Modelling Inter-Process Dependencies with High-Level Business Process Modelling Languages

Georg Grossmann **Michael Schrefl** **Markus Stumptner**

University of South Australia, Advanced Computing Research Centre, Mawson Lakes, SA 5095, Adelaide, Australia. E-mail: {cisgg, cismis, mst}@cs.unisa.edu.au.

Abstract

The work presented in this paper targets the software integration on the level of business process models. The goal is to create the behavioural description of an integrated system that is consistent with the behavioural descriptions of the original local systems intended to be integrated. We build the behavioural description from existing models of the local systems by inserting dependencies between them. By this means, simulation and verification of interactions between them is possible and incompatibilities can be identified at an early stage before a new system is introduced. So far, business process modelling languages have mainly focused on a single application system although B2B and enterprise application integration demand on models to express cross organisational communication and inter-process dependencies. In this paper, we investigate commonly used business process languages on their suitability to model inter-process dependencies. The result shows that there is no language which supports all identified dependencies directly and that all languages demand from the modeller to consider their low-level semantics which prevent him from focusing on the design. We propose a set of extensions of UML 2.0 Activity Diagrams to overcome these limits.

Keywords: BPMN, EPC, inter-process dependency, UML, WF-nets, YAWL.

1 Introduction

The aim of the work presented here 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 through process mining. The idea behind the integration approach is to identify dependencies, so called *inter-process dependencies*, between existing *local* business processes which create a new *global* business process. The global business process integrates the local ones and can either be used to replace them or to act as a mediator between them. The advantage of building on these dependencies is that the global business process does not need to be designed from scratch and the system integrator, who is usually 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 & Stumptner 2002). This approach is aligned with the current push towards Model Driven Architecture principles (Koehler et al. 2005).

The global business process is the major outcome of the integration approach. It defines the behaviour of a system that coordinates events and triggers on top of multiple local systems, similar to Distributed Event-Based Systems (Mühl et al. 2006). In event-based systems, an event notification service or publish/subscribe middleware mediates between the components of an event-based system (EBD) and conveys notification from publishers to subscribers that have registered their interest with a previously issued subscription. The approach presented here abstracts the integration from publish/subscribe middleware and apply object-oriented techniques, especially inheritance, to create a global business process that is consistent to local business processes defined by object life cycles (Schrefl & Stumptner 2002). The global process inherits activities from local business processes for observation and contains inter-process dependencies between these activities. An inter-process dependency indicates a published event by its source and the subscription of that event including an effect by its target. The approach assumes that the local systems are process-aware (Dumas et al. 2005), i.e., activities are executed according to a business process, and that events, like starting, completing, and cancelling of an activity, are accessible, e.g., through an API. 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 the employees accustomed work practise, i.e., the integration is hidden from the user and should not be replaced by a new integrated system. Figure 1 depicts three simplified processes of MON, RES, and RISK in BPMN notation (see also Section 3.4), which when composed form a *non-conformance main-*

This research was partially supported by the CIEAM CRC Project SI-302 "Improved OPAL Monitoring and Management System".

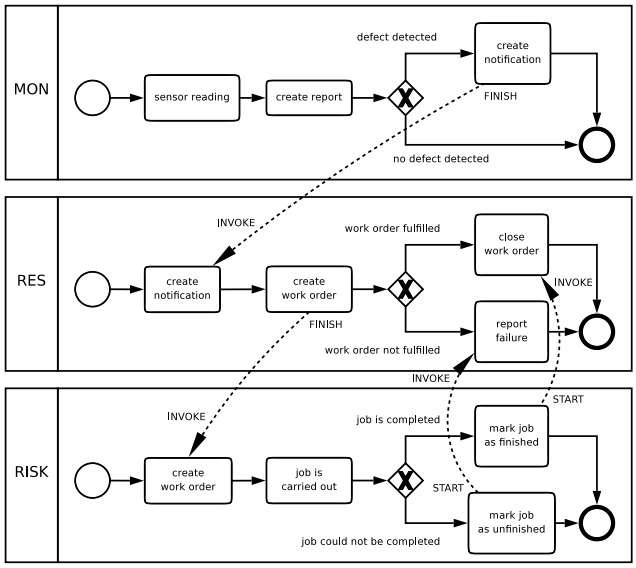

Figure 1: Example of the *non-conformance process* that is composed of MON, RES, and RISK process.

tenance process. A non-conformance maintenance is a set of unplanned work orders that result from evaluations performed in MON or 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, e.g., 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" when 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 paper 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 (e.g., (Garcia-Solaco et al. 1995)). The approach imposes a need to provide models that are capable of representing the behaviour of systems and their dynamic interaction, and is specific enough to allow testing these in the design phase. While a number of notations (Dumas et al. 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 the modeller has to consider the underlying execution semantics which prevents him from focusing on the design. The aim of this paper is to remove this burden from the modeller by proposing a set of extensions. The extensions are demonstrated on the Unified Modeling 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, based on management decisions or regulations such as ISO standards. On the technical side, data dependencies restrict the order of activity execution, as activities depend on certain data which might only be available at specific points in time. So far, inter-process dependencies have not been investigated independently from business rules and data dependencies in the context of modelling languages. The advantages of such an independent examination are (1) the ability to focus on the dependencies themselves apart from the environment, thus leading to (2) inter-process dependencies that are reusable in different domains, and (3) the ability to derive a comprehensive list of inter-process dependencies. This last benefit can be achieved by an in-depth analysis of *intra-process dependencies* and deriving inter-process dependencies from them.

In contrast, 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. 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) an *effect*, and (3) a post-condition or *target condition*. For example, in Petri nets, the source condition might be "token is in input place", the effect is "enable", and the target condition is "fire transition":

Conditions: In Petri nets, the source- and target condition can be defined by the position of a token. For example, a token resides in the input place of a transition before the transition fires (source-condition), and the token resides in the output place after the transition has fired (target-condition). Places are usually regarded as states and transitions as activities that change the state of a Petri net but cannot hold tokens. However, in common business process modelling languages, it is practical that activities may hold a token because activities take time

due to subactivities. Therefore, activities can also be part of a condition. Some modelling notations focus on a single primitive which are either activities, in case of UML Activity Diagrams, or states, in the case of UML State Machines. Many Petri net based notations support both, and map states to places and activities to a Petri net that defines the low-level behaviour of an activity with an implicit *activity state*, i.e., states and activities may hold a token. If we permit both activities and states to hold tokens and observe the passing of a token k through an activity a or a state s, then we can observe six phases which may represent a condition: (1) k enters a (written as ${}^\bullet a$), meaning a starts execution, (2) a holds k (written as $\dot{a}$), meaning a is executing, and (3) k leaves a (written as $a^\bullet$), meaning a finishes execution. The same phases can be observed for state s where (4) k enters s (written as ${}^\bullet s$), (5) s holds k (written as $\dot{s}$), and (6) k leaves s (written as $s^\bullet$). 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.

