Modelling and Enforcement of Inter-Process Dependencies with Business Process Modelling Languages

Georg Grossmann, Michael Schrefl and Markus Stumptner
University of South Australia, Advanced Computing Research Centre
Mawson Lakes, Adelaide, SA 5095, Australia
Email: {georg.grossmann, michael.schrefl, markus.stumptner}@cs.unisa.edu.au

Business process languages have been successfully applied in research and industry for the development of information systems. They are used for the analysis, design, construction, and verification of software applications and fit well into the model-driven paradigm. Although many business process languages have emerged in the last couple of years, they are in general not used for supporting the integration of existing information systems. In many cases, they are applied for designing an integrated system from scratch but less emphasis has been placed on integrating existing business process models which would be a natural consequence from a model-driven point of view.

In this article we identify inter-process dependencies for the integration of existing business processes and investigate their support from a modelling perspective by currently available business process modelling languages and from an enforcement perspective by an event driven architecture.

Keywords: BPMN, EPC, event-driven architecture, inter-process dependency, UML, WF-nets, YAWL

ACM Classifications: H.1.0 (Models and Principles – General), H.4.1 (Information Systems Application – Office Automation)

1. INTRODUCTION

The aim of this work is the consistent integration of software on the level of business processes. Business processes are a good candidate for modelling the behaviour of systems which can automatically be derived from existing systems as demonstrated by van der Aalst et al. (2004). We propose an integration approach based on so-called inter-process dependencies that are set between existing business processes. This article extends previous work that has received the best paper award at the Asia-Pacific Conference on Conceptual Modelling (APCCM) in 2008 (Grossmann et al., 2008). The extension includes a presentation of an event driven architecture for the enforcement of the identified inter-process dependencies in Sections 5 and 6.

Inter-process dependencies can be enforced by an event-based system (EBS) on top of local systems that execute business processes. A major advantage of building on these dependencies is that an EBS does not need to be designed from scratch and the system integrator, who is usually

1 This research was supported by the project “Dynamic Semantic Interoperability for Business Processes” funded by the Australian Research Council (ARC) under grant DP0988961.
familiar with the existing systems, can integrate them in an intuitive way. Having placed dependencies, it is possible to simulate and verify the interaction of local business processes and to identify incompatibilities and inconsistencies similar to those identified by Schrefl and Stumptner (2002). This approach is aligned with the current push towards Model-driven Architecture principles described by Koehler et al (2005).

The inter-process dependencies are the major outcome of the presented integration approach. They define the behaviour of a system that coordinates events and triggers on top of multiple local systems, similar to Distributed Event-Based Systems described by Mühl et al (2006). An event-based system (EBS) provides an event notification service, also known as publish/subscribe middleware that mediates between components of the EBS and conveys notification from publishers to subscribers that have registered their interest with a previously issued subscription. The presented approach abstracts the integration from publish/subscribe middleware and creates an EBS that is consistent to local business processes. Similar work by Schrefl and Stumptner (2002) as well as Küster et al (2007) investigated consistency criteria in the same context. An inter-process dependency between local business processes indicates a published event by its source and the subscription of that event including an effect by its target. The approach assumes, that local systems are process-aware, an important property of future generation information systems (Marlon Dumas, 2005). Activities are executed according to a business process and events like starting, completing, and cancelling of an activity, are accessible. Possible application scenarios are the integration of monitoring systems and enterprise application integration (EAI).

Example: A motivating example consists of three local business processes, a monitoring system MON, a resource planning system RES, and a risk management system RISK of a power plant POWER. MON monitors the equipment of POWER by reading values from sensors, and generates reports of the readings on a regular basis. RES administrates the maintenance routines of POWER, and RISK generates maintenance routine requirements for the equipment. Currently, the data exchange between the three systems is performed manually by employees. The aim of the integration is to automate the data exchange without interfering with the employees accustomed work practise, meaning that the integration is hidden from the user and existing systems should not be replaced by a new system. Figure 1 depicts three simplified processes of MON, RES, and RISK in BPMN notation, which when composed form a non-conformance maintenance process. A non-conformance maintenance is a set of unplanned work orders that result from evaluations performed in MON and RISK. The non-conformance maintenance process spans parts of all three processes and starts in process MON. On a regular basis, MON reads values from sensors installed on the power plant site and creates a report from the readings. If a defect is detected, a notification is created that needs to be exported to RES. There is some redundant functionality implemented in RES and RISK, for example, both systems administrate work orders. Work orders in RES are used for resource planning whereas work orders in RISK are necessary for completing the actual maintenance job. Therefore, if a work order is created in RES, then the same work order must be created in RISK. The responsible engineers receive necessary information from RES for performing the job on the plant site. If the job could be completed, the job is marked as “finished” in RISK and the work order is closed in RES. If the job could not be completed successfully then it is marked as “unfinished” in RISK and a failure report must be created in RES.

The inter-process dependencies are depicted by dotted arrows with two labels in Figure 1 and indicate events that cause an effect in another process. In the simplified example, each
event causes the invocation of a task the dependency is leading to. Labels “FINISH” and “START” indicate when the event is triggered at the source task of the dependency. “FINISH” means that the event is triggered when the task finishes execution, and “START” means that the task starts execution. The events of the two dependencies leading from MON to RES and RES to RISK are triggered when the task finishes because the output of the source task will be used as input for the target task. In case of the two dependencies leading from RISK to RES, the event is triggered when both source tasks start because the target tasks in RES can be executed in parallel but not before the decision about the job completion is made in RISK.

The article focuses on the control flow dependencies between objects. The underlying data flow that represents data exchange between activities of different objects is important, especially for instantiating the integrated business process, but structural based integration has been investigated intensively by related work (Cali et al., 2003; Garcia-Solaco et al., 1995). The integration approach imposes a need to provide models that are capable of representing the behaviour of systems and their dynamic interaction through events, and is specific enough to allow testing these in the design phase. While a number of notations investigated by Dumas et al. (2005) and Harvey (2005) have been developed for modelling the behaviour of information systems, they provide only limited support for modelling the types of inter-process dependencies that might occur in reality. The main limitation is that modellers have to consider the underlying execution semantics which prevents them from focusing on the design. The aim of this article is to remove this burden from the
Modelling and Enforcement of Inter-Process Dependencies with Business Process Modelling Languages

modellers by proposing a set of extensions. The extensions are demonstrated on the Unified Modelling Language (UML) and expressed in form of UML Stereotypes in the Activities meta model. In the following two sections we first demonstrate the drawbacks of the current conceptual models and then describe the set of stereotypes. We assume familiarity with Petri net semantics and the UML 2.0 notation.

2. INTER-PROCESS DEPENDENCIES ON SCHEMA LEVEL

Inter-process dependencies are usually derived from business rules and data dependencies and express in which order certain activities should be performed. Business rules are based on management decisions or regulations such as ISO standards and data dependencies are determined by data flow between input and output parameters of activities.

So far, inter-process dependencies have not been investigated independently from business rules and data dependencies and how they can be represented using a high-level business process modelling language. The advantages of an independent examination are (1) the ability to focus on the dependencies themselves apart from the environment, thus leading to (2) the identification of inter-process dependencies that are reusable in different domains, and (3) the ability to derive a comprehensive list of inter-process dependencies. An in-depth analysis of well-known intra-process dependencies can be used as a starting point to identify inter-process dependencies. We first explore the concept of intra-process dependencies and then apply the results to identify inter-process dependencies.

2.1 Exploring Intra-Process Dependencies

Intra-process dependencies define conditions of activity execution within a system and are implemented via common control structures that have been captured in the concept of workflow patterns identified by van der Aalst et al (2003). Taking Petri nets as an example, intra process dependencies are defined by the semantics of firing rules: A transition (1) may not fire if not all input places hold at least one token, (2) may fire if all input places hold at least one token, or (3) must fire if a transition is triggered which implies that the transition was enabled before (van der Aalst, 1998).

Observing “firing rules” from a higher point of view shows that they are composed of three parts: (1) a pre-condition or source condition, (2) a post-condition or target condition, and (3) an effect. A source condition is a state of an instance or a transition phase of an instance from one state into another that enables or triggers an effect. A target condition describes like the source condition a state of an instance or a transition phase of an instance. An effect describes if a target condition is enabled, triggered or disabled when a source condition holds (state) or happens (transition phase). For example, in Petri nets, if the source condition is “tokens are in all input places of transition $t$” then the effect is “$t$ fires” which consumes a token from each input place and the target condition is “instance enters $t$”.

Conditions: Source- and target condition specify either the state of an instance, i.e., an instance residing in a state or activity, or a transition phase of the instance in or out of an activity or state. Conditions in the context of intra-process dependencies are different to logic conditions in constraints because they also cover transition phases which refer to an event. However, they are similar in the way that they may specify the state of an instance which may also represent a logic condition.

In Petri nets, instances are represented by tokens, places are usually regarded as states and transitions as activities that change the state of a Petri net. Transitions (activities) in Petri nets
cannot hold tokens but some modelling languages like WF-nets adapted Petri net semantics in such a way that they can hold tokens. The reason for this is that activities usually take time due, for example, because of the execution of subactivities. Some modelling languages focus on a single primitive which are either activities, in case of UML Activity Diagrams, or states, in the case of UML State Machines. Some Petri net based notations support both, and map states to places and activities to a separate Petri net that defines the low-level behaviour of an activity with an implicit activity state meaning that states and activities can hold tokens. If both, activities and states, are permitted to hold tokens and the passing of a token through an activity or a state is observed, then six phases which may represent a source- or target condition can be identified: (1) $k$ enters $a$ (written as $*a$), meaning $a$ starts execution, (2) $a$ holds $k$ (written as $\delta a$), meaning $a$ is executing, and (3) $k$ leaves $a$ (written as $a*$), meaning $a$ finishes execution. The same phases can be observed for state $s$ where (4) $k$ enters $s$ (written as $*s$), (5) $s$ holds $k$ (written as $\delta s$), and (6) $k$ leaves $s$ (written as $s*$). It is assumed that only phases 2 and 5 consume time. An activity may take time because it consists of sub-activities that depend on internal and external events, and states represent “waiting positions” for events to happen.

**Effect:** The third property of an intra-process dependency, apart from source- and target-condition, is its effect. The effect is triggered by the source condition and forms an action in combination with a target condition. Three effects can be identified observing firing rules in Petri nets: (1) is enabled for “may fire”, (2) is triggered for “must fire”, and (3) is disabled for “may not fire”.

The analysis of intra-process dependencies showed that they consist of three parts. This result is the basis for identifying and specifying inter-process dependencies in the remainder of this section.

### 2.2 Identifying Inter-Process Dependencies

One significant difference between intra- and inter-process dependencies is that an inter-process dependency is set between two elements of different business processes whereas an intra-process dependency is set between two elements of the same process. We talk about a source business process when we talk about the process that holds the origin element and about the target business process when we talk about the process that holds the destination element of an inter-process dependency.

