The original paper is published: J. Mangler, S. Rinderle-Ma: Rule-Based Synchronization of Process Activities. Int'l Conf on Electronic Commerce, pp. 121 - 128 (2011)
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6046963&tag=1

Rule-Based Synchronization of Process Activities

J. Mangler and S. Rinderle-Ma
University of Vienna, Faculty of Computer Science
Workflow Systems and Technologies Group
Vienna, Austria
Email: {juergen.mangler,stefanie.rinderle-ma}@univie.ac.at

***Abstract*—Synchronization of running process instances has been identified as major challenge in literature and practice. Although process instances are, for example, often required to share resources such as printers or centrifuges, the necessary instance synchronization is not supported by most process engines. While existing (scarcely supported) patterns deal with the intra-process synchronization of activities, a model for a more generic synchronization mechanism is still missing. The contribution of this paper is two-fold. (1) We introduce a generic model to describe the state transitions of process instances at runtime, and based on this model (2) define a subscription based event / voting mechanism that enables arbitrary synchronization within and between running instances. In order to demonstrate the validity of our approach, we will conduct an extensive evaluation against existing synchronization patterns, as well as describe a generic rule engine prototype that implements the presented approach.**

I. Introduction

One purpose of Process Aware Information Systems (PAIS), such as WebSphere Process Server [1] or AristaFlow [2], is to support their users processes, in tracking and governing the progress of running process instance. In order to do so, the processes are made explicit through a process description at design time. Processes are started, executed, and monitored by a process engine based on this process description. Typical processes consist of a set of activities that rely on resources (human or machine) in order accomplish a certain process goal. As a process is basically a computer program describing a certain control flow logic, it possible to implement a very fine-grained model of how certain resources are used. Consider, for example, the following scenario: In a company several departments share a printer. The printer is specialized in large, printing jobs (e.g. folders or enveloped letters) which involves the retrofitting with special paper / envelopes. It is not efficient to always retrofit the printer with a different kind of paper each time a print job arrives because this wastes time and effort and leads to errors. As the departments already use a process management system to support their work, it is important to efficiently integrate the printing activity into these processes in to reduce waste.

Planning the concurrent use of a single activity (in our case the printer) is describe by a special group of process patterns by van der Aalst et al. [3]: **Multiple Instance Patterns**.

Multiple instance patterns are defined to *"describe situations where there are multiple threads of execution active in a process model which relate to the same activity"*. These situations may occur when either (a) an activity is able to invoke multiple instances of itself, (b) an activity is invoked as part of a loop that spawns concurrent execution threads, or (c) when two activities share the same implementation, and are concurrently invoked.

One problem with this kind of planning for a concurrent use of an activity is however: it demands to handle the planning from the point of view of the printer. The printer is retrofitted (one process instance is created) and then works on queued incoming jobs. Designing process on the basis of solving resource problems instead of designing them in order to describe business processes is arguably not good. For example consider the following additional scenarios (similar to [4]):

- Instead of the departments just contributing print jobs to a queue as part of the printing process, they could spawn their own printing process instances. In this case the printing activities from different process instances are overlapping and thus need to be synchronized.
- Different departments use the printer as part of their own custom processes. In this case printing activities from instances of different processes need to be synchronized.

The focus of this paper is to tackle the above mentioned scenarios by introducing a generic process / instance synchronization mechanism, that covers "Multiple Instance Patterns" for concurrent invocation of activities (a) within one instance of one process as described by van der Aalst et al., (b) between instances of one process, and (c) between instances of arbitrary processes.

The two main contributions of this paper are:

Extended Activity Lifecycle: As we want to avoid to tie the synchronization logic to some specific process engine properties or implementation, we define a simple extension of the common activity lifecycle, which is the base for all process engine implementations. Relying solely on the activity states and state changing events allows us to define a minimal but comprehensive mechanism that can easily be embraced by existing process engines.

Independent Rule Based Synchronization: "Multiple Instance Patterns" as defined by van der Aalst et al. outline not only simple scenarios that occur exactly as described, but actually refer to a class of scenarios that can include more complex behavior. For the printer example an intelligent job queue may be implemented, that not only deals with already submitted jobs, but predicts jobs that

may be submitted soon according to process instance progress. In order to make the synchronization as flexible and independent from the process engine as possible, we propose an ECA (Event Condition Action) rule based synchronization. In this paper we present an extensive evaluation of a rule based synchronization according to the above mentioned patterns.

The paper is structured as follows: in Sect. II we start by provide and discussing an extensive example scenario. In Sect. III we present a state-based interaction model for monitoring and manipulating process execution. In Sect. IV we present define the necessary vocabulary and concepts for rule based synchronization. In Sect. V we present a set of patterns to allow for synchronization of processes with parameters devised at design time or runtime. In Sect. VI we give a short presentation of our prototype and its connection to a process engine. Finally, we discuss related work in Sect. VII and present our conclusions in Sect. VIII.

II. A Use-Case for Process Instance Synchronization

Many businesses rely on lean manufacturing systems in order to operate effectively. In lean management the elimination of waste is an important goal [5]. Waste denotes e.g. unnecessary movement of materials, excess inventory not directly required for current manufacturing jobs, or extra steps taken by employees due to ever-changing manufacturing jobs.