Effects: The third property of a firing rule, next to source- and target condition, is its *effect*. The effect is caused by the source condition and coupled with the 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".

2.1 Identifying Inter-Process Dependencies

Not all possible combinations of source condition, effect, and target condition are useful for modelling inter-process dependencies. 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 dependencies between two conditions.

Conditions: An object-oriented point of view identifies impractical dependencies $d \in D$. Object-oriented systems clearly define 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*. In a business process, states represent the status of object properties whereas activities represent the execution of object methods. Under this assumption, it does not make sense to model states as target conditions of inter-process dependencies because they cannot be accessed directly from an external process. On the other hand, using states as a source condition may make sense. E.g., if an object *request* enters state "failed", it might be necessary to trigger a notification service to send an email to the sender of *request*. The event of exiting a state might be used as source condition as well. E.g., if an object *ship* leaves state "in port", then a notification might be sent to the logistics department that the ship has left the harbour. The conclusion from the above is that all possible conditions are candidates for *source conditions*.

Effects: An object-oriented point of view considers only activities as target condition because of data encapsulation. The externally caused execution of activities also 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, e.g., an order is cancelled while it is processed on the seller side. Further influence on the execution from outside is normally not allowed; e.g., an activity cannot be interrupted for a period of time. The two conditions "starting an activity" (${}^\bullet a$) and "finishing an activity" ($a^\bullet$) may represent a target condition where the latter condition may only be used in conjunction with *is triggered*. This leads to four possible *effects*, (1) enabling an activity (written as $\curvearrowright$a), (2) triggering an activity (written as $\rightarrow a$), (3) disabling an activity (written as $\not\curvearrowright a$), and (4) triggering (or forcing) the end of an activity execution, which means either completion (written as $\lightning_c a$) or cancelling an activity (written as $\lightning a$). Completing an activity a means that a finishes successfully and the instance that executed a enters all post-states of a, compared to a *commit* of a transaction. Cancelling a means that a finishes execution and the instance enters all pre-states of a, compared to a *rollback* of a transaction. The completion of an activity ($\lightning_c a$) by an external event will not be considered in the examples below because it occurs only in exceptional cases, e.g., an activity sends email-reminders constantly till an external event stops the loop.

Simple inter-process dependencies: *Simple inter-process dependencies* are defined by a condition within a source business process and an effect on an activity held by a target business process. A simple inter-process dependency $i := (c, e)$ consists of a source condition $c \in \{{}^\bullet n, n^\bullet\}$ where n is an activity or a state, and an effect $e \in \{\curvearrowright\text{a}, \rightarrow a, \not\curvearrowright a, \lightning_c a, \lightning a\}$ 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, e.g., ${}^\bullet n \curvearrowright a$ stands for $({}^\bullet n, \curvearrowright\text{a})$.

Complex inter-process dependencies: The two source conditions $\dot{a}$ and $\dot{s}$ have not been considered yet because they define a period of time where the length of the period is unknown. The combination of one of these conditions with a single effect e would lead to an inter-process dependency which does not define precisely when e is triggered. However, the combination of two simple inter-process dependencies allows to express precisely a constraint that must hold in a target business process during the period of time defined by the source condition. We therefore refer to a dependency holding $\dot{n}$ as source condition as a *complex inter-process dependency*. The two simple inter-process dependencies i_1 and i_2 that build a complex inter-process dependency m must be defined in a specific way. If m holds source condition $\dot{n}$ then i_1 must hold a source condition that defines the start of $\dot{n}$ which is ${}^\bullet n$, and i_2 must hold a source condition that defines the end of $\dot{n}$, i.e., $n^\bullet$. Furthermore, the effects of i_1 and i_2 must be complementary. This means, if i_1 defines $\curvearrowright$a as effect, then i_2 must hold $\not\curvearrowright a$ as effect, or vice versa. A complex inter-process dependency cannot hold $\lightning_c a$ as an effect where a is an activity because there exist no complement effect to it. An activity that has completed cannot be undone without executing a compensation activity.

Local and external condition: The effect of an *intra-process dependency* results in a state change if a local condition holds. The effect of an *inter-process dependency* may depend on a local- and an external condition. The external condition is an activity mode that can be changed externally. Local and external conditions are differentiated for *enabled* and *disabled* but not for *triggered* and *cancelled*. The reason is that an activity can be in local and external *enable-* or *disable mode* independently, which does not cause a state change of an object. On the other hand, the activity can be triggered or cancelled either locally or externally because these effects induces an immediate state change. This leads to different conditions required by inter-process dependencies: (1) an activity can only be enabled externally if it is disabled externally, (2) an activity can only be triggered externally if it is enabled locally and externally, (3) an activity can only be disabled externally if it is enabled externally, and

(4) an activity can only be cancelled or (5) completed externally if it is running. There are different scenarios where the necessary pre-conditions do not hold and the effect should be triggered externally. However, handling these scenarios will not be investigated further in this paper but in future work. By default, activities are regarded as externally enabled. In case of an *enabling inter-process dependency*, the target activity must be disabled externally either during an initialisation or by another inter-process dependency beforehand.

Dependency properties: The effect of an *intra-process dependency* results in a different *activity mode* that holds till another event occurs, e.g, an activity stays in *enable mode* till it is triggered. The *inter-process dependencies* may model different behaviour. If an activity is enabled externally it might stay enabled externally even after it has been triggered. Three different effects can be identified: (1) a *continuous* effect results in an activity mode that holds after another event has occurred, and a *one-time* effect results in a mode that is only active until another event occurs. One-time effects are distinguished further in (2) *immediate* and (3) *waiting* effects. Immediate effects ensures that the required conditions must hold at the time the effect is triggered so the state- and mode change occurs immediately. Waiting effects wait till the required conditions hold and then cause a state- and mode change. A thorough discussion of these effects for each dependency goes beyond this paper and will be part of future work. The conducted assessment investigates the support of at least one kind of effect. The proposed language extensions in Section 4 will define dependencies with a continuous effect for enabling and disabling and a waiting effect for triggering and cancelling.

