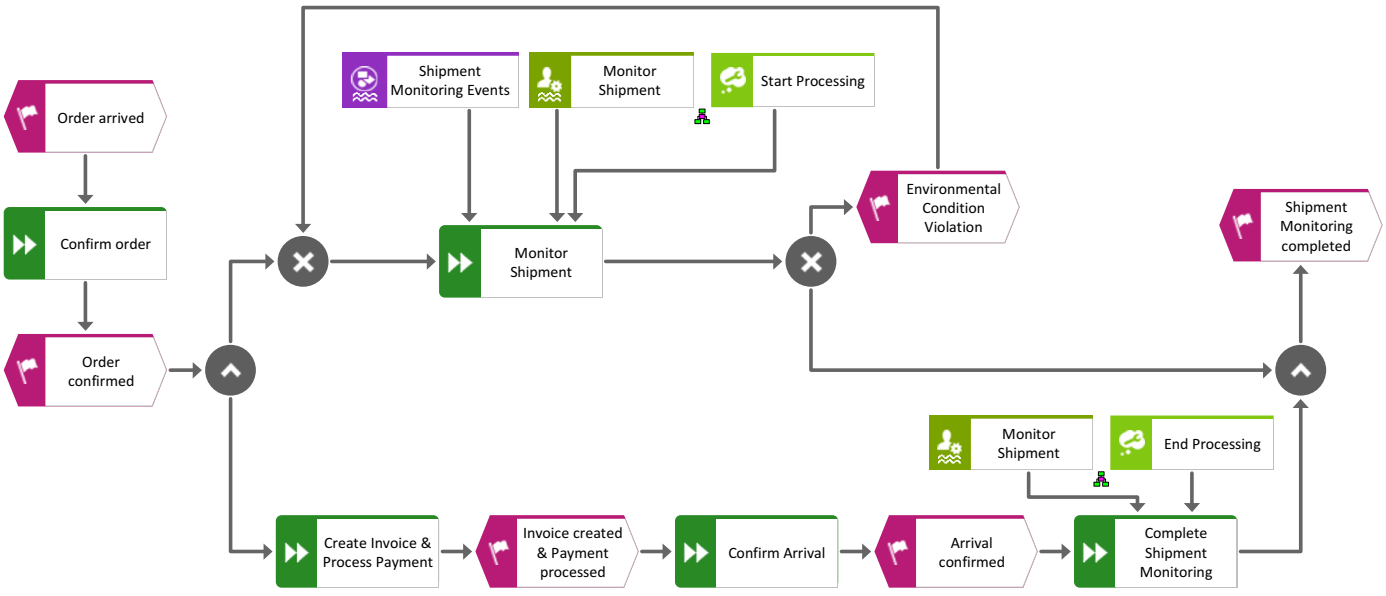

Figure 7: The Shipment Monitoring example BP as EPC utilizing an SPU with explicit completion

The implicit completion of an ESPT is realized with a modified conditional sequence flow; the condition determines when the ESPT completes. The explicit completion is realized with a dedicated signal. It is attached as non-interrupting signal to the boundary of the ESPT. Upon completion, either implicitly or explicitly, the ESPT stops processing, performs a clean shutdown, and passes on the control flow, i.e., no additional token is created at catching the shutdown signal. To trigger concurrent actions, ESPTs can activate outgoing sequence flow elements while remaining in the active state. Non-interrupting conditional events can be attached to the boundary of an ESPT to indicate a conditional activation of sequence flow elements. An ESPT can be modeled with combined explicit and implicit completion.

3.1.4. Related BPMN Concepts

Events are part of the BPMN specification. However, events in BPMN are meant to affect the control flow in a process [4, p. 233]. Events modeled as ESS do not exhibit this property; they are rather a source of business-relevant information that is exploited within the process. Due to these different semantics, events in the sense of the BPMN standard are not appropriate to model event stream input/output to/from SPUs.

From the task types contained in the BPMN standard, service tasks, business rule tasks, loop service tasks, and multiple instance service tasks share properties with SPUs. However, the modeling of SPUs with those existing BPMN task types is not feasible.

Service Tasks are containers for business functions that are implemented as SOA services. The execution semantics for service tasks state, that data input is assigned to the service task upon invocation; upon completion output data is available. For SPUs, this separation is not feasible; input data arrives continuously and output data can be available during task execution in form of output streams. Therefore, service tasks are no appropriate representation for SPUs. In *Business Rule Tasks*, event stream processing can be used to check conformance with business rules. However, event stream processing supports a wider application spectrum than conformance checking, e.g., real-time shipment tracking. Further, output in form of event streams is not part of business rule tasks; their purpose is to signal business rule evaluation results. *Loop Service Tasks* perform operations until a certain stop condition is met. However, the whole loop task is executed repeatedly, i.e., a repeated service call. This repeated execution of a business function depicts a different level of abstraction compared to continuous processing inside an SPU; SPUs perform continuous processing to complete a single business function. To use loop tasks for event stream processing, the process model would have to define the handling of single events rather than the handling of event streams. This conflicts with the abstraction paradigm of business functions and degrades coherence across the layers. *Multiple Instance Service Tasks* allow the execution of a task in parallel, i.e., parallel service calls. However, like loop tasks, this would require one task per event which conflicts with the intention to encapsulate business functions in tasks. In addition, the number of task instances executed in parallel is static and determined at the beginning of the task. This is not suitable for event processing since the number of events is not known a priori.

In general, BPMN tasks have no support for triggered completion required in event processing. In addition, event streams cannot be represented as input to and output from tasks. Thus, we extend BPMN with ESPTs. ESPTs support implicit and explicit completion, an essential part of SPU execution semantics. Further, we introduce ESSs as input to and output from ESPTs in the form of event streams.

3.1.5. Example: Modeling Shipment Monitoring with BPMN

To illustrate the application of our BPMN extensions, we model the monitoring of environmental conditions in the order process introduced in Section 1 with BPMN. Figures 9 and 10 show two variants with different completion strategies. The shipment monitoring is an SPU that receives monitoring events as input stream. This shipment monitoring SPU is modeled as an ESPT in BPMN; the monitoring events are assigned as an input ESS. The monitoring task can send a message event (as concurrent action) to indicate a violation of environmental conditions, e.g., temperature threshold exceeded. The message event can activate a task or trigger a different process for handling the exception; this exception handling is omitted here for brevity.

In Figure 9, the shipment monitoring is modeled with explicit completion semantics. As soon as the shipment has arrived, the monitoring is not required anymore. Thus, the monitoring task completion is triggered by sending the stop signal.