All possible intra-process dependencies can be used as a starting point to identify possible inter-process dependencies. Possible intra-process dependencies are specified by the combinations of source conditions, effects, and target conditions: If $C$ is the set of all possible source conditions, $V$ is the set of all possible target conditions, and $E$ is the set of all possible effects, then $D = C \times E \times V$ is the set of all possible intra-process dependencies. In this section we examine those possible dependencies and compare them to event-condition-action (ECA) rules which are suitable to describe inter-process dependencies.

ECA rules are well-known in the active database community and can be used for specifying dependencies between distributed systems (Paton and Diaz, 1999; Mühl et al., 2006). The event part of a rule describes a happening to which the rule may be able to respond. The condition part of the rule examines the context in which the event has taken place. The action describes the task to be carried out by the rule if the relevant event has taken place and the condition has evaluated to true (Paton and Diaz, 1999). In the following paragraphs we map elements of intra-process dependencies to parts of an ECA rule and identify which ECA rules can be used to describe practical inter-process dependencies from an object-oriented point of view.

**Events:** We use an event to describe an instance entering or leaving a state or activity in the source business process of an inter-process dependency. As described in Section 2.1, these events are also
included in the set \( C \) of source conditions that can be part of an intra-process dependency. Additionally to events, a source condition may specify a period of time, for example, while an instance is in a state \( s \) or activity \( a \) something should happen. The time period is determined by two events and is used in so-called composite inter-process dependency as described below.

**Actions:** An action defines the task that is performed in the target business process of an inter-process dependency and can be compared to the range of tasks performed by an intra-process dependency specified by the combinations \( T = E \times V \) of effects and target conditions. Table 1 shows all combinations and lists actions that refer to them.

An object-oriented point of view reduces the number of possible actions. The object-oriented execution model clearly defines publicly accessible and private data of objects, with the default being that object properties are private but can be accessed and updated through publicly available object methods. This concept is also known as *data encapsulation* and becomes relevant in the enforcement of inter-process dependencies. In a business process, states represent the status of object properties whereas activities represent the execution of object methods (Schreff and Stumptner, 2002). Under this assumption, inter-process dependencies that target a state can be ignored because they cannot be accessed directly from an external process. Therefore, the actions in the first three columns in Table 1 are not considered to be practical.

The externally triggered execution of activities follows certain rules in object-oriented systems. Publicly accessible activities are invocable from outside and, depending on the system, their execution can be stopped from outside which might result in an exception. Further influence on the execution from outside is normally not allowed; for example, an activity cannot be interrupted for a period of time or kept from completing successfully. Therefore all actions in column “executing an activity” in Table 1 and actions in column “finish an activity” in combination with “enable” and “disable” are not practical. We introduce a short-hand notation for the remaining five actions that are considered to be practical: (1) “enable an activity” is written as \( \triangleleft a \), (2) “trigger an activity” is written as \( \rightarrow a \), (3) “disable an activity” is written as \( \triangleright a \), and (4) “complete an activity” is written as \( \triangledown a \) and (5) “cancel an activity” is written as \( \triangledown a \). Completing an activity \( a \) means that \( a \) finishes successfully and the instance that executed activity \( a \) enters all post-states of \( a \), similar to a *commit* of a transaction.

<table>
<thead>
<tr>
<th>EFFECTS</th>
<th>TARGET CONDITIONS</th>
<th>Enter state</th>
<th>In state</th>
<th>Leave state</th>
<th>Start an activity</th>
<th>Executing an activity</th>
<th>Finish an activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable</td>
<td>Enable a state</td>
<td>Enable a state</td>
<td>Enable to leave a state</td>
<td>Enable an activity</td>
<td>Enable continue executing an activity</td>
<td>Enable completing an activity</td>
<td></td>
</tr>
<tr>
<td>Trigger</td>
<td>Move into a state</td>
<td>Move into a state</td>
<td>Leave a state</td>
<td>Trigger an activity</td>
<td>Continue executing an activity</td>
<td>Cancel or complete an activity</td>
<td></td>
</tr>
<tr>
<td>Disable</td>
<td>Disable a state</td>
<td>Disable a state</td>
<td>Hold in a state</td>
<td>Disable an activity</td>
<td>Pause executing an activity</td>
<td>Keep executing activity</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Possible actions specified by the combination of effects and post-conditions. Actions written in cursive are practical from an object-oriented point of view.
Cancelling \(a\) means that \(a\) finishes execution and the instance enters all pre-states of \(a\), similar to a *rollback* of a transaction. The completion of an activity \((a, \alpha)\) by an external event will not be considered further in the examples below because it occurs only in exceptional cases.

**Conditions:** As mentioned at the beginning of Section 2.2, the condition part of an ECA rule evaluates the context of an event. If the condition evaluates to true then the action is carried out. If it evaluates to false then either the rule is ignored, an exception is created, or other alternatives may be chosen (Paton and Diaz, 1999). Conditions can be compared to source conditions that describe a state of an instance in intra-process dependencies as discussed in Section 2.1 and are also evaluated for inter-process dependencies. However, we consider only conditions in this article that are required by the underlying business process execution semantics. Table 2 gives an overview of the conditions required for carrying out the previously identified actions. Each action can be triggered either internally or externally. In the former case two situations need to be distinguished: (1) If the activity that is involved with the action is not connected to an inter-process dependency then the same condition holds as for the corresponding intra-process dependency and (2) if the activity is also connected to an inter-process dependency then a must be enabled internally and externally. Those two cases are shown in the first four rows of Table 2. If the actions are triggered externally then a must also be connected to an inter-process dependency through which the actions are triggered. Those cases are shown in the last three rows of Table 2.

We extend activities that are connected to an inter-process dependency with a Boolean attribute called "*external enabling status*" to indicate at runtime whether an activity is externally enabled or disabled.

**Exception handling:** An exception may occur during runtime if an inter-process dependency is activated but the condition does not hold. We have not investigated exception handling in this work and concentrated on how the dependencies can be modelled using a well-known business process modelling language. An investigation of exception handling is part of future work. By default, activities are regarded as externally enabled. In case of an *enabling inter-process dependency*, the

<table>
<thead>
<tr>
<th>ACTIONS</th>
<th>CONTEXT</th>
<th>Activity (a) is only connected to an intra-process dependency</th>
<th>Activity (a) is connected to an intra- and inter-process dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable activity (a) internally</td>
<td></td>
<td>Activity (a) is disabled internally</td>
<td>Activity (a) is disabled internally</td>
</tr>
<tr>
<td>Trigger activity (a) internally</td>
<td></td>
<td>Activity (a) is enabled internally</td>
<td>Activity (a) is enabled internally and externally</td>
</tr>
<tr>
<td>Disable activity (a) internally</td>
<td></td>
<td>Activity (a) is enabled internally and not executing</td>
<td>Activity (a) is enabled internally and not executing</td>
</tr>
<tr>
<td>Cancel activity (a) internally</td>
<td></td>
<td>Activity (a) is executing</td>
<td>Activity (a) is executing</td>
</tr>
<tr>
<td>Enable activity (a) externally</td>
<td></td>
<td></td>
<td>Activity (a) is disabled externally</td>
</tr>
<tr>
<td>Trigger activity (a) externally</td>
<td></td>
<td></td>
<td>Activity (a) is enabled internally and externally</td>
</tr>
<tr>
<td>Disable activity (a) externally</td>
<td></td>
<td></td>
<td>Activity (a) is enabled externally and not executing</td>
</tr>
<tr>
<td>Cancel activity (a) externally</td>
<td></td>
<td></td>
<td>Activity (a) is executing</td>
</tr>
</tbody>
</table>

Table 2: Conditions of immediate-coupled actions
target activity must be disabled externally either during an initialisation phase or by another inter-
process dependency beforehand.

**Event-condition and condition-action coupling:** The execution of ECA rules is divided in phases
which include a **signalling phase** that refers to an event occurrence, an **evaluation phase** for
evaluating conditions, and an **execution phase** for carrying out actions. The order in which these
phases are executed depends on the **event-condition** and **condition-action** coupling (Paton and Diaz,
1999). The former determines when the condition is evaluated relative to the event that triggers the
rule. A condition-action coupling mode indicates when the action is to be executed relative to the
evaluation of the condition. Paton and Diaz (1999) survey three options for coupling:

- **immediate:** the condition (action) is evaluated (executed) immediately after the event (condition).
- **deferred:** the condition (action) is evaluated (executed) not necessarily at the earliest opportunity
  after the event (condition) is evaluated.
- **detached:** the condition (action) is evaluated (executed) within a different transaction from the
  event (condition).

Similar options can be observed in the context of inter-process dependencies. Events and
conditions are **detached** in inter-process dependencies because events occur in an external business
process that is monitored by a different transaction. Conditions (actions) are evaluated (executed)
internally and are coupled in two ways:

- **immediate:** the action is executed immediately after the condition is evaluated. In the remainder
  of this article actions that enable, disable or cancel an activity are immediate coupled.
- **deferred:** the action is executed once a condition evaluates to true. For example, if an activity that
  is internally disabled is triggered externally, then the activity is executed immediately when it gets
  enabled internally. Actions in inter-process dependencies that are specified as “trigger an activity
  externally” are deferred coupled in the remainder of this article and are regarded as **action
requests**. They are actually triggered when the conditions shown in Table 2 evaluate to true.

We have identified certain events, conditions and actions that are useful from an object-oriented
point of view to specify inter-process dependencies. We distinguish between primitive- and
**composite inter-process dependencies** similar to primitive and composite events in ECA rules
(Paton and Diaz, 1999). If the dependency is triggered by a single event like “starting an activity”
then we talk about a primitive dependency. If an inter-process dependency must hold during a time
period, for example, “while an activity is executed something must happen” then we talk about a
composite dependency because two primitive dependencies are required to specify it. The first
primitive dependency is triggered at the beginning of the period and the second dependency is
triggered at the end of the period.