Table 1 contains all 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 business processes *source* and *target*, respectively, and s represents a state in *source*. The dependencies are grouped according to their effect on b.

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 (Dumas et al. 2005). A step towards standardisation was recently 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 paper, we use UML terminology if not stated otherwise. A comparison of terms can be found in the appendix of (Grossmann et al. 2007).

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 paper in Table 2. 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.

3.1 WorkFlow Nets

WorkFlow nets (WF-nets) are based on Petri nets and are used frequently in research for modelling business processes and their integration (van der Aalst 1998, 2000). 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 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 second group of dependencies involves externally enabling the target task ($\curvearrowright$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, i.e., b is disabled by default. Dependency ${}^\bullet a \curvearrowright b$ 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 $a^\bullet \curvearrowright b$ is modelled by changing a to an AND-split that places a resource token when it finishes execution. Dependency ${}^\bullet s \curvearrowright b$ 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 $s^\bullet \curvearrowright b$ 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 ${}^\bullet a \curvearrowright b$ and $s^\bullet \curvearrowright b$ is marked with "∼". The remaining dependencies need more than one additional element and are therefore marked with "(+)".

Triggering dependencies: The third 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

DEPENDENCY	EXAMPLE
Enabling dependencies ($\curvearrowright$b):	
${}^\bullet a \curvearrowright b$	When a sports competition starts (a), access to the team statistics (b) is enabled.
$\overset{\bullet}{a} \curvearrowright b$	During travel booking (a), the optional reservation of a car is enabled (b).
$a^\bullet \curvearrowright b$	When booking a travel (a) finishes, printing the booking (b) is enabled.
${}^\bullet s \curvearrowright b$	When an order enters state "all parts delivered" (s), reclaiming the order is enabled (b).
$\overset{\bullet}{s} \curvearrowright b$	While a student is in state "is PhD candidate" (s), applying for an internship (b) is enabled.
$s^\bullet \curvearrowright b$	When the temperature of cooling water leaves a critical state (s), the experiment is enabled to start again (b).
Triggering dependencies ($\rightarrow b$):	
${}^\bullet a \rightarrow b$	When the backup of all computers starts (a), the backup of a single computer (b) is invoked.
$\overset{\bullet}{a} \rightarrow b$	While a sensor measures values (a), a backup sensor is activated (b) for value comparison.
$a^\bullet \rightarrow b$	When a security officer finishes his shift (a), another security officer must start his shift (b).
${}^\bullet s \rightarrow b$	When an operating system enters state "logged off" (s), an external backup is started (b).
$\overset{\bullet}{s} \rightarrow b$	While a power plant is in state "in operation" (s), a monitoring system (b) is running.
$s^\bullet \rightarrow b$	When a order leaves state "in archive" (s), a notification service (b) is invoked.
Disabling dependencies ($\not\curvearrowright b$):	
${}^\bullet a \not\curvearrowright b$	When an inspection of a power plant starts (a), turning on the power plant (b) is disabled.
$\overset{\bullet}{a} \not\curvearrowright b$	During the booking of a bargain travel (a), the cancellation of the hotel (b) is disabled.
$a^\bullet \not\curvearrowright b$	When a payment process finishes successfully (a), cancelling the payment (b) is disabled.
${}^\bullet s \not\curvearrowright b$	When a hotel enters state "booked out" (s), booking a room is disabled (b).
$\overset{\bullet}{s} \not\curvearrowright b$	While a room resides in state "occupied" (s), the cleaning service of the room is disabled (b).
$s^\bullet \not\curvearrowright b$	When a book leaves state "in archive" (s), changing the delivery address of the book (b) is disabled.
Cancelling dependencies ($\lightning b$):	
${}^\bullet a \lightning b$	When an experiment with high priority starts (a), a running experiment with lower priority (b) is cancelled.
$\overset{\bullet}{a} \lightning b$	When a experiment with higher priority starts (a), an ongoing experiment with lower priority (b) is cancelled and restarted after a has finished execution.
$a^\bullet \lightning b$	When the supervisor leaves the experiment (a), a participating assistant must stop working (b).
${}^\bullet s \lightning b$	When the temperature of cooling water enters a critical state (s), the running experiment (b) is stopped.
$\overset{\bullet}{s} \lightning b$	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.
$s^\bullet \lightning b$	When a hotel leaves state "rooms available" (s), a currently running hotel booking (b) is cancelled.

Table 1: Examples of inter-process dependencies.

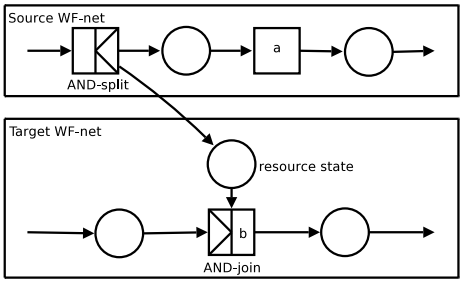

Figure 2: Dependency ${}^\bullet a \curvearrowright b$.

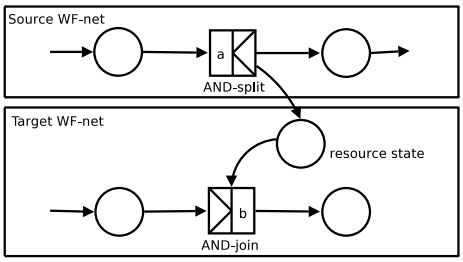

Figure 3: Dependency $a^\bullet \curvearrowright b$.

the same examples as shown in the previous group of dependencies. Hence, the same results are contained in Table 2.

Disabling dependencies: The first group of dependencies which we evaluate covers the external disabling of target task b ($\not\curvearrowright b$). Disabling b is modelled by an additional resource state connected to b that holds a token prior to run-time, i.e., 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 ${}^\bullet a \not\curvearrowright b$, it is consumed by b when it starts execution as shown in Figure 4. In dependency $a^\bullet \not\curvearrowright b$, 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 ${}^\bullet s \not\curvearrowright b$ 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 $s^\bullet \not\curvearrowright b$ is modelled in the same way as ${}^\bullet a \not\curvearrowright b$ (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 ${}^\bullet s \not\curvearrowright b$ 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, be-

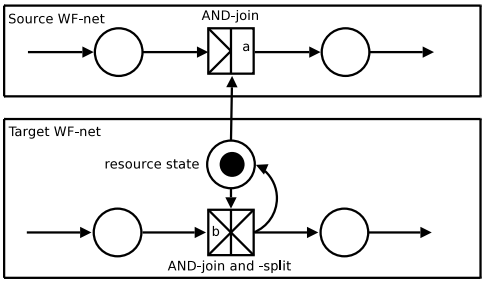

Figure 4: Dependency ${}^\bullet a \not\curvearrowright b$.

