Composing Time-Aware Web Service Orchestrations

Horst Pichler, Michaela Wenger, and Johann Eder

University of Klagenfurt, Department of Informatics-Systems, Austria*

Abstract. Workflow time management deals with the calculation of temporal thresholds for process activities, which allows forecasts about looming deadline violations. We present a novel approach to transform a web service orchestration into a time-aware orchestration, that contains temporal assessment and intervention logic. During process execution intervention strategies are triggered pro-actively to speed up a late process and to avoid upcoming violations of temporal constraints.

1 Introduction

Web service orchestrations are used to assemble processes from external processes and web-services to implement business processes. Expected process execution times and compliance to agreed upon deadlines rank among the most important quality measures [5,1]. To speed up processes and decrease the number of deadline violations should therefore be among the major objectives of business process management. This can be achieved by the application of workflow time management [17]. It deals with temporal aspects of time-constrained processes and aims at optimized, timely, and violation-free process execution. Based on explicit knowledge about process structure, activity durations, and deadlines it is possible to calculate temporal thresholds (internal deadlines) for each activity of a process. During run time, these thresholds are utilized to monitor the progress, forecast looming deadline violations, and to pro-actively trigger intervention strategies which speed up the remainder of the process, like skipping of optional activities, choosing alternative shorter paths, substituting activities, etc.

Time management approaches have mainly dealt with modelling of temporal aspects, checking satisfiability of temporal constraints, scheduling, and so on. The basic concepts of intervention strategies have been described (see Section 2), but how to model, implement, and apply them on process definitions and within process instances is still open. Suggested realization of time management functionality requires new process definition elements and additional logic within the process engine.

Our main goal is to close the gap between build and run-time concepts and enable time management for long-running web service orchestrations. The novel contribution of our approach is that we do not propose an extension of the process enactment service but make the process itself time aware and capable of triggering pro-active measures. So time-aware information systems can be realized without waiting for vendors to complement their process enactment services with up-to-date time management capabilities.

* Part of this work has been supported by EU Commission within the FET-STREP Project WS-Diamond.

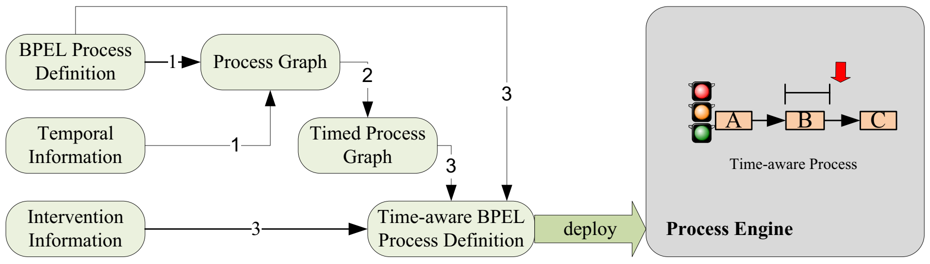


Fig. 1. Generation of a Time-aware Process Definition

The architecture of our approach is shown in Figure 1. Inputs to the process are a process model (in our realization a BPEL process definition), a representation of explicit temporal information (durations and deadlines) and a representation of intervention strategies. From the process model, augmented with temporal information, we calculate particular internal deadlines for all activities. Then we use this timed process graph and the intervention information to extend the process model with 2 aspects: (a) temporal state assessment and (b) intervention logic. Temporal state assessment monitors the execution of the process and compares the execution times with pre-calculated thresholds. If the process is not running within its tolerance, exceptions are raised. The intervention logic reacts to these exceptions and enables changes to the original process logic with the goal to regain a safe execution state. This results in an extended time-aware process with self-healing capabilities which can then be executed on any enactment service of that process model type.

To be more concrete we instantiated this general approach for WS-BPEL. However, this approach can be easily ported to any commercial workflow system, which supports exceptions and timed triggers in it's proprietary control flow language. Some more assumptions: we focus on long-running processes with asynchronous communication structures that take hours, days or even weeks, where the execution time of the process logic is negligible compared to the processing time of the individual activities.

The remainder of the paper is organized as follows: Section 2 provides an overview of related work and describes time management basics. Section 3 gives an overview of elements in a BPEL process definition and shows how to transform it into a graph representation, a prerequisite for the calculation of the timed graph in Section 4. In Section 5 we show how to define intervention and state assessment information. Section 6 describes how to transform the original definition into a time-aware process definition, based on the timed graph and intervention information. Finally we discuss our prototype and complexity in Section 7, and conclude the paper with Section 8.

