Event Stream Processing Units in Business Processes

Stefan Appel, Sebastian Frischbier, Tobias Freudenreich, Alejandro Buchmann

TU Darmstadt, Germany
lastname@dvs.tu-darmstadt.de

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 this 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 a BPMN extension to model SPUs. 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.

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 thresholds.

A common paradigm for the representation of information from sources like the Internet of Things or Cyber-physical Systems are 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 [4]. 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 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 BPs [25, 20], 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], a leader in business process management, we developed *Event Stream Processing Units* (SPUs) as such an integration concept.

In this paper we present SPUs. 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. SPUs encapsulate event stream processing logic at the abstraction level of business functions and hide implementation details. 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.

The paper is structured as follows: we introduce a logistics scenario as running example; we then derive requirements for the integration of event streams with BPs at the modeling, execution, and IT infrastructure layer. In Section 3, we introduce Event Stream Processing Tasks (ESPTs), a BPMN 2.0 extension to model SPUs. We present a mapping of ESPTs to BPEL as well as a runtime environment for SPUs. In Section 4, we discuss related work; we summarize our findings in Section 5.

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 [22]. The BP model describes interactions between these building blocks.

[1] www.softwareag.com

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, e.g., using the Business Process Modeling Notation (BPMN) [20]. The model is then transformed into an executable workflow expressed, e.g., with the Business Process Execution Language (BPEL) [19]. Typically, the workflow execution requires IT support, which is provided by a SOA and workflow management systems.

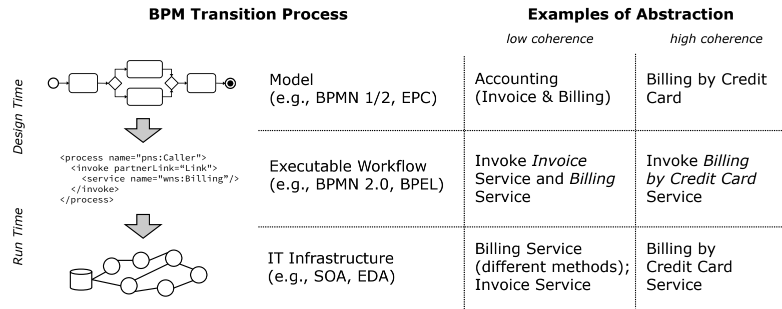

Fig. 1. Transition steps between process modeling, process execution, and IT infrastructure layer.

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 [21]. 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.

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 [22]. 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)*. While the actual event-processing functionality is encapsulated inside SPUs, event streams should be accessible by the modeler. Integrating event streams during modeling simplifies the transition to an executable workflow. Thus, the modeling notation has to provide *means to express event streams as input/output to/from SPUs (R_2)*. Finally, the model notion 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)*.

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 SPUs provided by the IT infrastructure (R_5)*. 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)*.

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 behave reactively 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)*. It must *provide containers for SPUs that represent business functions (R_9)*. Just like services, these *SPU containers must be manageable and capable of receiving the required data in form of event streams (R_{10})*.

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 Tasks* (ESPTs) to represent SPUs in BPMN process models. At the IT infrastructure layer, we adapt *Eventlets* [1] 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 form a SOA, SPUs form 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.

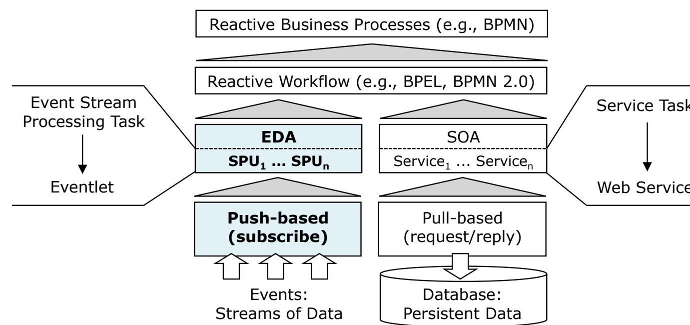

Fig. 2. Stream Processing Units (SPUs) as building blocks of an event-driven architecture (EDA)

3.1 Modeling Layer

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. We extend BPMN with building blocks that represent SPUs. The extension of BPMN is necessary to address the characteristics of SPUs determined by the streaming nature of event data. SPUs exhibit the following specific properties that cannot be expressed completely with existing 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.