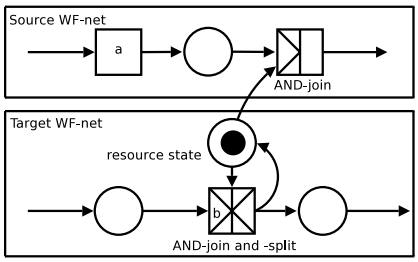

Figure 5: Dependency $a^\bullet \not\curvearrowright b$.

cause 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 $\not\downarrow b$, 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 $trigger_b$ 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 transaction *rollback* can be fired. The inhibitor arc will also be used later in Section 4 for defining the semantics of the language extensions. Table 2 marks all cancelling dependencies as unsupported by the WF-net specification because it does not support the extended Petri net semantics explained before.

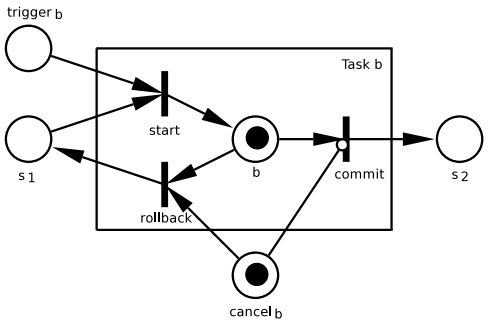

Figure 6: Cancelling task b modelled with an *inhibitor* arc from $cancel_b$ to *commit*.

3.2 Yet Another Workflow Language

Yet Another Workflow Language (YAWL) arose from the idea to support a set of control flow patterns[1] with moderate modelling effort and a formal foundation based on Petri net semantics. It consists of a set of extensions to WF-nets and YAWL workflows are also called *extended workflow nets* (EWF-nets). One such extension is the *remove token* functionality. It enables to model removing tokens from places so cancelling a task can be captured. This is achieved by 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.

Enabling and triggering dependencies: In contrast to WF-net, dependency ${}^\bullet a \curvearrowright b$ 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 ${}^\bullet a \curvearrowright b$ 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 ${}^\bullet a \not\curvearrowright b$ 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. These assumptions are necessary because it is not exactly defined when tokens are removed. The YAWL technical report states "...the moment the task executes all tokens in this area are removed." (cf., (van der Aalst & ter Hofstede 2002, p. 14)) but does not specify if they are removed at the beginning, at the end, or during the 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: An improvement com-

[1] http://www.workflowpatterns.com

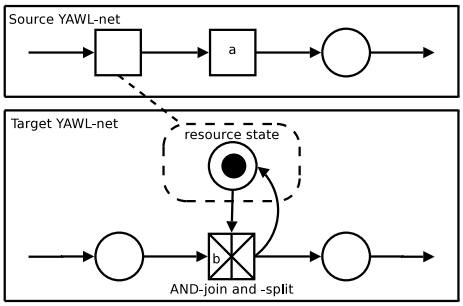

Figure 7: Dependency ${}^\bullet a \not\rightsquigarrow b$.

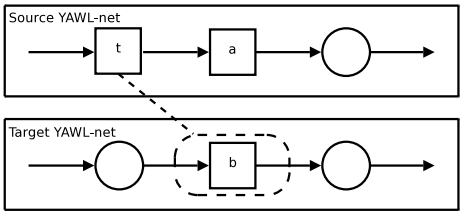

Figure 8: Dependency ${}^\bullet a \lightning b$.

pared to WF-nets is achieved by modelling cancelling dependencies due to the *remove token* functionality. However, it must be assumed that the token is either removed at the beginning or at the end of a task execution. Unfortunately, the technical report of YAWL does not define the remove token functionality precisely.

The dependency ${}^\bullet a \lightning b$ (depicted in Figure 8) assumes that a token is removed from b when a starts execution. The dependency $a^\bullet \lightning b$ is modelled in the same way, assuming that the token is removed when a completes execution.

The remaining dependencies ${}^\bullet s \lightning b$ and $s^\bullet \lightning b$ 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 ${}^\bullet s \lightning b$, it must be assumed that the token is removed when the task finishes execution. For $s^\bullet \lightning b$, it must be assumed that token removal happens at the end of the execution.

The modelling of dependency $\lightning b$ with the *remove token* functionality of YAWL may lead to undesired behaviour from an oo 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 & ter Hofstede 2002, p. 24)).

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[2] made significant changes especially to the semantics of UML Activity Diagrams by introducing Petri-net like semantics. (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) [3]). 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 (p. 380). However, object nodes belong to the data perspective in UML and can only be connected via object flows to activities (p. 376, 380). Actions cannot be connected to object nodes (p. 376), and all in- and outgoing edges of control nodes must be of the same type, i.e., either control flow or object flow (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* (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 $\lightning b$.

Since formal semantics for UML have not been widely adopted (Grossmann et al. 2005, Störrle 2004, Wohed et al. 2005), we have to make certain assumptions for modelling inter-process dependencies. One of these assumptions is that all control nodes, i.e., fork, join, decision, and merge, do not consume time. This would mean that all dependencies which are supported in some way were marked with "$\sim$" in Table 2. However, we have to make further assumptions for transmitting signals between processes. Therefore, we will ignore previously mentioned assumptions in the assessment of UML 2.0 Activities in Table 2 but mark dependencies with "$\sim$" which need further assumptions.

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 to send 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 (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

[2] During the writing of this paper, UML 2.1.1 was released but no significant changes between version 2.0 and 2.1.1 concerning the content of the paper have been identified.

[3] Page numbers in the remainder of this section refer to (OMG 2005).

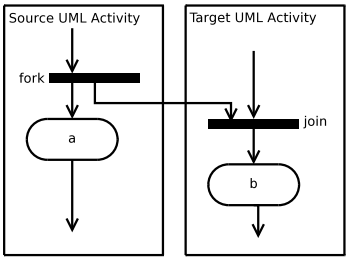

Figure 9: Dependency ${}^\bullet a \curvearrowright b$.

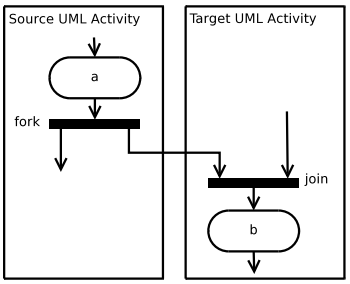

Figure 10: Dependency $a^\bullet \curvearrowright b$.