2 Related Work

Workflow time management architectures, as sketched in [9] or [14], consist of several components. [9] proposes a *build time component* that takes a control flow model and temporal information about activity duration and time constraints as input, and calculates thresholds for each activity in the process. The calculation of thresholds is frequently rooted in techniques based on temporal constraint networks to model and verify temporal information [12,19], others utilize project planning methods [13,11,19]. All these techniques apply variants of interval-timed graph-based models, which can also be used to calculate schedules for workflow execution. A node represents an activity or its start/end-event, edges represent precedence, constraints between nodes, and intervals are used to describe valid time frames for the execution of an activity or occurrence of an event. In this paper we adopted the approach of [10]. They extend the temporal model of [11] for asynchronous messaging patterns, an important prerequisite in web service environments (see also [18]). During run time a *prediction component* [9] correlates and compares time stamps of start and end-events with precalculated thresholds of corresponding activities. Based on this comparison the temporal state can be assessed, for example with the traffic-light model introduced in [17], or the duration and instantiation space model introduced in [15]. Delays, caused by unexpected waiting times or longer execution durations, will change the temporal state of the process. According to this state a *proactive component* [9] has to select intervention strategies that trigger actions within the process engine to speed up the process. Proactive intervention strategies, aiming at speeding up the process, may be applied according to the current temporal state to avoid looming deadline violations.

In this paper we apply intervention strategies that can be realized by implementing them within the process definition, like skipping optional tasks, execution of alternative paths or the parallel execution of sequential activities (see [22,11,8,16] for basic descriptions). Furthermore, we utilized an adaptation of the technique described in [6]: early escalation, which terminates a late process immediately, if the cost of finishing it, is higher than immediate escalation followed by termination. More escalation strategies, which can not be realized by changing the process definition, can be found in [8]. This includes load balancing strategies like resource redeployment (e.g., add resource capacity) or grouping similar tasks to batches to decrease the task preparation time. The closest approach to our's is [8] where time-awareness is also integrated into the process definition itself, by extending it with hard-coded conditional structures. Such a structure could for example be ”if (process late) then {perform two reviews} else {perform three reviews}”. Naturally, the statement ”process late” must be specified as a state-assessment expression, that compares time stamps of events (start or end of an activity) with a precalculated thresholds. The disadvantages of such an approach are obvious. Defining the process gets more complex as the designer has to specify state assessment mechanisms and temporal exception handling parts. New duration estimations or changed deadlines alter the thresholds, resulting in manual adaptations of the assessment expressions. Furthermore, intervention strategies for late processes may be added, changed or removed, which alters the process structure. And finally, with such a ”passive” approach it is not possible to handle activities that block process execution.

Specific temporal aspects of web service environments where examined in, e.g., [20,21]. [20] uses temporal abstractions of business protocols in a finite state machine formalism and [21] exploits an extension of a timed automata formalism for modeling time properties of web services. However, these approaches aim primarily at service compatibility, therefore proved to be unsuitable for our purposes.

3 Process Representations

In this section we describe the elements of BPEL (the process definition language of our realization), show how to transform it to a graph representation and augment it with temporal information, and describe subsequent timed graph calculations.

3.1 BPEL Representation

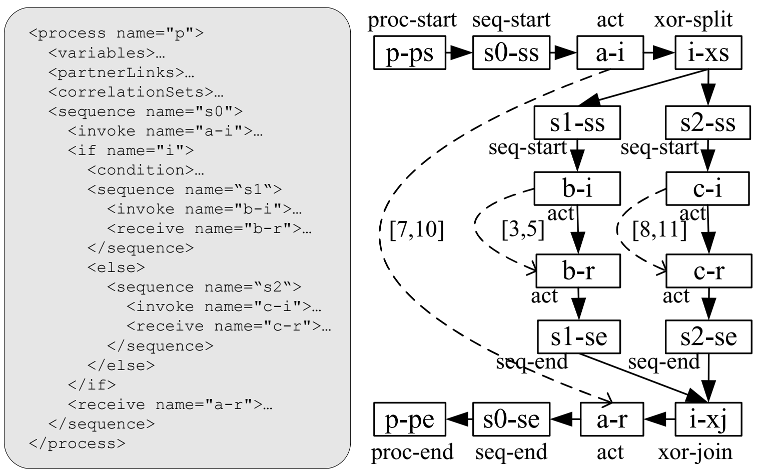
The original process is defined in WS-BPEL, also known as BPEL 2.0, which provides the following elements:

Declarative Elements to specify the environment of the process, like links to web services (*partner links*), process *variable*s, or *correlation sets*.
Basic Activities for the communication with web services: invoking or sending a message to a web service with *invoke*; *receive* a message from an external service; send a *reply* to an external service (as answer to a prior receive). Furthermore, activities to *assign* a value to a variable, delaying execution with *wait*, doing nothing with *empty*, and explicit termination with *exit*.
Structured Activities which define the control flow of the process by nesting of basic or structured activities: *sequence* for sequentially executed activities; *if* for conditional exclusive execution of activities; *flow* for parallel execution; *while*, *repeatUntil* and *forEach* for iterative execution; and *pick* for conditional execution based on the type of a received message. Additionally it is possible to declare *event* and *fault-handlers* within *scopes*, which handle thrown faults and raised events.

Some activity-types delay process execution because they wait for something (e.g., an incoming message). Therefore we consider the basic activities *receive* and *pick* as blocking. Furthermore, every structured activity is considered as blocking, if it contains a blocking basic or structured activity. The left-hand side of Figure 2 shows the skeletton of a BPEL process definition. To identify specific activities within a process definition, we use the *name*-attribute, an attribute that can be added to any BPEL-element. Note that this paper focuses on the regular flow in block-structured processes, therefore we do not consider flows with links and exception handlers.