In Figure 10, the shipment monitoring is modeled with implicit completion semantics. This requires the definition of a completion condition. In our example, we specify the shipment arrival: when the location of the shipment matches the destination address, the monitoring is completed. Other implicit completion conditions could be dedicated arrival events, e.g., arrival scans of shipment barcodes, or timeouts, e.g., no new monitoring events for the shipment arrive. The condition needs to be evaluated inside the SPU, thus support for different condition types depends on the technical infrastructure that executes SPUs.

The modeling of data flow between ESPTs is shown in Figure 11. The shipment monitoring SPU outputs a stream of events that indicate exceeded temperature thresholds. The *report threshold violation* SPU receives these events and implements the reporting, e.g., by sending an email in case temperature thresholds were exceeded multiple times. Completion of the reporting SPU is triggered explicitly after the confirmation of arrival of the shipment. The monitoring SPU completes implicitly.

ESPTs can also be modeled with implicit and explicit completion in parallel as shown in Figure 12. The implicit completion is the default case: the monitoring stops as soon as the shipment has reached its destination. In addition, an explicit completion is modeled: when a customer cancels the order, shipment monitoring becomes obsolete.

The output of ESPTs can affect process execution as shown in Figure 13. When an environmental condition violation is de-

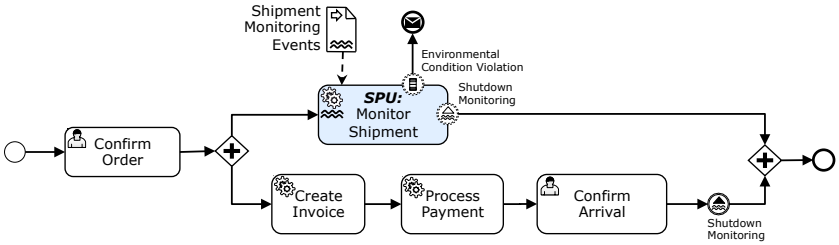

Figure 9: Shipment monitoring SPU that is stopped explicitly. The data input/output of the service tasks is omitted.

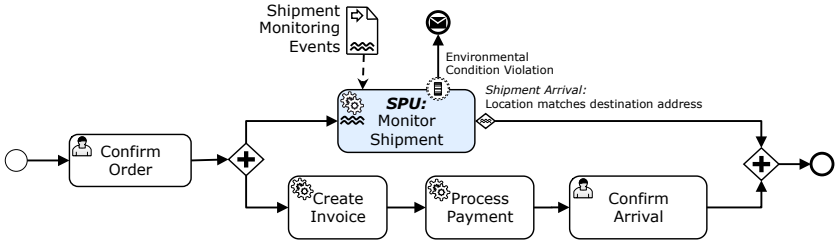

Figure 10: Shipment monitoring SPU that is stopped implicitly. The data input/output of the service tasks is omitted.

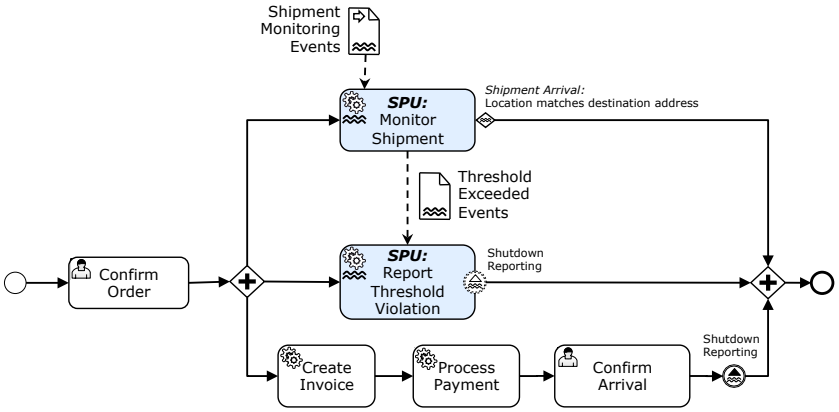

Figure 11: Interacting SPUs: Output from monitoring SPU is used as input for reporting SPU. The SPUs have different completion strategies.

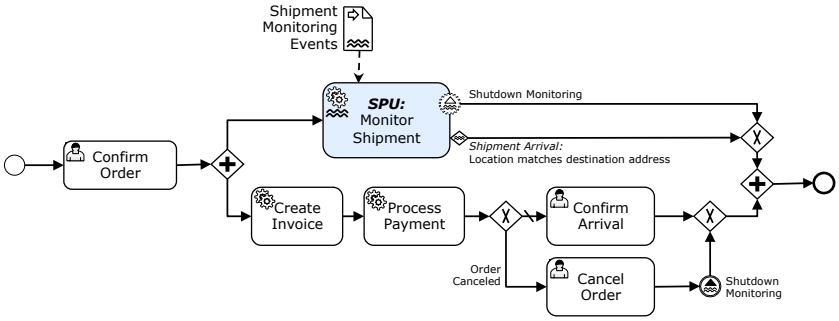

Figure 12: ESPT with implicit and explicit completion: Upon cancelation of an order the shipment monitoring is completed explicitly; upon shipment arrival, monitoring completes implicitly.

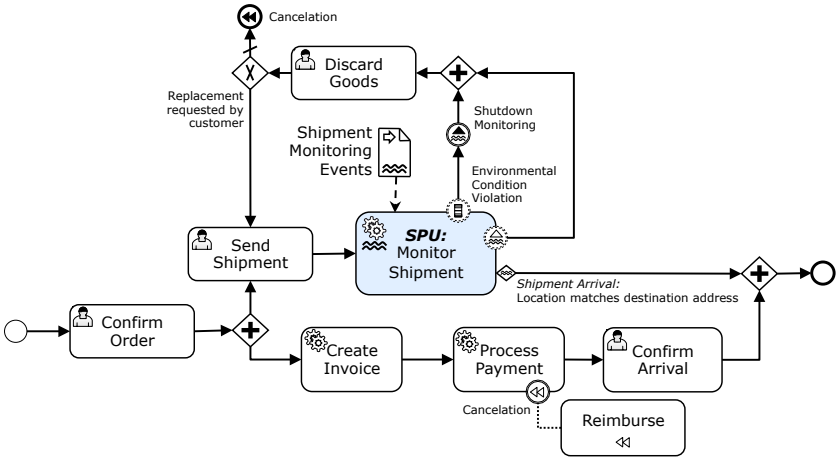

Figure 13: ESPT that affects process execution: Upon detection of an environmental condition violation the shipment monitoring is stopped and the customer decides upon cancelation of the order.

tected the shipment monitoring is stopped. After completion of the shipment monitoring ESPT, discarding of goods is triggered and the customer decides upon cancelation of the order. In this case a reimbursement is triggered via a compensation event.