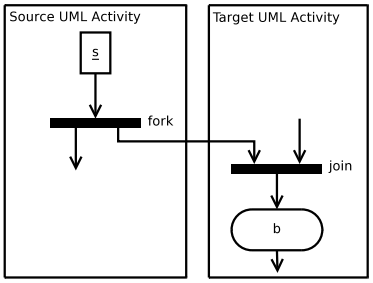

Figure 12: Dependency $s^\bullet \curvearrowright b$.

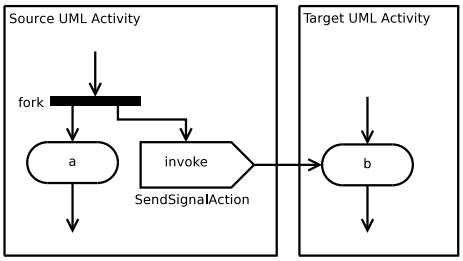

Figure 13: Dependency ${}^\bullet a \rightarrow b$.

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" (p. 394). Dependencies with an activity as source condition are modelled with a send signal action. Figure 13 depicts dependency ${}^\bullet a \rightarrow b$. The dependency $a^\bullet \rightarrow b$ can be modelled straight forward 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* (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, i.e., 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 an *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 (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 "$\sim$".

Cancelling dependencies: Dependencies with an activity as source condition can be captured by an interruptible activity region, as shown in Figure 14. For dependencies ${}^\bullet a \not\curvearrowright b$ and $a^\bullet \not\curvearrowright b$, 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.

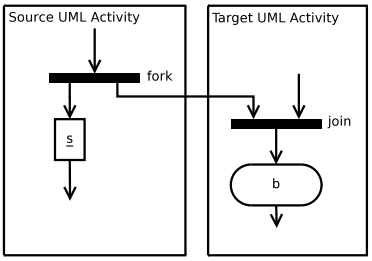

Figure 11: Dependency ${}^\bullet s \curvearrowright b$.

3.4 Business Process Modeling Notation

The Business Process Modeling Notation (BPMN) is a new standard for modelling business- and Web service processes. It was introduced by the Business Pro-

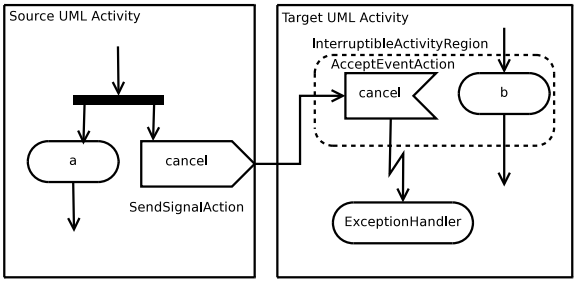

Figure 14: Dependency ${}^\bullet a \not\curvearrowright b$.

cess Management Initiative (BPMI) [4] and has been adopted by the Object Management Group (OMG). 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 provide *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 ${}^\bullet s$ and $s^\bullet$ of a state s can be captured by two intermediate events set in a sequence. The first event captures ${}^\bullet s$ and the second event $s^\bullet$. 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)[5]). A *send* task completes execution after sending a message (p. 65), and a *receive* task waits for a message to arrive and completes after receiving it (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 ${}^\bullet a \curvearrowright b$ is realised by setting $InMessage = "Enable"$ meaning that *enable* is sent when a starts execution as shown in Figure 15. In dependency $a^\bullet \curvearrowright b$, property $OutMessage = "Enable"$ is defined, meaning that *enable* is sent when a finishes execution. The dependencies ${}^\bullet s \curvearrowright b$ and $s^\bullet \curvearrowright b$ can be realised with two intermediate events in a sequence that represent ${}^\bullet s$ and $s^\bullet$, 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.

[4] http://www.bpmi.org

[5] Page numbers in the remainder of this section refer to (OMG 2006).

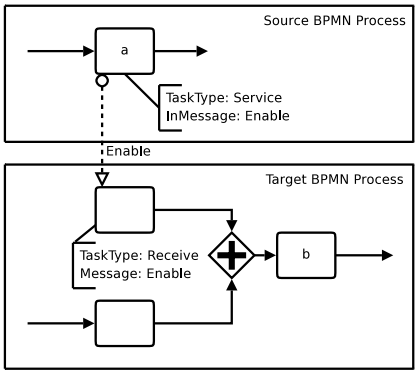

Figure 15: Dependency ${}^\bullet a \curvearrowright b$.

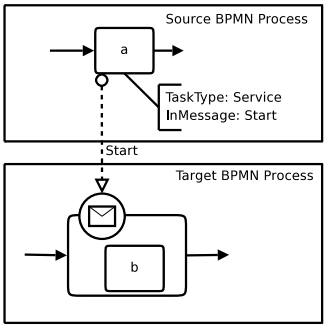

Figure 16: Dependency ${}^\bullet a \rightarrow b$.

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 the 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* messages 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)

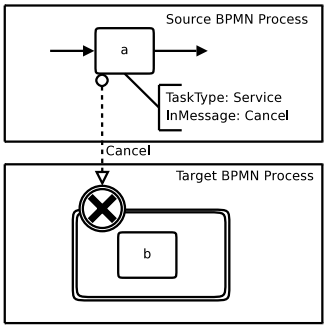

Figure 17: Dependency ${}^{\bullet}a \not\curvearrowright b$.

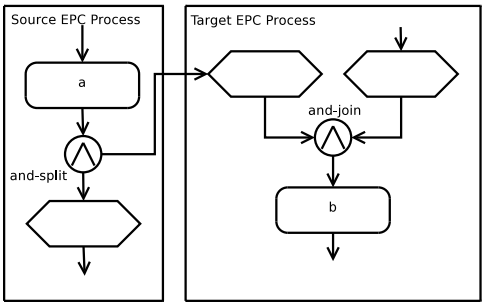

Figure 18: Dependency $a^{\bullet} \curvearrowright b$.

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 (Keller et al. 1992). The absence of formal semantics has led to several formalisation approaches, most of them Petri net related. We follow the approach of (van der Aalst 1999) for analysing inter-process dependencies in the EPC context. EPCs specify some syntactic constraints on how functions and events can be set in a sequence. On the path between two functions there must be an event, and on the path between two events there must be a function. These constraints are the reason that some dependencies cannot be modelled.