3.2 Process Graph

According to [10] and as visualized in Figure 2, a *process graph* consists of named nodes (rectangles with labels) connected by edges, which describe precedence constraints (solid arrows). Each node has a type (label above or below a node), which can either be activity (act) or an opening or closing node-type for structured elements: sequential execution of elements (seq-start, seq-end), 1-out-of-n exclusive conditional execution (xor-split, xor-join), parallel execution of a several paths (and-split, and-join), and the process itself (proc-start, proc-end). Furthermore, asynchronous communication relationships between invoking and receiving activities are represented as dashed arrows, augmented with service response times (angle-brackets on top of dashed arrows). The example process contains several nested structures: a sequence *s0* that executes *a-i*, followed by an if-conditional element *i*, which selects either *b-i* and *b-r* or *c-i*



```
<process name="p">
  <variables>…
  <partnerLinks>…
  <correlationSets>…
  <sequence name="s0">
    <invoke name="a-i">…
    <if name="i">
      <condition>…
      <sequence name="s1">
        <invoke name="b-i">…
        <receive name="b-r">…
      </sequence>
      <else>
        <sequence name="s2">
          <invoke name="c-i">…
          <receive name="c-r">…
        </sequence>
      </else>
    </if>
    <receive name="a-r">…
  </sequence>
</process>
```

Fig. 2. BPEL process and Graph-representation - Generation Step 1

and *c-r*, followed by a receiving *a-r*. Although the details are not specified in this shortened example, assume, that *a-i* asynchronously sends a message to an external service, whose response is received by *a-r*; furthermore, *b-i* and *c-i* send messages to services, whose response is received by *b-r* and *c-r*. Due to space limitations we omitted the XML-representation of the process graph and temporal information, which we used in our prototypical implementation.

3.3 Temporal Information

Additionally we need explicit *temporal information*, which are a maximum process duration (assume [15,15] for our running example) and response times of asynchronous relationships between invoking and receiving nodes. We apply [min, max]-intervals for temporal information (durations, deadlines), given in a specified time-unit, which will be days or hours (as applied in the running example) for long running processes. Expected service response times may stem from empirical knowledge (extracted from logs) or be estimated by experts. Especially when dealing with web services this information may also come from the service provider [5] or service directories, which may offer temporal information as part of their service descriptions [3]. The algorithm in [10] additionally requires explicit information about the execution duration of every node (ranging from millis to a few minutes at maximum), which we considered, compared to service response times, negligible. So we set all node durations to [0,0].

3.4 Transformation

How to transform a hierarchical block-structured process, which BPEL is, to a flattened graph representation is described in [4]. For the transformation we had to add the following BPEL-specific rules:

- every basic activity is represented as a node of type *act*
- the structured activities *sequence*, *if* and *flow* are represented as two nodes of corresponding type, which embrace inner nested activities: *seq-start* and *seq-end*, *xor-split* and *xor-join*, *and-split* and *and-join*
- the structured activity *pick* must be encapsulated in a sequence, which contains one node (the message receiver), followed by an xor-structure with a branch for each message type handler
- structured scope activities are interpreted as sequences

The following exceptions to above stated rules had to be considered: (a) iterative activities are represented by two nodes of type *act* connected by a precedence edge. Eventually nested elements are omitted.[1] Additionally these two nodes are connected with an asynchronous relationship edge, augmented with the specified, estimated (or calculated) execution time of the activity, specified in the temporal information file. (b) The same applies for *wait* activities, but here the duration can be extracted from the duration expression in the BPEL-definition. (c) Event, fault and compensation handlers of the original process definition are not considered, as time management focuses on the regular flow, and are therefore omitted in the graph representation.

It is basically possible to extract asynchronous, and therefore temporal, relationships (dashed arrows) between invoking and receiving activities from the BPEL process definition by interpreting the declaration parts (partnerLinks, etc.). However, this is outside the scope of this paper, and therefore we demand, that information about these relations must be specified by the process designer within the temporal information file. Asynchronous relationship edges and their response times can now be added to the graph in the final transformation step.

4 Timed Graph Calculation and State Assessment

Now we have a basic graph representation, augmented with a maximum duration and service response times (in hours) between invoking and receiving activities. Equipped with this information we can calculate the timed graph. Based on the process structure and explicit temporal information, it is is possible to determine remaining durations for each node in the process graph. The remaining duration interval represents the expected minimum and maximum execution duration of the path between node n and the end of the process. The calculation algorithm utilizes the graph specified above and explicit temporal information, and yields remaining durations for each node as visualized in Figure 3. The remaining duration of the first node proc-start also depicts the expected overall process duration. Due to space limitations we can not explain the details of this calculation (refer to [10]). Furthermore note that remaining durations or thresholds may be calculated with any interval-based approach that is capable of dealing with above mentioned control structures, where some even allow more complex time constraints (like [11], which introduces upper and lower-bound constraints).

