Jens Kolb et al. (Eds.): Enterprise Modelling and Information Systems Architectures,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2015 77

An Enhanced Communication Concept for Business Processes[1]

Felix Kossak[2], Verena Geist[2]

Abstract: Simple communication patterns often do not suffice for modelling the interplay between different business processes. In this paper, we introduce and formally specify an event-based communication concept for business process modelling, constituted by event trigger properties and event pools. We claim that this concept provides a much bigger scope for modelling communication than currently available concepts, particularly when actor and user interaction modelling are included.

1 Introduction

As long as business processes are modelled individually, simple communication patterns typically suffice for communication with an abstractly modelled environment and within the process. However, we often need to model the interplay between different processes as well, often between very heterogeneous systems. Due to growing demand to integrate different processes of different organisations – e.g. in the context of the European "Industry 4.0" initiative –, simple patterns like "Messages" or "Signals" do not always suffice.

Simple communication patterns, such as those provided by the BPMN 2.0 standard [Ob11], have the advantage of being relatively simple and easy to depict in diagrams with relatively intuitive symbols. But integration of automated processes, human actors (see e.g. [NG13, NC12]), and user interaction (see e.g. [KG12, ADG10]) demands more flexibility and customisation. In this paper, we propose a very general event concept for business process modelling, based on a set of *event properties* as well as *event pools*.

In general, a communication concept is supposed to serve different purposes in the context of business process modelling:

- The environment demands a new process instance to be started.
- A process instance notifies its environment that it has finished, that it needs to abort, and/or that an error has occurred.
- The default workflow may be left as in case of an error or of special circumstances, where e.g. compensation is required.

[1] The research reported in this paper supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

[2] Software Competence Center Hagenberg GmbH (SCCH), Softwarepark 21, 4232 Hagenberg im Mühlkreis, Austria, {felix.kossak,verena.geist}@scch.at

- Different processes need to synchronise, or a process with its environment.
- Data need to be exchanged.

We claim that the proposed communication concept provides a much bigger scope for serving all these purposes than currently available concepts, which we review in Section 5. This review shows that what is obviously missing is a *general* concept for covering different ways of communication in BPM which is *concrete* enough to be integrated in tools in a straightforward way. This is what our paper aims to contribute.

The notion of an "event" has been used ambiguously in the literature, including the BPMN 2.0 standard, where it typically (though not always) denotes a flow node rather than something which happens at a certain point in time. Therefore we need to introduce a clear notation. We try to stay close to the well-established BPMN 2.0 standard, whose basic knowledge we assume. To summarise our notation:

> An *event node* either throws or catches *triggers* at certain points in time. Every trigger is of one particular *trigger type* (e.g. "Message" or "Error"), and an event node can throw or catch triggers of one or several different trigger types, which are defined in its *event definitions*.

In the following, we will use Abstract State Machine (ASM) notation [BS03] to formalise our concepts. We expect that the notation provides an intuitive understanding without familiarity with the formal semantics of ASMs. The reader may look at it as pseudocode, with special attention to understandability. Yet this notation makes the semantics of the concepts unambiguous and the complete ASM-based specification, of which we show the important parts, can be easily and provably correct refined towards software code.

The specification is embedded in a comprehensive business process modelling concept, the *Hagenberg Business Process Modelling Method* (*H-BPM*), whose core is largely based on BPMN 2.0. A formal specification of the core model has been published in [Ko14], which also demonstrates the requirements for the communication concept. The whole method, including actor, user interaction, and data modelling is presented in [Sc15].

This paper is structured as follows. In Section 2, we introduce a set of event trigger properties to generalise trigger types, opening a wide scope of customised communication. In Section 3, an event pool concept is presented for flexible distribution of triggers, enhancing also flexibility and comfort for human participants. We evaluate the combination of both concepts in Section 4, where we also look at standard trigger types again. Section 5 reviews related work and Section 6 gives a summary.