The elimination of waste by supporting efficient use of resources is also a primary goal of process instance synchronization. Consider, for example, a company with an in-house printing facility, that prints thousands of fliers, letters and posters a day. To minimize waste as defined above, the company wants to introduce the following rules:

There are three print queues for letters, fliers, and posters. The activity reflecting the print job constitutes the synchronization point, i.e., the activity where the different print queues are to be synchronized. Empirical data shows that there is a certain amount of serial letters every morning. Therefore the employees have to prepare letter production first. When the other queues reach a certain fill level, the letter production is stopped and the machinery is prepared for flier or poster production. This process is repeated until the end of the day, under the premise that each form is at least processed once.

The printing facility operates independently of other facilities. Processes of other facilities are to be kept as simple as possible. It is possible to send a request-for-demand and later the actual print job. In order to avoid a massive spooling server the actual print-jobs are sent as late as possible.

By analyzing this use-case we found different properties:

1) Several independent business entities exist.
2) Each business entity has its own processes.
3) In order to enable a simple printing activity to be incorporated in existing processes we can utilize a simple rule-based synchronization.

In the following we want to justify the third statement above: In a process environment it is advisable to rely on rules when, first of all, a certain interrelationship spawns multiple processes. Traditionally this is solved by introducing additional activities in conjunction with triggers and mutexes, which tend to slightly clutter the process description. This makes the processes harder to understand and maintain and therefore is a source for hidden errors. Additionally when a certain interrelationship is subject to sporadic change and or involves significant conditional branching or decision-making a rule-based approach is advisable. Thus, for synchronizing process instances we opt for a rule-based approach.

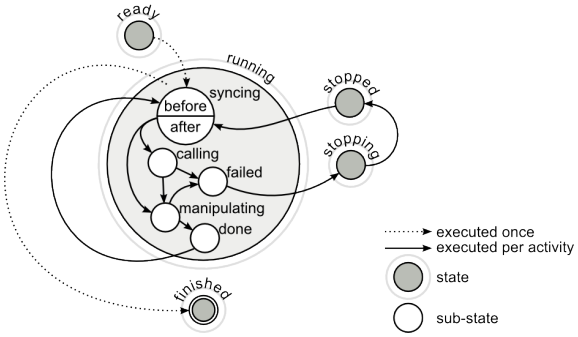


Fig. 1. Generic Process Engine State Chart

III. Observable Lifecycle of Instances and Activities

Before discussing synchronization it is important to deal with the lifecycle of instances and activities. The notion of an activity lifecycle is covered in a wide range of publications, ranging from discussions about BPEL implementations [6] and exception handling [7], to API descriptions [8].

All models more or less share three basic states for an activity: *ready*, *processing*, *finished*. These states are then extended by additional states for error and recovery (like, e.g., retrying, failed) [9]. As the goal of this paper is to allow for synchronization by an external synchronization engine (see Section VI), it is crucial describe a fine grained state model (as well as state changes) observable from the outside as a series of events.

In order to further streamline the model, we differentiate between states that are related to process instance, and states that are related to the execution of activities. For this paper we explicitly exclude states that deal with exception handling or repair.

A. Instance Level States

We differentiate between the following states (see Fig. 1):

ready: The initial state after a client requests a new instance. It is possible to set the context, endpoints and process description.

running: Denotes that an activity is executed. The progress of this execution is reflected by a series of states for each activity (i.e. Activity Level States).

stopped: Whenever an error occurs (e.g., connection problems to services that represent activities), an exception

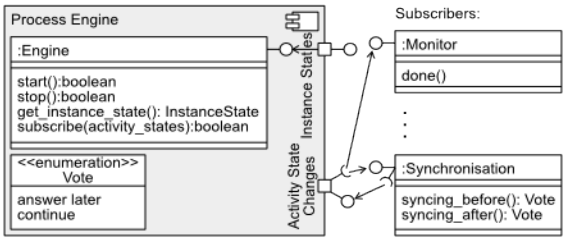


Fig. 2. Interaction between Synchronization and Process Engine

may be raised that puts the process engine into the state *stopped*.

finished: This is the final state of a process instance. Information like the context, endpoints or process description is read-only.

B. Activity Level States

While *running*, the process engine goes through a series of states **per activity**. As multiple activities may be executed in parallel, this states are not observable from the outside, but instead they are communicated as a series of events:

calling: Active during a connection to an external service that implements an activity. It occurs when actual services are called.

failed: Whenever an error occurs a failed state for this specific activity is communicated and the process instance state is set to *stopped*.

manipulating: Active while the result of a call to an external service is merged into the context. It also occurs for activities that only manipulate context.

done: Occurs for all activities after they are committed. This also marks the point where changes to the context are observable from the outside.

The above mentioned states are in a similar form to be found in all process engine. Especially for synchronization we include the following two states:

syncing: Can be used by to decide if the control flow can continue or if the engine has to be stopped (in case of a synchronization error). The idea is to keep activities in the *syncing* state, in or order to wait for activities in parallel branches of the same process instance or activities in different process instances. *Syncing before* is a state every activity holds before state *calling*. *Syncing after* is a state every activity holds after state *done*.

C. Utilizing States

In order to utilize the states we assume that states and state-changes are communicated to subscribers (see Fig. 2). Subscribers are modules or external services such as monitors or a synchronization service, that are interested in a particular process instance execution. These subscribers can *subscribe()* to "Activity Level States" changes, such as *syncing*.

We further assume that subscribers can interact with process instances by:

- Triggering "Instance Level States" *running* and *stopped* by calling *start()* and *stop()*.
- By voting for **"Activity Level States: syncing before and after"**. We define voting as allowing an activity to progress to the next change. Voting also includes the possibility to prohibit this state change (*answer later*) by delaying the vote and thus delaying execution.