[1] As cyclic structures are problematic for time management calculations, since the actual number of iterations will not be known in advance, we applied a common solution for now: interpreting the loop as one complex activity with an estimated or calculated overall duration.

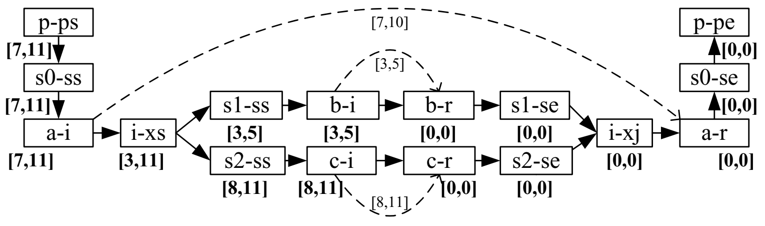


Fig. 3. Timed Graph - Generation Step 2

For run-time purposes we need thresholds, relative to the start time of the process, that shall not be exceeded in order to meet the process deadline. For this we adapted the idea of the traffic light model, described in [17]. It's lights represent the temporal states: (1) *Green* indicates, that the process can be finished within the calculated process duration. (2) *Yellow* indicates, that the process can be finished within the specified maximum duration. Future delays should be avoided, as the process already started consuming buffer time. Proactive intervention is advised. And otherwise it is (3) *Red*, which indicates, that all available buffer time has been consumed, and that missing the deadline is likely. Proactive intervention is inevitable, the process must be sped up.

For illustration purposes we applied a very simple (rather pessimistic worst-case) approach for temporal assessment: we calculate two state switching thresholds for each node/activity *n*, based on the upper bound of a specified maximum process duration interval *maxduration.ub = 15* hours, the upper bound of the calculated process duration interval *calcduration.ub = 11* hours, and the upper bound of the node's remaining duration interval *n.rduration.ub*, as follows:

– *n.greenToYellow := calcduration.ub - n.rduration.ub*
– *n.yellowToRed := maxduration.ub - n.rduration.ub*

Note, that state assessment for blocking structured activities must use the remaining durations of the corresponding end node. This calculation yields, e.g., for the node a-i the following thresholds: *a-i.greenToYellow = 0* hours after process start and *a-i.yellowToRed = 4* hours after process start. As *a-i* is the first basic activity in the process, it will be reached within (milli)seconds (rounded to 0 hours) after process start, therefore the temporal state will most probably never switch to yellow or red at this position. However, theoretically *a-i* could consume up to four hours before the state switches to red - this time is also called *buffer time*. For the receiver activity *b-r*, we determine the following values: *b-r.greenToYellow = 11* and *b-r.yellowToRed = 15*. If, for example, *b-r* did not receive its message until 11 (hours after process start) then the state switches to yellow and we could start an intervention, e.g., interrupt the waiting activity *b-r* and invoke an alternative fast (more "expensive") service which returns a message immediately: the process is in time, but the costs increased. You will notice, that the threshold-values of *b-r* are equal to the specified and calculated process duration (and equal to the thresholds of *c-r* and *a-r*). This means that each of these activities is allowed to consume the whole buffer time of the process, leaving no buffer for subsequent activities. For descriptions of fairer buffer distribution techniques refer to [7].

5 Interventions

Proactive time management needs information about how to intervene when the temporal state changes. We support the following intervention strategies, which may be applied on any BPEL-activity (basic or structured).

Optional Execution. Skipping optional activities can be used on any element, which is not absolutely necessary for the successful completion of the process. Therefore the element, or activities nested within this element, should not have communication relationships to another element in the process (e.g. skipping an invoke activity may block the process at the matching receive activity).

Parallelization of Sequence. Parallelization of sequences forces the parallel execution of elements of a sequence. Note, that elements nested in the sequence (or their subelements) may have communication relationships between each other. In the worst case, such an execution will again be sequential.

Alternative Path. A late process can sped up by executing a faster alternative (basic or structured) activity, instead of the original one. Again, eventually existing communication relationships between elements must be considered: e.g., an alternative for an invoke-activity which calls another service may block the process at the matching receive-activity.

Dynamic Service Selection. This strategy is a variant of Alternative Path that offers multiple alternatives. In case the process is late, the fastest of several variants must be selected and executed. We assume, that the list of matching candidates has been preselected. Alternatively one could also apply a QoS-based adaptive service-retrieval technique, which automatically finds compatible candidates [23].

Early Termination. Early termination of a late process, depicted by *terminate*, aims at the avoidance of costs resulting from further process execution and exception-handling actions at the end. Although we do not consider the cost factor in this paper (cp. [6]), we offer this policy as a last resort, only to be used in extreme cases, as it does not consider side-effects on integrated processes or services.