Our extensions to BPMN are shown in Figure 3. We introduce *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.

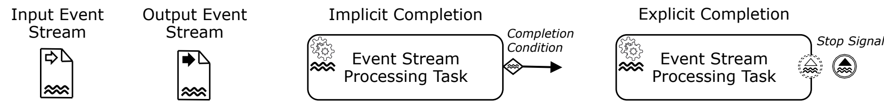

Fig. 3. Extensions to BPMN: Event Stream Specifications (ESSs) and Event Stream Processing Tasks (ESPTs)

Definition 1. *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 2. *An* ***Event Stream Processing Task (ESPT)*** *($\rightarrow$ R_1, R_3, R_4) 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 [20, p. 428]) is triggered implicitly or explicitly ($\rightarrow$ R_5). 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. To trigger concurrent actions, ESPTs can activate outgoing sequence flow elements while remaining in the active state.*

Related BPMN Concepts Events are part of the BPMN specification. However, events in BPMN are meant to affect the control flow in a process [20, 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. Thus, due to the different semantics, events in the sense of the BPMN standard are not appropriate to model SPUs.

To avoid unnecessary extensions of BPMN, we evaluated different BPMN task types as alternatives to ESPTs. From the task types contained in the BPMN 2.0 standard, service tasks, business rule tasks, loop service tasks, and multiple instance service tasks share properties with SPUs.

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.

Example: Shipment Monitoring To illustrate the application of our BPMN extensions, we model the monitoring of environmental conditions in the order process introduced in Section 1. Figures 4 and 5 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 4, 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.

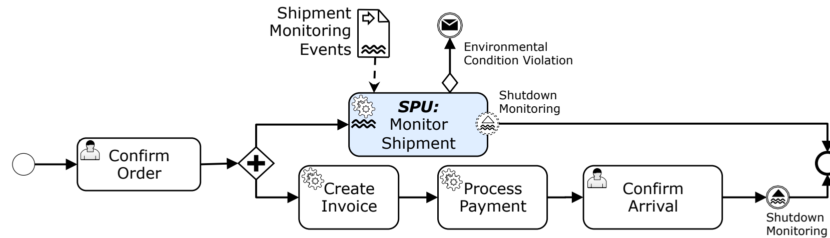

Fig. 4. Shipment monitoring SPU that is stopped explicitly. The data input/output of the service tasks omitted.

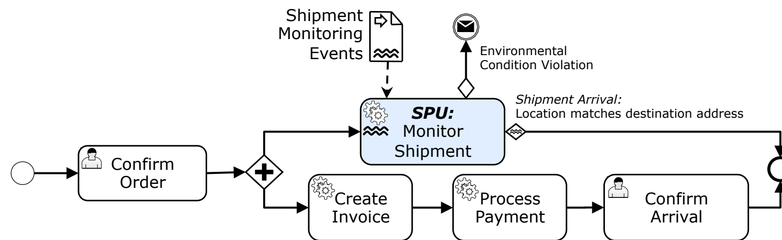

Fig. 5. Shipment monitoring SPU that is stopped implicitly. The data input/output of the service tasks omitted.

In Figure 5, 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.

3.2 Workflow Execution Layer

The support of BPs by an IT infrastructure requires a transition from the graphical process notation to an executable format. With this technical representation, tasks of a process model can be executed by technical components of the IT infrastructure. The BPMN 2.0 standard itself specifies such a technical representation of the graphical model. 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 graphical notation and interfaces of IT components so that the process can be executed automatically. The transition from a graphical model towards a technical representation requires adding additional technical information necessary for the execution.

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 is shown in Figure 6: 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.

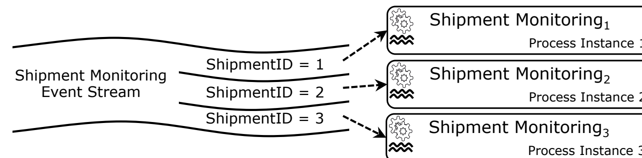

Fig. 6. Process execution: Event Stream Processing Tasks (ESPTs) receive sub streams of events

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.