IV. Rule-Based Synchronization

As laid out in Sect. III we will rely on some simple foundations for implementing synchronization. First of all, external services can subscribe to process instances. A running process instance communicates its progress through a series of states changes pushed to subscribers . A subscriber may *stop* a process instance as well as *continue*, or *delay* the execution of a process instance. For a stopped process instance, a subscriber may furthermore *skip* activities by changing the control flow pointer as well as *restart* the process instance.

As the logic for synchronizing activities spawns multiple instances we decided for a declarative (rule-based) approach. As mentioned before this allows for a great degree of flexibility in comparison to static implementation of patterns.

A. Rules Syntax

For our simple rule engine we rely on Event Condition Action (ECA) rules [10], which incorporate many familiar concepts found in many information systems:

We define a set of events $E = \{e_1, \ldots, e_n\}$ where each event e_i has a set of attributes: $e_i \leftarrow \{x_1, \ldots, x_n\}$. Events are for example triggered by state changes received from running process instances.

As a rule engine also needs to have an internal state (i.e. variables) in order to e.g. only trigger an action after a minimum number of events occurred, we introduce the notion of contexts $C = \{c_1, \ldots, c_n\}$. Each context c_i consists of one or more variables: $c_i \leftarrow \{v_1, \ldots, v_n\}$. The context may be changed in the *action* part of the rule. A single context variable may be directly referred as $c_i^{v_j}$. The purpose of contexts is to group the variables together, that hold the state of the rule engine. We define a change of a variable inside a group to also trigger an event, that can potentially be processed by a rule.

Finally we define action calls $A = \{a_1 \ldots a_n\}$. In addition to modifying the context one or more calls a_i may be used in the action part of the rule. These calls represent the invocation of custom external functionality, in our case to control a process instance.

A rule is triggered either by an event e_i or by modifying a context c_i (changing one or more context variables v_i). The condition part consists of a set of propositions $P = \{p_1, \ldots, p_n\}$ about event attributes x_n and/or context variables $c_i^{v_j}$ (e.g. to test if an event attribute matches a certain value). The condition part of a rule triggered by an event may include propositions about contexts.

TABLE I
Rule Quick Reference

Events	$E =$	$\{e_1, \ldots, e_k\}$
Attributes		$e_i \leftarrow \{x_1, \ldots, x_l\}$
Single Attribute		$e_i^{x_j}$
Contexts	$C =$	$\{c_1, \ldots, c_m\}$
Variables		$c_i \leftarrow \{v_1, \ldots, v_n\}$
Single Variable		$c_i^{v_j}$
Conditions (Propositions)	$P =$	$\{p_1, \ldots, p_p\}$, check E and C
Shortcut Values	$S =$	$\{s_{x_i}^1, \ldots, s_{x_i}^r\}$
Named Shortcut Value		$s_{x_i}^{name} = value$
Actions	$A =$	$\{a_1 \ldots a_o\}$, modify C

As a rule is potentially triggered multiple times, we will use the $\forall$ quantifier and henceforth denote rules:

$$\begin{aligned}\forall\langle e_y, c_{z_1}, \ldots, c_{z_q}\rangle \bullet\ & \exists e_y \wedge e_y \leftarrow \{x_1, \ldots, x_n\} \wedge e_y \in E \wedge \\ & \exists c_{z_1} \wedge \ldots \wedge \exists c_{z_q} \wedge \\ & c_{z_1} \leftarrow \{v_{z_1}^1, \ldots, v_{z_1}^n\} \ \wedge \ldots \wedge \\ & c_{z_q} \leftarrow \{v_{z_q}^1, \ldots, v_{z_q}^n\} \wedge \\ & c_{z_1} \in C \ \wedge \ldots \wedge c_{z_q} \in C \wedge \\ & p_1 \wedge \ldots \wedge p_n \\ \Rightarrow\ & a_1, \ldots, a_o\end{aligned}$$

In this context we use e_y to denote an existing event e_i identified by y. The same semantic is used in conjunction with $c_{z_1}, \ldots, c_{z_q}$. To make this more readable we will from now on use the following equivalent notion:

$$\begin{aligned}\forall\langle e_i, c_1, \ldots, c_n\rangle : p_1, \ldots, p_p \bullet \\ a_1, \ldots, a_o\end{aligned}$$

where all expressions related to the naming and existence-checking of events, contexts and variables are left out. Furthermore the propositions and actions are visually separated with a $\bullet$. These can be read as

$$\forall\langle \text{Event}\rangle : \text{Condition} \bullet \text{Action}$$

with an event being either triggered from outside (e_i) or triggered through changing a context c_i in any other rule.

In the propositions the event attributes have to be compared to actual values and these values may clutter the rules. Therefore we introduce a set of shortcut values $S = \{s_{x_i}^1, \ldots, s_{x_i}^r\}$. Values are bound to event attributes x_i, and associated with unique symbolic identifiers $1 \ldots r$: e.g., $s_{x_i}^{name} = val$ denotes that the value *val* for attribute x_i can be identified by $name$.

In case one or more values $s_{x_i}^{name}$ exist, each rule with an event that includes x_i can (1) use each associated *id* as a shortcut (to simplify notation), and (2) is not invoked if a value other than defined ones is pushed: the values also act as a filter, to reduce the number of propositions.