3.1.6. Transformation from EPC to BPMN

EPCs can be used for abstract process models from a business perspective; processes are modeled at the CIM layer. For the execution of processes a technical process representation is required that correlates with an executable process representation. Since BPMN is a suitable notation for such technical models, EPC models can be transformed to BPMN in order to support automated process execution. This also requires a transformation of SPUs. Since SPUs are independent of a concrete modeling notation a mapping of SPUs in EPCs to SPUs in BPMN is possible. The transformation from EPCs to BPMN can also be partly automated. This is, for example, supported by the Software AG ARIS business process platform.

Theoretical approaches for a mapping from EPCs to BPMN are given in [20]. The basic concept is the transformation of EPC functions into a BPMN tasks. EPC events are disregarded as long as they are not decision conditions for connectors. Operators, organizational elements, and data objects are directly transformed into their BPMN representations. In ARIS, for example, functions supported by a screen object are mapped to BPMN user tasks; functions assigned with an organizational object only are mapped to manual tasks. Analogously, we define the mapping from EPC functions that are supported by ESPSs to ESPTs in BPMN as shown in Figure 14.

An EPC function that is supported by an ESPS with implicit completion is mapped to an ESPT with implicit completion, i.e., the completion condition assigned to the connection between the EPC function and the capability is mapped to the outgoing sequence flow of the ESPT ($\Leftrightarrow_a$ in Figure 14). An EPC function that is supported by an ESPS with explicit completion is represented by an ESPT with explicit completion; the completion is triggered by an SPU intermediate boundary non-interrupting signal event ($\Leftrightarrow_b$ in Figure 14). This event is trig-

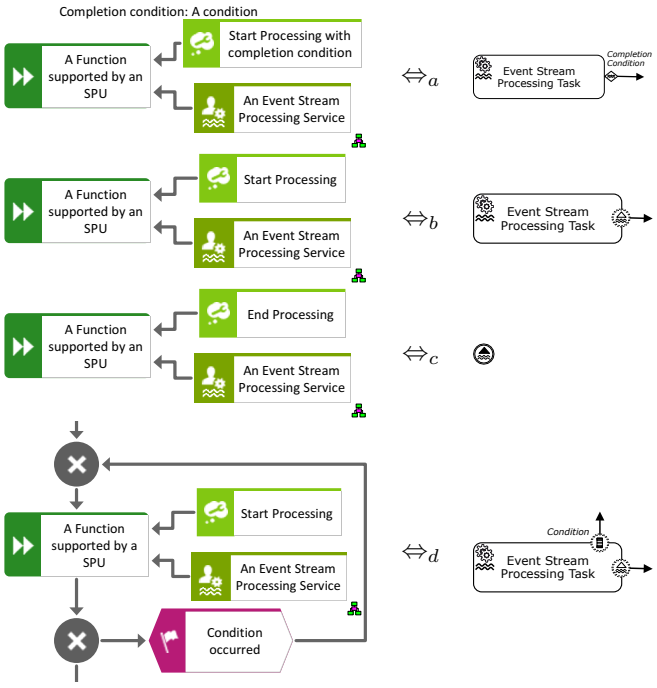

Figure 14: Mapping of SPUs between EPC and BPMN

gered by the SPU signal intermediate throwing event, which is the BPMN representation for the explicit completion capability used by an EPC function ($\Leftrightarrow_c$ in Figure 14). ESPTs in BPMN allow non-interrupting conditional events to be attached to the boundary of an ESPT for a conditional triggering of subsequent actions. In EPCs this is modeled with an event loop enclosing solely the event stream processing function ($\Leftrightarrow_d$ in Figure 14). The function supported by an ESPS is not completed in this case; the control flow directly returns. The outgoing EPC event is consumed elsewhere and triggers concurrent functions.

Event streams in EPC are mapped to their corresponding elements in BPMN: an EPC input ESS (*is input for* association) is mapped to the BPMN input ESS; an EPC output ESS (*has as output* association) is mapped to the BPMN output ESS. In EPCs the distinction between input and output event stream is based on the type of the association connection, in BPMN individual elements for both cases exist.

3.2. Workflow Execution Layer

The execution of BPs by an IT infrastructure requires a transition from the technical process model to an executable process format. The BPMN 2.0 standard itself specifies such execution semantics; the standard also provides examples for the mapping between BPMN and BPEL. Independent of the concrete technical representation format, the goal is to bridge the semantic gap between the technical model notation and interfaces of IT components so that the process can be executed automatically. The transition from a technical process model towards an executable process representation requires adding additional technical details.

For different task types and control flow components, execution languages provide executable representations. When the mapping of graphical process task and process control flow elements is complete and all necessary data is specified, the process execution engine is able to execute instances of the process. Each instance reflects a concrete business transaction, e.g., processing of Order No. 42. For each process instance, the execution engine orchestrates the different tasks, passes on task input and output data, and evaluates conditions specified in the control flow. Examples are the execution of BPMN service tasks and human tasks: a service task can be executed by calling a web service. For this, the execution engine needs the service address as well as the input data to send to a service and the format of the expected output data from this service. For the execution of human tasks, process execution engines typically provide a front end to perform the work necessary to complete the task.

At the execution layer we define the technical details that allow ESPTs to be mapped to IT components. The mapping mechanism has to take into consideration that events arrive indefinitely and are not known when the control flow reaches an ESPT. Thus, the data input must be specified as a *subscription* for desired events that arrive during the execution period of an ESPT. During process execution, this subscription has to partition the event stream in process instance specific sub streams: when a process instance is created for a certain business task, e.g., processing of Order No. 42, the event stream has to be partitioned in sub streams of events relevant for the different order process instances. This requires events to hold an attribute that allows an association with a process instance. Monitoring events contain, for example, a shipment ID. This is shown in Figure 15: a monitoring task must be active for each process instance. This task instance has to receive all monitoring events for the shipment that is handled in this process instance. Given that each event carries a shipment ID, each monitoring task instance can issue a subscription for the appropriate events using the shipment ID as filter. When the process instance ID correlates with the shipment ID, the subscription can also be derived by the process execution engine on the basis of the process instance ID.