**Primitive inter-process dependencies:** **Primitive inter-process dependencies** are defined by an
event that occurs in a source business process and an action that is executed in a target business
process. A primitive inter-process dependency \( i = (c, e) \) consists of an event \( c \in \{n, \rho, a\} \) where \( n \)
is an activity or a state, and an action \( e \in \{\rho, a, \rho, a, a\} \) where \( n \) and \( a \) must belong to different
business processes, and \( a \) is an activity. Source condition and effect are written side by side as short
hand notation, for example, \( *n \rho a \) stands for \( *(n, \rho, a) \)

**Composite inter-process dependencies:** A composite inter-process dependency describes a state \( s_i \)
in which a target business process must be while the source business process is in a certain state \( s_j \).
where $s_s$ is either $\bullet a$ or $\circ s$ and $a$ is an activity and $s$ is a state. The state $s_s$ is either activity is enabled, executing, disabled, or not executing. A composite inter-process dependency is defined by two primitive dependencies $i_1$ and $i_2$ where the event of $i_1$ indicates when the source business process enters $s_s$ and the event of $i_2$ indicates when the source business process leaves $s_s$. The action of $i_1$ brings the target business process into state $s_s$ and the action of $i_2$ brings the target process into a state complement to $s_s$, for example, if $s_s$ is “activity is enabled” then the complement state is “activity is disabled”. Modelling a composite dependency with two primitive dependencies has the advantages that the start and end of the time period while a source state is in a specific state are indicated.

A composite inter-process dependency is $i := (n, e)$ where $n$ is an activity or a state and $e$ is an action as a composite inter-process dependency which is defined by a pair $(i_1, i_2)$ of primitive dependencies. Dependencies $i_1$ and $i_2$ are specified by following four rules: (1) $i_1$ must hold an event that defines the start of $n$ which is $n^*$ and (2) $i_2$ must hold an event that defines the end of $n$ which is $\neg n^*$ and (3) the action defined in $i_1$ is $e$, i.e., it is equal to the action specified by the composite inter-process dependency and (4) the action of $i_2$ must be complementary to $e$, i.e., if $e = \neg \neg a$ then the effect of $i_2$ is $\neg a$.

Table 3 contains all identified inter-process dependencies that are considered to be practical in real-world business processes. Each dependency is explained through an example where $a$ and $b$ are activities of source and target business process, respectively, and $s$ represents a state in the source process. The dependencies are grouped according to their action. Within a group, dependencies are distinguished by their event that is triggered in the source process. Each group contains six dependencies as identified previously.

3. ANALYSIS OF HIGH-LEVEL BUSINESS PROCESS MODELLING LANGUAGES

Currently, a number of languages are widely used for business process modelling, in particular Petri net based or UML based ones, or other activity- or event-based languages mentioned by Dumas et al (2005) and Harvey (2005). A step towards standardisation was taken by the Object Management Group (OMG) in accepting the Business Process Modeling Notation (BPMN) 1.0 draft (OMG, 2006). In this section we are going to analyse Petri net based (WF-nets, YAWL), UML based (UML Activities), and activity/event centred languages (BPMN, EPC) in terms of their suitability for designing inter-process dependencies. Each modelling language defines its own terminology. For each language we use the corresponding terminology and in the remainder of the article, we use UML terminology if not stated otherwise. A comparison of terms can be found in the Appendix of (Grossmann, 2008).

We have analysed each modelling language to determine whether it supports the dependencies identified in Section 2. Possible outcomes were (1) the language supports the dependency directly (written “+”), (2) it supports the dependency indirectly (written “+(+)”), (3) it supports the dependency under certain assumptions (written “~”), or (4) it does not support the dependency (written “--”). Direct support means that a single node or arc can express the dependency whereas indirect support means that a network of elements is necessary to model it. An overview of the results is shown in the second half of the article in Table 4. Some examples illustrate the dependencies modelled in the specific languages. Each example includes a section of a business process source and a section of business process target which hold the source condition and the target activity $b$, respectively. Complex dependencies will not be investigated because they are composed of simple dependencies which also holds for all modelling languages presented here.
### Modelling and Enforcement of Inter-Process Dependencies with Business Process Modelling Languages

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabling dependencies ( (\sqsubseteq) )</td>
<td></td>
</tr>
<tr>
<td>( a \sqsubseteq b )</td>
<td>When a sports competition starts ( a ), access to the team statistics ( b ) is enabled.</td>
</tr>
<tr>
<td>( a \sqsubseteq b )</td>
<td>During travel booking ( a ), the optional reservation of a car is enabled ( b ).</td>
</tr>
<tr>
<td>( a' \sqsubseteq b )</td>
<td>When booking a travel ( a ) finishes, printing the booking ( b ) is enabled.</td>
</tr>
<tr>
<td>( s \sqsubseteq b )</td>
<td>When an order enters state “all parts delivered” ( s ), reclaiming the order is enabled ( b ).</td>
</tr>
<tr>
<td>( s \sqsubseteq b )</td>
<td>While a student is in state “is PhD candidate” ( s ), applying for an internship ( b ) is enabled.</td>
</tr>
<tr>
<td>( s' \sqsubseteq b )</td>
<td>When the temperature of cooling water leaves a critical state ( s ), the experiment is enabled to start again ( b ).</td>
</tr>
<tr>
<td>Triggering dependencies ( \to b )</td>
<td></td>
</tr>
<tr>
<td>( a \to b )</td>
<td>When the backup of all computers starts ( a ), the backup of a single computer ( b ) is invoked.</td>
</tr>
<tr>
<td>( a \to b )</td>
<td>While a sensor measures values ( a ), a backup sensor is activated ( b ) for value comparison.</td>
</tr>
<tr>
<td>( a' \to b )</td>
<td>When a security officer finishes his shift ( a ), another security officer must start his shift ( b ).</td>
</tr>
<tr>
<td>( s \to b )</td>
<td>When an operating system enters state “logged off” ( s ), an external backup is started ( b ).</td>
</tr>
<tr>
<td>( s \to b )</td>
<td>While a power plant is in state “in operation” ( s ), a monitoring system ( b ) is running.</td>
</tr>
<tr>
<td>( s' \to b )</td>
<td>When an order leaves state “in archive” ( s ), a notification service ( b ) is invoked.</td>
</tr>
<tr>
<td>Disabling dependencies ( \not\to b )</td>
<td></td>
</tr>
<tr>
<td>( a \not\to b )</td>
<td>When an inspection of a power plant starts ( a ), turning on the power plant ( b ) is disabled.</td>
</tr>
<tr>
<td>( a \not\to b )</td>
<td>During the booking of a bargain travel ( a ), the cancellation of the hotel ( b ) is disabled.</td>
</tr>
<tr>
<td>( a' \not\to b )</td>
<td>When a payment process finishes successfully ( a ), cancelling the payment ( b ) is disabled.</td>
</tr>
<tr>
<td>( s \not\to b )</td>
<td>When a hotel enters state “booked out” ( s ), booking a room is disabled ( b ).</td>
</tr>
<tr>
<td>( s \not\to b )</td>
<td>While a room resides in state “occupied” ( s ), the cleaning service of the room is disabled ( b ).</td>
</tr>
<tr>
<td>Cancelling dependencies ( \not\in b )</td>
<td></td>
</tr>
<tr>
<td>( a \not\in b )</td>
<td>When an experiment with higher priority starts ( a ), an ongoing experiment with lower priority ( b ) is cancelled.</td>
</tr>
<tr>
<td>( a \not\in b )</td>
<td>When an experiment with higher priority starts ( a ), an ongoing experiment with lower priority ( b ) is cancelled and restarted after ( a ) has finished execution.</td>
</tr>
<tr>
<td>( a' \not\in b )</td>
<td>When the supervisor leaves the experiment ( a ), a participating assistant must stop working ( b ).</td>
</tr>
<tr>
<td>( s \not\in b )</td>
<td>When the temperature of cooling water enters a critical state ( s ), the running experiment ( b ) is stopped.</td>
</tr>
<tr>
<td>( s \not\in b )</td>
<td>When the temperature of cooling water enters a critical state ( s ), the running experiment ( b ) is stopped and restarted after the temperature has fallen below the critical level.</td>
</tr>
<tr>
<td>( s' \not\in b )</td>
<td>When a hotel leaves state “rooms available” ( s ), a currently running hotel booking ( b ) is cancelled.</td>
</tr>
</tbody>
</table>

### 3.1 WorkFlow Nets

Work by van der Aalst (1998, 2000) proposed WorkFlow nets (WF-nets) based on Petri nets for modelling business processes and their integration. WF-nets define transitions of Petri nets as tasks and places as states between tasks (also called conditions). A task is specified by low-level behaviour which is hidden in a WF-net. The low-level behaviour consists of transition start that is followed either by transition commit or rollback, and places, which allow to model the trigger and the execution state. Representing an execution state explicitly allows to model the assumption that
Modelling and Enforcement of Inter-Process Dependencies with Business Process Modelling Languages

a task takes time. WF-nets support four different types of tasks depending on their triggers: *Automatic* tasks are triggered the moment they are enabled, *user* tasks are triggered by human participants, *message* tasks are triggered by an external event, and *time* tasks are triggered by a clock. Routing is supported by AND-split, AND-join, XOR-split, and XOR-join. During the evaluation, we modelled each dependency by adding elements to an initial structure of two WF-nets.

**Enabling dependencies**: The first group of dependencies involves externally enabling the target task \( \sim b \). Like disabling \( b \), enabling \( b \) is modelled with an additional resource state connected to \( b \) but a resource token is placed during the execution of the source process, meaning that \( b \) is disabled by default. Dependency \( a \sim b \) is modelled by inserting an AND-split that places a resource token in the resource place before \( a \) executes, as shown in Figure 2. Task \( b \) is enabled at the same time as \( a \) starts execution assuming that \( a \) is triggered immediately when it is enabled. The dependency \( a \sim b \) is modelled by changing \( a \) to an AND-split that places a resource token when it finishes execution. Dependency \( s \sim b \) is captured in the same way as the previous dependency where \( s \) is the post-state of the AND-split because a token does not need time when passing from AND-split to \( s \). The dependency \( s \sim b \) can be captured with an additional AND-split where \( s \) is a pre-state of the AND-split. It must be assumed that the AND-split does not consume time; otherwise \( b \) is not enabled at the same time as a token leaves \( s \). Due to assumptions, support for \( a \sim b \) and \( s \sim b \) is marked with "~". The remaining dependencies need more than one additional element and are therefore marked with "+".

**Triggering dependencies**: The second group includes dependencies which trigger the target task \( \rightarrow b \). WF-nets support four types of triggers as mentioned before. *Message* tasks are adequate for modelling dependencies in this group because they represent tasks that are triggered by an external event. However, triggers in WF-net do not possess an element notation which can be connected to an external flow. Therefore, it is not possible to model when the external event is sent from another WF-net. As an alternative, we can assume that target task \( b \) is always triggered automatically, and when \( b \) is enabled externally, it is triggered immediately. By this means, we set “enable externally” equal to “trigger” and receive the same examples as shown in the previous group of dependencies. Hence, the same results are contained in Table 4.