2 Trigger Properties

Trigger types, such as those provided by BPMN, can be distinguished by different purposes, as their usual names (such as "Message") suggest. However, more generally, most

of them can also be distinguished by different properties, mostly properties concerning the distribution of triggers, answering questions like the following (cf. [Al10]):

- Is the trigger intended for a particular process or (potentially) for different processes (like a fire alert for employees of different companies in a single office building)?
- Is the trigger intended for a particular process *instance*, e.g. for a particular business case, or may it be caught by any instance of a given process (like a call to a help desk may be answered by *any* employee concerned)?
- If there is no particular recipient (no particular process instance) addressed, is it sufficient that *just one* process and process instance deals with the trigger or should more or even all of the potential recipients react (as in the case of a fire alert)?
- Is it obligatory that someone deals with the trigger (as in the case of a help desk call), or is it optional (as in "There is a special offer today in the canteen")?
- Can the trigger only be caught instantly, or is it valid for some time (or indefinitely)?

When we define respective properties for triggers, we can use them to identify e.g. a signal as a trigger which should be broadcast to all processes (of a given set) and to all process instances and which should be sustained even when some actor has already reacted to it.

However, all the possible combinations of possible values for different properties (reduced by possible constraints) offer more scope than a small set of standard trigger types. We now introduce a set of key properties for triggers and necessary constraints on them.

We need to consider a system in which several communicating processes are running concurrently. Thus we need to identify, for any trigger, one or more recipient processes.

```
shared recipientProcesses : triggers → Set
```

recipientProcesses is a function from the set of triggers to the power set of *processes*. It is a **shared** function, which means that both the process considered and its environment can set the value of this function for a particular trigger. The given process needs to set the value for triggers which it throws (sends), while the environment (including other processes) need to set the value for triggers which the process in question shall receive.

We stipulate that if no recipient process is identified, all processes running on or visible to the workflow engine shall receive the trigger, except if a particular public event pool is specified (see below).

Next, we may want to identify a particular event node of the target process. For instance, there may be alternative start nodes and the environment wants to determine where exactly the new process instance shall start. If no *recipientNode* is specified, any suitable event node within the target process may catch the trigger (if no further constraints apply).

```
shared recipientNode : triggers → flowNodes
```

There are dependencies between *recipientProcesses* and *recipientNode*. For a start, if no particular recipient process is identified, or more than one process is identified, then we cannot specify a particular event node:

```
assert
  forall trigger ∈ triggers holds
    if recipientProcesses(trigger) = undef or
        | recipientProcesses(trigger) | ≠ 1 then
      recipientNode(trigger) = undef
```

Above, the keyword **undef** denotes "undefined" and the vertical bars around "recipientProcesses(trigger)" denote the cardinality of this set of processes.

If a *recipientNode* is defined, then the only member of *recipientProcesses* must be the parent of the *recipientNode*. Thereby we also make sure that only event nodes that are direct children of a given process can be addressed (for propagation into sub-processes, see further below).

```
assert
  forall trigger ∈ triggers holds
    if recipientNode(trigger) ≠ undef then
      forall process in recipientProcesses(trigger) holds
        process = parentNode(recipientNode(trigger))
```

Note that in combination with the previous assertion, we can derive that when *recipientNode* is defined, then *recipientProcesses* must be defined as well and the cardinality of *recipientProcesses* must be 1; thus **forall**, above, actually ranges over a single process.

If a recipient process is specified but no particular recipient node, then we shall be able to specify whether the trigger may be *propagated* into sub-processes (recursively). This corresponds to the distinction between the two concepts of *direct resolution* and *propagation* in BPMN ([Ob11, p. 234f]).

```
shared mayBePropagated : triggers → Boolean
```

If *recipientNode* is specified, then propagation is obviously not desired:

```
assert
  forall trigger ∈ triggers holds
    if recipientNode(trigger) ≠ undef then
      mayBePropagated(trigger) = false
```

We may also want to address a particular process *instance*. E.g. when a customer has placed an order and subsequently asks when they can expect delivery, then this request must be linked with the proper process instance associated with the respective order number. An order number is an example of *correlation information*. In general, this can be any piece of information through which a particular process instance can be identified. To make correlation possible, the same *correlationInfo* must be shared by the respective properties of both process instance and trigger. (The term "correlation information" is also used in BPMN. Also compare with the *correlation sets* of WS-BPEL [OA07, Sect. 9].)