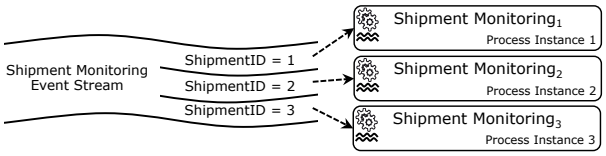

Figure 15: Process execution: Event Stream Processing Tasks (ESPTs) receive sub streams of events

The subscription parameters are essential for the instantiation of an ESPT. Like the input data passed on to a service during a service call, the subscription is part of the input data during an ESPT instantiation. Further, when the ESPT is modeled with an implicit completion, the completion condition is part of the input data required for the instantiation. As for ESPT completion, different ESPT instantiation strategies are possible. The push-based nature of stream processing allows an implicit creation of ESPT instances upon the arrival of appropriate events. In addition, ESPT instances can also be created explicitly by the process execution engine. When switching from explicit to implicit instantiation, the responsibility of instantiation moves from the process execution engine to the IT infrastructure. Implicit instantiation is useful when the moment of instantiation cannot be determined by the execution engine. It is also the more natural approach with respect to the characteristics of event streams; application logic is executed as soon as appropriate events are available. We support both instantiation schemes to allow for a high flexibility ($\rightarrow R_8$). Independent of the instantiation scheme, a subscription does not guarantee the availability of events, e.g., that events for Shipment No. 42 are published. Explicitly instantiated ESPTs can use a timeout to detect such an absence of events. With implicit instantiation, ESPT instances are not created in this case; the execution environment can detect and report this.

3.2.1. ESPT Instantiation

The execution of a BP leads to process instances that may run in parallel. Each ESPT in the model has corresponding ESPT instances that are created during process execution. Each ESPT instance processes the event streams relevant for a particular process instance (see Figure 15). The process execution engine can create an ESPT instance explicitly during the execution of a process instance. The subscription parameters required for the explicit instantiation must be derived per process

instance; they define the sub stream of events that has to be processed by a particular ESPT instance, e.g., monitoring events for Shipment No. 42. The *explicit instantiation* is specified as follows ($\rightarrow R_6, R_7, R_8$):

```
EsptInstantiate(EsptName, EventStreamFilter,
    SubStreamAttribute,
    SubStreamId [, CompletionCondition])
```

For the monitoring example, the explicit instantiation of a monitoring task for Shipment No. 42 without and with completion condition is:

```
EsptInstantiate(MonitorShipment,
    MonitoringEvent,
    ShipmentId, 42)

EsptInstantiate(MonitorShipment,
    MonitoringEvent,
    ShipmentId, 42,
    "destination.equals(location)")
```

An ESPT is referenced by name: `EsptName`, e.g., `Monitor Shipment`. The subscription parameter has three parts: First, a general filter for events of interest that applies to all instances of an ESPT is specified as `EventStreamFilter`, e.g., monitoring events. Second, the `SubStreamAttribute` defines the part of the event data that partitions the event stream with respect to ESPT instances, e.g., the shipment ID; both are static expressions and derived based upon the ESS used in the model. Third, the `SubStreamId` defines the concrete event sub stream for which an ESPT instance should be created, e.g., Shipment No. 42. The `SubStreamId` is dynamic and derived per process instance by the execution engine at run time, e.g., based on the process instance ID. The optional `CompletionCondition` can be specified for implicit completion, e.g., defining a time out.

With implicit instantiation, the process execution engine only registers a static subscription pattern for an ESPT once, e.g., with the registration of the process. Since events arise in a push-style manner, the IT infrastructure is able to create ESPT instances implicitly at run time. The *implicit instantiation* is specified as follows ($\rightarrow R_6, R_7, R_8$):

```
EsptRegister(EsptName, EventStreamFilter,
    SubStreamAttribute
    [, CompletionCondition])
```

For the shipment monitoring example, the ESPT registration is:

```
EsptRegister(MonitorShipment, MonitoringEvent,
    ShipmentId)
```

In contrast to explicit instantiation, the execution engine is not responsible for the dynamic subscription part anymore. Rather, the IT infrastructure ensures, that an ESPT instance is created for each distinct value of the `SubStreamAttribute`, e.g., for each shipment ID.

Upon implicit instantiation, the creation of ESPT instances is not synchronized with the control flow of the process execution. ESPT instances are created based upon the availability of events, i.e., as soon as events for a certain entity instance are available the corresponding ESPT instance is created. The availability of events for Shipment No. 42, for example, results directly in the creation of an ESPT instance that processes these events. This happens independently of the control flow of the process execution. In cases where this behavior is not desired, a synchronization step has to be performed between the beginning of the actual event processing and the control flow of the process execution. Two cases have to be taken into consideration: First, the process control flow reaches an ESPT and no ESPT instance has been created yet. Second, an ESPT instance is created although the control flow has not reached the ESPT.

In the first case, the process execution blocks and proceeds after the completion of an ESPT instance; this behavior is equivalent to the execution semantics of, e.g., service tasks. However, the process execution engine has no control over the instantiation and thus relies on the IT infrastructure. In the second case, the already created ESPT instance has to wait for the control flow of the process execution engine. This can be achieved by implementing a lock after the creation of an ESPT instance; this lock is released upon a dedicated signal from the process execution engine. Since a dedicated signal from the process execution engine is required, the explicit creation of ESPT instances is an alternative in such cases.

ESPTs with Output Event Streams. ESSs used as output of ESPTs can be mapped to advertisements. Advertisements inform the IT infrastructure about the event types published by event producers; they allow the validation of subscriptions, i.e., whether events are potentially available for an issued subscription. The validation of subscriptions allows the identification of inconsistencies between demand and availability of event streams upon registration of ESPTs. The event types for the advertisements are derived from output ESSs are included as additional parameter at the instantiation or registration of ESPTs:

```
EsptInstantiate(EsptName, EventStreamFilter,
    SubStreamAttribute,
    SubStreamId [, CompletionCondition]
    [, PublishedEventType])

EsptRegister(EsptName, EventStreamFilter,
    SubStreamAttribute [, CompletionCondition]
    [, PublishedEventType])
```

A shipment monitoring SPU with implicit completion that outputs *threshold exceeded events* (see Figure 11) is then registered as follows:

```
EsptRegister(MonitorShipment, MonitoringEvent,
    ShipmentId,
    "destination.equals(location)",
    ThresholdExceededEvent)
```

When the *report threshold violation* SPU is registered, the IT infrastructure knows that the demanded *threshold exceeded event* type is available:

```
EsptRegister(ReportExceededThreshold,
    ThresholdExceededEvent,
    ShipmentId)
```

3.2.2. ESPT Completion

For the explicit completion of an ESPT instance, the process execution engine has to advise the IT infrastructure to perform a shutdown of particular ESPT instances, e.g., the shipment monitoring of Shipment No. 42. The completion command is specified as follows ($\rightarrow R_8$):

```
EsptComplete(EsptName,SubStreamId)
```

The `SubStreamId` identifies the ESPT instance that should be completed. In the monitoring example for Shipment No. 42, the following completion command is issued after the arrival confirmation task:

```
EsptComplete(MonitorShipment,42)
```

We distinguish between the control commands to manage ESPTs and the ESPT execution semantics. The control commands to register, instantiate, and complete ESPTs follow a request/reply pattern. Thus, our integration approach of event streams with BPs can be mapped to web service invocations. Web service invocation capabilities are part of most process execution engines so that ESPTs can be registered, instantiated, or completed; the ESPT name as well as further subscription and completion parameters are specified as variables in the service invocation. In addition to service invocation mechanisms, it might be necessary to implement a back channel for control flow purposes. Implicitly completing ESPT instances might have to notify the process execution engine about completion. This is the case when the control flow waits for a completion of an ESPT, e.g., when an ESPT is used before a BPMN AND-Join.