Enabling dependencies: In the group *enabling dependencies*, only ${}^{\bullet}a \curvearrowright b$ and ${}^{\bullet}s \curvearrowright b$ 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, e.g., in contrast to UML 2.0 Activities. We have chosen the formal definition of object life cycles (Schrefl & Stumptner 2002) as a foundation because life cycles allow to model business processes from an object-oriented point view. 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 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.

DEP.	WF-net	YAWL	UML 2.0 Activities	BPMN	EPC
Enabling dependencies ($\curvearrowright$b):					
${}^{\bullet}a \curvearrowright b$	$\sim$	(+)	(+)	(+)	−
$\dot{a} \curvearrowright b$	(+)	(+)	(+)	$\sim$	−
$a^{\bullet} \curvearrowright b$	(+)	(+)	(+)	(+)	(+)
${}^{\bullet}s \curvearrowright b$	(+)	(+)	(+)	(+)	(+)
$\dot{s} \curvearrowright b$	(+)	(+)	$\sim$	$\sim$	−
$s^{\bullet} \curvearrowright b$	$\sim$	$\sim$	(+)	(+)	−
Triggering dependencies ($\rightarrow b$):					
${}^{\bullet}a \rightarrow b$	$\sim$	(+)	(+)	(+)	−
$\dot{a} \rightarrow b$	(+)	(+)	$\sim$	(+)	−
$a^{\bullet} \rightarrow b$	(+)	(+)	(+)	(+)	(+)
${}^{\bullet}s \rightarrow b$	(+)	(+)	−	(+)	(+)
$\dot{s} \rightarrow b$	(+)	(+)	−	(+)	−
$s^{\bullet} \rightarrow b$	$\sim$	$\sim$	−	(+)	−
Disabling dependencies ($\not\curvearrowright b$):					
${}^{\bullet}a \not\curvearrowright b$	(+)	(+)	$\sim$	$\sim$	−
$\dot{a} \not\curvearrowright b$	(+)	(+)	$\sim$	$\sim$	−
$a^{\bullet} \not\curvearrowright b$	(+)	(+)	$\sim$	$\sim$	−
${}^{\bullet}s \not\curvearrowright b$	$\sim$	$\sim$	$\sim$	$\sim$	−
$\dot{s} \not\curvearrowright b$	$\sim$	$\sim$	$\sim$	$\sim$	−
$s^{\bullet} \not\curvearrowright b$	(+)	(+)	$\sim$	$\sim$	−
Cancelling dependencies ($\not\downarrow b$):					
${}^{\bullet}a \not\downarrow b$	−	$\sim$	$\sim$	(+)	−
$\dot{a} \not\downarrow b$	−	$\sim$	$\sim$	$\sim$	−
$a^{\bullet} \not\downarrow b$	−	$\sim$	$\sim$	(+)	−
${}^{\bullet}s \not\downarrow b$	−	$\sim$	−	(+)	−
$\dot{s} \not\downarrow b$	−	$\sim$	−	(+)	−
$s^{\bullet} \not\downarrow b$	−	$\sim$	−	(+)	−

Table 2: Comparison of modelling languages: + directly supported, (+) indirectly supported, $\sim$ 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. It 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, i.e., 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. The basic building block for modelling behaviour are activities and are adopted for the formalisation approach. 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/activity final*, 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 *α (initial node), where for no $t \in T : (t, \alpha) \in F$; α is the only state with this property. There is a nonempty set of distinguished states in S, the final states Ω (activity final nodes), where for no $s \in \Omega$ and no $t \in T$:$(s, t) \in F$ and the states in Ω 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 $t \in T$ *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

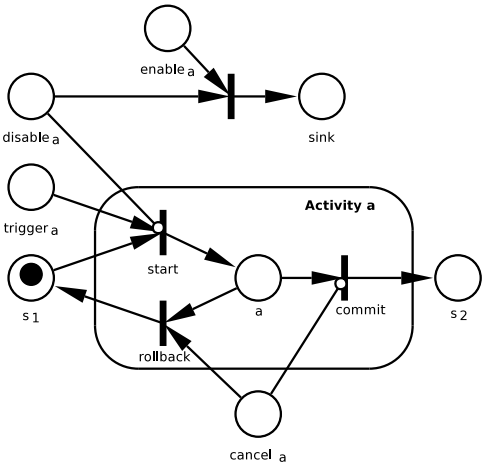

Figure 19: Low-level behaviour of activity a.

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 a 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) σ of an object type O is a subset of $S \cup T$. We denote the initial LCS $\{\alpha\}$ by A.*

The low-level semantics of an activity a are defined by extended Petri net semantics as shown in Figure 19. Places s_1 and s_2 represent pre- and post-states of a, respectively. Place s_1 enables transition *start* that represents the execution start of activity a. It fires if a token resides in places s_1 and $trigger_a$, and no token resides in place $disable_a$. The latter condition is defined by an *inhibitor* arc leading from $disable_a$ to *start*. The semantics of an inhibitor arc were explained in Section 3.1. After *start* has fired, a token is produced in place a which represents the *execution state* of activity a. If a token resides in a and no token resides in $cancel_a$ then the internal behaviour of activity a is not determined. Either transition *commit* fires, i.e., activity a finishes execution successfully, or transition *rollback* fires, i.e., activity a cancels execution. In case of a successful execution, a token is produced in post-state s_2, otherwise, a token is produced in the pre-state s_1. If a token resides in a and a token resides in $cancel_a$ then the behaviour is deterministic because the inhibitor arc connecting $cancel_a$ and *commit* gives priority to *rollback* over *commit*. Places $disable_a$ and $cancel_a$ will be used for modelling disabling and cancelling of a externally, as explained later. Place $enable_a$ will be used to enable a again, after it has been disabled. Place *sink* has the simple function to collect tokens that result from enabling an activity. Places $trigger_a$ and $cancel_a$ are accessible internally and externally whereas places $disable_a$ and $enable_a$ are only accessible externally.

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 $trigger_a$, $disable_a$, $enable_a$, and $cancel_a$ 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*, and *isCancelled* that hold a value of type *boolean*. Their default value is *false*. Place $enable_a$ is not implemented by an activity property but a function that sets $isDisabled = false$.