**Disabling dependencies**: The third group of dependencies which we evaluate covers the external disabling of target task \( b \) (\( \neg b \)). Disabling \( b \) is modelled by an additional resource state connected to \( b \) that holds a token prior to runtime, meaning that \( b \) is enabled externally by default. \( b \) is changed to an AND-join and -split that consumes the resource token when starting execution and returns it
after completion. By this means, \( b \) stays enabled externally after execution as shown in Figures 4–5. On the side of the source process, the resource token is consumed by a different transition in each dependency. In dependency \( a \rightarrow b \), it is consumed by \( b \) when it starts execution as shown in Figure 4. In dependency \( a \rightarrow b \), target \( b \) is disabled at the moment that \( a \) finishes execution. This can be captured with an additional AND-join that consumes tokens from the post-state of \( a \) and the resource state. The join is triggered immediately because it is an automatic task as shown in Figure 5. The dependency \( s \rightarrow b \) can be captured by a task that consumes the resource token and that is directly connected to \( s \). However, it must be assumed that this task does not consume time. The dependency \( s \rightarrow b \) is modelled in the same way as \( a \rightarrow b \) (depicted in Figure 4), where \( s \) is the pre-state of the AND-join. No time passes by when a token leaves \( s \) and enters the AND-join. All dependencies need more than one introduced element to model the dependencies and, therefore, they are supported indirectly. The dependency \( s \rightarrow b \) is only supported under assumptions mentioned earlier.

All examples shown before suffer two disadvantages. First, they imply an effect on the source WF-net. If \( b \) is executing then \( a \) cannot execute, because a missing resource token blocks the execution. This problem can be solved by decomposing \( b \) into its low-level behaviour and connecting the resource state with two arcs, going to and from the start transition within \( b \). A second disadvantage is that \( a \) can only be executed once, which means it cannot be part of an execution loop in the source WF-net.

**Cancelling dependencies:** The fourth group covers dependencies which cancel a target task. In WF-net, there exists no element which can model cancelling a task. However, cancelling \( b \) can be modelled by decomposing \( b \) into its low-level behaviour and applying semantics of extended Petri nets as explained by Peterson (1977). The decomposition is necessary to model transition rollback of \( b \) explicitly so it can be triggered externally. Figure 6 illustrates \( b \), where \( b \) is decomposed in three transitions start, commit, and rollback, and an execution place with the same name as the task that holds a token while \( b \) is executing. Task \( b \) has a pre-state \( s_1 \) and post-state \( s_2 \), and a place trigger, which models the trigger of \( b \). As mentioned already, transition rollback must be triggered for cancelling \( b \). For this, two conditions must be hold, first, rollback must be enabled, and second, rollback must be given priority over commit. This can be achieved by zero-testing: the introduction of arcs from a place \( s \) to a transition \( t \) which allow the transition to fire only if the place \( s \) has zero tokens in it, also called inhibitor arcs. For enabling rollback, a new place called cancel\(_b\) is introduced with an outgoing arc to rollback. An inhibitor arc going from cancel\(_b\) to commit is added that gives rollback priority over commit. The inhibitor arc is drawn according to Peterson (1977) as...
shown in Figure 6. If a token resides in $cancel_b$ and another token resides in execution state $b$ then only transition rollback can be fired. The inhibitor arc will also be used later in Section 4 for defining the semantics of the language extensions. Table 4 marks all cancelling dependencies as unsupported by the WF-net specification because it does not support the extended Petri net semantics explained before.

3.2 Yet Another Workflow Language

Yet Another Workflow Language (YAWL) arose from the idea to support a set of control flow patterns with moderate modelling effort and a formal foundation based on Petri net semantics, see also van der Aalst et al (2002). It consists of a set of extensions to WF-nets and YAWL workflows are also called extended workflow nets (EWF-nets).

One of those extensions is the remove token functionality. It is relevant when modelling disabling and cancelling dependencies because it enables the removal of tokens from places. The visual representation consists of a dashed rounded rectangle and lines which are connected to a task. If this task is executed then all tokens within the rectangle are removed. According to the YAWL technical report: “... the moment the task executes all tokens in this area are removed.” (cf. van der Aalst and ter Hofstede, 2002, p.14). It is assumed that the tokens are removed when the task starts execution. In contrast to this, a revised version of YAWL called newYAWL was released by Russell et al (2007) which defines that a token is removed when a task finishes execution, (cf. Russell et al, 2007, p.13). The language newYAWL supports workflow patterns also from a data and resource perspective, and models a remove token functionality with a so-called cancellation region. Here, we investigated the description found in van der Aalst and ter Hofstede (2002).

Enabling and triggering dependencies: In contrast to WF-net, dependency $a \leftarrow b$ can be modelled in YAWL without the assumption that $a$ must be an automatic task because two tasks can be connected directly. This means that the state between AND-split and $a$ in Figure 2 that models the WF-net version of dependency $a \leftarrow b$ can be removed and $a$ is executed immediately after the AND-split. The remaining dependencies are supported in the same way as WF-nets, since no extensions are provided which would enhance their support.

Disabling dependencies: Disabling dependencies are indirectly or only under certain assumptions supported by simple WF-nets. YAWL improves the support in some cases by taking advantages of the newly introduced remove token functionality. The advantage lies in the expressiveness of the language. It became possible to model an activity $a$ that can be executed while $b$ is executing and...
that can be part of an execution an execution loop because it can be executed several times. These two circumstances cannot be realised by WF-nets as explained in Section 3.1. The dependency $a \not\in b$ can be modelled slightly different to simple WF-nets as depicted in Figure 7. However, we have to assume that the task that removes tokens does not take time and that $a$ is triggered immediately after tokens are removed, so that disabling $b$ happens at the moment that $a$ starts execution. The remaining dependencies can be modelled in a similar way as shown in Figure 7, assuming that the task that removes tokens does not consume time. Due to the assumptions made for modelling the dependencies, the results are the same as for WF-nets.

**Cancelling dependencies:** The dependency $a \not\in b$ (depicted in Figure 8) assumes that a token is removed from $b$ when $a$ starts execution. The dependency $a \not\in b$ is modelled in the same way, assuming that the token is removed when $a$ completes execution.

The remaining dependencies $s \not\in b$ and $s \not\in b$ are modelled by an additional task where $s$ is the post-state and pre-state of that task, respectively. This task causes the token removal from $b$. For $s \not\in b$, it must be assumed that the token is removed when the task finishes execution. For $s \not\in b$, it must be assumed that token removal happens at the end of the execution.

The modelling of dependency $\not\in b$ with the remove token functionality of YAWL may lead to undesired behaviour from an object-oriented point of view. If a token represents a business process instance or an identifier of an object, it is not defined what happens with that instance or object id once it is removed. In case of a process instance, it may mean that a business case is not processed further or is aborted without executing a compensation task. A practical solution would be to leave the token in the net and rollback the target task of the dependency. However, YAWL does not include a rollback transition compared to the low-level behaviour of a WF-net task (cf. van der Aalst, 1998, p.26 and van der Aalst and ter Hofstede, 2002, p.24).

![Figure 7: Dependency $a \not\in b$.](image)

![Figure 8: Dependency $a \not\in b$.](image)
3.3 UML 2.0 Activities

UML has been accepted by the software industry as well as in research which made it a de-facto standard for modelling during object-oriented analysis and design. A major revision of UML in version 2.0 made significant changes especially to the semantics of UML Activity Diagrams by introducing Petri-net like semantics (Note: During the writing of this article, UML 2.1.2 was released but no significant changes between version 2.0 and 2.1.2 relevant to the content of the article have been identified). Engels et al (2005) use it as a fundamental tool for discussing various process modelling perspectives, like control- and data flow, pre- and post-conditions, hierarchical process composition, process interaction, and exception handling.

UML 2.0 specifies two types of nodes which may contain activity behaviour, Actions and Activities. Both have the same shape but differ in their ability to be decomposed further and in their execution semantics. Activities may contain sub-activities and may consume tokens from some input edges whereas actions cannot be decomposed further and only start execution if all input edges offer at least one token, (cf. OMG, 2005, p.301, 308). Routing is supported by fork-, join-, decision-, and merge nodes which have the same semantics as AND-split, AND-join, XOR-split, and XOR-join in WF-nets, respectively.

UML Activities do not provide an element for states. As alternative, the element ObjectNode can be used which provides a property inState that may specify a set of states of an object type (OMG, 2005, p.380). However, object nodes belong to the data perspective in UML and can only be connected via object flows to activities (OMG, 2005, p. 376, 380). Actions cannot be connected to object nodes (OMG, 2005, p.376), and all in- and outgoing edges of control nodes must be of the same type, meaning that either control flow or object flow (OMG, 2005, p.349, 363, 369, 374). Join nodes may have incoming edges of different types but in this case the outgoing edge must be of type object flow (OMG, 2005, p.369). This syntax constraint makes it difficult to model inter-process dependencies that involve states as source conditions as demonstrated later.

UML Activities provide elements for cancelling activities. The elements consist of a specific region int of type InterruptibleActivityRegion, a send signal action send outside of int, and an accept event action accept within int where send can send tokens over an edge to accept. If accept receives a token from send then all tokens within int are removed. Action accept might be connected to an exception handler which is an activity outside of int and that is invoked after tokens were removed. These elements will be used later for modelling $\not b$.

Since formal semantics for UML have not been widely adopted (Grossmann et al, 2005; Störrle, 2004; Wohed et al, 2005), we have to make one assumption on the execution semantics of UML that hold for all inter-process dependencies: All control nodes including fork, join, decision, and merge, do not consume time. In order to rate the support of different dependencies in more detail we do not mark all dependencies with “~” by default but on the basis of the previous mentioned assumption. For example, if a dependency is marked with “~” then this means that further assumptions are required apart from control nodes not consuming time.

Enabling dependencies: Enabling dependencies can be captured by a network consisting of fork- and join nodes. This network is able to overcome the lack of resource state and -token. A fork node on the source diagram splits the local control flow and allows sending a token to the other process. Outgoing edges of fork nodes have the ability to store the tokens until they are accepted by the destination node (OMG, 2005, p.363). This is important, as otherwise the control flow in the source diagram would be blocked from further execution.
The join node in the target diagram ensures that \( b \) is only executed if both local and external control flows offer tokens. This construct is used for modelling all dependencies shown in Figures 9–12.

**Triggering dependencies**: Triggering dependencies can be modelled using the UML send signal action which is directly connected to the target activity. A send signal action is defined as “an action that creates a signal instance and transmits it to the target object, where it may cause the execution of an activity” (OMG, 2005, p.394). Dependencies with an activity as source condition are modelled with a send signal action. Figure 13 depicts dependency \(*a \rightarrow b\. The dependency \( a^* \rightarrow b\)
can be modelled straightforward by connecting $a$ directly to the send signal action. All dependencies holding a state as source condition cannot be modelled because outgoing object flows from an object node cannot be connected to a send signal action (OMG, 2005, p.376). Therefore a send signal action cannot be set in a sequence with an object node that represents a state.