The specification of interventions binds activities of the process, depicted by their name, to a certain intervention-behavior, which shall be invoked instead of the activity, if the process is in the given temporal-state. We defined an XML-structure for intervention information, specified by the following DTD.

```
<!ELEMENT interventions (intervention)+ >
<!ELEMENT intervention (intervene)+ >
<!ATTLIST intervention activity CDATA #REQUIRED >
<!ELEMENT intervene ((terminate|optional|parallelize|alternative|dynamic), bpel?>
<!ATTLIST intervene when (yellow|red) #REQUIRED >
<!ELEMENT terminate EMPTY >
<!ELEMENT skip EMPTY >
<!ELEMENT parallelize EMTPY >
<!ELEMENT alternative (bpel) >
<!ELEMENT dynamic (variant)+ >
<!ATTLIST dynamic objective (green|yellow|red) #REQUIRED >
<!ELEMENT variant (bpel) >
<!ATTLIST variant duration CDATA >
<!ELEMENT bpel (declarations,actions) >
<!ELEMENT declarations (#PCDATA) >
<!ELEMENT activity (#PCDATA) >
```

A set of *interventions* for a specific process may contain several *intervention* elements. Each refers to a basic or structured *activity*-name in the BPEL-process and contains *intervene* elements, that define which intervention strategy to apply, *when* a certain temporal state is assessed. An Intervene-element must contain one element of type *terminate, skip, parallelize, alternative,* or *dynamic*, which specifies the strategy to apply. Furthermore, an intervene-element may contain an optional *bpel* element, which defines a BPEL-activity (basic or structured) and necessary BPEL-declarations, which we do not further specify here (variables, partnerLinks, etc.). This BPEL code will be executed before the intervention itself takes place; it may for instance be used to notify the process-owner about temporal state changes. The following example defines interventions for two activities of a (fictituos) process.

```
<interventions>
    <intervention activity="a_sequence" >
        <intervene when="red"> <optional/> </intervene>
        <intervene when="yellow"> <parallelize/> </intervene>
    </intervention>
    <intervention element="an_activity">
        <intervene when="yellow"> <optional/> </intervene>
        <intervene when="red" >
            <terminate />
            <bpel>
                <declarations> ... BPEL declarations ... </declarations>
                <activity> ... BPEL activity (e.g. notify owner)... </activity>
            </bpel>
        </intervene>
    </intervention>
</interventions>
```

Intervene-elements of type *alternative* must contain *bpel* declarations and code of the alternative. For interventions of type *dynamic* we must specify multiple execution *variants*, augmented with information about the (expected) duration for each alternative. The attribute *objective* defines the desired temporal state after the execution of a variant. The *duration* of a variant (may be a structured activity) can be calculated with the calculation algorithm explained above. The following intervention specification refers to activities within our example process. The sequence s1 shall be skipped if the process enters state red, and if it enters state yellow alternative code shall be executed (e.g. invoking a fast service and receiving it's message). If the process waits too long at the the receiving activity *a-r* it shall either select a faster variant (yellow) or even terminate (red). Note, that both activities are considered 'blocking', as they wait for an incoming message.

```
<interventions>
    <intervention element="s1">
        <intervene when="red"> <optional/> </intervene>
        <intervene when="yellow">
            <alternative>
                <bpel> declarations and activity for alternative </bpel>
            </alternative>
        </intervene>
    </intervention>
    <intervention element="a-r">
        <intervene when="red"> <terminate/> </intervene>
        <intervene when="yellow">
            <dynamic objective="green">
                <variant duration="[6,9]">
                    <bpel> declarations and activity for alternative1 </bpel>
                </variant>
                <variant duration="[3,5]">
                    <bpel> declarations and activity for alternative3 </bpel>
                </variant>
                <variant duration="[4,7]" >
                    <bpel> declarations and activity for alternative2 </bpel>
                </variant>
            </dynamic>
        </intervene>
    </intervention>
</interventions>
```

6 Generation of a Time-Aware Process Definition

A time-aware process definition is an extension of the original process definition with state assessment and intervention mechanisms for specified process-parts, to be executed in case given thresholds are violated. The generation is based on the original process definition, the timed graph, and intervention information, and consists of the following basic steps:

```
[works on copy of original process definition]
add process-level extensions
for each activity x in bottom-up order (deepest nestings first)
    if exists intervention for x
        if x is non-blocking
            add activity-level extensions for non-blocking activity x
        elseif x is blocking
            add activity-level extensions for blocking activity x
        end-if
    end-if
end-for
```

6.1 Process-Level Extension

First the original process definition must be extended on the top-level, as visualized in the BPMN-diagram [2] on the left-hand side of Figure 4 (elements of the original process are displayed grey-shaded). We decided to use BPMN as graphical representation instead of BPEL-code due to space limitations.

1. Nest the top-level activity of the process within a new sequence-activity *top_seq*
2. Insert assignment for the current time *t*, which is the start time of the process.
3. Initialize the temporal *state* by assigning it the value 'green'.