B. Prerequisite Vocabulary for Rule-Based Synchronization

In the following sections we want to apply our rule-based approach to synchronize process instances. Prior to this we have to define the following vocabulary that is shared for several synchronization patterns we present. Synchronization is possible based on only two synchronization events due to the fact that we synchronize either before or after execution of a rendezvous point.

$$e_{before} \leftarrow \{\text{instance}, \text{position}, \text{endpoint}\}$$
$$e_{after} \leftarrow \{\text{instance}, \text{position}, \text{endpoint}\}$$

These events are triggered before and after an activity is called, and may be used to delay the execution of an activity or the execution of the next activity by means of voting, as described in Sect. III. Both events share the same attributes. Further, we can rely on a simple set of action calls A, derived from the state-based model. Altogether:

- $x_{\textbf{instance}}$ is the process instance that pushed the event.
- $x_{\textbf{position}}$ is the unique id of the activity in the process description of a certain instance. When an activity is called several times in a loop, the position is the same for each iteration. It is persistent not across process instances.
- $x_{\textbf{endpoint}}$ is the unique identifier of an activity. It is persistent across process instances.
- $a_{\textbf{continue}}$ continues the control flow for a given activity and instance.
- $a_{\textbf{skip}}$ skips an activity in a given instance. This requires a series of steps: stop process instance, change position information of a given activity to *"after"* (see Sect. III), restart process instance.

A concrete example for a rule as described above is

$$\forall\langle e_{before}\rangle : e_{before}^{endpoint} = \text{http://someuri} \ \bullet \ a_{continue}(x)$$

For the attribute values of an event e_{before}

$$x_{instance} = \text{Instance 1}$$
$$x_{position} = \text{Activity C}$$
$$x_{endpoint} = \text{http://someuri}$$

this rule triggers the corresponding process engine (in syncing state) to continue "**Activity C**" of "**Instance 1**". In the above example only x_{endpoint} is part of the condition of the rule, but not x_{instance} and x_{position}. This results from the fact that the rule is intended to fire for all possible values of x_{instance} (e.g. *Instance 1*) and x_{position} (e.g. *Activity C*).

V. Rule-Based Synchronization Patterns

In this section we specify the rules necessary to synchronize on activities.[3] and evaluate our approach by applying them to the scenario presented in Sect. II:

P34: Static Partial Join for Multiple Instances.
P35: Canceling Partial Join for Multiple Instances.
P36: Dynamic Partial Join for Multiple Instances.

We will show that our simple method covers synchronization of multiple instances of an activity used in different processes or process instances[1].

For easier comprehension of the patterns described in the next subsections, please use:

[1]As our approach is based on events, it is also possible to synchronize on activities inside a single process instance.

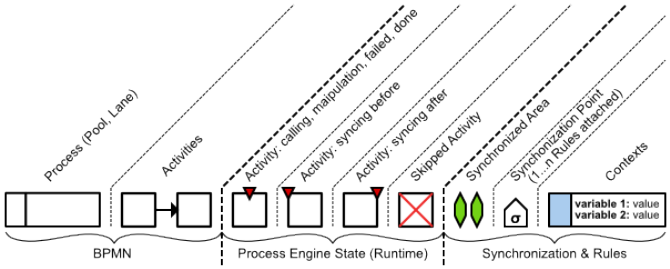


Fig. 3. Graphical Notation to Describe Synchronization

- The Rule Quick Reference Table I.
- Fig. 3 shows our BPMN based graphical notation to describe synchronizations. This extension is necessary, because we have to describe synchronization between processes or process instances, as well as describe process instance snapshots that represent different times during execution (runtime).

A. P34 - Static Partial Join:

In our print queue use-case (cf. Sect. II) patterns P34 (Fig. 4(a)) and P35 (Fig. 4(b)) occur e.g. during the preparation of the printing facility. When the facility is put into configuration mode, three independent test teams have to prepare the facility, extract test data, and file a test report. When the configuration is done once, the teams look at different aspects of the test data yielded by the test mode. Whenever two teams successfully complete the configuration and testing, the printing facility is prepared for production.

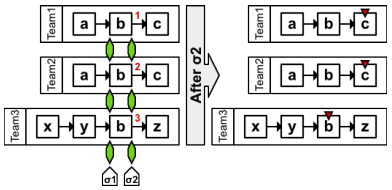
Individual teams (independent processes or branches) are represented by swimlanes. They all have an activity b to gather data from the test run. When all three teams are ready to start b, the test-run can be started. Whenever two teams finish gathering data through b, the teams are allowed to move on and produce a summary. The remaining team continues with the data gathering, although the result has no consequences. All activities b have to be finished before a new configuration run.

To realize this behavior we define the following values and contexts as a basis for the rules:

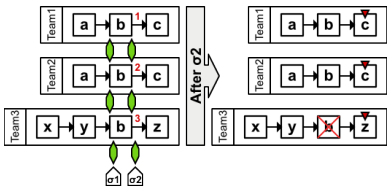
$$s^{b}_{endpoint} = \text{http://b} \qquad c_{\text{wait}} \leftarrow \{\text{buf} : \text{Array}\}$$
$$c_{\text{sync}} \leftarrow \{\text{run} : \text{Array}, \text{fin} : \text{Array}\}$$

We defined two synchronization points: We use σ_1 to group all rules that are hooked to e_{before} and σ_2 to group all rules that are hooked to e_{after}.