As we do not want to restrict the form of correlation information, we define an own universe (data type), "*correlationInfo*", whose implementation is left open. We re-use the name for the respective properties of both triggers and process instances. Note that *correlationInfo* of instances is **controlled**, which means that this property can only be set within the process in question, i.e. by the process engine.

```
shared correlationInfo : triggers → correlationInfo

controlled correlationInfo : instances → correlationInfo
```

In the next section, we will introduce event pools (represented by the universe *eventPools*), some of which may be directly addressed by a trigger.

```
shared recipientPool : triggers → eventPools
```

Another important trigger property shall indicate whether it suffices that one actor reacts to it or not. In other words, shall the trigger be deleted once it has been caught by some event node or shall it be sustained so others can catch it as well?

```
shared deleteUponCatch : triggers → Boolean
```

Next, we want to specify whether a trigger is supposed to be caught instantaneously or if it shall be sustained for some time, and if so, for how long. There are at least three possible ways to define a *timeout*:

- in terms of an absolute point in time ("until 1 Feb 2015, 15:00");
- in terms of a time span from the creation of the trigger; or
- in terms of a particular hour, day of the week, week, etc. after the creation of the trigger ("until the following Friday, 14:00").

More exotic variants are imaginable, but we think that *at least* those should be supported, requiring the following properties:

- The first variant requires a simple time property, *timeout*.
- The second variant requires a duration, *lifetime*, in combination with a *timestamp* of the time of creation of the trigger.
- The third variant also requires a *timestamp*, along with a "semi-relative" time property, allowing for values like "the 5th of the following month", "November of the same year", etc., for which we use an abstract universe, *RelativeTime*; we call the respective trigger property *relativeTimeout*.

```
shared timestamp : triggers → Time

shared timeout : triggers → Time
```

```
shared lifetime : triggers → Time

shared relativeTimeout : triggers → RelativeTime
```

lifetime and *relativeTimeout* require a *timestamp*.

```
assert
  forall trigger ∈ triggers holds
    if lifetime(trigger) ≠ undef or
        relativeTimeout(trigger) ≠ undef then
      timestamp(trigger) ≠ undef
```

Furthermore, at most one of the functions *timeout*, *lifetime*, and *relativeTimeout* may be defined for a particular trigger.

If neither *timeout* nor *lifetime* nor *relativeTimeout* are defined, then either the process engine has defined a default lifetime which will come into effect or the trigger does not expire as long as any potential recipient process is running.

Finally, in many cases, the process that sent a given trigger is of interest. For instance, we would like to know which process sent an "Error" or "Escalation" trigger. Even the throwing event node may be of interest, and as the process can be derived from that, we define the *senderNode* as a trigger property. (Note that the sender instance can be derived from the sender process in combination with *correlationInfo*.)

```
shared senderNode : triggers → flowNodes
```

Additionally, we retain the property *triggerType* (as in BPMN, with values like "Message", "Signal", "Error", etc.) for the following reasons:

- The BPMN trigger types "Signal", "Error", and "Escalation" cannot be distinguished by the other properties, yet "Error" and "Escalation" have algorithmic significance for the workflow.
- The relatively small number of trigger types defined by BPMN, reflecting the most common communication needs, can be represented by symbols which are relatively easy to identify and to remember and render a diagram much easier to understand.
- We want to remain compatible with the BPMN standard as far as possible.

```
shared triggerType : triggers → eventTriggerTypes
```

However, there is a certain redundancy of information shared between the *triggerType* and other properties, and we must assure consistency. We will discuss the respective relations further below.

For the following considerations, we further stipulate that triggers must be uniquely identifiable and that duplication always leads to *different* triggers.

3 Event Pools

If we want to enable users to choose in which order to process messages (and possibly other event triggers), we have to give them a kind of "pool" into which event triggers are delivered and from which users can pick. This concept is already well established in the form of the "inbox" of an email client. The pool concept we are going to introduce is also influenced by that proposed for S-BPM (see [Fl12]); S-BPM lays a special focus on the viewpoint of actors (or "subjects").

We not only want users to be able to choose the order in which to process triggers but also to be able to opt-in for additional, non-obligatory trigger sources, like certain kinds of news (like RSS feeds). This can be enabled by giving users access to certain additional event trigger pools, i.e. pools not directly associated with a particular process.

Furthermore, there are certain kinds of event triggers, like signal, error, or compensation triggers, which may be supposed to be caught by more than one process or sub-process. One way to handle this is to duplicate such events for every potential recipient. Another possibility, at least for the conceptual level, is to deposit such a trigger in a pool which is not associated with a particular process but which is "public".

So a process might have access to different event pools, some private, some public. However, a user might want to have a single view on all the relevant pools. To this end, we can define a view on all the triggers from all the pools relevant for a particular process by means of a virtual pool which we call the process's *inbox*. For the abstraction of the throwing of triggers, we further define an *outbox* for each process.