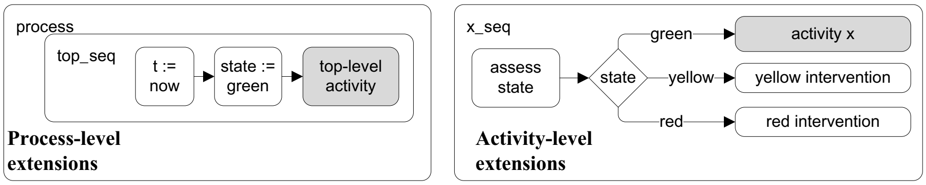


Fig. 4. Process-level Extensions and Extensions for non-blocking Activities

We used XQuery and XPath in our BPEL-prototype for accessing, comparing and manipulating variables, as they offer a rich function-base for diverse purposes, along with datatypes for the structured representation of dates, times and durations.

6.2 Generation of Intervention Logic

The intervention logic for an activity consists of a state assessment mechanism to determine the current temporal state, conditional structures to select the corresponding intervention, and timed triggers for blocking elements.

State Assessment Mechanism. An important part of intervention logic is the assessment of the current temporal state. The following shows a simplified pseudo-code representation of the necessary state assessment extensions for an activity *x* and it's thresholds:

```
rel_time := currentTime() - t;
if (rel_time <= x.greenToYellow) then state := green
  elsif (rel_time <= x.yellowToRed) then state := yellow
  else state := red
```

State assessment is based on a comparison of the relative time (duration since start of the process) and the corresponding node-dependent threshold value. For non-blocking activities it takes place before the invocation, and for blocking activities during their execution.

Basic Intervention Extensions for Non-blocking Activities. The BPMN-diagram on the right-hand side of Figure 4 shows the extensions for a non-blocking activity *x*.

1. Replace activity *x* with a new if-activity *x_if* (omitted in the diagram).
2. Add three branches and conditions for states green (if), yellow and red (elseif).
3. Insert activity *x* into the if-branch.
4. Generate intervention handling code for the elseif-branches (see details below).
5. Nest *x_if* within a new sequence-activity *a_seq*.
6. Insert state assessment elements before *x_if* into *a_seq*.

Intervention Extensions for Blocking Activities. For blocking activities such a passive intervention mechanism is not sufficient. The prediction component must additionally check threshold-violations during the execution of this element. Therefore it is necessary to add time-triggered logic as visualized in Figure 5. We used several mechanisms: timed triggers (circle with clock) which invoke corresponding event-handlers, throwing of faults (fat circle with flash-symbol), and catching of faults within a fault-handler (double circle with flash-symbol). Within a BPEL-scope we use fault-handlers, which catch named faults (exceptions) that are thrown within this scope. Furthermore, it is possible to define time triggered event-handlers based on the onAlarm-element, which periodically executes specified code (basically a concurrent sub-process). The diagram is to be interpreted as follows: the control flow enters the scope and starts the (blocking) activity. An onAlarm event-handler periodically calls state assessment, which checks if the temporal state has changed. If this is the case, it will immediately

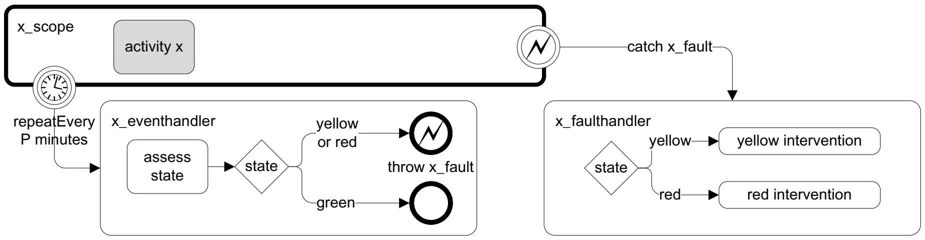


Fig. 5. Activity-level Extensions for blocking Activities

throw *x_fault*, which is caught by *x_faulthandler*. If the state is still green the event handler will return control to the regular control flow (the execution of *x*). The generation of extensions for blocking activities consists of the following steps:

1. Nest activity *x* within a new scope *x_scope*.
2. Add *x_eventhandler* to the scope, containing an onAlarm-element with a specified repeatEvery-period, including state assessment logic including a throw-element.
3. Add *x_faulthandler* to the scope, which contains state-dependent intervention handling code.

Generating Intervention Handler Code. Intervention handler code is executed when the temporal state changes to yellow or red, and must be integrated in the corresponding if-branches. Intervention code for an activity *x* and a certain temporal state is determined by the specified *intervene*-element within the intervention information (see Section 5) and basically generated as follows.

1. Insert a sequence *x_int_seq* within the corresponding if-branch.
2. Add type-dependent intervention code to *x_int_seq* (for details see below).
3. If the optional bpel-element exists within the intervene element:
   (a) add the bpel-activity before the intervention code that was generated in step 2
   (b) add the related bpel-declarations (variables, partnerLinks, etc.) on process level

Step 2 generates type-dependent intervention code: for an intervention of type *optional* we add the BPEL-activity *empty* and for *terminate* we simply add the BPEL-activity *exit*. Adding BPEL-code for an *alternative* path is equal to steps 3.a) and 3.b). To *parallelize* a sequence we generate a flow element in the corresponding if-branch of the intervention logic, which contains a duplicate of every activity within the sequence-activity (including already generated intervention logic of nested activities).