To ease the understanding of the process we will use the test scenario depicted in Fig. 5. Above the activities small triangles denote the progress. A triangle at the left upper or right upper corner means, that the activity waits for a continue, either before or after its execution. A triangle in the middle of an activity means that it is currently executed. More than one



(a) P34: Static Partial Join



(b) P35: Canceling Partial Join

Fig. 4. Static & Canceling Partial Join for Multiple Instances

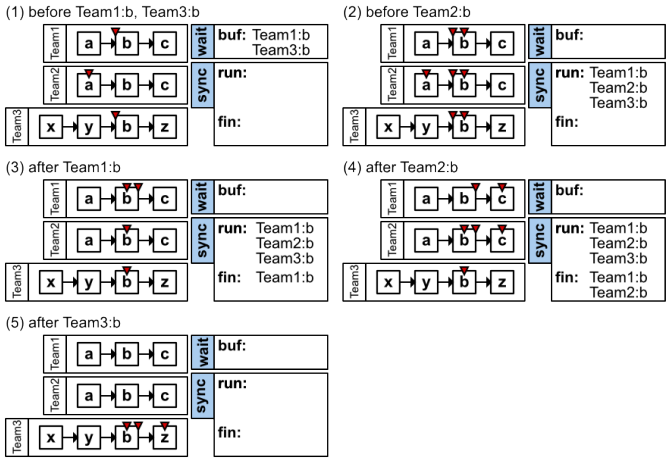


Fig. 5. P34 Example Scenario

triangle per swimlane shows the progress made by the events in the caption (e.g. after p1:b).

$$\forall \langle e_{before} \rangle : \varnothing \bullet c^{buf}_{wait}.append\left(e^{instance}_{before}, e^{position}_{before}\right) \quad \text{(P34-}\sigma\text{1-a)}$$

$$\forall \langle c_{wait} \rangle : c^{buf}_{wait}.length \geqslant 3 \bullet$$
$$c^{run}_{sync} := first\ 3\ from\ c^{buf}_{wait},$$
$$foreach\ c^{run}_{sync} as\ t\ \{a_{continue}\,(t_{instance}, t_{position})\}$$
(P34-σ1-b)

Rule P34-σ1-a fires when e_{before} occurs. In Fig. 5-(1) it fires two times, when team 1 and team 3 are ready to start the configuration run.

Whenever P34-σ1-a fires, the activity, identified by the attributes **instance** and **position**, is added to c^{buf}_{wait}. There is **no condition**, as through the introduction of a shortcut value $s^{a}_{endpoint}$, rules implicitly only match rules when the given endpoint is matching the value in $s^{b}_{endpoint}$.

In Fig. 5-(2) rule P34-σ1-a fires again, leading to three entries in c^{buf}_{wait}. This context change triggers rule P34-σ1-b.

The consequence is, that three activities are moved to c^{run}_{sync} and $a_{continue}$ is called for each of them. While c^{run}_{sync} is not empty, e_{after} activities are saved in c^{buf}_{wait} (rule P34-σ1-a) but it is not possible that rule P34-σ1-b is triggered.

$$\forall \langle e_{after} \rangle : \varnothing \bullet c^{fin}_{sync}.append\left(e^{instance}_{after}, e^{position}_{after}\right) \quad \text{(P34-}\sigma\text{2-a)}$$

$$\forall \langle c_{sync} \rangle : c^{fin}_{sync}.length = 2 \bullet \\ foreach\ c^{fin}_{sync}\ as\ t\ \{a_{continue}\,(t_{instance}, t_{position})\} \quad \text{(P34-}\sigma\text{2-b)}$$

$$\forall \langle c_{sync} \rangle : c^{fin}_{sync}.length > 2 \bullet \\ a_{continue}\left(c^{fin}_{sync}.last_{instance}, c^{fin}_{sync}.last_{position}\right) \quad \text{(P34-}\sigma\text{2-c)}$$

$$\forall \langle c_{sync} \rangle : c^{fin}_{sync}.length = c^{run}_{sync}.length \bullet \\ c^{fin}_{sync} := Array, c^{run}_{sync} := Array \quad \text{(P34-}\sigma\text{2-d)}$$

Rule P34-σ2-a fires when e_{after} occurs, as seen in Fig. 5-(3). This happens when one team successfully completes the configuration run. As a consequence the activity is copied to c^{fin}_{sync}. Rule P34-σ2-a fires again for Fig. 5-(4), when the second team completes the configuration run. This also leads to the firing of rule P34-σ2-b, because two teams finishing the configuration run means that the production run is possible. The consequence is that these two teams can continue with the last activity in their process, the submission of a test report.

The third team continues gathering data from the configuration run, although the findings have no consequences. When the team is finished, as seen in Fig. 5-(5), the overall configuration run is deemed over. This triggers rule P34-σ2-c, therefore team 3 can immediately continue with filing the report. As soon as all activities are finished, also rule P34-σ2-c is triggered, c^{run}_{sync} and c^{fin}_{sync} are set to empty. Therefore P34-σ1-b may trigger whenever three activities are available.

B. P35: Canceling Partial Join

This join is very similar to P34. The buffering of e_{before} events and the triggering, denoted by σ_1 is the same. This means that rules P34-σ1-a and P34-σ1-b are reused.

$$\forall \langle e_{after} \rangle : \varnothing \bullet c^{fin}_{sync}.append\left(e^{instance}_{after}, e^{position}_{after}\right), \\ c^{run}_{sync}.delete\left(e^{instance}_{after}, e^{position}_{after}\right) \quad \text{(P35-}\sigma\text{2-a)}$$

$$\forall \langle c_{sync} \rangle : c^{fin}_{sync}.length = 2 \bullet \\ foreach\ c^{fin}_{sync}\ as\ t\ \{a_{continue}\,(t_{instance}, t_{position})\} \\ foreach\ c^{run}_{sync}\ as\ t\ \{a_{skip}\,(t_{instance}, t_{position})\} \\ c^{fin}_{sync} := Array, c^{run}_{sync} := Array \quad \text{(P35-}\sigma\text{2-b)}$$

The rules for σ_2 are simpler for P35. Again, when one team successfully completes the configuration run, e_{after} occurs and therefore rule P35-σ2-a fires. As a consequence the activity is copied to c^{fin}_{sync}, but additionally it is removed from c^{run}_{sync}.