3.2.3. ESPT Mapping in BPEL

Business process models that contain ESPTs can be mapped to BPEL. However, the BPEL standard [8] does not support all concepts required for a complete mapping of the different instantiation and completion strategies. ESPTs with explicit instantiation and explicit completion can be mapped to standard BPEL: the explicit instantiation is realized as web service call. The return from this call is blocked by the IT infrastructure until the ESPT instance is explicitly stopped by an `EsptComplete` service invocation. Explicit instantiation and completion in BPEL are as follows:

```
<invoke partnerLink="EsptWebService"
    operation="EsptInstantiate"
    inputVariable="explicitInstantiateParams"
    outputVariable="completed"/>

<invoke partnerLink="EsptWebService"
    operation="EsptComplete"
    inputVariable="explicitCompletionParams"/>
```

With implicit instantiation, single ESPT instances are transparent to the process execution engine. The registration of ESPTs has to be performed once with the registration of a process; the ESPT instances are then created automatically. The BPEL standard does not support hooks for service invocation upon the registration of new processes. Thus, a BPEL execution engine has to be extended with these capabilities to support implicit instantiation of ESPTs. The hook for execution at process registration can be part of the BPEL code itself; when a new process is registered and checked, this part of the process is executed only once:

```
<atRegistration><invoke
    partnerLink="EsptWebService"
    operation="EsptRegister"
    inputVariable=
      "implicitInstantiateParams"/>
    </atRegistration>
```

When an ESPT is invoked implicitly, there is no BPEL web service invocation in each process instance. Thus, a blocking service invocation cannot be used to interrupt the control flow until completion of an ESPT instance. Rather, the process execution engine has to be notified externally about the completion of an ESPT instance so that the control flow can proceed. Extensions to BPEL engines to react on such external triggers have been proposed, e.g., in [21] and [22]. The ESPT can be mapped to a barrier that is released when the ESPT instance signals its completion.

3.3. IT Infrastructure Layer

SPUs require a technical representation at the IT infrastructure layer. In [14] we present a suitable component model and runtime infrastructure to encapsulate event stream processing. We introduce event applets, in short *Eventlets*, as service-like abstraction for event stream processing. Our model benefits from concepts known from services; it hides application logic so that Eventlets represent business functions. We extend the runtime environment presented in [14] to allow for the integration with BP execution engines. We now introduce the main concepts of Eventlets to make this paper self-contained; we then present the extensions to the Eventlet middleware. We adapt the more general Eventlet nomenclature of [14] to fit the terminology of this paper.

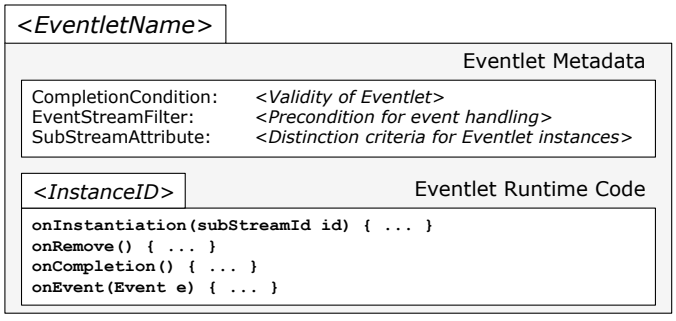

Figure 16: Eventlet structure: Eventlet metadata and Eventlet runtime methods

Eventlets encapsulate event stream processing logic with respect to a certain entity, e.g., shipments ($\rightarrow R_{10}$). An Eventlet instance subscribes to events of a certain entity instance, e.g., Shipment No. 42 ($\rightarrow R_{11}$). The basic structure of an Eventlet is shown in Figure 16. The grouping attribute to define the sub stream of events associated with a certain entity instance

is specified as Sub Stream Attribute[2] in the Eventlet metadata, e.g., the shipment ID. Further, the metadata holds the Completion Condition[3], e.g., a timeout, as well as the Event Stream Filter[4] as a general subscription filter applied by all Eventlet instances, e.g., monitoring event. Eventlet instances are created implicitly or explicitly ($\rightarrow R_9$). With implicit instantiation the middleware ensures that an Eventlet instance is active for each distinct value of the sub stream attribute, e.g., for each shipment in transport. With explicit instantiation, Eventlet instances are created manually by specifying a concrete sub stream attribute value, e.g., Shipment No. 42. The completion of Eventlet instances is triggered implicitly by the completion condition or explicitly by a command ($\rightarrow R_9$). Eventlet instances run in a distributed setting and have a managed lifecycle; application logic can be executed upon instantiation, removal, completion, and upon event arrival ($\rightarrow R_{11}$).

In our monitoring example, an Eventlet holds application logic to detect temperature violations. This can involve a lookup in a database at instantiation to retrieve the temperature threshold for a certain shipment. It can also involve issuing complex event processing (CEP) queries to rely on the functionality of a CEP engine for temperature violation detection. An evaluation of CEP queries encapsulated in Eventlets is presented in [14]; we show that Eventlets can process a couple of thousands events per second while being scalable across machines. The semantics of ESPT execution (cf. Section 3.2) are implemented by the Eventlet middleware. The `EsptInstantiate` and `EsptRegister` invocations provide the Eventlet middleware with the metadata to explicitly or implicitly create Eventlet instances. For implicit instantiation, the middleware creates a so-called Eventlet Monitor; it analyzes the event stream and detects the need to create Eventlet instances as soon as events of a new entity instance, e.g., a new shipment, occur. Like services, Eventlets are managed in a repository and identified via the `EsptName`.

3.3.1. Eventlet Middleware Extension

The Eventlet middleware infrastructure uses the Java Message Service (JMS) for event dissemination. JMS supports publish/subscribe communication with event content sensitive subscriptions. Our implementation supports events in attribute-value and XML representation. For attribute-value events, the Event Stream Filter is specified as JMS message selector in a SQL-like syntax. The Sub Stream Attribute is the name of an attribute, e.g., shipmentID. For XML events, Event Stream Filter and Sub Stream Attribute are specified as XPath expressions on the event content. For implicit completion of Eventlet instances, timeouts are supported.