The last intervention mechanism, *dynamic* service replacement, has to select one out of several replacement alternatives. Selection is based on the current delay of the process, the position and duration of replacement variants, and the objective (desired goal state). Code generation is best explained by an example. In Section 5 we defined a yellow-intervention of type *dynamic* for activity a-r, with three replacement variants ordered by maximum duration: one with a duration of [6,9], one with [4,7] and one with [3,5]. The objective is green, which means that the execution of the selected alternative should be finished until the green end-threshold of a-r. The calculation of green and yellow thresholds for activity a-r yielded: *c_greenToYellow=11* and *c_yellowToRed=15* (hours after the process start). With this information we generate the following code:

```
if (state = yellow)
    rel_time := currentTime() - t;
    timeframe := a-r.greenToYellow - rel_time;
    if (timeframe >= 9) bpel-code of variant
      elseif (timeframe >= 7) bpel-code of variant
      else bpel-code of fastest variant
```

Again we applied a worst-case approach: first we calculate the relative time (duration since process start), followed by the calculation of the time frame. The time frame is the difference between the threshold of the goal state (greenToYellow) and the relative time. The duration of the selected variant must fit into this time frame. Therefore we compare the time frame with the upper bound values (worst case) of the duration intervals specified for each alternative (in decreasing order) and add corresponding BPEL-code. The fastest variant will be selected, even if it does not fit in the time frame. This approach assumes that the faster a service the more expensive it will be, and shall therefore only be selected if absolutely necessary. For our prototype we implemented an additional version, which allows to specify variants ordered by preference – a preferred service will be selected when it fits into the time frame, even if a slower service exists, that also fits into the time frame.

Furthermore, we designed and implemented an improved version for alternative and dynamic interventions on blocking activities, which exploits the following finding: if the (blocking) activity is almost finished, it is not a wise decision to interrupt it and execute one of the variants. Therefore we added a special treatment on state-assessment-level for these types, which checks (on state change) if the remaining execution duration of the activity is less than the duration of the variant that fits into the time frame. If this is the case no fault will be thrown, and the original activity is allowed to finish.

7 Prototypical Implementation and Complexity Considerations

We implemented a proof-of-concept Java-based transformation prototype and tested several processes with the open source engine ActiveBPEL. The advantage of this approach is, that it adds pro-active time-awareness to the process, which results in less deadline violation. The process designer is not addressed with complicated calculations and programming of tedious intervention logic. Furthermore, if the effect of current intervention strategies is insufficient, then temporal information, intervention strategies or temporal assessment can be changed easily, and used to generate new time-aware process definition. On the downside we have to state, that the generated process will contain considerably more elements than the original process definition. The number of additional elements varies heavily, depending on various parameters, the nesting-depth, the number of specified interventions, and the complexity of intervention logic. Still, it can be predicted by using the tables in Figure 6. For specific interventions both tables must be combined, e.g., when defining an intervention for non-blocking sequence-activity (with n = 3 nested basic activities), where the yellow-intervention is parallelize and the

Extension Types: Number of additional Elements

	var decl.	var assign	seq	if		scope	fault handler	throw	catch	event handler	on Alarm	**Sum**
				#	br*							
process**	3***	2	1	0	0	0	0	0	0	0	0	**6**
non-blocking	0	3	1	1	3	0	0	0	0	0	0	**8**
blocking	0	3	1	3	7	1	1	1	1	1	1	**20**

*number of branches incl. conditions ** always included *** includes rel_time for state assessment

Intervention Types: Number of additional Elements (w/o optional BPEL code)

	act & decl.	flow	seq	if		var assign	**Sum**
				#	br		
skip	1	0	0	0	0	0	**1**
terminate	1	0	0	0	0	0	**1**
parallelize	n	1	0	0	0	0	**n + 1**
alternative	b + d	0	0	0	0	0	**b + d**
dynamic	(v*b) + (v*d)	0	1	1	v	2	**(v*b) + (v*d) + 4**

n … #activities nested in activity for which intervention has been declared
v … #variants (activities) for dynamic replacement
b … avg. #activities nested in activities, which describe one variant
d … avg. #declarative elements (partnerLinks, etc.) for one variant

Fig. 6. Tables for Prediction of Number of Additional Elements

red-intervention is skip, we calculate the number of additional elements as: sum process + sum non-blocking + sum skip + sum parallelize = 6 + 8 + 1 + 4 = 19 additional elements. In nearly all cases the complexity will be linear, with one exception: nested parallel structures, where increase is exponential in the number of parallelizations in a nesting path, as all nested elements must be duplicated on each level with parallelization during the bottom up generation of intervention logic. Therefore, we propose to specify interventions only for selected mission-critical parts and for parts which have the potential to significantly speed up the process. A related problem is, that a user who monitors the progress of the process will see a rather complicated transformed process.