In summary, the event pool concept we are proposing comprises the following pool types:

- a *private* event pool for each process or sub-process for triggers which are only visible for event nodes that are directly within this (sub-)process (this corresponds to "direct resolution" in BPMN);
- a *group* event pool for each (sub-)process for triggers which are visible also within sub-processes of this (sub-)process, recursively, to enable propagation;
- *public* event pools to which processes can subscribe or to which several processes can be mandatorily subscribed (by the process designer);
- a virtual *inbox* for each (sub-)process to provide a single view on all relevant pools; and
- an abstract *outbox* for each (sub-)process to hide the details of delivering triggers thrown within this (sub-)process in accordance with the triggers' properties.

Within private and group event pools, triggers for a particular process instance can be identified by correlation information.

We now introduce event pools in more detail. We assume a universe (data type) *eventPools*, on which the rules (algorithmic functions) *AddTrigger* and *RemoveTrigger* as well as a derived function (derived property) *containsTrigger* are defined.

An event pool may or may not be associated with a particular (sub-)process, i.e. an *ownerProcess*. A public event pool is associated with the *environment* instead. We also assume that the *environment* has a pool for receiving triggers.

For the sake of simplicity, we assume that there is a fixed number of event pools with fixed associations in a given run of a process engine. Consequently, we model the function *ownerProcess* as *static* (i.e. it cannot change during runtime).

```
static ownerProcess : eventPools → processes ∪ { environment }
```

A derived function can identify all event pools owned by a particular (sub-)process or by the environment:

```
derived eventPools : processes ∪ { environment } → Set
eventPools(process) =
  { pool | pool ∈ eventPools and ownerProcess(pool) = process }
```

An event pool associated with a particular (sub-)process may be *private*; else, it is a group event pool. If an event pool associated with the environment is *private*, it is supposed to receive triggers addressed to the environment. If an event pool associated with the environment is *not private*, it is a public event pool.

```
static private : eventPools → Boolean
```

We can then define:

- a *private event pool* as a pool with *ownerProcess*(*pool*) ∈ processes and *private*(*pool*) = **true**;
- a *group event pool* as a pool with *ownerProcess*(*pool*) ∈ processes and *private*(*pool*) = **false**;
- a *public event pool* as a pool with *ownerProcess*(*pool*) = environment and *private*(*pool*) = **false**; and
- the environment's event pool (for triggers addressed to the environment) as a pool with *ownerProcess*(*pool*) = environment and *private*(*pool*) = **true**.

We define a *default public event pool* which is visible for all processes and to which e.g. signals can be distributed if their destination is not further specified:

```
static defaultPublicEventPool : → eventPools

assert
  ownerProcess(defaultPublicEventPool) = environment and
      private(defaultPublicEventPool) = false
```

Additional public event pools may be defined by the business process designer.

We may want event pools to have further properties such as access rights, but we do not consider more properties in this place.

We assert that every process has exactly one private event pool and one group event pool. The environment has one unique private event pool.

We can now identify the unique event pools of a given process by derived functions:

```
derived privateEventPool : processes → eventPools
privateEventPool(process) =
  choose pool in eventPools(process) with private(pool) = true do
    return pool

derived groupEventPool : processes → eventPools
  choose pool in eventPools(process) with private(pool) = false do
    return pool
```

The *visiblePublicEventPools* of a process are all the public event pools to which the process in question has, or has been, subscribed:

```
monitored visiblePublicEventPools : processes → Set
```

The *defaultPublicEventPool* must be visible for all processes:

```
assert
  forall process ∈ processes holds
    defaultPublicEventPool ∈ visiblePublicEventPools(process)
```

The *visibleEventPools* of a process are then the *visiblePublicEventPools* plus the private and group event pools.

```
derived visibleEventPools : processes → Set
visibleEventPools(process) =
    eventPools(process) ∪ visiblePublicEventPools(process)
```

We can now define the *inbox* of a process as a view showing all triggers available in any of the *visibleEventPools*.

```
derived inbox : processes → Set
inbox(process) =
  { trigger | forsome pool ∈ visibleEventPools(process) holds
      containsTrigger(pool, trigger) }
```

From a process's viewpoint, for throwing a trigger it shall suffice to put it into an *outbox*.

```
shared outbox : processes → eventPools
```