ESPT Instantiation and Completion The execution of a BPs 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 6). 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, timeout(120sec))
```

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 ESPT instances 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.

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)
```

Although ESPTs have different execution semantics than BPMN service tasks, 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.

ESPT Mapping in BPEL Business process models that contain ESPTs can be mapped to BPEL. However, the BPEL standard [19] 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 a `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 [14] and [12]. 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 [1] 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 [1] 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 [1] to fit the terminology of this paper.

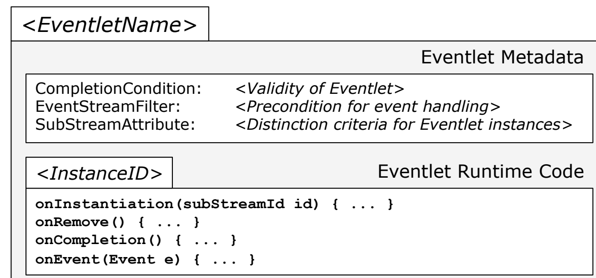

Fig. 7. Eventlet structure: Eventlet metadata and Eventlet runtime methods

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 7. 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 [1]. 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`.

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 [1] to support ESPT execution. As shown in Figure 8, 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 mid-

[2] Referred to as Instantiation Expression in [1].
[3] Referred to as Validity Expression in [1].
[4] Referred to as Constant Expression in [1].

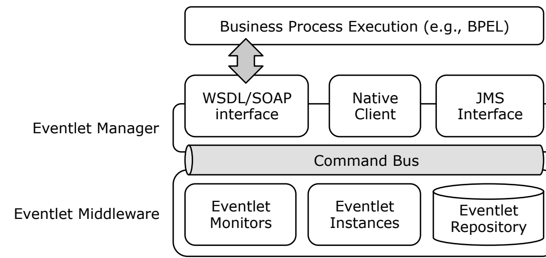

Fig. 8. Eventlet middleware access via web service

dleware 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.

4 Related Work

Events are part of various BP modeling notations like BPMN 2.0 and event-driven process chains (EPCs) [13, 25]; they trigger functions/tasks and influence the process control flow. The incorporation of (complex) events leads to more reactive and dynamic processes. This is a core concept in event-driven architectures (EDA) [6, 18] or event-driven SOA [16]. 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 [2, 3, 5, 9, 26]. 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 [17]. 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 [22]. 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 [7] to event stream processing and introduces SPUs as an appropriate abstraction.

At the execution layer, Juric [12] presents extensions to BPEL that allow service invocations by events. In [23], 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 [8]. CEP is also part of BP execution environments like JBoss jBPM/Drools [11]. In [15], 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 [10]. While event stream processing queries can run centralized, e.g., a single CEP query processes monitoring data of all shipments, our middleware instantiates Eventlets for each entity instance, e.g., one CEP query per shipment. This encapsulation of event stream processing logic is related to *design by units* described in [24]. It improves scalability and fosters elasticity; in [1] 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.

5 Conclusion

In collaboration with Software AG, we identified the need to integrate event streams with BP modeling and execution. Rather than single events, event streams are considered as business-relevant units in this context. We developed an approach for this integration at the modeling, execution, and IT infrastructure layer. Our approach introduces SPUs: a consistent abstraction for event stream processing across the layers. Like services, SPUs encapsulate business functions; they use event streams as business-relevant data sources. This allows intuitive modeling from the business perspective. The abstraction paradigm of SPUs leads to a high coherence across the layers. This minimizes the transition effort from the graphical model notation to the executable process description and from the executable process description to the IT infrastructure. Our approach is a clear separation of concerns; SPUs are declarative, the (imperative) application logic resides solely at the technical layer inside Eventlets.

The contributions of this paper are: 1) SPUs as abstraction to encapsulate event stream processing as business functions, 2) an extension of BPMN 2.0 with ESPTs and ESSs to model SPUs, 3) a mapping of ESPTs and ESSs to an executable process description, and 4) an extension of our Eventlet middleware to interface with BP execution engines. 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 IT infrastructure layer. We illustrate our approach with the running example of a shipment monitoring SPU inside an order-to-delivery process.

In ongoing work we are 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 extensions to the event-driven process chain (EPC) notation to support SPUs in EPCs.