**Disabling dependencies:** Disabling dependencies are not directly supported by UML Activities. This is due to the fact that there exists no element comparable to a resource state in WF-nets with a pre-existing “resource token”. Instead, disabling an activity $b$ can be emulated with the introduction of a fork node that has the initial node as predecessor. The function of the fork is to distribute a token to an object node when the business process starts execution. The object node serves as a resource state and is directly connected to $b$, meaning that $b$ is enabled when the process starts execution. Disabling $b$ can be modelled by an interruptible activity region that holds the object node. An example of an interruptible activity region is depicted in Figure 14. Activity $b$ is disabled by sending a cancel signal from a send signal action that is received by an accept event action within the interruptible activity region. In a next step, a token is traversed from the accept event action to an Exception Handler activity which causes the removal of all tokens within the interruptible activity region. In UML, the time between sending and receiving a signal is undefined (OMG, 2005, p.274), and it must be assumed that cancel is received immediately or the inter-process dependency does not hold. Therefore, the support for modelling disabling dependencies by UML Activities is marked with “~”.

**Cancelling dependencies:** Dependencies with an activity as source condition can be captured by an interruptible activity region, as shown in Figure 14. For dependencies $a \triangleright b$ and $a^* \triangleright b$, it must be assumed that the transmission of the signal does not take time. Similar to triggering dependencies, dependencies with a state as source condition cannot be modelled because of syntactic constraints between object nodes, object flows, and actions.

### 3.4 Business Process Modelling Notation

The Business Process Modelling Notation (BPMN) is a new standard for modelling business- and Web service processes. It was introduced by the Business Process Management Initiative (BPMI) and has been adopted by the Object Management Group (OMG). BPMN 1.1 was released in 2008 and included minor changes to the notation and a new event type called signal event compared to its predecessor. However, there is no difference in the support of inter-process dependency between the two BPMN versions. A more detailed description of the changes in the latest release can be found in Decker and Schreiter (2008).
Similar to UML Activities, BPMN is an activity centred modelling language and does not provide a modelling primitive for states between activities. On the other hand, BPMN provides **events** that capture situations between activities. Events are categorised in **start**, **intermediate**-, and **end** events where start events have no incoming and end events no outgoing **sequence flow**. Each event may specify a trigger. The triggers that will be used later for modelling dependencies are **message**, in combination with start events, and **cancel**, in combination with intermediate events. A **message** trigger specifies that a message arrives from a participant and triggers the start of a process. A **cancel** trigger is used within a **transactional sub-process** and is triggered if a cancel message has been received. The phases $s$ and $s'$ of a state $s$ can be captured by two intermediate events set in a sequence. The first event captures $s$ and the second event $s'$. However, it is not common practise to model two intermediate events in a sequence and might confuse the modeller.

Modelling two business processes and their interactions within one BPMN diagram is directly supported by BPMN compared to the other languages. If more than one process is defined, then each process is placed in a **pool** that defines the container of a participant. **Activities** within a pool can be **sub-processes** and may contain sub-activities or sub-tasks. Tasks cannot be decomposed further. Activities are connected with **sequence flows**, where a sequence flow cannot cross pool boundaries. For crossing pool boundaries, **message flows** are provided which capture the interaction between two participants. Messages are handled by different types of tasks. Relevant task types for inter-process dependencies are **service**, **send**, and **receive**. A **service** task holds two attributes, **InMessage** and **OutMessage**, which are sent when the task starts and completes execution, respectively (cf. OMG, 2006, p.64). A **send** task completes execution after sending a message (OMG, 2006 p.65), and a **receive** task waits for a message to arrive and completes after receiving it (OMG, 2006 p.64). We assume here that sending a message does not consume time.

**Enabling dependencies**: Enabling a task in another pool can be modelled by introducing a message flow leading to a receive task that enables the target task $b$. The message flow transfers a message **enable**, from the source BPMN process to a receive task in the target process. This task is connected to $b$ with an AND-join that merges the control flows from the receive task and other local tasks executed prior to $b$. The source condition of enabling dependencies are realised by sending message **enable** at different points in time. This can be modelled by making use of the task attributes **InMessage** and **OutMessage** explained above. Dependency $a \xrightarrow{\text{enable}} b$ is realised by setting **InMessage** = “Enable” meaning that enable is sent when $a$ starts execution as shown in Figure 15. In dependency $a \xrightarrow{\text{enable}} b$, property **OutMessage** = “Enable” is defined, meaning that enable is sent when $a$ finishes execution. The dependencies $s \xrightarrow{\text{enable}} b$ and $s' \xrightarrow{\text{enable}} b$ can be realised with two intermediate events in a sequence that represent $s$ and $s'$, respectively. Representing the former dependency, the first event in the sequence holds trigger **message**, whereas in latter dependency, the second event holds the message trigger. In both dependencies, the event with the trigger is connected to the receive task in the target process.

**Triggering dependencies**: Triggering dependencies can be captured by a **start event** and a **message** trigger. A start event can only be attached to a process. Therefore, the task which should be triggered must be placed in a sub-process of the target process and the start event is attached to the border of the sub-process as shown in Figure 16. The start event is triggered by message start which is received from the source process. Depending on the different dependencies within the group, message **start** is sent at different points in time as explained for enabling dependencies.

**Disabling dependencies**: Disabling dependencies are not directly supported by BPMN for the same reason mentioned for UML Activities: there is no element in the language that is comparable to a
resource state with a pre-existing “token” element in WF-nets. It could be emulated with a set of Data Object nodes that serve as a resource state but this would overload a business process and make it more difficult to understand from the modeller perspective.

Cancelling dependencies: Dependencies in this group are modelled in analogy to triggering dependencies. A task can be cancelled if it is placed within a transactional sub-process that has an intermediate event attached to its border; the event is triggered if a cancel message has been received as shown in Figure 17. When the cancel message is sent depends on the specific dependency and is defined by task properties as explained for triggering dependencies.

3.5 Event-Driven Process Chains
Event-driven Process Chains (EPC) have become a widespread process modelling technique because of the success of products such as SAP R/3 and ARIS. EPCs describe the flow of control of business processes as a chain of functions, events, and logical connectors. Functions represent activities in a business process. An event expresses a pre-condition (trigger) for a function or a post-condition that signals the termination of a function. Logical connectors AND, OR, and XOR are used according to their names to build the control flow of a process in a natural way as described...
Modelling and Enforcement of Inter-Process Dependencies with Business Process Modelling Languages

Enabling dependencies: In the group enabling dependencies, only $a \rightarrow b$ and $s \rightarrow b$ can be modelled. The remaining dependencies are not supported due to the syntactical constraints. The two supported dependencies can be captured by inserting an additional event and an AND-join in the target diagram. Both dependencies are modelled in the same way and shown in Figure 18.

Triggering dependencies: We model triggering dependencies in the same way as the dependencies of the previous group but with a different point of view. We regard the additional event in the target process as a “trigger” event. This is allowed because EPCs do not specify events further and leave their definition open for the modeller. Hence, the support of dependencies in this group is equal to the support of the previous group.

Disabling and cancelling dependencies: Disabling and cancelling dependencies are not supported by EPCs because elements similar to resource place and -token as well as mechanisms for cancelling a function do not exist.

4. EXTENSIONS FOR BUSINESS PROCESS MODELLING LANGUAGES

Business process modelling languages need a formal foundation for representing inter-process dependencies. A formal foundation allows modelling inter-process dependencies without ambiguities and without the necessity to consider assumptions, for example, in contrast to UML 2.0 Activities. We have chosen the formal definition of object life cycles as a foundation because life cycles allow to model business processes from an object-oriented point view (Schrefl and Stumptner, 2002). Object life cycles (OLC) provide a higher-level, overall picture on how instances of object types may evolve over their lifetime, whereas programming languages represent the behaviour by a set of operations. OLCs determine the legal order of activity and states of an object type. In the examples presented later, each diagram represents an OLC of an object type. OLCs are comprised of activities, states and edges corresponding to transitions, places, and arcs of Petri nets, where tokens represent object identifiers belonging to an object of a specific type that changes its state over time.

In the following, we propose a set of extensions to business process modelling languages for...
Modelling and Enforcement of Inter-Process Dependencies with Business Process Modelling Languages

modelling OLCs and inter-process dependencies. The goal of the extensions is (i) to support all identified inter-process dependencies directly and (ii) to express their semantics in form of extended Petri nets. We will demonstrate the set of language extension on the hand of UML 2.0 Activities and would like to stress that the extension can be applied to the other languages as well. The extensions consist of additional activity properties and the introduction of two element types. The first element type is *State*, which is only introduced if no equivalent element is already provided by the language; the second element type is *Link*, where links are further specified by a specific *link type*. Before we explain the extensions in detail, we give a formal definition of the notation which results from the mapping of UML to OLC.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>WF-net</th>
<th>YAWL</th>
<th>UML 2.0 Activities</th>
<th>BPMN</th>
<th>EPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabling dependencies (⊙ b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*a ⊙ b</td>
<td>~</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>–</td>
</tr>
<tr>
<td>* a ⊙ b</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>~</td>
<td>–</td>
</tr>
<tr>
<td>*a' ⊙ b</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(†)</td>
</tr>
<tr>
<td>*s ⊙ b</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td>*s' ⊙ b</td>
<td>(+)</td>
<td>(+)</td>
<td>~</td>
<td>~</td>
<td>–</td>
</tr>
<tr>
<td>Triggering dependencies (→ b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*a → b</td>
<td>~</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>–</td>
</tr>
<tr>
<td>a → b</td>
<td>(+)</td>
<td>(+)</td>
<td>~</td>
<td>(+)</td>
<td>–</td>
</tr>
<tr>
<td>a' → b</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(†)</td>
</tr>
<tr>
<td>s → b</td>
<td>(+)</td>
<td>(+)</td>
<td>~</td>
<td>(+)</td>
<td>–</td>
</tr>
<tr>
<td>s' → b</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>(~)</td>
<td>–</td>
</tr>
<tr>
<td>Disabling dependencies (≠ b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*a ≠ b</td>
<td>(+)</td>
<td>(+)</td>
<td>~</td>
<td>~</td>
<td>–</td>
</tr>
<tr>
<td>a ≠ b</td>
<td>(+)</td>
<td>(+)</td>
<td>~</td>
<td>~</td>
<td>–</td>
</tr>
<tr>
<td>a' ≠ b</td>
<td>(+)</td>
<td>(+)</td>
<td>~</td>
<td>~</td>
<td>–</td>
</tr>
<tr>
<td>*s ≠ b</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>–</td>
</tr>
<tr>
<td>s ≠ b</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>–</td>
</tr>
<tr>
<td>s' ≠ b</td>
<td>(+)</td>
<td>(+)</td>
<td>~</td>
<td>~</td>
<td>–</td>
</tr>
<tr>
<td>Cancelling dependencies (∧ b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*a ∧ b</td>
<td>–</td>
<td>~</td>
<td>~</td>
<td>(+)</td>
<td>–</td>
</tr>
<tr>
<td>a ∧ b</td>
<td>–</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>–</td>
</tr>
<tr>
<td>a' ∧ b</td>
<td>–</td>
<td>~</td>
<td>~</td>
<td>(+)</td>
<td>–</td>
</tr>
<tr>
<td>s ∧ b</td>
<td>–</td>
<td>~</td>
<td>~</td>
<td>(~)</td>
<td>–</td>
</tr>
<tr>
<td>s' ∧ b</td>
<td>–</td>
<td>~</td>
<td>~</td>
<td>(~)</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 4: Comparison of modelling languages: + directly supported, (+) indirectly supported, ~ supported under certain assumptions, – not supported.
4.1 Formal Semantics of UML 2.0 Activities

It is well-known that UML Activities lack formal semantics. Due to this fact, some of the inter-process dependencies can only be modelled under certain assumptions as described in Section 3.3. A formal definition of UML 2.0 Activities based on the definition of OLCs eliminates ambiguities and simplifies the specification because it covers only a subset of the elements which are sufficient.