We assume that some delivery service will pick triggers up from the *outbox* and distribute them according to their properties. For any *trigger*:

- If *recipientProcess*(*trigger*) is **undef** or empty, then the *trigger* shall be delivered to a public event pool; if additionally *recipientPool*(*trigger*) = **undef**, then the *trigger* shall be delivered to the *defaultPublicEventPool*.
- If there is some *process* in *recipientProcesses*(*trigger*) and *mayBePropagated*(*trigger*) = **true**, then the *trigger* shall be delivered to the group event pool of each specified process.
- If there is some *process* in *recipientProcesses*(*trigger*) and *mayBePropagated*(*trigger*) = **false**, then the *trigger* shall be delivered to the private event pool of each specified process.
- If *environment* ∈ *recipientProcesses*(*trigger*), then the *trigger* shall (also) be delivered to the environment's (private) event pool.

When a particular process instance has reacted to a trigger in a public event pool, we set a controlled function *hasBeenCaughtByInstance* to true so that the instance will not react twice. The function value is **false** by default and set to **true** once the instance in question has reacted. Note that the process in question can always be identified via the instance.

```
controlled hasBeenCaughtByInstance : triggers × instances → Boolean
```

This concludes an outline of the major features of the proposed enhanced communication concept for business processes.

4 The Scope of Possible Communication and Standard Trigger Types

We now evaluate the scope of communication which the proposed concept enables. We start with a comparison with the BPMN standard, which describes "different strategies to forward the *trigger* to catching **Events**: publication, direct resolution, propagation, *cancellations*, and *compensations*" [Ob11, p. 234]. Cancellation and compensation do not actually constitute different ways of delivering triggers, but the actual delivery strategies can be handled by our proposal:

- **Publication** within a process can be achieved by specifying a recipient process of the trigger and leaving the recipient node undefined; publication across processes can be achieved by specifying a public event pool as the recipient pool.
- **Direct resolution** can be achieved by specifying a recipient node.
- **Propagation** can be achieved by setting *mayBePropagated* to **true**.

Aldred defines "process integration patterns" [Al10], many of which are relevant for our concept. Aldred distinguishes the following "dimensions" of communication:

- **Participants**: 1–1, 1–many, or many–many; the first two can be covered by setting *deleteUponCatch* to **true** for 1–1 and **false** for 1–many, and also by choosing a

suitable event pool, e.g. a public event pool for 1–many. The case of many–many can be handled by allowing different senders to send triggers to a particular public event pool (with *deleteUponCatch* set to **false**).

- **Uni-directional / bi-directional**: this is a matter of process design (though a public event pool could aid in bi-directional communication).
- **Synchronous / asynchronous**: this can be supported via the *timeout / lifetime* properties of triggers.
- **Thread-coupling**: this is a matter of process design.
- **Time** (whether two participants need to both be participating in an interaction at the exact same moment): this can be supported via the *lifetime* property, which is set to zero (or a minimum) for immediate communication.
- **Direct / indirect contact**: indirect contact between communication partners that need not know each other can be supported by public event pools.
- **Duplication**: in our concept, duplication *can* (but need not) be replaced by setting the trigger property *deleteUponCatch* to **false** and possibly using a public event pool.

Patterns of process instantiation, however, as e.g. discussed in [DM09], are a matter of process design and not of trigger design.

So it turns out that our concept covers a wide range of communication patterns.

"Standard" event trigger types as defined by the BPMN standard can be matched to particular settings of trigger properties as proposed here:

- A **Message** trigger has a single recipient process, *deleteUponCatch* is **true**, and there is no timeout.
- A **Signal** trigger has *deleteUponCatch* set to **false** and *mayBePropagated* is **true**.
- An **Error** trigger is in effect a special-purpose **Signal** trigger. The same holds for an **Escalation** trigger.
- A **Cancel** trigger has defined *recipientProcesses*, *mayBePropagated* is **true**, *deleteUponCatch* is **false**, and timeout is minimal.
- A **Compensation** trigger and a **Terminate** trigger have the same properties as a **Cancel** trigger (except the *triggerType*).

(Note that we do not consider **Link** triggers as they do not actually serve communication.)