We extended the Eventlet middleware in [14] to support ESPT execution. As shown in Figure 17, the Eventlet middleware is configured and controlled using a command bus. This command bus is realized as a set of JMS queues and topics to which all middleware components connect. We added a web service interface to the Eventlet Manager. The new interface accepts service invocations as described in Section 3.2 and uses the internal command bus to start or stop Eventlet Monitors and Eventlet instances. The web service interface is implemented as Java Enterprise application. The Eventlet middleware can be deployed on multiple application servers and use a JMS infrastructure in place. It is designed for scalability: Eventlet instances can run on arbitrary machines.

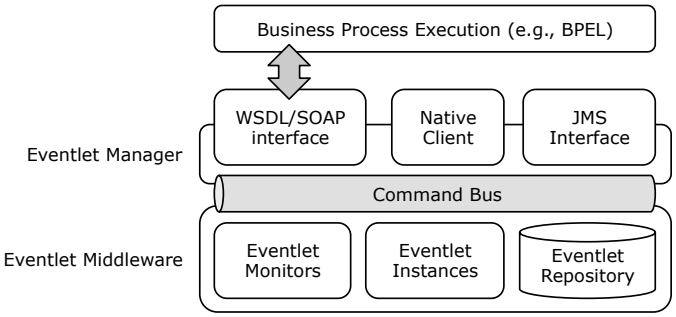

Figure 17: Eventlet middleware access via web service

4. Implementation: Process Modeling and Execution

We implemented SPU modeling capabilities for EPCs and BPMN in Software AG's ARIS platform. ARIS is a business process platform for business process analysis, enterprise architecture, and governance, risk & compliance. ARIS supports modeling of processes with EPCs as well as with BPMN. For our implementation we used ARIS Design Server 9.0 and the ARIS Architect 9.0. To support SPU modeling in EPCs and BPMN within ARIS, we added new notation element symbol types and connection attributes for EPCs (see Figure 3) and BPMN (see Figure 8) to the configuration of the ARIS server; the ARIS server acts as central repository for process models and process model components.

For EPCs we added symbol types for:

- a cluster/data model object (derived from cluster symbol type) to model ESSs;
- a service type object (derived from business service symbol type) to model ESPSs; and
- an application system type object (derived from software service symbol type) to model the technical representation of ESPSs.

For BPMN we added symbol types for:

- two cluster/data model types (derived from data input/output symbol types) to model ESSs;
- a function type object (derived from service task symbol type) to model ESPTs; and
- two event object types (derived from signal intermediate event symbol type) to model SPU signal intermediate events (throwing and non-interrupting) required for explicit completion.

[2]Referred to as Instantiation Expression in [14].
[3]Referred to as Validity Expression in [14].
[4]Referred to as Static Expression in [14].

Further, we added an attribute type and an attribute type symbol for BPMN to model the condition for implicit completion.

With the added symbol types, attribute symbol types, and attribute types it is possible to model SPUs and event streams in EPC and BPMN diagrams. All new modeling elements are usable in different diagram types as the symbols they are derived from. They can be used, for example, in application system type diagrams, all types of EPCs, function allocation diagrams, and service allocation diagrams.

Process Model Execution

One goal of BPM is to enable the automated execution of business process models. In Section 2 we introduce the different layers in such a model-to-execute (M2E) process and show that a high coherence across the process modeling layer, the process execution layer, and IT infrastructure layer is necessary. Our approach of SPUs is a mechanism to encapsulate event stream processing to achieve such a high coherence and to provide the foundation for M2E.

Our extensions to EPCs allow business experts to create abstract BP models that contain SPUs; our extensions to BPMN along with the proposed EPC-to-BPMN mapping allow the transformation of abstract EPC models to technical BPMN models. These BPMN models are further refined during the M2E process and brought to execution. The execution as such is detailed in Section 3.2 where the mapping between ESPTs and service invocations is presented.

We are working with Software AG on the implementation of M2E mechanisms that support SPUs. In the following we illustrate the M2E workflow and show how SPU integration is achieved. Our M2E approach is based upon the Software AG ARIS, CentraSite, and webMethods product suites. Business process analysis and process modeling is supported by the ARIS Architect; it provides an integrated platform where process models are created and governed collaboratively. At the beginning of the M2E workflow EPC process models are created with the ARIS Architect; these models reflect an abstract business perspective. In the next step a transformation process is applied; abstract EPC process models are mapped into a logical process model represented with BPMN. In ongoing work we are working on the customization of the ARIS model transformation framework, which performs the EPC-to-BPMN mapping. We are integrating the mapping of ESPSs in EPCs to ESPTs in BPMN. This allows a partly automated transformation from EPC models with ESPSs to BPMN models.

After the transformation the resulting BPMN model needs to be refined by a process engineer. This involves the adaptation to technical concepts and restrictions, e.g., adapting events for inter and intra process communication and error handling scenarios. Finally, changed process elements are linked to the related elements in the original EPC model using the process alignment capability of ARIS. This allows a synchronization of the process between abstract and technical layer; refinements in the EPC model are applied - if possible automatically - to the BPMN model and vice versa.

The next step in the M2E workflow is the transition from a technical model in ARIS to an executable process representation that is deployed to a process execution system. For SPUs this involves linking the Eventlet middleware via its web service interface to ESPTs of the BPMN model (see Section 3.2). In Software AG's M2E the executable process is created with the Software AG Designer. Process developers import the technical process model from ARIS in the Software AG Designer. In the Designer the process is represented as a technical BPMN diagram and technical implementations are mapped to process steps. This is supported by the Software AG CentraSite service repository: the web service interface of our Eventlet middleware is registered with CentraSite. This allows an easy assignment of the Eventlet service (including invocation parameters) to ESPTs in the model. Currently, only explicit instantiation and completion of SPUs is supported. Implicit instantiation would require a service invocation at process registration, implicit completion would require a feedback mechanism from the Eventlet middleware to the process execution environment.

Processes are synchronized between the webMethods platform and the ARIS platform. Changes to technical processes are propagated to the abstract process models and vice versa. During this round tripping, approval steps ensure that processes at the IT level and at the business level remain synchronized. The executable process model is deployed to the webMethods Integration Server where the individual process steps are executed by the different components of the webMethods BPMS; SPUs are executed by the Eventlet middleware controlled by web service invocations from webMethods BPMS.