Initial- and activity final node: The subset of UML includes initial- and activity final node for starting and completing an object life cycle. However, we had to adapt the semantics of activity final nodes to the definition of OLC. A token represents an object identifier and may be present in different (activity) states at the same time. This means that in more than one (activity) state the same token may reside, similar to pointers in a programming language that resides in different (activity) states that refer to the same token. In contrast to this, more than one place in a Petri net cannot hold the same token at the same time. An activity final node from an OLC point of view has the task to collect all pointers which refer to the same token. In contrast, according to the UML specification, the first token that enters the activity final node (1) is destroyed and (2) causes the removal of remaining tokens (cf. OMG, 2005, p.320). This behaviour is not aligned with the definition of OLCs, which determine the legal order of activity and states: first, a token referred by a pointer that is destroyed implies the removal of all remaining tokens which refer to the same object identifier, meaning that subsequent activities and states of the removed pointers are skipped, and second, remaining tokens with different object identifiers are removed as well, which is not desired.

Activities: The basic building blocks for modelling behaviour are activities and are adopted for the formalisation approach.

Control flow nodes: For splitting and merging control flows, nodes decision, merge, fork, and join are included and correspond to XOR-split, OR-join, AND-split, and AND-merge in WF-nets, respectively.

Each UML node is mapped either to an activity or to a state in an OLC. Initial-, activity final-, decision, and merge nodes are mapped to states, and activity-, fork-, and join nodes are mapped to activities. Control flows are mapped to arcs. We call the notation that results from the mapping of UML 2.0 Activities to OLC “Activity State Diagram”.

Definition 1 (Activity State Diagram). An Activity State Diagram (ASD) \( B_O = (S_O,T_O,F_O) \) of an object type \( O \) (the subscripts are omitted if \( O \) is understood) consists of a set of states \( S \neq \emptyset \) (initial node/activity final nodes, decision-merge nodes), a set of activities \( T \neq \emptyset \) (activities, actions, fork-/join nodes), \( T \cap S = \emptyset \), and a set of arcs \( F \subseteq (S \times T) \cup (T \times S) \). There is a distinguished state in \( S \), the initial state \( \alpha \) (initial node), where for no \( t \in T \): \( (t,\alpha) \in F \); \( \alpha \) is the only state with this property. There is a nonempty set of distinguished states in \( S \), the final states \( \Omega \) (activity final nodes), where for no \( s \in S \) and no \( t \in T \): \( (s,t) \in F \) and the states in \( \Omega \) are the only states with this property.

We say an activity \( t \in T \) may consume a token (or object identifier) from a state \( s \in S \) if and only if \( (s,t) \in F \), and \( t \) may produce a token into \( s \in S \) if and only if \( (t,s) \in F \). Due to the underlying Petri net semantics, an Activity State Diagram determines the legal sequences of states and activities, and thus the legal sequence in which activities may be applied: an activity may be applied to an object if the object is contained in every pre-state of the activity and it is enabled. If an activity on some object has been executed successfully, the object is contained in every post-state of the activity but in no pre-state unless that pre-state is also a post-state. Unlike Petri nets, where a transition is automatically fired if every pre-state contains a token, an activity in an ASD diagram...
must be explicitly invoked (triggered) for an object which is in every pre-state of the activity. In addition, and unlike Petri nets, activities take time. Therefore, during the execution of an activity on an object, the object resides in an implicit state named after the activity. This state is referred to as activity state. Thus, we can say that every instance of an object type is at any point in time in one or several (activity) states of its object type, which are jointly referred to as life cycle state.

**Definition 2 (Life cycle state).** A life cycle state (LCS) $\sigma$ of an object type $O$ is a subset of $S \cup T$. We denote the initial LCS $\{\alpha\}$ by $A$.

The definition of Object Life Cycles complements existing business process modelling languages with formal execution semantics and with consistency criteria for the specialisation of system behaviour (Schrefl and Stumptner, 2002). Later aspect plays an important role in the abstraction of existing business processes and tailoring of reference processes to specific requirements.

The low-level semantics of an activity $a$ are defined by extended Petri net semantics and shown in Figure 19. Places $s_1$ and $s_2$ represent pre- and post-states of $a$, respectively. Place $s_1$ enables transition $\text{start}$ that represents the execution start of activity $a$. It fires if a token resides in places $s_1$, $\text{triggered}_a$, and no token resides in place $\text{disabled}_a$. Latter place is connected with an inhibitor arc leading to transition $\text{start}$. The inhibitor arc indicates that if $\text{disabled}_a$ holds a token then $\text{start}$ cannot fire as explained in Section 3.1. By default, no token resides in $\text{disabled}_a$. If an enabling dependency is connected to an activity $a$ then a token is placed in $\text{disabled}_a$ during an initial phase that runs when a process instance is created.

After $\text{start}$ has fired, a token is produced in place $a$ that represents the execution state of activity $a$. In case of a successful execution of activity $a$, a token is placed in $\text{completed}_a$ which causes transition $\text{commit}$ to fire and a token is produced in post-state $s_2$. If $a$ fails then a token is placed in $\text{cancelled}_a$ and transition $\text{rollback}$ is fired. In both cases, inhibitor arcs connected to places $\text{cancelled}_a$ and $\text{completed}_a$ and transition $\text{commit}$ and $\text{rollback}$, respectively, make the internal behaviour of $a$ deterministic. Places $\text{disabled}_a$, $\text{triggered}_a$, $\text{cancelled}_a$ and $\text{completed}_a$ need to be accessible from outside so an event system can instantiate an inter-process dependency.

4.2 Extensions of UML 2.0 Activities
In this section, three extensions are made to the UML 2.0 Activities specification.
Activity properties: In previous section, places triggered, disabled, completed, and cancelled defined part of the low-level semantics of an activity $a$. They are implemented in UML by extending the UML node type Activity with properties isTriggered, isDisabled, isCompleted and isCancelled that hold a value of type boolean. Their default value is false.

State: Definition 4.1 includes states which are not part of the UML Activity specifications. In UML Activities, object nodes can be used as an alternative to states but due to syntax constraints, for example, object nodes may not have incoming or outgoing control flows, they cannot be used in the same way as states in WF-nets. Therefore, we introduced an explicit representation of states and added node type StateNode to the UML 2.0 Activities meta model. StateNode is defined as a subclass of ActivityNode in form of a UML Stereotype where incoming and outgoing edges must be of type ControlFlow. The formal semantics of states were explained in Section 4.1. The graphical notation for states is adopted from the ObjectNode specification.

Links: The elements defined in UML 2.0 Activities are not sufficient for modelling inter-process dependencies directly as shown in Section 3.3. We have introduced a new edge type, Link, to handle this situation. A new subclass Link is added to class ActivityEdge of the UML specification. We specify four different link types as subclasses of Link according to their effects on target conditions. A disable link is a link that disables an activity externally. An enable link enables an activity externally which was disabled before. An invoke link triggers an activity externally, a force to complete link causes a running activity to be finished, and a cancel link cancels an activity externally. The moment when a link disables, enables, triggers, or cancels depends on its source condition.

The semantics of links are defined by an additional Petri net arc that connects the low level behaviour of two business processes together. Table 5 contains four examples of dependencies holding an activity as source condition. They are modelled by links in UML notation shown on the left hand side in each example. The Petri net that corresponds to the dependency is shown on the right hand side of each example. A link in UML corresponds to one or two arcs within a Petri net. For better readability the corresponding arcs are highlighted by a dotted line.

The semantics of links that have a state as source condition are also defined by an additional Petri net but this Petri net is more complex. If source condition $s$ is modelled then its semantics are defined by a Petri net with an additional arc for each activity $a_i$ that is directly connected to $s$ where each additional arc is leading from the commit transition of activity $a_i$. Similar to this, if $s^{*}$ is modelled then its semantics are defined by a Petri net with an additional arc for each activity $a_i$ to which $s$ is directly connected to where each additional arc is leading from the start transition of activity $a_i$.

The UML notation for links consists of a directed arrow which is labelled with the name of the link type in UML Stereotype notation. A second label, attached to the origin of the link, specifies when the link is activated. The label on the origin can be either start, running, or finish and corresponds to the conditions $a$, $a^*$, and $a^{**}$, respectively, if the source node is an activity $a$. If the source node of a link is a state $s$ then the label is either enter, while, or leave which correspond to $s$, $s^*$, and $s^{**}$, respectively. Examples for the UML notation of disable, enable, invoke, and cancel links are shown on the left side of each example in Table 5.

5. EVENT DRIVEN ARCHITECTURE

Pietzuch et al (2007) emphasise an Event-Driven Architecture (EDA) for a loose coupling of information systems. Event-based system realise a publish/subscribe communication paradigm in
which subscribers are able to express their interest in events that are generated by publishers and that match their registered interest as described by Eugster et al. (2003). Publishers are in this case the producers of information and subscribers are the consumers of the information where information is denoted by the term event and the act of delivering it by the term notification.

In a basic interaction scheme, a subscriber calls a subscribe() operation on the event service, without knowing the sources of these events, and a publisher calls a publish() operation for propagating events, without actually knowing the recipients. This mechanism allows decoupling the
Modelling and Enforcement of Inter-Process Dependencies with Business Process Modelling Languages

production and consumption of information and increases scalability. Event-based interaction withdraws the control of interaction from the participating partners (Mühl et al., 2006), an important advantage for business environments that need to adapt to changes quickly and cost efficiently.

Example: A power plant monitoring system, as introduced in Section 1, needs to be extensible in order to communicate with new systems. New systems may include recently developed mathematical models such as the one presented by Daigle et al. (2007) that extend the current capabilities of a monitoring system, for example, with simulation and diagnosis functionality.