State: Definition 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, e.g., 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 the previous section. 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 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 *disable*, *enable*, *invoke*, and *cancel* links with an activity as source condition are defined by extended Petri net semantics and depicted by dotted arrows as shown on the right side of Figures 20–23. The dotted arrow emphasises the link semantics from the low-level behaviour of activities a and b. Activity a, the source of the inter-process dependency, is depicted only with its internal behaviour. Places, such as $trigger_a$, $disable_a$, $enable_a$, and $cancel_a$, are not significant for defining link semantics and left out. The semantics of links with a state as source condition require a more complex representation. The source condition ${}^\bullet s$ can only be captured by the *commit* transition of all activities which are directly connected to s, whereas $s^\bullet$ can only be caught by the *start* transition of all activities to which s is directly connected to. The depiction of the Petri net semantics can be found in the technical report of this paper (Grossmann et al. 2007).

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 phases ${}^\bullet a$, $\dot{a}$, and $a^\bullet$, 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 ${}^\bullet s$, $\dot{s}$, and $s^\bullet$, respectively. Examples for the UML notation of some link types are shown on the left side of Figures 20–23.

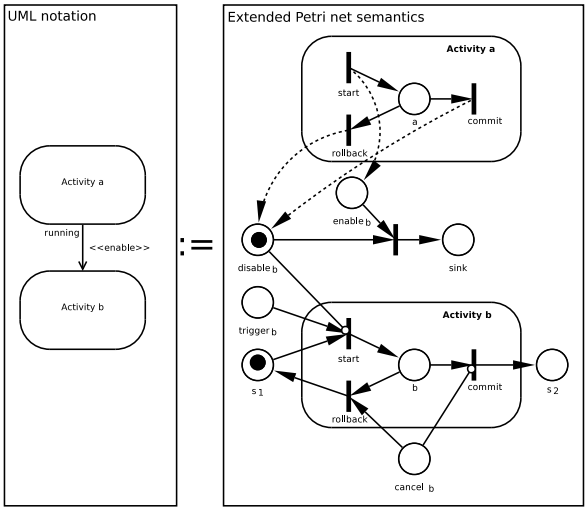

Figure 20: Semantics of an *enable* link with source condition $\dot{a}$: During the execution of a, b is enabled.

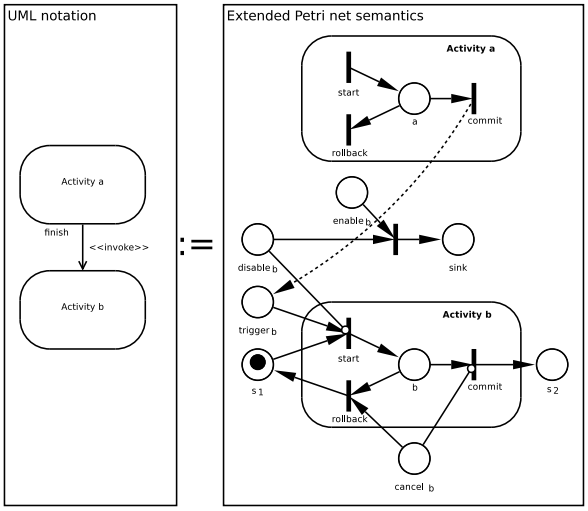

Figure 21: Semantics of an *invoke* link with source condition $a^\bullet$: When a finishes execution, b is triggered.

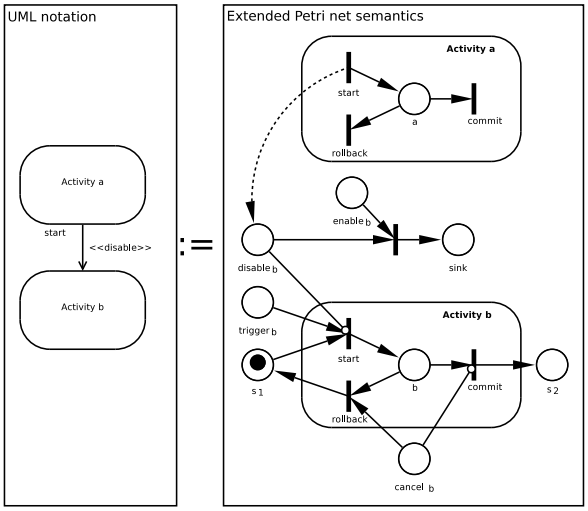

Figure 22: Semantics of a *disable* link with source condition ${}^\bullet a$: When a starts execution, b is disabled.

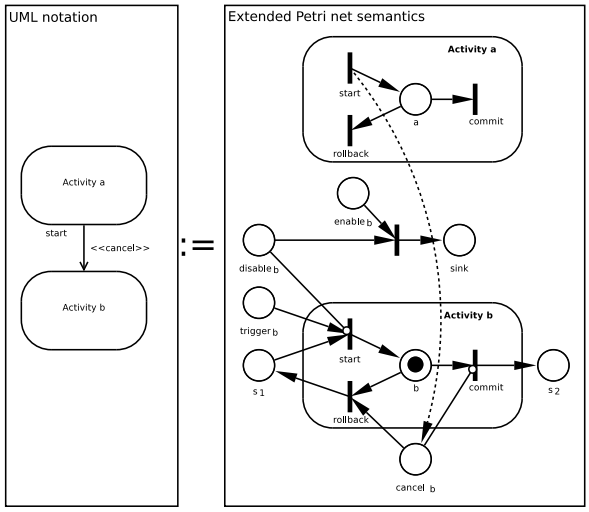

Figure 23: Semantics of a *cancel* link with source condition $^\bullet a$: When a starts execution, b is cancelled.

5 Related Work

A number of comparisons of business process languages has been conducted. Wohed et al. (Wohed et al. 2005, 2006) analysed UML 2.0 Activities and BPMN for the modelling support of workflow patterns. One category of the workflow patterns are control flow patterns which specify the order of activity flow within a workflow process. However, unlike the work presented here, Wohed et al. did not investigate cross-process dependencies. Söderström et al. (Söderström et al. 2002) developed a business process meta model that can be used for the comparison and the translation of modelling languages. Within the meta model, a concept called *location* is defined that specifies where a certain event takes place. This concept can be related to *source condition* in our work. Söderström et al. found that EPCs, UML 1.3 State Diagrams, and Business Modelling Language (BML) do not support this concept.

The interaction across boundaries of traditional business process management was investigated by Barros et al. (Barros et al. 2005). 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 interaction and inter-organisational workflows. Inter-organisational workflows have been proposed for inter-connection of business processes (van der Aalst 2000, Chebbi et al. 2006). Schulz et al. (Schulz & 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, e.g., a task is temporarily suspended or a task is created but was not started yet.