5. Related Work

Events are part of various BP modeling notations like BPMN and EPCs [7, 3, 4]; they trigger functions/tasks and influence the process control flow. Event processing is often applied to monitor process execution. Events can be detected by a process execution engine and depict, for example, activation/completion of activities or sending/receiving of messages; events can also be defined implicitly based upon object state changes, e.g., changes in data records [23, 24]. Based upon such events, control flow deviations in single process instances can be detected [25]; it is also possible to check conformance in cross-organizational processes based upon message exchanges [26]. In contrast, our approach focuses on event processing as part of single process instances. Such incorporation of (complex) events leads to more reactive and dynamic processes. This is a core concept in event-driven architectures (EDA) [27, 28] or event-driven SOA [29]. However, event streams do not have explicit representations in BPMN or EPCs. Currently, event streams have to be modeled explicitly as multiple events, e.g., using loops that process events. Such explicit modeling of complex events and event processing is for example presented in [30, 31, 32, 33, 34]. The problem is, that process models are often created by business experts without detailed knowledge about technical details of event processing. Further, to make models intuitively understandable, modelers should use as few elements as possible with self-explaining activity labels [16]. Thus, activities should represent business functions. Services are a successful abstraction mechanism to support this. Services

represent business functions and exhibit a data input/output interface [6]. Process models do not (and should not) contain the application logic of a service; this is left to service developers who can use more appropriate modeling notations to describe the technical details. Thus, the approach in this work confers basic service concepts [35] to event stream processing and introduces SPUs as an appropriate abstraction. We concentrate on control flow oriented business process models represented with BPMN and EPCs. However, the SPU concept is also applicable to alternative approaches for managing business operations and processes, e.g., the Guard-Stage-Milestone (GSM) approach, which allows for a more declarative modeling of business activities [36]. In GSM, SPUs can be represented by stages. Instantiation is modeled as guards: a stage is activated when an event stream becomes available (implicit instantiation guard) or when an event triggers the start of event stream processing (explicit instantiation guard). Completion is modeled as milestones: an implicit completion milestone holds the SPU completion condition; and explicit completion milestone waits for an event published by other stages.

At the execution layer, Juric [22] presents extensions to BPEL that allow service invocations by events. In [37], Spiess et al. encapsulate event sources as services. Both approaches do not address event streams as input/output to/from components; rather than a stream of events, single events are understood as business relevant.

At the technical layer, event streams are well-known. CEP is supported by a variety of tools, e.g., the Esper CEP engine [38]. CEP is also part of BP execution environments like JBoss jBPM/Drools [39]. In [40], BP modeling techniques are used to express CEP queries. Event stream processing is integrated bottom-up; CEP queries and rules are specified at the technical layer. In contrast, we propose a top-down approach where business entity-centric event streams are visible as input/output of ESPTs at the modeling layer. Event streams can be as business relevant as, e.g., input/output data of services. Thus, like service task input/output is explicit in models, event streams are explicit at the modeling layer in our approach.

The event stream processing application logic inside Eventlets can be simple rules, CEP queries, or complex event processing networks as described in [41]. Our middleware instantiates Eventlets for each entity instance, e.g., one CEP query is issued per shipment. This encapsulation of event stream processing logic is related to *design by units* described in [42]. It improves scalability and fosters elasticity; in [14] we show the scalability benefits of CEP query encapsulation in Eventlets. The more process instances require entity-centric stream processing, the more Eventlets are instantiated and vice versa.

6. Conclusion

In collaboration with Software AG, we developed SPUs to provide an abstraction for event stream processing. The contributions of this paper are:

- SPUs as abstraction to encapsulate event stream processing as business functions;
- extensions of EPCs and BPMN 2.0 to model SPUs;
- a mapping of SPU-containing process models between EPCs and BPMN;
- a conversion of technical BPMN process models to an executable process representation;
- an extension of our Eventlet middleware to interface with BP execution engines; and
- an integration of our EPC and BPMN extensions in Software AG ARIS and in the Software AG model-to-execute workflow.

We take semantics of event processing into account and support implicit as well as explicit instantiation and completion strategies. Event stream processing techniques, like CEP, are widely adopted. Our approach encapsulates them and makes event stream processing available coherently across the BP modeling, BP execution, and the IT infrastructure layer.

SPUs in general and ESPSs and ESPTs in particular depict tools that enable modeling event stream processing in BPs. Developing a comprehensive understanding and description of the resulting execution semantics, however, is challenging. SPUs issue subscriptions; but successfully issued subscriptions do not guarantee that demanded events are published. Further, BP models can describe complex situations, which require, for example, transactional behavior. In such cases SPUs require compensation functionality to support rollbacks although event producers and consumers are logically decoupled.

SPUs enable the combination of push- and pull-based interactions within BPs. Thus, the interplay of push- and pull-based components in the context of large and complex processes is a topic of further investigation. Design and interaction patterns need to be derived to better understand the resulting interdependencies at the BP modeling, the BP execution, and the IT infrastructure layer. Especially in cross-organizational processes aspects like data integration and quality of service at the IT infrastructure layer require novel solutions. At the BP modeling layer, meta-model extensions that describe SPUs are necessary to provide a basis for interoperability.

In ongoing work we are enhancing the integration between the process execution layer and the IT infrastructure layer; a backchannel needs to be incorporated to fully support implicit instantiation and completion semantics. We are also enhancing our Eventlet middleware. We implement support for more types of completion conditions and investigate complex expressions as triggers for the instantiation of Eventlets. We are also working on automated transformations between EPC process models and an executable process representation to support a seamless M2E workflow. The implementation of our approach with Software AG's ARIS and webMethods platforms is also the basis for an evaluation of our approach in a real-world environment to investigate how the modeling of event stream processing by means of SPUs is adopted.

Acknowledgements

We thank Dr. Walter Waterfeld, Software AG, Germany, for the valuable feedback and insights into process modeling practice. This work is partly funded by: German Federal Ministry of Education and Research (BMBF) under research grants 01IS12054 (Software Campus), 01IC12S01V (SINNODIUM), and 01IC10S01 (EMERGENT); LOEWE Priority Program Dynamo PLV supported by the LOEWE research initiative of the state of Hesse/Germany. The authors assume responsibility for the content.

References

[1] S. Frischbier, M. Gesmann, D. Mayer, A. Roth, C. Webel, Emergence as competitive advantage - engineering tomorrow's enterprise software systems, in: 14th International Conference on Enterprise Information Systems (ICEIS), Poland, 2012.

[2] A. Buchmann, S. Appel, T. Freudenreich, S. Frischbier, P. E. Guerrero, From calls to events: Architecting future BPM systems, in: 10th International Conference on Business Process Managment (BPM), Estonia, 2012.

[3] W. van der Aalst, Formalization and verification of event-driven process chains, Information and Software Technology 41 (1999) 639 – 650.

[4] Object Management Group (OMG), Business process model and notation (BPMN), version 2.0, 2011.

[5] S. Appel, S. Frischbier, T. Freudenreich, A. Buchmann, Event stream processing units in business processes, in: 11th International Conference on Business Process Management (BPM), China, 2013.