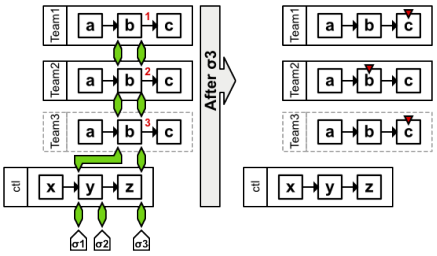


Fig. 6. P36: Dynamic Partial Join for Multiple Instances

Rule P35-σ2-b is triggered when two teams successfully gather the test data. This means that the configuration run is finished (all remaining activities $s^{b}_{endpoint}$ are skipped) and all three teams file their report. This is done by calling $a_{continue}$ for all activities in c^{fin}_{sync}, and then calling a_{skip} for all activities in c^{run}_{sync}. Finally c^{run}_{sync} and c^{fin}_{sync} are set to empty.

C. P36: Dynamic Partial Join

While P34 and P35 are design time patterns, with a fixed number of activities to start or end synchronization, P36 is a runtime pattern. Again we interpret this pattern not in the context of a single instance, but we want to allow it to synchronize on multiple independent process instances.

By applying the pattern to our example we can come up with the following scenario (see Fig. 6): The printing facility, consisting of several subsystems, has sustained damage that has to be repaired. To keep the downtime minimal, several teams work in parallel, on different issues. Initially two teams are dispatched. During the process it may be necessary to dispatch additional teams to work on new problems. When a number of teams finish reparation of critical subsystems, the printing facility may return to operational status, although other teams may be still busy working on minor subsystems. The supervision of the repair is to be done by a control team.

As in P34 we assume, that some teams may finish their work, without consequences for the operation. For the example depicted in Fig. 6 we assume activity a to represent the assembly of the team and necessary resources, activity b represents the repair act, yielding standardized repair protocols, whereas c represents a detailed report. To realize P36 we need the following vocabulary:

$$s^{syn}_{endpoint} = \text{http://b} \quad c_{wait} \leftarrow \{\text{buf} : \text{Array}\}$$
$$s^{ctla}_{endpoint} = \text{http://x} \quad c_{sync} \leftarrow \{\text{run} : \text{Array}, \text{fin} : \text{Array}\}$$
$$s^{ctld}_{endpoint} = \text{http://y} \quad c_{ctl} \leftarrow \{\text{buf} : \text{Array}, \text{add} : \text{false}, \text{done} : \text{true}\}$$

Rules P36-σ1-a and P36-σ1-b realize queues for activities $s^{syn}_{endpoint}$ and $s^{ctla}_{endpoint}$, meaning that a repair team or control team is ready.

$$\forall \langle e_{before} \rangle : e^{endpoint}_{before} = \text{syn} \wedge c^{add}_{ctl} = \text{false} \bullet \\ c^{buf}_{wait}.append\left(e^{instance}_{before}, e^{position}_{before}\right) \quad \text{(P36-}\sigma\text{1-a)}$$

$$\forall \langle e_{before} \rangle : e_{before}^{endpoint} = \text{ctla} \bullet \\ c_{ctl}^{buf}.append\left(e_{before}^{instance}, e_{before}^{position}\right) \quad \text{(P36-}\sigma\text{1-b)}$$

Rule P36-σ1-c describes that as soon as a control team is ready and no other repair processes are running (c_{sync}^{run} is empty), the repair may start. For this purpose the context c_{ctl} holds two variables: *add* tells if additional teams are still requested to join the repair effort, *done* tells if the printing facility is repaired to a point where it can be restarted again. These variables are set to initial values, then all waiting repair teams are dispatched.

$$\forall \langle c_{sync}, c_{ctl} \rangle : c_{sync}^{run}.empty? \wedge c_{ctl}^{buf} > 0 \bullet \\ c_{ctl}^{done} := false, c_{ctl}^{add} := true, \\ remove\ c_{ctl}^{buf}.first, move\ c_{wait}^{buf}\ to\ c_{sync}^{run}, \\ foreach\ c_{sync}^{run} as\ u\ \{a_{continue}\ (u_{instance}, u_{position})\} \quad \text{(P36-}\sigma\text{1-c)}$$

Rule P36-σ1-d describes that repair teams which are ready for work, are allowed to join the repair, as long as c_{ctl}^{add} is *true* (when it is *false* rule P36-σ1-a is firing).

$$\forall \langle e_{before} \rangle : e_{before}^{endpoint} = \text{syn} \wedge c_{ctl}^{add} = \text{true} \bullet \\ c_{sync}^{run}.append\left(e_{before}^{instance}, e_{before}^{position}\right), \\ a_{continue}\left(e_{before}^{instance}, e_{before}^{position}\}\right) \quad \text{(P36-}\sigma\text{1-d)}$$

Rule P36-σ2-a fires whenever the control team decides that no more teams are necessary, represented by the finishing of activity x: c_{ctl}^{add} is set to *false*.

$$\forall \langle e_{after} \rangle : e_{after}^{endpoint} = \text{ctla} \bullet c_{ctl}^{add} := true \quad \text{(P36-}\sigma\text{2-a)}$$