5 Related Work

We have already commented on BPMN [Ob11] and on the "process integration patterns" of Aldred [Al10] in the context of YAWL in the previous section. Some of the "dimen-

sions" of communication discussed by Aldred concern process design rather than pure communication mechanisms. Moreover, Aldred's patterns are not translated into formal mechanisms which can be straightforward integrated in tools.

More generally, the Workflow Patterns of van der Aalst, ter Hofstede, et al. [vdAtH] (on which YAWL is based) address various perspectives relevant for event handling. Regarding the control-flow patterns, events are in particular involved in *implicit termination*, *deferred choice*, and in several *cancellation* patterns. There are also two patterns that explicitly describe the notion of *triggers*. However, support for external data interaction patterns and for triggering work execution (see *auto-start* patterns) is limited.

By adopting the concept of YAWL, Mendling et al. [MNN05] define an extension to EPC to enhance support for workflow patterns. They introduce e.g. cancellation areas to support *cancellation* patterns. However, the focus of EPC is on semi-formal process documentation rather than formal process specification.

The event pools of S-BPM [Fl12] provided inspiration for the pool concept introduced here. The pools of S-BPM are tailored for actor comfort, but are not embedded in a wider delivery concept. S-BPM provides some extra "configuration parameters" for input pools, whose most important application appears to be the enforcement of synchronous communication, which is handled differently in our more general concept.

WS-BPEL [OA07] supports correlation, propagation, and definition of timeouts (by *message* and *alarm* events); however, it shows deficiencies regarding the generality of specifying event handlers and event consumption.

Lucchi and Mazzara [LM07] propose a framework for generic event and error handling in business processes by reducing the amount of different mechanisms for exception, event, and compensation handling in WS-BPEL to a single mechanism based on the idea of event notification. The resulting specification helps simplify BPEL models and implementations in the area of Web services orchestration similar to our improvements for BPM.

Common event-driven patterns are presented by Etzion and Niblett in [EN11]. The authors regard BPM as a related technology to event processing and reflect current trends, e.g. event-driven architecture and asynchronous BPM, and future directions. They propose basic and dimensional patterns including common temporal patterns as we do. There are also certain parallels concerning pattern policies, e.g. consumption or cardinality policies.

A set of service interaction patterns is proposed by Weske in [We12]. The patterns primarily apply to the service composition layer; however, an issue common to our proposed concept is the classification according to the number of involved participants.

Herzberg et al. [HMW13] introduce so-called *process events* that enrich events that occur during process execution with context data to create events correlated to the proper process descriptions. They address correlation as a main issue of their event processing platform. However, they concentrate on business process monitoring and analysis rather than modelling.

The WED-flow approach of Ferreira et al. [Fe10] proposes data states to integrate event processing into workflow management systems. Data states store required information for event-handling, thereby increasing backward and forward recovery options. In contrast to our work, the WED-flow approach does not define control flow but triggers over attribute values (wed-states), yielding the flow as a consequence of satisfied conditions.

The Complex Event Processing (CEP) discipline [Lu02], an emerging technology dealing with event-driven behaviour, and its combination with BPM is a main topic of interest [BDG07] and used e.g. for *Event-Driven Business Process Management* [Am09] to detect and react to possible errors within processes and also to support dynamic business process adaptation [HSD10] or business process exception management [Li14].

6 Summary

We introduced a communication concept for advanced business process modelling which enables modelling of a wide range of different communication styles. We showed how different communication patterns can be modelled by a combination of a set of event trigger properties and a few different types of event pools. Event pools make it also possible to model flexibility for human actors, such as the ability to process messages in a custom order or to subscribe to optional communication sources (such as news).

We have compared our communication concept in particular with BPMN as well as with the patterns introduced by Aldred [Al10]. We think it is obvious that our concept is much more general as that of BPMN-style modelling languages and is able to meet all relevant communication needs identified by Aldred.

The communication concept we have proposed is part of an overall BPM method developed at the Software Competence Center Hagenberg, Austria, which we call the *Hagenberg Business Process Modelling Method (H-BPM)*; it is outlined in [Sc15].

Acknowledgement: This publication was supported by the *AdaBPM* project, which is funded by the FFG under the project number 842437.

References

[ADG10] Atkinson, C.; Draheim, D.; Geist, V.: Typed business process specification. In: EDOC'10. IEEE Computer Society, pp. 69–78, 2010.

[Al10] Aldred, L.: Process integration. In (ter Hofstede, A. M.; van der Aalst, W. M. P.; Adams, M.; Russell, N., eds): Modern Business Process Automation: YAWL and its Support Environment, pp. 489–511. Springer, Heidelberg, 2010.