8 Conclusions

The prediction and proactive avoidance of deadline violations decreases costs of processes and increases their quality of service. Existing approaches describe how to model and calculate temporal information for these purposes, but do not show how to apply corresponding interventions on running processes. Therefore we enhanced the original process definition with additional interval-based temporal and intervention information, and showed how to transform it into a time-aware process definition, which pro-actively avoids looming deadlines and that can be executed on any engine that supports BPEL. To achieve this we utilized inherent control-flow features of the process definition language to integrate time-triggered predictive and proactive intervention mechanisms, with a focus on blocking activities that wait for messages of delayed external services. Current and future research comprises handling of non-blocked structures (flows with links), exception handlers, and how to compensate already finished activities.

References

1. Cardoso, J., Sheth, A., Miller, J.: Workflow Quality of Service. In: Proc. of the Int. Conf. on Integration and Modeling Technology (IEIMT/IEMC). Kluwer Publishers, Dordrecht (2002)
2. OMG: Business Process Modelling Notation (BPMN) 1.1. OMG Specification (2008)

3. W3C: OWL-S: Semantic Markup for Web Services. W3C Member Submission (2004)
4. Eder, J., Gruber, W.: A Meta Model for Structured Workflows Supporting Workflow Transformations. In: Manolopoulos, Y., Pokorný, J., Sellis, T.K. (eds.) ADBIS 2006. LNCS, vol. 4152. Springer, Heidelberg (2006)
5. Gillmann, M., Weikum, G., Wonner, W.: Workflow Management with Service Quality Guarantees. In: Proc. of ACM SIGMOD Int. Conf. on Management of Data. ACM Press, New York (2002)
6. Panagos, E., Rabinovich, M.: Predictive Workflow Management. In: Proc. of the 3rd Int. Workshop on Next Generation Information Technologies and Systems, Neve Ilan, Israel (1997)
7. Kao, B., Garcia-Molina, H.: Deadline Assignment in a Distributed Soft Real-Time System. IEEE Transactions on Par. Dist. Systems 8(12) (1997)
8. van der Aalst, W.M.P., Rosemann, M., Dumas, M.: Deadline-based Escalation in Process-Aware Information Systems. BPM Center Report, BPM-05-05, BPMcenter.org (2005)
9. Eder, J., Pichler, H., Vielgut, S.: An Architecture for Proactive Timed Web Service Compositions. In: Eder, J., Dustdar, S. (eds.) BPM Workshops 2006. LNCS, vol. 4103, pp. 323–335. Springer, Heidelberg (2006)
10. Eder, J., Pichler, H., Vielgut, S.: Avoidance of Deadline-violations for Inter-org. Business Processes. In: Proc. of the 7th Int. Baltic Conf. on DBs and Inf. Systems. IEEE Press, Los Alamitos (2006)
11. Eder, J., Panagos, E., Rabinovich, M.: Time Constraints in Workflow Systems. In: Jarke, M., Oberweis, A. (eds.) CAiSE 1999. LNCS, vol. 1626, p. 286. Springer, Heidelberg (1999)
12. Haimowitz, I.J., et al.: Temporal Reasoning for Automated Workflow in Health Care Enterprises. In: Adam, N.R., Yesha, Y. (eds.) Electronic Commerce 1994. LNCS, vol. 1028. Springer, Heidelberg (1996)
13. Marjanovic, O., Orlowska, M.: On Modeling and Verification of Temporal Constraints in Production Workflows. Knowledge and Information Systems 1(2) (1999)
14. Marjanovic, O., Orlowska, M.: Workflow Temporal Manager. In: Proc. of the Australian Workshop on Intelligent Desicion Support and Knowledge Management, Sydney, Australia (1998)
15. Marjanovic, O., Orlowska, M.: Dynamic Verification of Temporal Constraints in Production Workflows. In: Proc. of the Australasian Database Conf. IEEE Computer Society, Los Alamitos (2000)
16. Baggio, G., et al.: Applying Scheduling Techniques to Minimize the Number of Late Jobs in Workflow Systems. In: Proc. of ACM 2004 Symp. on Applied Computing. ACM Press, New York (2004)
17. Eder, J., Panagos, E.: Managing Time in Workflow Systems. In: Workflow Handbook 2001, Future Strategies Inc. (2001) ISBN 0-970-35090-2
18. Newcomer, E.: Understanding Web Services. Addison-Wesley, Reading (2002)
19. Bettini, C., et al.: Free Schedules for Free Agents in Workflow Systems. In: Proc. of 7th Int. Workshop on Temporal Representation and Reasoning. IEEE Computer Society Press, Los Alamitos (2000)
20. Benatallah, B., Casati, F., Ponge, J., Toumani, F.: On temporal abstractions of web service protocols. In: CAiSE 2005 Forum Short Paper Proceedings, CEUR-WS.org (2005)
21. Kazhamiakin, R., et al.: Representation, verification, and computation of timed properties in web service compositions. In: Proc. of Int. Conf. on Web Services. IEEE Comp. Society, Los Alamitos (2006)
22. Pozewaunig, H., et al.: ePERT – Extending PERT for Workflow Management Systems. In: Proc. of Symp. on Adv. in Databases and Information Systems, Nevsky Dialect (1997)
23. Ardagna, D., et al.: PAWS: A Framework for Executing Adaptive Web-Service Processes. IEEE Software 24(6) (2007)