[6] M. Papazoglou, Service-oriented computing: concepts, characteristics and directions, in: 4th International Conference on Web Information Systems Engineering (WISE), Italy, 2003.

[7] G. Keller, A.-W. Scheer, M. Nüttgens, Semantische Prozeßmodellierung auf der Grundlage "Ereignisgesteuerter Prozeßketten (EPK)", Inst. für Wirtschaftsinformatik, 1992.

[8] OASIS Web Services Business Process Execution Language (WSBPEL) TC, Web services business process execution language (BPEL), version 2.0, 2007.

[9] C. Ouyang, M. Dumas, W. van der Aalst, A. ter Hofstede, J. Mendling, From business process models to process-oriented software systems, ACM Transactions on Software Engineering and Methodology 19 (2009) 2:1–2:37.

[10] M. K. Chandy, O. Etzion, R. von Ammon, 10201 executive summary and manifesto – event processing, in: Event Processing, number 10201 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2011.

[11] A. Meyer, S. Smirnov, M. Weske, Data in business processes, EMISA Forum 31 (2011) 5–31.

[12] J. Becker, M. Rosemann, C. Uthmann, Guidelines of business process modeling, in: W. Aalst, J. Desel, A. Oberweis (Eds.), Business Process Management, volume 1806 of *Lecture Notes in Computer Science*, Springer, 2000, pp. 30–49.

[13] P. Eugster, P. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM Computing Surveys (CSUR) 35 (2003) 114–131.

[14] S. Appel, S. Frischbier, T. Freudenreich, A. Buchmann, Eventlets: Components for the integration of event streams with SOA, in: 5th IEEE International Conference on Service-Oriented Computing and Applications (SOCA), Taiwan, 2012.

[15] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, Service-oriented computing: State of the art and research challenges, IEEE Computer Journal 40 (2007) 38–45.

[16] J. Mendling, H. Reijers, W. van der Aalst, Seven process modeling guidelines (7pmg), Information and Software Technology 52 (2010) 127 – 136.

[17] J. Miller, J. Mukerji, MDA Guide, Version 1.0.1, 2003.

[18] A.-W. Scheer, M. Nüttgens, ARIS Architecture and Reference Models for Business Process Management, in: W. Aalst, J. Desel, A. Oberweis (Eds.), Business Process Management, volume 1806 of *Lecture Notes in Computer Science*, Springer, 2000, pp. 376–389.

[19] S. Stein, Modelling Method Extension for Service-Oriented Business Process Management, Ph.D. thesis, Christian-Albrechts-Universität zu Kiel, Kiel, Germany, 2009.

[20] V. Hoyer, E. Bucherer, F. Schnabel, Collaborative e-business process modelling: Transforming private EPC to public BPMN business process models, in: 5th International Conference on Business Process Management (BPM) Workshops, Australia, 2007.

[21] R. Khalaf, D. Karastoyanova, F. Leymann, Pluggable framework for enabling the execution of extended BPEL behavior, in: International Conference on Service Oriented Computing (ICSOC) Workshops, Austria, 2007.

[22] M. B. Juric, WSDL and BPEL extensions for event driven architecture, Information and Software Technology 52 (2010) 1023–1043.

[23] N. Herzberg, A. Meyer, O. Khovalko, M. Weske, Improving business process intelligence with object state transition events, in: 32nd International Conference on Conceptual Modeling (ER), China, 2013.

[24] N. Herzberg, A. Meyer, M. Weske, An event processing platform for business process management, in: 17th International Enterprise Distributed Object Computing Conference (EDOC), Canada, 2013.

[25] M. Weidlich, H. Ziekow, J. Mendling, O. Günther, M. Weske, N. Desai, Event-based monitoring of process execution violations, in: 9th International Conference on Business Process Management (BPM), France, 2011.

[26] A. Baouab, O. Perrin, C. Godart, An optimized derivation of event queries to monitor choreography violations, in: 10th International Conference Service-Oriented Computing (ICSOC), China, 2012.

[27] P. Chakravarty, M. Singh, Incorporating events into cross-organizational business processes, IEEE Internet Computing 12 (2008) 46 –53.

[28] B. M. Michelson, Event-driven architecture overview, Patricia Seybold Group (2006).

[29] O. Levina, V. Stantchev, Realizing event-driven SOA, in: 4th International Conference on Internet and Web Applications and Services (ICIW), Italy, 2009.

[30] A. Barros, G. Decker, A. Grosskopf, Complex events in business processes, in: 10th International Conference on Business Information Systems (BIS), Poland, 2007.

[31] B. Biörnstad, C. Pautasso, G. Alonso, Control the flow: How to safely compose streaming services into business processes, in: IEEE International Conference on Services Computing (SCC), USA, 2006.

[32] A. Caracaş, T. Kramp, On the expressiveness of BPMN for modeling wireless sensor networks applications, in: 3rd International Workshop on Business Process Model and Notation (BPMN), Switzerland, 2011.

[33] A. Estruch, J. Heredia Álvaro, Event-driven manufacturing process management approach, in: 10th International Conference on Business Process Managment (BPM), Estonia, 2012.

[34] M. Wieland, D. Martin, O. Kopp, F. Leymann, SOEDA: A method for specification and implementation of applications on a service-oriented event-driven architecture, in: 12th International Conference on Business Information Systems (BIS), Poland, 2009.

[35] A. Elfatatry, Dealing with change: components versus services, Communications of the ACM 50 (2007) 35–39.

[36] R. Hull, E. Damaggio, F. Fournier, M. Gupta, F. T. Heath III, S. Hobson, M. Linehan, S. Maradugu, A. Nigam, P. Sukaviriya, R. Vaculin, Introducing the guard-stage-milestone approach for specifying business entity lifecycles, in: 7th International Workshop on Web Services and Formal Methods (WS-FM), USA, 2010.

[37] P. Spiess, S. Karnouskos, D. Guinard, D. Savio, O. Baecker, L. Souza, V. Trifa, SOA-based integration of the internet of things in enterprise services, in: IEEE International Conference on Web Services (ICWS, USA, 2009.

[38] EsperTech Inc., Esper Complex Event Processing Engine, 2013.

[39] JBoss.com, Drools - The Business Logic integration Platform, 2013.

[40] S. Kunz, T. Fickinger, J. Prescher, K. Spengler, Managing complex event processes with business process modeling notation, in: 2nd International Workshop on Business Process Model and Notation (BPMN), Germany, 2010.

[41] O. Etzion, P. Niblett, Event processing in action, Manning Publications Co., 2010.

[42] S. Tai, P. Leitner, S. Dustdar, Design by units: Abstractions for human and compute resources for elastic systems, IEEE Internet Computing 16 (2012) 84 –88.