This paper focused on the identification of inter-process dependencies and the assessment of different modelling languages for their support. However, an evaluation of the identified dependencies is missing so far. One solution to this is proposed by Green and Petre (Green & Petre 1996). They developed the "Cognitive Dimensions Framework" for the evaluation of information-based artefacts and notations, which includes 14 dimensions that can be regarded as a set of criteria. For example, the dimensions include *viscosity*, *hidden dependencies*, and *abstraction level*. Viscosity targets the future changes of a system. Related to the presented approach, this dimension tackles possible changes after inter-process dependencies have been introduced, e.g., investigation of local changes and their impact on the global business processes, or changes made at the global business process and the impact on the execution of local processes. Due to the fact that presented inter-process dependencies are based on Petri net semantics, the impacts can be analysed by well-known verification techniques for Petri net based workflows (van der Aalst 1998). *Hidden dependencies* deal with relationships between two components such that one of them is dependent on the other, but the dependency is not fully visible. A set of inter-process dependencies might be introduced which cause a deadlock that is not obvious to the modeller. An example would be a circle dependency between three activities a, b, and c defined by $^\bullet a \curvearrowright b$, $^\bullet b \curvearrowright c$, and $^\bullet c \curvearrowright a$ which would cause a deadlock. Such incompatibilities of inter-process dependencies need to be identified in the future. The *abstraction level* deals with the number of high-level concepts and their acceptance by the user. This dimension was the main driving force behind the work presented here. It was shown that an abstraction of inter-process dependencies through so called *links* simplified the models considerably compared to current modelling languages. The inter-process dependencies were derived from well-known intra-process dependencies, so it can be assumed that their semantics are easy to understand by business process modellers. An in-depth analysis of these and remaining cognitive dimensions will be part of an evaluation in future work.

6 Conclusion

We have defined a set of *inter-process dependencies* between two conditions which were derived from *intra-process dependencies* and analysed commonly used business process modelling languages for their support. Since it could be shown that none of the presented languages support all dependencies directly, we propose a set of extensions for the investigated languages and demonstrated them at hand of UML 2.0 Activities. 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, an implementation of the extensions in a model checker for identifying incompatibilities between inter-process dependencies is planned.

Acknowledgements: We wish to thank Wolfgang Mayer and the anonymous reviewers for useful discussions and comments on an earlier version of this paper.

References

Barros, A., Dumas, M. & ter Hofstede, A. (2005), Service Interaction Patterns, *in* 'Proc. of BPM', LNCS 3649, Springer, pp. 302–318.

Chebbi, I., Dustdar, S. & Tata, S. (2006), 'The view-based approach to dynamic inter-organizational workflow cooperation', *DKE* **56**, 139–173.

Dumas, M., van der Aalst, W. & ter Hofstede, A. (2005), *Process-Aware Information Systems*, John Wiley & Sons.

Engels, G., Förster, A., Heckel, R. & Thøne, S. (2005), Process Modeling Using UML, *in* 'Process-Aware Information Systems (Dumas et al., eds.)', John Wiley & Sons, pp. 85–117.

Garcia-Solaco, M., Saltor, F. & Castellanos, M. (1995), *Semantic heterogeneity in multidatabase systems*, Prentice Hall, pp. 129–202.

Green, T. & Petre, M. (1996), 'Usability analysis of visual programming environments: a cognitive dimensions framework', *Journal of Visual Languages and Visual Computing* **7**, 131–174.

Grossmann, G., Ren, Y., Schrefl, M. & Stumptner, M. (2005), Behavior Based Integration of Composite Business Processes, *in* 'Proc. of BPM', LNCS 3649, pp. 186–204.

Grossmann, G., Schrefl, M. & Stumptner, M. (2007), Modelling Inter-Process Dependencies with High-Level Business Process Modelling Languages, Technical report, UniSA, ACRC. http://www.mrgg.at/documents/TR-UNISA-CIS-KSE-07-08.pdf.

Keller, G., Nüttgens, M. & Scheer, A. (1992), 'Semantische Processmodellierung auf der Grundlage Ereignisgesteuerter Processketten (EPK)', Veröffentlichungen des Instituts für Wirtschaftsinformatik. In German.

Koehler, J., Hauser, R., Sendall, S. & Wahler, M. (2005), 'Declarative techniques for model-driven business process integration', *IBM Systems Journal* **44**(1), 47–65.

Mühl, G., Fiege, L. & Pietzuch, P. R. (2006), *Distributed Event-Based Systems*, Springer.

OMG (2005), 'Unified Modeling Language: Superstructure 2.0'. formal/05-07-04.

OMG (2006), 'Business Process Modeling Notation Specification (BPMN)'. dtc/06-02-01.

Peterson, J. L. (1977), 'Petri Nets', *ACM CSUR* **9**(3), 223–252.

Schrefl, M. & Stumptner, M. (2002), 'Behavior-consistent Specialization of Object Life Cycles', *ACM TOSEM* **11**(1), 92–148.

Schulz, K. A. & Orlowska, M. E. (2004), 'Facilitating Cross-Organisational Workflows with a Workflow View Approach', *DKE* **51**(1), 109–147.

Söderström, E., Andersson, B., Johannesson, P., Perjons, E. & Wangler, B. (2002), Towards a Framework for Comparing Process Modelling Languages, *in* 'Proc. of CAiSE', Springer, pp. 600–611.

Störrle, H. (2004), Semantics of Control-Flow in UML 2.0 Activities, *in* 'Proc. of HCC 2004', IEEE Press, pp. 235–242.

van der Aalst, W. (1998), 'The Application of Petri Nets to Workflow Management', *Journal of Circuits, Systems, and Computers* **8**(1), 21–66.

van der Aalst, W. (1999), 'Formalization and Verification of Event-driven Process Chains', *Information and Software Technology* **41**(10), 639–650.

van der Aalst, W. (2000), 'Loosely Coupled Interorganizational Workflows: modeling and analyzing workflows crossing organizational boundaries', *Information and Management* **37**(2), 67–75.

van der Aalst, W. & ter Hofstede, A. (2002), YAWL: Yet Another Workflow Language, Technical report, Queensland University of Technology, Brisbane.

Wohed, P., van der Aalst, W., Dumas, M., ter Hofstede, A. & Russell, N. (2005), Pattern-based Analysis of the Control-flow Perspective of UML 2.0 Activity Diagrams, *in* 'Proc. of ER', LNCS 3716.

Wohed, P., van der Aalst, W., Dumas, M., ter Hofstede, A. & Russell, N. (2006), On the Suitability of BPMN for Business Process Modelling, *in* 'Proc. of BPM', LNCS 4102, Springer, pp. 161–176.