Acknowledgements We thank Dr. Walter Waterfeld, Software AG, Germany, for the valuable feedback and insights into process modeling practice. Funding by German Federal Ministry of Education and Research (BMBF) under research grants 01IS12054, 01IC12S01V, and 01IC10S01. The authors assume responsibility for the content.

References

1. S. Appel, S. Frischbier, T. Freudenreich, and A. Buchmann. Eventlets: Components for the integration of event streams with SOA. In *SOCA*, Taiwan, 2012.
2. A. Barros, G. Decker, and A. Grosskopf. Complex events in business processes. In *BIS*, Poland, 2007.
3. B. Biörnstad, C. Pautasso, and G. Alonso. Control the flow: How to safely compose streaming services into business processes. In *SCC*, USA, 2006.
4. A. Buchmann, S. Appel, T. Freudenreich, S. Frischbier, and P. E. Guerrero. From calls to events: Architecting future bpm systems. In *BPM*, Estonia, 2012.
5. A. Caracaş and T. Kramp. On the expressiveness of BPMN for modeling wireless sensor networks applications. In *BPMN*, Switzerland, 2011.
6. P. Chakravarty and M. Singh. Incorporating events into cross-organizational business processes. *IEEE Internet Computing*, 12(2):46 –53, 2008.
7. A. Elfatatry. Dealing with change: components versus services. *Communications of the ACM*, 50(8):35–39, 2007.
8. EsperTech Inc. Esper Complex Event Processing Engine, 2013.
9. A. Estruch and J. Heredia Álvaro. Event-driven manufacturing process management approach. In *BPM*, Estonia, 2012.
10. O. Etzion and P. Niblett. *Event processing in action*. Manning Publications Co., 2010.
11. JBoss.com. Drools - The Business Logic integration Platform, 2013.
12. M. B. Juric. WSDL and BPEL extensions for event driven architecture. *Information and Software Technology*, 52(10):1023 – 1043, 2010.
13. G. Keller, A.-W. Scheer, and M. Nüttgens. *Semantische Prozeßmodellierung auf der Grundlage "Ereignisgesteuerter Prozeßketten (EPK)"*. Inst. für Wirtschaftsinformatik, 1992.
14. R. Khalaf, D. Karastoyanova, and F. Leymann. Pluggable framework for enabling the execution of extended BPEL behavior. In *ICSOC/WESOA*, Austria, 2007.
15. S. Kunz, T. Fickinger, J. Prescher, and K. Spengler. Managing complex event processes with business process modeling notation. In *BPMN*, Germany, 2010.
16. O. Levina and V. Stantchev. Realizing event-driven SOA. In *ICIW*. Italy, 2009.
17. J. Mendling, H. Reijers, and W. van der Aalst. Seven process modeling guidelines (7pmg). *Information and Software Technology*, 52(2):127 – 136, 2010.
18. B. M. Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2006.
19. OASIS Web Services Business Process Execution Language (WSBPEL) TC. Web services business process execution language (BPEL), version 2.0, April 2007.
20. Object Management Group (OMG). Business process model and notation (BPMN), version 2.0, January 2011.
21. C. Ouyang, M. Dumas, W. van der Aalst, A. ter Hofstede, and J. Mendling. From business process models to process-oriented software systems. *ACM Transactions on Software Engineering and Methodology*, 19(1):2:1–2:37, Aug. 2009.
22. M. Papazoglou. Service-oriented computing: concepts, characteristics and directions. In *WISE*, Italy, 2003.
23. P. Spiess, S. Karnouskos, D. Guinard, D. Savio, O. Baecker, L. Souza, and V. Trifa. SOA-based integration of the internet of things in enterprise services. In *ICWS*, USA, 2009.
24. S. Tai, P. Leitner, and S. Dustdar. Design by units: Abstractions for human and compute resources for elastic systems. *IEEE Internet Computing*, 16(4):84 –88, 2012.
25. W. van der Aalst. Formalization and verification of event-driven process chains. *Information and Software Technology*, 41(10):639 – 650, 1999.
26. M. Wieland, D. Martin, O. Kopp, and F. Leymann. SOEDA: A method for specification and implementation of applications on a service-oriented event-driven architecture. In *BIS*, Poland, 2009.