Rule P36-σ3-a represents that teams that finish their repair are put on standby as long as the control team not decided that the facility can be restarted (c_{ctl}^{done} is false). The teams that are in standby mode are moved to c_{sync}^{fin}.

$$\forall \langle e_{after} \rangle : e_{after}^{endpoint} = \text{syn} \wedge c_{ctl}^{done} = \text{false} \bullet \\ c_{sync}^{fin}.append\left(e_{after}^{instance}, e_{after}^{position}\right) \quad \text{(P36-}\sigma\text{3-a)}$$

As soon as the control team decides that the printing facility can be restarted (rule P36-σ3-b, c_{ctl}^{done} is set to *true*), all teams currently in standby (c_{sync}^{fin}) are allowed to continue. All teams teams still working are allowed to continue as soon as they finish (rule P36-σ3-c).

$$\forall \langle e_{after} \rangle : e_{after}^{endpoint} = \text{ctle} \bullet c_{ctl}^{done} := \text{true}, \\ foreach\ c_{sync}^{fin} as\ u\ \{a_{continue}\ (u_{instance}, u_{position})\} \quad \text{(P36-}\sigma\text{3-b)}$$

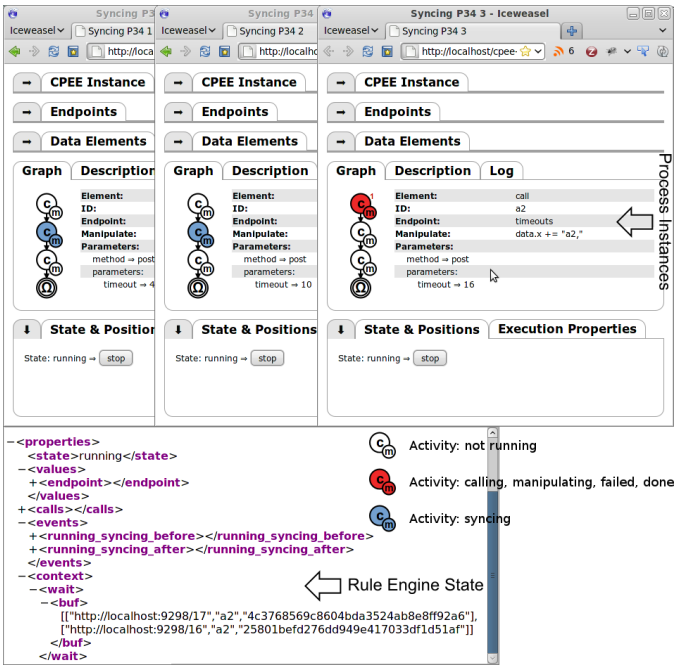


Fig. 7. Screenshot of Synchronized Process Instances

$$\forall \langle e_{after} \rangle : e_{after}^{endpoint} = \text{syn} \wedge c_{ctl}^{done} = \text{true} \bullet \\ c_{sync}^{fin}.append\left(e_{after}^{instance}, e_{after}^{position}\right), \\ a_{continue}\left(e_{after}^{instance}, e_{after}^{position}\right) \quad \text{(P36-}\sigma\text{3-c)}$$

As in P34, when all teams are finished, c_{sync}^{run} and c_{sync}^{fin} are set to empty.

$$\forall \langle c_{sync} \rangle : c_{sync}^{fin}.length = c_{sync}^{run}.length \bullet \\ c_{sync}^{fin} := Array, c_{sync}^{run} := Array \quad \text{(P34-}\sigma\text{2-d)}$$

VI. Implementation

In Fig. 7 the visual representation of a set of three instances can be seen in the upper half, whereas in the lower half, the internal representation of the state of the rule engine is visible.

Our prototype synchronization engine is connected to the CPEE (Cloud Process Execution Engine) by Stuermer et al. [11], [12] which supports the all the concepts described in Sect. III including HTTP based subscription and voting mechanisms.

For this prototype implementation we not only implemented the patterns mentioned in this paper, but we also identified and implemented additional synchronization related patterns. The implementation including rules realizing additional patterns (17, 18, 39, 40) can be found under: http://www.pri.univie.ac.at/workgroups/wee/?t=rulespattern.

VII. Related Work

Early approaches [13] for process instance synchronization proposed a database transaction inspired approach: ensure that the execution of activities is serializeable. Some years later,

multi instantiation of activities has been addressed by the workflow patterns project [3]. There are implementations of multi instantiation patterns with and without a priori runtime knowledge [14], [15], [16]. However, no comprehensive implementation of synchronization patterns has been presented yet. Specifically, there is no implementation in connection with processes executed within the cloud. Aside of workflow patterns, synchronization between instances of different process types has been addressed by Heinlein [17] based on a central unit that is controlling whether inter-process dependencies are fulfilled. While Heinlein proposes only a set of instructions to execute the activities in multiple processes in a particular order (akin to serialization in databases), we concentrate on parallel execution of activities in combination with complex synchronization logic (e.g. execute n of m activities). Heinleins approach can be implemented as special use-case of our approach. Approaches in the context of semantic compliance [18], [19], [20] of processes have merely focused of verifying constraints, often implemented as rules, for certain instances of one process type. However, they have not addressed inter-instance dependencies so far. Thus, the approach presented in this paper can be seen as complementary to these approaches.