Event-based systems provide two advantages compared to peer-to-peer connections proposed by conventional service oriented architectures (SOA). First, partners need not participate actively in the interaction at the same time, also known as time decoupling mentioned in Eugster et al. (2003), and second, events can be broadcast to multiple partners at the same time. The first aspect is beneficial if it is unknown when and by whom a transaction is started, or when a reply will arrive. The latter aspect is required if information needs to be delivered quickly to several partners, for example, information about an emergency due to a component failure in a power plant.

Eugster et al. (2003) point out that an EBS reduces coordination and thus synchronisation requirements where interacting partners do not need to know each other. This is also known as space decoupling and implies that partners communicate only with an event system and do not keep a reference to a partner.

These properties are not suitable for scenarios similar to the one introduced in Section 1 because the execution of a business process may consist of several instances that are related to some instances of other processes. A reference is therefore required to identify related instances in order to send messages to the correct recipient.

The presented proposal adapts some properties, such as space decoupling, to fit the requirements of business transactions. Publish/subscribe mechanisms can be combined with SOA and might be the future architecture for EAI and B2B integration solutions. One disadvantage of event-based systems is the lack of standards. However, some standards in combination with SOA are currently under investigation by Monsieur et al. (2007).

Leymann (2006) pointed out the strong relationship between business process modelling and Service Oriented Architecture. Processes are a good candidate for modelling the behaviour of services and for modeling the integration of systems because they provide adequate elements for specifying the orchestration and choreography of services. This point of view together with the above mentioned advantages of an EDA support strongly the idea of combining both approaches in a single framework as proposed in this work.

The following sections describe requirements for local systems that implement local business processes to support inter-process dependencies by an event-based system.

5.1 Expressing EDA Functionality

Transactions are an essential interaction mechanism in distributed systems and ensure consistency between related instances. We distinguish between local (component) and global (distributed) transactions similar to distributed transaction management systems (Connolly and Begg, 2004).

A local transaction $l$ is modelled by the inner behaviour of an activity $a$ shown in Figure 19 which is hidden from the modeller and defined by a Petri net. The significant parts of the Petri net are transitions $\text{start}$, $\text{commit}$ and $\text{rollback}$. A local transaction starts, commits or rolls back if the corresponding transition within an activity fires.
A global transaction $g$ consists of a set of local transactions and ensures that its local transactions distributed over different business processes are executed by corresponding instances. This is important in cases where several instances of a business process are running concurrently.

If a local transaction of a global transaction fails and rolls back then it can be executed again without cancelling the global transaction. If a global rollback is required due to the local fail, similar to a two-phase commit (2PC) protocol, then this can be modelled explicitly. An example of a business process that models a 2PC protocol is shown in Grossmann et al (2005).

Space decoupling is one advantage of a traditional event-based system. However, for modelling transactions it is not suitable as mentioned in Section 5. In order to model transaction with inter-process dependency we distinguish between initial and reply dependencies which require the two event types broadcasting and directed.

**Broadcasting events**: Broadcasting events can be found in traditional EBS. When an event is observed, it is broadcast through an event channel. The sender of the event is not aware of the event receivers. Only the event based systems is aware of them because of their subscriptions to the event. In the context of business processes, we later redefine broadcasting events in the way that they are directed to a certain business process but not to a certain instance of that process.

**Directed events**: We introduce directed events so a sender is able to determine the receiver of an event. This is an important feature to ensure that a specific business process instance receives the event. The only difference to a broadcasting event is that a directed event has a business process instance ID attached and only an instance with that ID can receive it.

**Initial dependencies**: An initial dependency indicates the start of a global transaction. By default, the first inter-process dependency between a source business process $B_s$ and a target business process $B_t$ that occurs in a control flow sequence is an initial dependency that is associated with an event profile of type broadcasting.

**Example**: Activities MON.create_notification and RES.create_work_order in Figure 1 are part of an initial dependency.

**Reply dependencies**: A reply dependency $d_r$ that connects from process $B_s$ to process $B_t$ is a dependency that follows another dependency $d_i$ in a control flow sequence. The dependency $d_i$ connects the same processes but in the opposite direction from $B_t$ to $B_s$. Both dependencies $d_i$ and $d_r$ deal with the same process instances during an execution where $d_i$ is called a reply dependency to $d_r$. It must be ensured that for each event sent by an initial dependency there is one event sent by a reply dependency. To facilitate this property, we assume that an initial- and a corresponding reply dependency are not part of a control flow loop.

By default, all dependencies from $B_s$ to $B_t$ that follow a dependency from $B_t$ to $B_s$ in a control flow sequence are reply dependencies and associated with a directed event profile.

**Example**: The activities mark_job_as_finished and mark_job_as_unfinished of RISK in Figure 1 are part of a reply dependency to the initial dependency leading from RES.create_work_order to RISK.create_work_order.

The coordination of sending information between partners has to be handled externally in an event based system and requires certain functionality provided by local systems:

**Public functions**: Each activity $a$ in a business process has certain properties, for example, disabled, triggered, completed and cancelled that need to be accessible externally so an action of an
inter-process dependency can be performed from outside. The public functions enable(), disable(),
invoke(), force-complete(), and cancel() provide access to those properties.

The properties of an activity \( a \) are defined by places \( \text{disabled}_a, \text{triggered}_a, \text{completed}_a \) and \( \text{cancelled}_a \) in a Petri net that defines the low level behaviour of activity \( a \) as shown in Figure 19. They are located outside of the activity border indicating that they can be connected with another business process. The call of a public function is defined by connecting arcs to and from those places.

**Example:** Table 5 includes dependency \( \alpha \rightarrow \beta \) on the top left: While activity \( a \) is executing, activity \( b \) is enabled. A token is set in place \( \text{disabled}_b \) during an initial phase because of dependency \( \alpha \rightarrow \beta \) as explained in Section 4.1. The arc leading from \( \text{disabled}_b \) to transition \( \text{start} \) of activity \( a \) express that function enable\((b)\) is called when \( a \) starts executing and the arcs leading from \( \text{commit} \) and \( \text{rollback} \) of activity \( a \) to \( \text{disabled}_b \) express the call of function enable\((b)\).

**Observing events:** In order to instantiate an inter-process dependency that has an activity as event, transitions \( \text{start} \), \( \text{commit} \), and \( \text{rollback} \) of activity \( a \) must be observable by an event system. For dependencies having entering or leaving a state as an event, tokens passing on incoming and outgoing arcs of a state must be observable.

### 5.2 Components

The proposed architecture consists of the component types event detector, event channel, rules engine, and event which can also be found in work by Michelson (2006). An event detector recognises event occurrences and deposits them on an event channel that serves as message backbone transporting events between detectors, rule engines, and subscribers. A rule engine evaluates produced events against rules and initiates actions according to them. Rules may include complex business rules, for example, if certain events appear in combination then a new business process instance is created. In the presented architecture only simple rules are supported where subscriptions are compared with incoming events and actions are taken if they match. If an exception occurs then an exception event is created by the rule engine and published on the event channel. Events are created in a local system using event profiles and can be subscribed using subscription profiles.

**Event profile:** An event profile \( R \) of an inter-process dependency \( d \) is a tuple \((e,v,U,S,P)\) that consists of an event \( e \), a delivery type \( v \), two business process instance IDs (BPIIDs) \( U \) and \( S \), and a set \( P \) of input parameters. Each BPIID consists of an object type- and object instance ID \((t,i)\) where \( U \) identifies the process instance publishing \( e \) and \( S \) identifies the process instance subscribed to \( e \). Pair \( S \) is needed by the rule engine to deliver \( e \) to a specific subscriber if \( e \) is a directed event.

Set \( P \) contains parameter values that are required by function invoke\((a,P)\) called by the rule engine in case of a trigger dependency. If the event of a dependency is \( \alpha^* \) or \( \alpha^\ast \) of an activity \( a \) then the input parameters of \( a \) can be used for \( P \). If the event is \( \alpha \) then the output parameters of \( a \) can be used.

An event profile describes a specific delivery type \( t \) which is either broadcasting or directed. A broadcasting profile does not contain information about a subscriber, which means that \( S \) is undefined and can be received by any system subscribing to event \( e \). A broadcasting event is used to initiate transactions by an initial dependency (explained in Section 5.1). A directed profile contains information about a specific subscriber in \( S \) and events of this profile will be delivered only...
to this subscriber. Directed events are used for enacting reply dependencies (explained in Section 5.1).

An event is consumed by a subscriber which can communicate with an event-based in two ways. Paton and Diaz (1999) identified two communication models that are applied in the presented approach as well. The push model allows a producer of an event to initiate the transfer of event data to subscribers. The pull model permits a subscriber of events to request the event data from a producer. We use the push model for initiating enabling or disabling activities and the pull model for invoking, forcing-to-complete, or cancelling activities because the enforcement of the last three actions require local conditions to be checked beforehand as explained in Section 2.2.

A local system subscribes for events by submitting subscription profiles.

Subscription profile: A subscription profile \( O \) of a dependency \( d \) is a tuple \( (e, n, t_U, S, f, c) \) where \( e \) is an event, \( n \) is a notification type where \( n \) can be push or pull, \( t_U \) is the business process ID of the publisher, \( S \) is the business process instance ID of a subscriber, \( f \) is a handle for calling a public function implemented by the subscriber as explained in Section 5.1, and \( c \) is the optional condition similar to those used in event-condition-action (ECA) rules.

6. ENFORCEMENT
The enforcement of inter-process dependencies at runtime requires proper setup for each modelled inter-process dependency before business processes are instantiated. During setup a rule engine identifies possible event occurrences and partners that are interested in particular events. The setup consists of a sequence of function calls that is explained according to the event flow from an event generator to an event subscriber.

6.1 Event Generation
When an inter-process dependency \( d \) is created during design time, an event profile \( R \) is generated for \( d \) by the local system that executes the source business process of \( d \). During runtime, profile \( R \) is instantiated and advertised to the event channel.

Creating an event profile: A profile \( R \) as defined in Section 5.2 is created for a dependency \( d \) with some default values and advertised to the event system. The event \( e \) of \( R \) written \( e(R) \) is equal to the event that is defined by the source of \( d \). The delivery type \( v \) of \( R \) written \( v(R) \) is broadcasting if \( d \) is an initial dependency as explained in Section 5.1, or directed if \( d \) is a reply dependency. The business process instance IDs \( U \) and \( S \) (which hold the IDs of producer and subscriber) as well as the input parameters in \( P(R) \) are determined during runtime when \( R \) is instantiated. After the default values are assigned, profile \( R \) is advertised by calling function advertise() as explained by Pietzuch et al (2007) to announce the intention of publishing event \( e \).