[Am09] von Ammon, R.; Emmersberger, C.; Ertlmaier, T.; Etzion, O.; Paulus, T.; Springer, F.: Existing and future standards for event-driven business process management. In: Proc. of the 3rd ACM Int. Conf. on Distributed Event-Based Systems. ACM, pp. 24:1–24:5, 2009.

[BDG07] Barros, A.; Decker, G.; Grosskopf, A.: Complex events in business processes. In: Business Information Systems. Springer, pp. 29–40, 2007.

[BS03] Börger, E.; Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Berlin, Heidelberg, 2003.

[DM09] Decker, G.; Mendling, J.: Process instatiation. Data & Knowledge Engineering, 68(9):777–792, 2009.

[EN11] Etzion, O.; Niblett, P.: Event Processing in Action. Manning Publications, 2011.

[Fe10] Ferreira, J.; Wu, Q.; Malkowski, S.; Pu, C.: Towards flexible event-handling in workflows through data states. In: SERVICES-1. IEEE, pp. 344–351, 2010.

[Fl12] Fleischmann, A.; Schmidt, W.; Stary, C.; Obermeier, S.; Börger, E.: Subject-Oriented Business Process Management. Springer, Berlin, Heidelberg, 2012.

[HMW13] Herzberg, N.; Meyer, A.; Weske, M.: An event processing platform for business process management. In: Proc. of the 2013 17th IEEE Int. Enterprise Distributed Object Computing Conf. IEEE, pp. 107–116, 2013.

[HSD10] Hermosillo, G.; Seinturier, L.; Duchien, L.: Using complex event processing for dynamic business process adaptation. In: Proc. of the 2010 IEEE Int. Conf. on Services Computing. IEEE, pp. 466–473, 2010.

[KG12] Kopetzky, T.; Geist, V.: Workflow charts and their precise semantics using abstract state machines. In: EMISA. LNI. Gesellschaft für Informatik e.V., pp. 11–24, 2012.

[Ko14] Kossak, F.; Illibauer, C.; Geist, V.; Kubovy, J.; Natschläger, C.; Ziebermayr, T.; Kopetzky, T.; Freudenthaler, B.; Schewe, K.-D.: A Rigorous Semantics for BPMN 2.0 Process Diagrams. Springer, 2014.

[Li14] Linden, I.; Derbali, M.; Schwanen, G.; Jacquet, J.-M.; Ramdoyal, R.; Ponsard, C.: Supporting business process exception management by dynamically building processes using the BEM framework. In: Decision Support Systems III, volume 184 of LNBIP, pp. 67–78. Springer International Publishing, 2014.

[LM07] Lucchi, R.; Mazzara, M.: A pi-calculus based semantics for WS-BPEL. The Journal of Logic and Algebraic Programming, 70(1):96 – 118, 2007.

[Lu02] Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley Professional, 2002.

[MNN05] Mendling, J.; Neumann, G.; Nüttgens, M.: Yet another event-driven process chain. In: Business Process Management, pp. 428–433. Springer, 2005.

[NC12] Natschläger-Carpella, C.: Extending BPMN with Deontic Logic. Logos Verlag Berlin, 2012.

[NG13] Natschläger, C.; Geist, V.: A layered approach for actor modelling in business processes. Business Process Management Journal, 19:917–932, 2013.

[OA07] OASIS: , WS-BPEL 2.0. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`. Accessed 2014-11-03., 2007.

[Ob11] Object Management Group: , Business Process Model and Notation (BPMN) 2.0. `http://www.omg.org/spec/BPMN/2.0`. Accessed 2014-11-03., 2011.

[Sc15] Schewe, K.-D.; Geist, V.; Illibauer, C.; Kossak, F.; Natschläger-Carpella, C.; Kopetzky, T.; Kubovy, J.; Freudenthaler, B.; Ziebermayr, T.: Horizontal Business Process Model Integration. In: Transactions on Large-Scale Data-and Knowledge-Centered Systems XVIII, pp. 30–52. Springer, 2015.

[vdAtH] van der Aalst, W.M.P.; ter Hofstede, A.H.M.: , Workflow Patterns Homepage. http://www.workflowpatterns.com. Accessed 2015-07-20.

[We12] Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer Science & Business Media, 2012.