In addition, compliance approaches are interesting in the way they implement the constraints: Among the formal languages proposed for compliance rule modeling, we often come across temporal logics such as linear temporal logic (LTL) or computational tree logic (CTL) [21]. Both are fragments of first-order logic (FOL). Due to its linear time semantics, which is more suitable in the business process context [22], LTL has been clearly preferred over CTL which has branching time semantics. Nonetheless, expressivness of both, LTL and CTL, is necessary for expressing semantic constraints, but is not very suitable for implementing synchronization rules: Synchronization deals not mainly with complex dependencies to establish the order of tasks, but with the description of circumstances for the (parallel) execution of tasks. Thus identifying the tasks (events and conditions) can be simpler, while additionally it is necessary to deal with the runtime aspect (actions and context variables). As discussed in the paper, we found ECA rules perfectly sufficing our requirements for synchronization.

VIII. Conclusion

In this paper we presented a simple mechanism for synchronizing process activities founded on a simple extension of the established activity lifecycle. Based on ECA rules it is possible to synchronize activities inside single instances, and more important: synchronize independent process instances. In comparison to existing approaches, synchronization is only loosely coupled with process engines (by just delaying the execution of activities). It also focuses on synchronization as defined by existing process patterns, instead of serialization of process descriptions. Future work will include more sophisticated dynamic synchronization mechanisms that can predict activity instantiation (negative example: print jobs that arrive only one second late, have to wait for a full day) and thus further reduce resource waste.

References

[1] A. Iyengar, V. Jessani, and M. Chilanti, *WebSphere Business Integration Primer: Process Server, BPEL, SCA, and SOA*. IBM Press, 2007.

[2] P. Dadam, M. Reichert, S. Rinderle-Ma, A. Lanz, R. Pryss, M. Predeschly, J. Kolb, L. T. Ly, M. Jurisch, U. Kreher, *et al.*, "From ADEPT to AristaFlow BPM suite: A research vision has become reality," in *1st Int'l. Workshop on Empirical Research in Business Process Management (ER-BPM '09)*, vol. LNBIP, (Ulm, Germany), Springer, Sept. 2009.

[3] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.

[4] C. Heinlein, "Workflow and process synchronisation with interaction expressions and graphs," in *Int'l Conference on Data Engineering*, IEEE, 2001.

[5] J. P. Womack, D. T. Jones, and S. . S. A. (Firm), *Lean thinking*. Simon & Schuster Audio, 1996.

[6] F. Curbera, R. Khalaf, W. A. Nagy, and S. Weerawarana, "Implementing BPEL4WS: the architecture of a BPEL4WS implementation," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, p. 1219–1228, 2006.

[7] F. Curbera, R. Khalaf, F. Leymann, and S. Weerawarana, "Exception handling in the BPEL4WS language," *Business Process Management*, p. 1021–1021, 2003.

[8] M. MSDN, "Windows workflow foundation, activity lifecycle." http://msdn.microsoft.com/en-us/library/cc303703.aspx, Jan. 2010. Last Access: 2011-04-01.

[9] J. Eder, J. Mangler, E. Mussi, and B. Pernici, "Using stateful activities to facilitate monitoring and repair in workflow choreographies," in *Proceedings of the 2009 Congress on Services - I*, pp. 219–226, IEEE Computer Society, 2009.

[10] J. Bae, H. Bae, S. Kang, and Y. Kim, "Automatic control of workflow processes using ECA rules," *IEEE Trans. on Knowl. and Data Eng.*, vol. 16, no. 8, pp. 1010–1023, 2004.

[11] G. Stuermer, J. Mangler, and E. Schikuta, "Building a modular service oriented workflow engine," in *Service-Oriented Computing and Applications (SOCA), 2009 IEEE International Conference on*, p. 1–4, 2010.

[12] J. Mangler, G. Stuermer, and E. Schikuta, "Cloud process execution engine - evaluation of the core concepts," *1003.3330*, Mar. 2010. [Last accessed: 17.03.2010].

[13] G. Alonso, D. Agrawal, and A. E. Abbadi, "Process synchronization in workflow management systems," in *Parallel and Distributed Processing, 1996. Eighth IEEE Symposium on*, p. 581–588, 1996.

[14] S. Rinderle and M. Reichert, "Data-Driven process control and exception handling in process management systems.," in *Proc. CAiSE'06*, LNCS 4001, p. 273–287, 2006.

[15] A. Guabtni and F. Charoy, "Multiple instantiation in a dynamic workflow environment," in *CAiSE*, pp. 175–188, 2004.

[16] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns | evaluations | products - FLOWer," June 2010.

[17] C. Heinlein, "Synchronization of concurrent workflows using interaction expressions and coordination protocols," in *CoopIS/DOA/ODBASE*, pp. 54–71, 2002.

[18] A. Awad, G. Decker, and M. Weske, "Efficient compliance checking using BPMN-Q and temporal logic," in *Int'l Conference Business Process Management*, p. 326–341, 2008.

[19] S. Sadiq, W. Sadiq, and M. Orlowska, "A framework for constraint specification and validation in flexible workflows," *Information Systems*, vol. 30, no. 5, p. 349 – 378, 2005.

[20] L. Ly, S. Rinderle-Ma, and P. Dadam, "Design and verification of instantiable compliance rule graphs in Process-Aware information systems," in *Proc. Int'l Conf. on Advanced Systems Engineering*, pp. 9–23, 2010.

[21] A. Ghose and G. Koliadis, "Auditing business process compliance," in *Int'l Conference on Service Oriented Computing* (B. Krämer, K. J. Lin, and P. Narasimhan, eds.), vol. 4749 of *LNCS*, p. 169–180, Springer-Verlag, 2007.

[22] Y. Liu, S. Müller, and K. Xu, "A static compliance-checking framework for business process models," *IBM Systems Journal*, vol. 46, no. 2, p. 335–261, 2007.