Example: The example in Figure 1 shows dependency

\*RISK.mark_job_as_finishedRES.close_work_order. When this dependency is modelled a profile, \( R \) is created for it. \( R \) holds the start of activity

\*RISK.mark_job_as_finished as event \( e \) and the delivery type is directed because the dependency is a reply dependency to dependency

RES.create_work_order\*→RISK.create_work_order.

Instantiating an event profile: Profile \( R \) for dependency \( d \) is instantiated during runtime and then published to the event channel. For a business process instance \( h \), the instantiation takes place when an event of \( h \) is observed. During instantiation, \( U(R) \), the event producer ID, is assigned the ID of \( h \).
If the delivery type of \( R \) is \textit{directed} then \( S(R) \) is assigned the ID of a subscriber that should receive \( e \).

If \( R \) was created for a \textit{triggering dependency} then input parameters are assigned to \( P(R) \). After all required values of \( R \) are assigned, profile \( R \) is published on the event channel by calling publish(\( R \)) as defined by Pietzuch \textit{et al.} (2007).

\textbf{Example:} When an instance \( h \) enters activity RISK.mark_job_as_finished then the profile \( R \) created in the previous example is instantiated with \( U(R) \) being set to the ID of \( h \). If the delivery type is \textit{directed} then a subscriber instance ID that was previously received by dependency RES.create_work_order \( \rightarrow \) RISK.create_work_order is assigned to \( S(R) \). No values are assigned to the input parameters \( P(R) \) in the example.

\subsection*{6.2 Event Subscription}

When a dependency \( d \) is created during design time then a \textit{subscription profile} \( O \) is generated for \( d \) by the local system that executes the target business process of \( d \). During runtime, profile \( O \) is instantiated when a business process instance is created and subscribed to the event system.

\textbf{Creating a subscription profile:} A subscription profile \( O \) as explained in Section 5.2 is generated with default values for a dependency \( d \).

The event of \( O \) (written \( e(O) \)) is identified by the source of \( d \). The notification type of \( O \) (written \( n(O) \)) is \textit{push} if the function \( f \) of \( O \) is enable() or disable(), or \textit{pull} if \( f \) is invoke(), force-complete() or cancel(). The function of \( O \) (written \( f(O) \)) depends on the dependency \( d \) as explained in Section 5.1. The ID of the source business process is assigned \( t_U \). If a condition is defined for \( d \) then it is assigned to \( c(O) \). The business process instance ID of the subscriber is assigned during runtime when \( O \) is instantiated.

\textbf{Example:} A profile \( O \) is created by RES for the dependency 'RISK.mark_job_as_finished\( \rightarrow \)RES.close_work_order' shown in Figure 1. Event \( e(O) \) is the start of activity RISK.mark-job-as-finished. The function \( f \) is invoke(close-work-order) and the notification type is \textit{push} because \( f \) is an invoke function. The producer ID \( t_U \) is RISK. The optional condition \( c \) is not defined in this case and stays empty.

\textbf{Instantiating a subscription profile:} A subscription profile \( O \) is instantiated when a new process instance is created and published to the event channel. During instantiation, the ID of \( h \) is assigned to \( S(O) \). Then function subscribe(\( O \)) as described by Pietzuch \textit{et al.} (2007) is called.

\textbf{Example:} The profile \( O \) which was created in the previous example is instantiated when a new instance \( h \) of RES is created. The ID of \( h \) is assigned to \( S(O) \) and function subscribe(\( O \)) is called by RES.

It is the task of the rule engine to process the events when they are published by local systems.

\subsection*{6.3 Event Processing}

The rule engine is responsible for distributing events. It consumes new published events on the event channel and calls public functions to execute actions if certain conditions hold. Two types of events may be received:

\textbf{Broadcasting event is published:} If the rule engine receives an event of a profile \( R \) with delivery type \textit{broadcasting} then it searches for a previously received subscription profile \( O \) which holds three conditions: (1) \( e \in R \) is equal to \( e \in O \) meaning that the events match, (2) the business process...
ID of the event producer in $U(R)$ is equal to $t_U(O)$, and (3) condition $c(O)$ executed with the input parameters $P(R)$ returns true. If $c$ is empty then the last condition is disregarded.

**Example:** If an instance of MON finishes activity `create_notification` shown in Figure 1 then a broadcasting event is published because of the initial dependency `MON.create_notification`→`RES.create_notification`. When function `RES.create_notification` is enabled, RES sends a notification to the event system for pulling the event that invokes the activity. After the event system has received the event from MON and a notification from RES, it calls function `invoke(create_notification)`.

**Directed event is published:** If the rule engine receives a directed event then it searches for a subscription profile $O$ that holds the same conditions mentioned for broadcasting events. The condition is that the business process instance IDs of the subscriber defined in both profiles must match.

If a profile $O$ is found then an action defined by function $f(O)$ is executed. Depending on the notification type $n(O)$ function $f(O)$ is called in two different ways:

$n(O)$=push: The rule engine calls a public function using the function handle defined in $f(O)$. The function implements the effect that is similar to the action of an event-condition-action rule. It is either enabling or disabling an activity $a$.

$n(O)$=pull: The rule engine calls a public function using handle $f(O)$ after it has received a notification by a function call `notify_cb_wait(sub_handle)` from the subscriber where `sub_handle` is the subscription handle of $O$ that was received by the subscriber during subscription. The function that is accessible by $f$ implements the action of the ECA rule which is either triggering, forcing-to-complete, or cancelling an activity $a$.

The subscriber of $O$ calls `notify_cb_wait(sub_handle)` under a certain condition: If $f(O)$ is an `invoke(a)` function where activity $a$ is an activity then `notify_cb_wait()` is called if activity $a$ is locally and externally enabled. This means that all pre-states of activity $a$ must hold a token and place $\text{disabled}_a$ must not hold a token in the corresponding Petri net of activity $a$.

If $f(O)$ is `force-complete(a)` or `cancel(a)`, then the condition holds if activity $a$ is executing.

**Example:** If a RISK instance $h$ starts executing activity `mark_job_as_finished` then a directed event to a RES instance that corresponds to $h$ is published on the event channel. If the corresponding instance has placed a notification on the channel that activity `close_work_order` is enabled then public function `RES.invoke(close_work_order)` is called.

It may happen that an immediate action that is triggered by an event $e$ cannot be executed immediately. This is the case if $e$ is subscribed by a profile with notification type pull and the corresponding function `notify_cb_wait()` has not been called by the subscriber when the event is published. As a result an exception event is published on the event channel that needs to be subscribed by an exception handler.

7. RELATED WORK

The interaction across boundaries of traditional business process management was investigated by Barros et al (2005, 2007). On the level of message exchange and conversions, interaction- and correlation patterns were identified. The difference to the presented work is that it is on a different abstraction level and that these patterns were not mapped to the business process modelling languages investigated here. Possible application areas of the language extensions we presented are
Modelling and Enforcement of Inter-Process Dependencies with Business Process Modelling Languages

modelling interaction and inter-organisational workflows. Inter-organisational workflows have been proposed for inter-connection of business processes by van der Aalst (2000) and Chebbi et al (2006). Schulz and Orlowska (2004) investigated the communication aspects between workflows and defined control flow dependencies and state dependencies. For synchronising control flows only AND-split and AND-joins were used. The definition of state dependencies is similar to phases of a token passing an activity node. The main difference to our work is that state dependencies were defined between a public and private workflow where the public workflow serves as a proxy for the private workflow. In this situation the workflow management system has access to further states that a task can obtain, for example, a task is temporarily suspended or a task is created but was not started yet.

Related work for the enforcement of inter-process dependencies can be found in the area of event driven architecture (EDA). Active database systems are one candidate that realises dependencies between distributed systems by event-condition-action rules as described by Paton and Diaz (1999) and they are written in a textual notation. A common publish/subscribe API was described by Pietzuch et al (2007) who investigated existing systems and standards. Some of the API functions are used in this article.

8. CONCLUSION
We have defined a set of inter-process dependencies between two conditions and analysed commonly used business process modelling languages with respect to their support. Since none of the presented languages support all dependencies directly, we proposed a set of extensions for the investigated languages and demonstrated them through UML 2.0 Activities. We proposed an event driven architecture for the enforcement of inter-process dependencies using event- and subscription profiles.

In the future, we plan to identify categories of scenarios where specific dependencies are useful. For example, dependencies involving states as source condition play an important role in monitoring systems where critical states must be observed whereas dependencies between activities might be more frequent in business-to-business relationships. Furthermore, we are going to implement a model checker for identifying incompatibilities of inter-process dependencies during their instantiation.

REFERENCES

320 Journal of Research and Practice in Information Technology, Vol. 42, No. 4, November 2010
Modelling and Enforcement of Inter-Process Dependencies with Business Process Modelling Languages


Modelling and Enforcement of Inter-Process Dependencies with Business Process Modelling Languages

BIOGRAPHICAL NOTES

Georg Grossmann is a Research Fellow at Advanced Computing Research Centre (ACRC) within the University of South Australia (UniSA). He works in the area of business process integration and obtained his PhD in 2008 from UniSA. His doctoral thesis focused on the behaviour-based integration of object oriented information systems and was awarded with the Ian Davey Research Thesis Prize for the most outstanding research thesis at UniSA in 2008. Before coming to Australia, he obtained a MSc in Economics and Computer Science jointly from the University of Vienna and the Vienna University of Technology. Current research interests include behaviour-based integration of web services, ontology-driven integration and distributed event-based systems.

Michael Schrefl received his Dipl.-Ing. degree and his Doctorate from Vienna University of Technology, Vienna, Austria, in 1983 and 1988, respectively. During 1983–1984 he studied at Vanderbilt University, USA, as a Fulbright scholar. From 1985 to 1992 he was with Vienna University of Technology. During 1987–1988, he was on leave at GMD IPSI, Darmstadt, where he worked on the integration of heterogeneous databases. He was appointed Professor of Information Systems at Johannes Kepler University of Linz, Austria, in 1992, and Professor in Computer and Information Science at University of South Australia in 1998. He currently leads the Department of Business Informatics – Data and Knowledge Engineering at University of Linz, with projects in data warehousing, workflow management, and web engineering. Dr Schrefl is a member of ACM and IEEE Computer Society.

Markus Stumptner received his MS and PhD degrees in computer science from Vienna University of Technology, Austria, in 1985 and 1990, respectively. He is full Professor of Computer Science at the University of South Australia, Adelaide, where he directs the Advanced Computing Research Centre. His research interests include object-oriented modelling, knowledge representation, and model-based reasoning in areas such as database and service integration, configuration and diagnosis. He has published more than 150 papers in journals and refereed conferences. He is a member of the ACM and IEEE Computer Society and a member of the Australian Computer Society Steering Committee for Artificial Intelligence.